# Tool Support for Collaborative Creation of Interactive Storytelling Media

Paula Klinke, Silvan Verhoeven, Felix Roth,
Linus Hagemann, Tarik Alnawa,
Jens Lincke, Patrick Rein, Robert Hirschfeld

**Technische Berichte Nr. 141**

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam


Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 141

Paula Klinke | Silvan Verhoeven | Felix Roth | Linus Hagemann | Tarik Alnawa |
Jens Lincke | Patrick Rein | Robert Hirschfeld

# Tool Support for Collaborative Creation of Interactive Storytelling Media

Universitätsverlag Potsdam

**Bibliografische Information der Deutschen Nationalbibliothek**
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar

**Universitätsverlag Potsdam 2022**
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665



Druck: docupoint GmbH Magdeburg

**ISBN 978-3-86956-521-7**

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
https://doi.org/10.25932/publishup-51857
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-518570

# Abstract

Scrollytellings are an innovative form of web content. Combining the benefits of books, images, movies, and video games, they are a tool to tell compelling stories and provide excellent learning opportunities. Due to their multi-modality, creating high-quality scrollytellings is not an easy task. Different professions, such as content designers, graphics designers, and developers, need to collaborate to get the best out of the possibilities the scrollytelling format provides.

Collaboration unlocks great potential. However, content designers cannot create scrollytellings directly and always need to consult with developers to implement their vision. This can result in misunderstandings. Often, the resulting scrollytelling will not match the designer's vision sufficiently, causing unnecessary iterations. Our project partner Typeshift specializes in the creation of individualized scrollytellings for their clients. Examined existing solutions for authoring interactive content are not optimally suited for creating highly customized scrollytellings while still being able to manipulate all their elements programmatically. Based on their experience and expertise, we developed an editor to author scrollytellings in the lively.next live-programming environment. In this environment, a graphical user interface for content design is combined with powerful possibilities for programming behavior with the morphic system. The editor allows content designers to take on large parts of the creation process of scrollytellings on their own, such as creating the visible elements, animating content, and fine-tuning the scrollytelling. Hence, developers can focus on interactive elements such as simulations and games. Together with Typeshift, we evaluated the tool by recreating an existing scrollytelling and identified possible future enhancements.

Our editor streamlines the creation process of scrollytellings. Content designers and developers can now both work on the same scrollytelling. Due to the editor inside of the lively.next environment, they can both work with a set of tools familiar to them and their traits. Thus, we mitigate unnecessary iterations and misunderstandings by enabling content designers to realize large parts of their vision of a scrollytelling on their own. Developers can add advanced and individual behavior. Thus, developers and content designers benefit from a clearer distribution of tasks while keeping the benefits of collaboration.

# Zusammenfassung

Scrollytellings sind innovative Webinhalte. Indem sie die Vorteile von Büchern, Bildern, Filmen und Videospielen vereinen, sind sie ein Werkzeug um Geschichten fesselnd zu erzählen und Lehrinhalte besonders effektiv zu vermitteln. Die Erstellung von Scrollytellings ist aufgrund ihrer Multimodalität keine einfache Aufgabe. Verschiedene Berufszweige wie Content-Designer:innen, Grafikdesigner:innen und Entwickler:innen müssen zusammenarbeiten, um das volle Potential des Scrollytelingformats auszuschöpfen. Jedoch können Content-Designer:innen Scrollytellings nicht direkt selbst erstellen, sondern müssen ihre Vision stets gemeinsam mit Entwickler:innen umsetzen. Dabei können unnötige Iterationen über das Scrollytelling auftreten, wenn dieses den Visionen der Content-Designer:innen noch nicht entspricht. Außerdem können Missverständnisse entstehen. Unser Projektpartner Typeshift hat sich auf die Erstellung von, für seine Kund:innen individualisierten, Scrollytellings spezialisiert.

Aufbauend auf Typeshifts Erfahrungen und Expertise haben wir einen Editor entwickelt, um Scrollytellings in der Live-Programmierumgebung lively.next zu erstellen. In lively.next wird eine graphische Oberfläche für die Erstellung von Inhalten mit weitreichenden Möglichkeiten zur Programmierung von Verhalten durch das Morphic-System kombiniert. Der Editor erlaubt es Content-Designer:innen eigenständig große Teile des Erstellungsprozesses von Scrollytellings durchzuführen, zum Beispiel das Erzeugen visueller Elemente, deren Animation sowie die Feinjustierung des gesamten Scrollytellings. So können Entwickler:innen sich auf die Erstellung von komplexen interaktiven Elementen, wie Simulationen oder Spiele, konzentrieren. Zusammen mit Typeshift haben wir die Nutzbarkeit unseres Editors durch die Nachbildung eines bereits existierenden Scrollytellings evaluiert und mögliche Verbesserungen identifiziert. Unser Editor vereinfacht den Erstellungsprozess von Scrollytellings. Content Designer:innen und Entwickler:innen können jetzt beide an demselben Scrollytelling arbeiten. Durch den Editor, der in lively.next integriert ist, können beide Parteien mit den ihnen bekannten und vertrauten Werkzeugen arbeiten.

Durch den Editor verringern wir unnötige Iterationen und Missverständnisse und erlauben Content-Designer:innen große Teile ihrer Vision eines Scrollytellings eigenständig umzusetzen. Entwickler:innen können zusätzliches, individuelles Verhalten hinzufügen. So profitieren Entwickler:innen und Content-Designer:innen von einer besseren Aufgabenteilung, während die Vorteile von Zusammenarbeit bestehen bleiben.

# Contents

# 1 Introduction to Scrollytellings as Interactive Media

In this chapter, we introduce the domain of scrollytellings and explain the difference to other interactive media. We want to take a more detailed look at how consumers on the web interact with scrollytellings and experience them.

Throughout the project, we worked closely with our project partner Typeshift. Typeshift is an agency for digital content that creates interactive websites. One of the products they create is scrollytellings.

The creation process of scrollytellings requires a combination of media creation and software development and consequently exhibits challenges of both of them. To understand how these challenges arise, we describe the components of scrollytellings as interactive media and their creation.

To create such a scrollytelling, people of different professions and backgrounds need to collaborate closely. The core ideas and envisioned message of the scrollytelling need to be transferred between all involved parties. Yet, this transfer is complicated when content designers and developers cannot collaborate in a shared medium. To solve this, we built an editor to have a UI-based tool helping both sides.

## 1.1 Domain

Whereas most websites are interactive in the sense that something is happening due to the consumers' interactions, our focus is explicitly on those websites wherewith their interaction consumers drive the story. While there are different kinds of interactive web pages, we primarily focus on scrollytellings.

Scrollytellings are a kind of web page where the main interaction happens through scrolling while the content is transferred through a told story.

Those stories are the focus since a lot of interactive web pages our project partner creates are scrollytellings. Scrollytellings are becoming increasingly popular for presenting content as used, for example, by the New York Times. Furthermore, scrollytellings are created on a wide range of other topics, including the development of the web,[1] herd immunity against COVID-19,[2] or about why millennials are facing a tough financial future.[3]

[1] https://webflow.com/ix2 (last accessed on 2021-07-28).

[2] https://www.zeit.de/wissen/2021-06/herd-immunity-calculator-covid-end-of-pandemic (last accessed on 2021-07-28).

[3] https://highline.huffingtonpost.com/articles/en/poor-millennials/ (last accessed on 2021-07-28).

To better understand the challenges arising in the creation process of scrollytellings, we first discuss interactive media in general and then gain a deeper insight into scrollytellings and their characteristics.

## 1.2 Interactive Media

Nowadays, authors of new technologies and digital media, in general, have a common goal: "the transformation of experience"[18](p.98). They thrive more and more towards a setting where the consumer is no longer only the audience and recipient but rather the interactor [18]. This term, by its meaning, already suggests that the consumer takes over an active role in the experience.

Traditional media like newspapers and books usually offer only limited opportunities for interaction. To achieve a transformation of experience, richer and more interaction is helpful, so the web and computers are often used. Computers themselves offer many different possibilities to interact with and to get involved as a consumer. When discussing the combination of text, audio, pictures, and video, we chose the term *modality* over *multimedia* as media is a more general communication tool like text, books, or movies [56]. A modality here is an independent channel for communication between a computer and a human. Media on the web offers the opportunity to combine different *modalities*. Hence, we call people experiencing interactive media neither reader nor viewer but consumer. Examples of modalities are animations, hyperlinks, videos, and audio. Using different modalities increases the number of different sensory channels which are used to interact with the consumers. In this work, we define *interactivity* as such that the increase of modalities increases the interactivity of a medium [56].

Consuming media, consumers can easily forget their environment and might not be able to distinguish between the real world and the experience anymore. This is when experience transformation happens, and consumers rather experience media than consume it.

To better understand the consequences of interactive media for consumers, we want to define the terms *presence* and *immersion*. Since scrollytellings are interactive media, as we will see in section 1.3, we have a look at the levels of interactivity and challenges arising in the authoring process of interactive media. We are doing so to understand better what is required from authors of interactive media to see what challenges our project partner faces.

### 1.2.1 Definition

Interactive media is a widely used term with slightly varying definitions, meanings, and usage in literature. One definition is that consumers can access information through their interaction [49]. This includes traditional media like books and newspapers too since consumers need to turn pages to get the information. The definition excludes only movies; since watching a movie, consumers do not need to interact with the device to get information. Nevertheless, in the broadest sense of

the definition, even watching, for example, a DVD-ROM, consumers need to push a play button and can change the speed of the shown content or the chapter.

In the following, we focus on more technical details of interactive media.

In the context of this work, when talking about interactive media, we have video games or websites in mind that consist of a combination of audio, video, text, and graphics. Interactive media usually allows consumers to have some control over the way of the media presentation and to interact with the media [15]. We see interaction here alters the medium due to the consumers' input [17].

Interacting with the media at hand lets the consumers absorb the content at their own pace. Nowadays, the focus is more and more on building immersive experiences. Thus, letting the consumers forget about their surroundings and getting the feeling of presence (see subsection 1.2.2) for the virtual world. Even though this might have happened already in traditional media like books, making the consumer an interactor, when creating immersive media, makes them feel even more involved in the story.

The extent to which consumers can interact with media and the possibilities of how they interact differ a lot. Consumers can interact through the keyboard and mouse in a classical way or gestures on a touchscreen or touchpad. With new technology like virtual reality, authors of interactive media can increase immersion, and the probability of presence rises a lot. Due to the development of those new technologies, borders between the consumers' life and the stories they are experiencing blur even more.

Interactive media puts the consumer in control over parts of the experience, and by this, interactive media systems are more immersive. To understand why interactive media is immersive and creates presence in the consumers, we want to look at what those terms represent.

### 1.2.2 Presence

Presence is the "subjective experience of being in one place or environment, even when one is physically situated in another"[63]. It is a misperception of the brain which leads consumers into thinking they were located in the experience.

Effects of presence are dependent on the medium and the person consuming it. The effects can include flinching or ducking synchronous to something happening in the experience [40]. Even though reviewers of movies or video games may not talk about presence, they use "breathtaking"[4] or "pulse-pounding"[5] to describe the experience of the media they are reviewing. Additionally, some people experience vection, the illusion of self-movement, and, resulting from that, sometimes simulation sickness. Other effects are training skills, as, for example, aircraft pilots do with simulations of airplane flights. Additional effects of presence are enjoyment,

[4] https://www.theguardian.com/film/2020/sep/16/the-eight-hundred-review-chinese-alamo-warehouse-regiment-shangai-war (last accessed on 2021-07-28).

[5] https://murder-mayhem.com/best-thrillers-you-can-watch-right-now (last accessed on 2021-07-28).

involvement (consumers experiencing information instead of abstractly processing it), desensitization, and even the memory and social judgment might be changed. Usually, when people have experienced presence, they cannot remember where the information came from but are, when experiencing more presence, reportedly more likely to remember the information [40]. That is why presence is essential when using the aforementioned effects in fields like education, health and medicine, computer science, and many more [41]. The misperception we call presence gets increasingly common.

While many of the effects of presence remind us more of movies or virtual reality, presence is important for interactive media in general and especially websites and scrollytellings. It is important as, for example, better remembering received information is also one goal of scrollytellings.

### 1.2.3 Immersion

Immersion and presence and, resulting from that, how to measure and compare them across different media and levels of interactivity are highly discussed terminology. While some use immersion in a very technical and measurable manner, defining it as "the extent to which the actual system delivers a surrounding environment, one which shuts out sensations from the real world, [...]"[55] others define immersion as "a psychological state characterized by perceiving oneself to be enveloped by, included in, and interacting with an environment that provides a continuous stream of stimuli and experiences"[63].

Thus, a system is immersive when making consumers able to be surrounded by and involved in the experience. At the same time, presence is a state of mind where the consumers forget about their surroundings and "are in the experience". To create this experience, factors for immersion are, besides others, the modes of interaction, control over, and the perception of being included in the experience [63].

### 1.2.4 Levels of Interaction

Having a better understanding of the effects of presence, we understand that we want to have an immersive system with a higher chance of creating presence in the consumer. To be able to maximize the possibility of consumers experiencing presence, we will have a look at the different levels of interaction and their effects on consumers.

When thinking about interaction, authors should not only thrive to more interactivity as it raises the consumers' interest but should also think about how the consumers perceive different levels of interaction. Here we will have a look specifically at interactive websites. Although there are no absolute guidelines for how authors should design interactive media, we will discuss some modes of interactions and their effect on consumers.

**Enabling Interaction** Studies have found that consumers will interact more when having different possibilities of interacting with websites [56]. It seems that the

simple existence of links, a comment section, and video and audio parts increases the perceived positive perception of a website. Nevertheless, having to navigate more through a website, consumers rated the structure of the website worse [56]. It takes up a lot of the capacity of the consumers' minds and ultimately leads them to not focus on the possibilities of interaction anymore. Thus, authors should make interactions on websites possible because consumers do enjoy it while at the same time not giving too many choices as not to overwhelm them.

Other studies found that even uninvolved or uninterested consumers tend to have greater involvement in a website when needing to interact with it to access content [56]. However, for all groups of consumers, interactivity seems to bring them more in touch with the content than when only reading a static text without interaction. They seem to engage more with the content and remember it better when needing to interact with the site by, for example, clicking. Even though the structure of websites might be received as worse when consumers need to navigate a lot, they state to have learned more about the topic.

When asked about peripheral content on a page, consumers reviewed low interactive advertisements with animations as more involving and leading to a better review of the product, whereas animations had no impact on highly interactive advertisements [56]. Animations on high interactive ads seemed to be seen more as a distraction than a helpful feature.

From the ongoing scientific body of findings on how consumers perceive interactivity, this limited list of examples already extensively demonstrates that interactivity used right, can have a positive impact on how consumers perceive a website. It gets consumers more involved, they get to know the topic better, and they have a greater feeling of having learned something.

However, as interactive media offers many possibilities, authors should also limit the interaction possibilities not to mislead and overwhelm consumers.

**Limiting Interaction** This raises the question of how much interaction is too much interaction and how to find a balance between giving consumers a choice and leading them through the experience.

Technology-based interactive media started with CD-ROM and DVD-ROM. Having placed games on those playback devices or any other closed system, there are only a finite number of possible interactions the authors have thought of, thus limiting interaction possibilities [15](p.7). Having this limited set of possibilities, authors have thought of it requires them to think about what consumers might want to do and at the same time restrict consumers' interactions to get the desired experience. Authors are also in a position to limit the possibilities as such that consumers are neither overwhelmed nor confused.

Nowadays, the web offers games as well as interactive articles. On the web, consumers can have possibly an infinite number of hyperlinks to external resources and, therefore, infinite possibilities of what to see and how to interact with the media at hand. Due to the infinite possibilities, designers and authors of interactive media on the web must limit and maintain some control over the possibilities of interactions for the experience to be effective for the consumer [15](p.7). Consumers

are overwhelmed and review websites with too much interactivity, as we have seen, not as positive.

In this work, we focus on scrollytellings which limit the main interaction, as the name implies to scrolling. We will investigate them further in section 1.3.

But first, we will have a closer look at the challenges that arise when creating interactive media.

### 1.2.5 Challenges

As aforementioned, interactive media is not a new form of presenting content to consumers, and there already exist studies about what works well and what does not [56]. There are still various challenges that remain when creating interactive media. Due to the wide range of challenges, we will only consider a few examples.

One of the challenges in creating interactive media is to make consumers aware of the different supported possibilities of interaction [15](p.18). Interactive media may suffer from low discoverability as it may not be obvious to consumers that the main interaction in scrollytellings is to scroll. For example, when we created the first longer scrollytelling during our project and gave it to consumers, many reported that they did not know what to do with it. Even though when testing it with our team, due to our shared knowledge, it was clear that we need to scroll, but the consumers were confused, and some reportedly left the page frustrated. With that specific scrollytelling, we failed in making our consumers aware of how to interact with the web page.

General design challenges can be applied to interactive media as well. For example, when using icons to show what interactions are possible, authors should choose a limited number and stick to them. Nevertheless, icons are not always apparent to consumers [15](pp.18–19). To make them more understandable, authors can also use labels. Even though it might seem obvious and applies to other fields as well, authors should use obvious abbreviations and clear and unique headings to guide consumers through the experience.

While those are some challenges for authors when creating interactive content, there are other requirements towards the authors themselves, which might be challenging. Since computer-based interactive media often consists of different modalities, authors of interactive media need to know film, writing, photography, programming, graphic design, animation design, and a lot more [49]. For this reason, the creation of interactive media is often a collaborative effort between different specialists. For such a team to be able to work effectively, respective tools are required. To make the work more effective, we built our editor, which will be introduced in section 1.5 and covered in more detail in chapter 3.

Looking at interactive web articles, other technical challenges occur. One is that web technologies change fast and often [21]. Authors of interactive media do not only need to think about how to build and publish these articles today but also how to archive the article so that their work is preserved. One example is flash animations which were widely spread, and nowadays, most mainstream browsers block those

completely. It is even recommended to uninstall the Adobe Flash Player.[6] Another challenge of creating such articles is the creation for different devices and different bandwidths [21]. A web article and its interaction possibilities may look good and work on desktop devices but needs to be adapted for mobile devices in interaction possibilities and screen size. Additionally, making interactive web articles accessible is more complicated compared to static writing. Combining audio, video, text, and interaction, authors need to think about new ways to make this content accessible for everybody.

Nevertheless, authors of interactive content should always consider whether or not interactivity might help consumers understand and engage more with the content or whether it is more a way of distraction [21].

Despite all the challenges, interactive media have an impact and let consumers learn more because, through the opportunity of interaction and experience at their own pace, interactive media and, therefore, scrollytellings are more immersive and have a higher possibility for creating presence. Understanding what interactive media in general is and what to consider creating interactive media, we want to take a closer look at scrollytellings in specific and see how they integrate into interactive media.

## 1.3 Scrollytellings

Now that we have an overview of interactive media, we will take a closer look at scrollytellings. While conventional web pages often consist of text that consumers read, scrollytellings let them experience a story through scrolling. Scrolling is the primary interaction and represents the velocity of how fast the story evolves. Due to scrolling representing the velocity, consumers have control over the evolution of the story.[7]

Scrollytellings are interactive media since scrollytellings consist of different modalities and consumers get information only when interacting with the web page. In addition, scrollytellings can have parts where consumers directly interact with the content, for example, via a touch gesture or mouse to change an infographic and learn about the consequences of different events.

Newspaper publishers who use scrollytellings are more and more present in today's journalism and content creation for interactive web pages. For example, the New York Times created the scrollytellings "Why the Mexico City Metro Collapsed"[30] and "Snow Fall"[5]. We will have a closer look at both scrollytellings in subsection 1.3.3

The increasing popularity of scrollytellings is because consumers have a multi-dimensional experience of the story, which they can encounter at their own speed.

[6] https://www.adobe.com/products/flashplayer/end-of-life.html (last accessed on: 2021-07-28).

[7] We use velocity and speed of an animation interchangeably when referring to the rate of change of animated properties.

Combining texts, graphics, videos, animations, and sounds with interactive elements creates an experience where different senses are involved. Additionally, consumers may adjust the speed at which they receive the scrollytelling's content, which helps to absorb the topic.

Having had a small introduction on what makes scrollytellings different, a definition of scrollytellings follows. Afterward, we will look at two examples and see how scrollytellings differ from books, movies, video games, and traditional web pages.

### 1.3.1 Definition

The term scrollytelling is a portmanteau of the word "scrolling" and the word "storytelling". These words refer to how the story evolves, which is through scrolling and the often vertical arrangement of the story. While at the same time, they are telling a story in contrast to writing about the daily news.

"Snow Fall"[5] is one of the first scrollytellings published in 2012 by the New York Times. Some identified it as being highly influential [64, 48]. We will have a closer look at that specific scrollytelling in subsection 1.3.3.

Scrollytellings are mainly linearly told stories. The written text makes up the core of the web pages, accompanied by other elements like video, pictures, and animations. Often they are stand-alone web pages [64].

### 1.3.2 Structure and Elements of Scrollytellings

Given the above definition, we now want to explore the design space of scrollytellings. While looking at specific examples of scrollytellings, we want to get a broader impression of what scrollytellings consist of.

Most of the scrollytellings we have seen during our project are linear stories with no chapters or sub-pages but rather one stand-alone web page. "Snow Fall", as we will see, stands out as an exception here since it consists of chapters. Moreover, each chapter is a small story on its own.

In addition to being an experience consisting of different modalities, scrollytellings play with the experience of scrolling. Scrolling is interaction consumers are highly used to and do not think about anymore. Being used to scroll a page also scrolling the page's content down, it might confuse consumers at first that scrolling down in a scrollytelling makes the story evolve more than moving the page. As a stylistic device, new content can come into sight from different directions of the screen. Moving the page in another direction than down breaks with the consumers' assumption of shifting the viewport when they are scrolling a page down but at the same time leads to immersion and the feeling of control for the consumers.

Since scrollytellings change the meaning of scrolling and consumers do not necessarily change only the viewport when scrolling down, scrollytelling authors are responsible for arranging how consumers see the content. While on traditional web pages with static pictures, consumers are responsible for scrolling exactly as far so that they can, for example, have a good look at a picture. In scrollytellings, those

pictures could fade in, and authors can decide exactly when the picture comes into sight. The possibility of arranging the content takes some of the responsibility from the consumers back to the authors.

Since scrollytellings transfer a lot of the content to the consumers by animations and graphics, it is essential for the animations to "feel natural". That an animation "feels natural" is influenced, for example, by good timing, which can be the temporal relationship between objects appearing or moving. Misunderstandings may arise for consumers when animations do not have good timing or are not triggered at the correct scroll position [61].

While creating scrollytellings is more complex than creating a static web page, it also gives authors much freedom. The degree to which a web page has scrollytelling elements varies a lot. While some scrollytellings might have some scroll-based animations and graphics at first, they might evolve into a more static web page later. Other scrollytellings might feel like a movie with many time-based animations, which only sometimes require consumers to interact with. Furthermore, some scrollytellings have all of the above elements all the way through.

Additionally, the scrollytelling "Über Flockenbau und Sturzflüge"[8] built by our project partner has build-in stops to prevent consumers from scrolling too fast through the web page. Even though it seems like a good idea to pause at some points where consumers have to either read or see important parts or not scroll through the experience too fast, we have not yet discovered this mechanic in any other scrollytelling in our research.

Now that we have an abstract understanding of what scrollytellings are, we want to have a closer look at some examples.

### 1.3.3 Two Examples of Scrollytellings

In the following section, we discuss two exemplary scrollytellings published by the New York Times in greater detail. The first one is "Snow Fall" [5] from 2012, which is considered by some to be a breaking point in digital journalism [11] and starting point for digital long forms [48]. The second one is "Why the Mexico City Metro Collapsed"[30] from 2021. We chose those two examples since "Snow Fall" is one of the first scrollytellings and in contrast to that a recent scrollytelling with "Why the Mexico City Metro Collapsed". Both were built by the New York Times, indicating that even though "Snow Fall" was created back in 2012, the genre is still around today and gaining popularity.

**"Snow Fall"** "Snow Fall: The Avalanche at Tunnel Creek"[5] is a scrollytelling that won the Pulitzer Prize in December 2012. It is "the urtext" [11] of a scrollytelling. "Snow Fall" is a story about an avalanche at Tunnel Creek in Washington State in the Western US in February 2012, which killed 3 out of a group of 16 skiers.

[8] https://typeshift.io/snowflakes/ (last accessed on: 2021-07-28).

Thus “Snow Fall” consists of a written story accompanied by pictures, videos of the involved skiers, animations, and audio snippets of interviews with the survivors. Having five chapter-like parts, the scrollytelling “Snow Fall” leaves it up to the consumer to unfold the story of the 16 skiers. Especially videos and animations playing in silent loops create the impression of silence in the snow and set the atmosphere for the scrollytelling. Pictures of the mountain pass and skiers make the consumer dive into the story. In addition to that, the effect of photographs and videos scrolling away like a curtain makes the scrolling more playful and gradually reveals content. It is an homage to the curtains of theaters and cinemas [11]. Changes in the location are often represented using big pictures, looped films, or animated maps to visualize the difference. All location changes have in common that they are using the curtain effect. One, arguably sad, the key scene of “Snow Fall” is a video filmed with a GoPro that records the discovery of one of the three dead group members. While being a multi-dimensional experience, “Snow Fall” does not replace text but instead accompanies it and enriches it to build an even better understanding of the story [11].

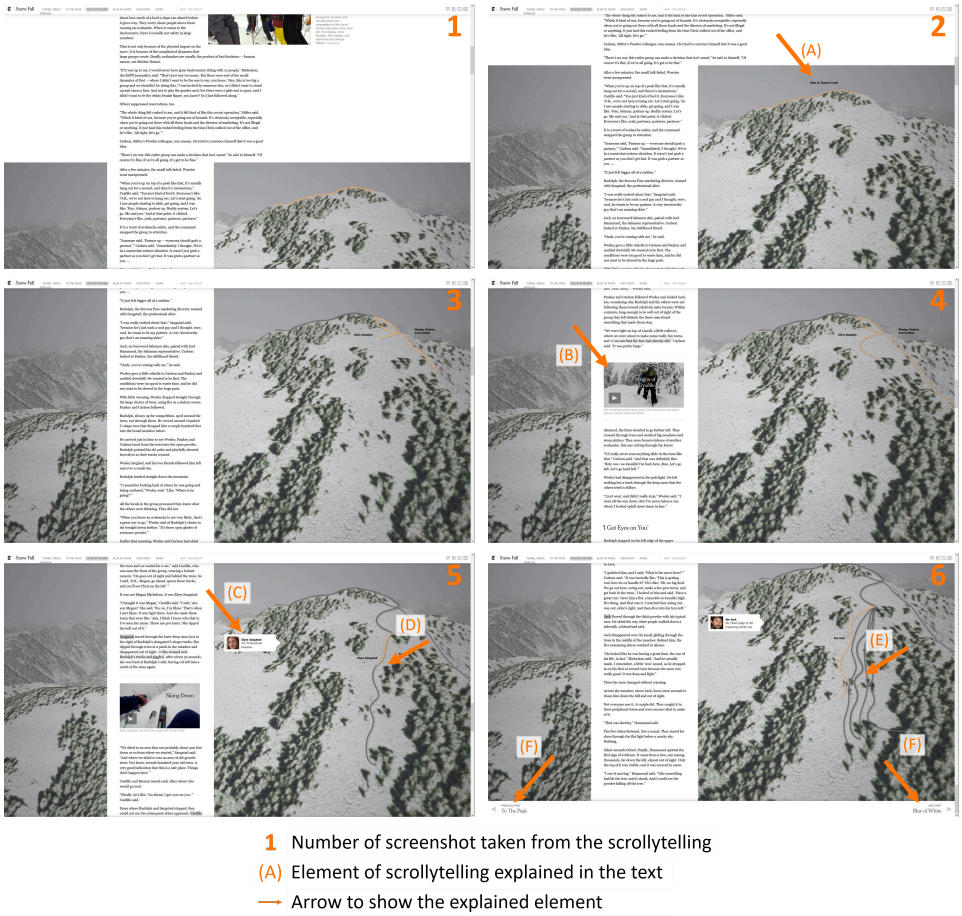


**Figure 1.1:** A scroll based animation from “Snow Fall: The Avalanche at Tunnel Creek”[5]

While "Snow Fall" contains multiple time-based animations, we want to take a look at one specific scroll-based animation from the end of the third chapter in Figure 1.1. In the first screenshot, we can see how a white page, consisting of text and containing only some pictures, transitions into a page where a drawn picture with an animation becomes the focus besides the text. As seen in screenshot 1 (A), orange lines with annotations appear based on the scroll position. When scrolling further, the part that was already shown turns gray and fades into the background while the new elements appear orange. Those lines visualize the paths the skiers took that day. Even though the animation takes much attention from the consumer, we can see in screenshot 4 that still videos in the written story are shown with (B) and audio snippets. While the animation visualizes the paths the skiers took, the told story with video and audio creates the atmosphere around it. In doing so, the author made "Snow Fall" more immersive. Thus, the consumers do not need to know or research the surroundings to understand which places the author describes in the story. To establish a sense of the skiers' movement, some of the lines evolve and grow while scrolling, as highlighted in (D). This can be seen in screenshot 5. In addition to the evolving orange line, the author annotated the animation with a photograph and short description of the person whose path is in focus at this moment. One example of an on-site photograph can be seen in (C). In screenshot 6, (E) highlights all the routes taken by the skiers and where they ended. We see the complete picture, and the animation is over. Additionally, we see in screenshot 6 the aforementioned division into chapters. With the left and right-pointing buttons (F), the consumer can now navigate forward or backward in the story.

Apart from being an engaging online experience, "Snow Fall" is also a well-researched and written story. While researching the story, the author John Branch, for example, interviewed all survivors of the avalanche and spoke to snow scientists from Alaska.[9] Combining both a well-written good researched story and fine-tuned elements like videos, pictures, audios, and animations made this story widely known and inspired other story writers to emulate "Snow Fall" [11].

**"Why the Mexico City Metro Collapsed"** Now that we analyzed one of the first scrollytellings, we will now have a look at a scrollytelling which was created and published about nine years later in the New York Times too. The scrollytelling "Why the Mexico City Metro Collapsed" [30] was published in 2021 and is a story about the possible reasons for why the metro in Mexico City collapsed, causing 26 fatalities.

"Why the Mexico City Metro Collapsed" starts with a full-screen picture and the display of the heading. Nevertheless, in contrast to "Snow Fall" the starting picture of "Why the Mexico City Metro Collapsed" evolves into an animation controlled by the consumer's scroll. Zooming in, a driving metro, small at first, creates the impression of zooming into the story. White boxes with text scrolling on top of the animation do not take too much attention from zooming into the surrounding. Since having looked at an animation that helps consumers understand the content in the example

[9] https://www.pulitzer.org/files/2013/feature-writing/branchentryletter.pdf (last accessed on 2021-07-28).

above, we want to have a look at an animation that creates an atmosphere and takes consumers into the story. The animation is shown in Figure 1.2.


**Figure 1.2:** A scroll based animation from "Why the Mexico City Metro Collapsed"[30]

Figure 1.2 shows the animation opening the story. First, we see the title of the story. When the title is scrolled out to the top, the background animation stays, and the story begins. Screenshot 2 (A) shows a small metro that starts its ride while scrolling down. Consumers focus on the animation since the text is provided only in short snippets on white boxes that scroll through while the animation in the back continues. We can see such a text box in screenshot 2 in (B). The dotted arrows suggest the direction of the text boxes in screenshots 1, 2, 4, and 5 scroll on top of the animation. The train starts its ride, and we are following while once in a while, in the white text boxes, excerpts from a personal story, start to come up. While the train continues its way, the perspective changes into zooming onto the train and getting closer until it fades out, as shown in screenshot 6, and the article starts. This

animation does not focus on illustrating parts of the story or explaining it but instead aims to take consumers with it to look at the Mexico City metro.

Afterward, an article-like text starts, accompanied by another schematic animation illustrating what could have already been read in the text. It explains and visualizes in more detail what went wrong when constructing the metro. Those animations take consumers back into full-screen by looking at the overpass' high-level constructions first and then focusing on smaller architectural details of the constructions, succeeded by much text. It is accompanied by schematic and realistic pictures as well as a video from that night. The pictures and the video are embedded statically into the site and scroll with the text. But the pictures are not only standing for themselves; some of the pictures of the collapse provided are also annotated to show the important aspects and to provide visual proof for what was stated in the surrounding text. Thus, "Why the Mexico City Metro Collapsed" evolves more into a static traditional web page.

In the end, the article gives an outlook on a new train track under construction which might have the same construction issues as the metro line.

**Learnings** To conclude, one can say that even though "Snow Fall" was created nine years before "Why the Mexico City Metro Collapsed", both share some elements. Both start with full-screen visuals; "Snow Fall" with a looped video and "Why the Mexico City Metro Collapsed" with a scroll-driven animation. Likewise, each of these scrollytellings contains raw footage from around the accident.

Nevertheless, "Snow Fall" is much longer and the modality-richer scrollytelling. It has five chapters, and throughout the entire scrollytelling animations, videos and pictures are placed. In contrast, "Why the Mexico City Metro Collapsed" only starts with two animations but later is more like an article accompanied by pictures rather than a modality-rich scrollytelling.

We learned from examining both scrollytellings that for a scrollytelling both is needed: a story in the form of well-written texts and different elements like animations or videos, which need to be highly fine-tuned to feel natural together.

Due to this fact, we can conclude that we need different people with different abilities working closely together for scrollytelling creation.

### 1.3.4 How Scrollytellings Differ From Other Media

Now that we have seen in subsection 1.3.1 that scrollytellings are a story told through scrolling and have looked at examples, we want to see how scrollytellings differ from other media. Scrollytellings are a type of web page where many different aspects from other media are combined. However, major differences can be found.

For a better understanding of what is possible when creating a scrollytelling, how scrollytellings differ, and why they are their type of media, we will compare them to physical books, movies, computer games, and traditional static web pages. For all those kinds of media, we want to discuss:

- the need and possibility of interaction to get information,
- how much consumers are involved in the experience,
- whether consumers get information at their speed,

- how consumers can navigate to resources outside of the media,
- whether authors need to consider timing,
- whether the media is a combination of different modalities,
- whether it is easily printable,
- whether it is easily referenceable, and
- whether the story needs to be linear.

Recounting the main characteristics of scrollytellings: Scrollytellings need consumers to interact with them to get information. While they might also have videos or time-based animations, most of the experience happens when consumers scroll on the web page. Sometimes scrollytellings can even have interactive parts where consumers interact with the mouse or touch gestures. This way, consumers get information at their preferred speed and are involved in the experience. With scrollytellings, it is easy to embed links to foreign web resources outside of the scrollytelling into the experience, which does not require consumers to switch to another medium and come back later. Scrollytellings are a combination of different modalities. Thus, consisting of text, graphics, animations, audios, and videos, and scrollytellings are often linear stories. Additionally, in scrollytellings, authors need to think about the content and the timing of the different modalities working together. In scrollytellings, an important part is the presentation of the content, so authors have to think about what parts of a scrollytelling consumers need to see together and how they should appear. Nevertheless, printing out scrollytellings is usually not an option due to the importance of scroll-based animations. At the same time, it can be hard to reference scrollytellings, especially when they do not consist of different chapters. Describing at which point in scrolling down something is located is not an easy task for consumers.

With the above questions to be answered about the other media, we want to see what scrollytellings have in common and what sets them apart from other types of media.

**Comparison to Books** Printed books are the oldest of the here compared mediums. According to the definition that interactive media is the media consumers need to interact with to get information, books are indeed, in the broadest sense, interactive media, just like scrollytellings. Consumers need to turn pages to be able to read the book.

With books being interactive media, consumers are involved in the experience and can define the speed at which they want to learn about the content. Like scrollytellings, books can combine different modalities since they can consist of pictures and text. Nevertheless, they are bound to static pictures, while scrollytellings can also have animations and videos. Similar to scrollytellings, elements are arranged on book pages and presented to consumers so that consumers no longer have to worry about the arrangement of the elements like the text and graphic combination. While there exist books with different options of how a story can evolve combined in one book, it is more common that books are telling a linear story just like scrollytellings.

In contrast to scrollytellings and media on the web, it is difficult to make references from a book to an outside source. When consumers read a book, and it references another book, website, or video, they need to put the book away, leave the medium, and return later. Timing in a book is not essential since there are static pictures that consumers can observe at once due to the pages a book consists of. Furthermore, books are no continuous experience due to the pages while scrollytellings except when having chapters can easily be one continuous experience. Nevertheless, pages enable the opportunity of referencing easily and reasonably. Due to the continuous experience of scrollytellings, referencing something in a scrollytelling is difficult. Since a scrollytelling does not consist of pages, consumers might see only parts of a line of text or picture when authors did not create it in another way.

Scrollytellings are linear story-based web articles. Thus, authoring scrollytellings, by definition, has a lot in common with text editing. Nevertheless, scrollytellings are a lot more than just text and pictures, and therefore, authors need more possibilities to create them.

**Comparison to Movies** Movies are the first digital medium in our comparison. Movies per se do not count as interactive media since consumers do not need to interact with them to retrieve information. Even though one can argue that watching a DVD is interactive since consumers need to push a play button or can jump between different chapters, we want to look at only the experience itself of watching the movie.

What movies have in common with scrollytellings is that they can consist of different modalities. In movies, as in scrollytellings, text with graphics, videos, and animations can be combined. Nevertheless, text is not too common in movies, except in title cards. Additionally, in movies, timing is an important issue. Animations and cuts, for example, need to be timed correctly so that consumers do not get confused and have a pleasant experience. The importance of timing is the same as in scrollytellings. In addition, movies and scrollytellings can both be a very visual experience. Like scrollytellings, movies have only one predefined and therefore linearly told story. Additionally, movies and scrollytellings have in common that they are not easily printed due to being a continuous experience.

However, unlike scrollytellings, consumers of movies have no need and no possibility to interact with and influence the story. Additionally, consumers cannot influence the speed at which they receive the information. Nevertheless, referencing movies is more straightforward than referencing scrollytellings since time is a discrete value, available to be read and jumped to or sought. In contrast, in scrollytellings, the scroll position usually is not disclosed to consumers.

Even though movies are not interactive, they already combine different modalities and have a linearly told story. They are already a complex medium that needs different people to create it, like story writers, movie editors, animation artists, or people who shoot the movie.

Now we have seen that we can think about scrollytellings as of movies where the time is reflected in the scroll position. This conceptual model works except for the fact that we can embed time-based animations and interactive parts in the scrollytellings too.

**Comparison to Computer Games** While in movies, consumer interaction is not possible, computer games are designed for consumers to interact with. Computer games are a prime example of interactive media as consumers interact with them for the game to continue.

Depending on the computer game, consumers often are more involved in playing it than when experiencing a scrollytelling. Even though in scrollytellings, consumers can experience presence as well. Just like scrollytellings, computer games can consist of different modalities and often are an audio-visual experience. Authors need to time different modalities like audio and video correctly to create the right atmosphere for the consumers. Similar to scrollytellings, video games are not printable, and depending on the computer game, it can be hard to reference exact points in the game.

In contrast to scrollytellings, the story being told in a computer game does not need to be linear. Nevertheless, in computer games, the story can be linear to various degrees depending on the game mechanics. For example, a tutorial or a player required to visit checkpoints or craft specific items before being allowed to proceed can be a guided, strictly linear story. On the one hand, there often are many different possible scenarios in computer games, and the consumers choose between them with their interactions. On the other hand, there are scrollytellings which are mainly linear-told stories. Nevertheless, scrollytellings can also have small games included for the consumers to interact with.

To summarize, computer games are more interactive than scrollytellings and have, in many cases, a lot more interaction-driven stories. Nevertheless, interaction in the form of small games or simulations can be part of a scrollytelling.

**Comparison to Traditional Web pages** Traditional web pages are concept-wise the closest relatives to scrollytellings of the compared media. With traditional web pages, we are thinking, for example, about news articles where the publishers put the content which was printed in a newspaper on a web page. They might have hyperlinks, pictures, or even videos but are static web pages where scrolling only shifts the viewport. Since consumers need to interact with the web page to get information, traditional web pages are interactive media, just like scrollytellings.

Traditional web pages and scrollytellings have in common that consumers can and have to scroll down and, by doing so, interact with them. By scrolling, consumers get actively involved in the experience while getting information at their speed. They have a linearly told story and can be a combination of different modalities. Additionally, traditional web pages like scrollytellings make it easy to reference resources outside of the experience, and other resources can even be embedded.

In contrast to scrollytellings, consumers have to arrange the site on a traditional web page. Since graphics are placed in the text, consumers need to scroll to the correct position to look closer at them. At the same time, they might need to scroll back and forth to see a graphic referenced by the text. Web pages are easily printable and can be read like a book afterward. Printing removes the small viewport that is otherwise moved by scrolling. Depending on the web page, referencing can be possible when headings or paragraphs contain linkable HTML anchors.

Now we have seen that traditional web pages have a lot in common with scrollytellings. Nevertheless, traditional static web pages can be viewed as one long page with text and graphics, while for scrollytellings, animations and timing with the scroll position are highly important.

### 1.3.5 Classification of Scrollytellings

Table 1.1 offers a combined overview of the different types of media and their characteristics, as listed in subsection 1.3.4. At this moment, we shortened "computer games" to game and "traditional web pages" to web page. Adjustable speed refers to the question of whether consumers get the information at their speed.

| Characteristics | Book | Movie | Game | Web page | Scrollytelling |
|---|---|---|---|---|---|
| need of interaction | x | - | x | x | x |
| adjustable speed | x | - | x | x | x |
| including resources | - | - | x | x | x |
| timing is important | - | x | x | - | x |
| modality combination | static | x | x | x | x |
| printing possibility | x | - | - | x | - |
| referencing possible | x | x | - | x | - |
| linear story | x | x | - | x | x |

**Table 1.1:** Table to compare the different characteristics of media shown in subsection 1.3.4. "x" highlights the presence while "-" highlights the absence of this characteristic.

We can see, traditional web pages in their characteristics are close to scrollytellings; except for the fact that timing is an important part of scrollytellings. Therefore authors need to consider the timing for scrollytellings as well. While for traditional web pages, there can be a text document and some pictures placed in between the text, our conceptual model of scrollytellings needs to be different from a traditional web page. Printing a traditional web page gives us a similar experience to reading it on the web, while this is not true for scrollytellings. Traditional web pages can have referenceable sections and chapters, which is not as common for scrollytellings. As we have seen in subsection 1.3.3 an essential part of scrollytellings is animations that the consumers drive by scrolling.

This reminds us more of a movie where it is possible to regard time as a continuously changing scroll position at a fixed pace. In movies, editors can plan how content is presented to the consumer, different animations need to be fine-tuned to feel natural and need to be placed in the correct order. Thus, in a sense, we can think about scrollytellings as about movies.

Sometimes scrollytellings have interactive parts so that consumers can alter the content, try out impacts on different scenarios or play a small game within the scrollytelling. Due to that, there can be characteristics of computer games in scrollytellings as well.

**Summary of Scrollytellings** In this section, we have seen a definition of scrollytellings, looked at various examples, and found similarities and differences to other media. Scrollytellings consist of very different parts in different proportions. Thus authors can think about scrollytellings in very different ways. Authors can think about them as traditional web pages with, for example, scroll-based animations and pictures fading in and out in consequence, or they can think about them as a movie where consumers control the progress via scrolling and embed interactive content. Other authors might create an experience that reminds us more of a game that only has some small parts controlled by scrolling.

To support our project partner Typeshift in creating custom scrollytellings and enable a high variety of scrollytelling designs, we collaborated with them and combined different aspects of scrollytellings. Yet, during our project, we focused foremost on the interpretation of scrollytellings as a scroll-driven movie. Additionally, the scrollytellings we had in mind when creating our editor were not as text-driven as the examples we have seen in this section. We had mainly one example[10] in mind during our project, which is why our editor and this work is more focused on the animation of different objects rather than having a long, told story.

In section 1.2 we have seen that due to the different components interactive media, and therefore scrollytellings, consist of, different people need to work closely together. To better understand why specifically, our project partner wanted a tool to create scrollytellings more easily and what our solution is, we will have two short sections covering both.

## 1.4 Our Project Partner Typeshift

Now we give an overview of our project partner's process for creating web content and some of the difficulties that occur.

Typeshift is an agency for digital content, creating, besides other products, scrollytellings. It consists of content designers and developers. The content designers speak with their clients and create the storyboards. Storyboards are sequences of pictures and texts describing different scenes of a scrollytelling. The developers then have to build the web pages. This results in highly customized and individual scrollytellings.

Figure 1.3 depicts the current workflow. It starts with (1) content designers talking to their clients and creating a story. If customized and complex graphics are required, they will also talk to external graphic designers. Those graphics can be pictures

[10] https://typeshift.io/snowflakes/ (last accessed on: 2021-07-28).

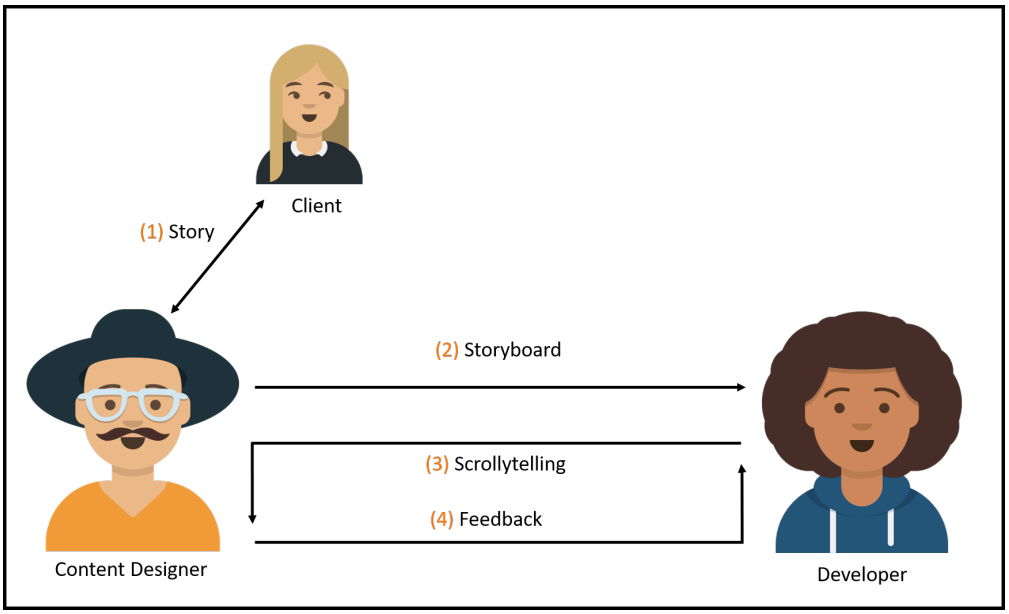


**Figure 1.3:** Current workflow of our project partner Typeshift

and Lottie animations (see subsection 4.3.5) which will then be integrated into the scrollytelling. The story created by the content designers and clients, as well as the graphics designers' files, need to be transferred to the developers (2). Often the story gets transferred in the form of a storyboard. The transfer is difficult, and misunderstandings happen frequently because the content designers already have a vision in their mind which also needs to be transferred. At the same time, not all visions of content designers might work for consumers in reality.

The developers then build the scrollytelling according to the storyboard. They will give the current version of the scrollytelling to the content designers, as seen in (3). When the content designers experience the first version of the scrollytelling, quite often, small details will still require changes. Those details can be timing, placement of objects, or the overall feeling when interacting with the scrollytelling. In the current process, this feedback is usually expressed and transferred in writing. The content designers will provide the feedback (4) using an external tool like writing a message in another application. Currently, discord[11] or e-mail is used at the moment. These frequent and prolonged feedback loops have proven to be very inefficient.

In addition to having to transfer the vision of the content, it is also hard to explain at what point which interactive or animated element needs to be placed or timed differently. As explained in subsection 1.3.2 animations make up an essential part of a scrollytelling which is why it is important for them to feel natural and have the desired effect. In addition to the fact that it is hard to express in written form how to time a scrollytelling's elements, content designers are not able to try things out for themselves to see what feels the best but instead ask developers to implement the

[11] https://discord.com/ (last accessed on 2021-07-28).

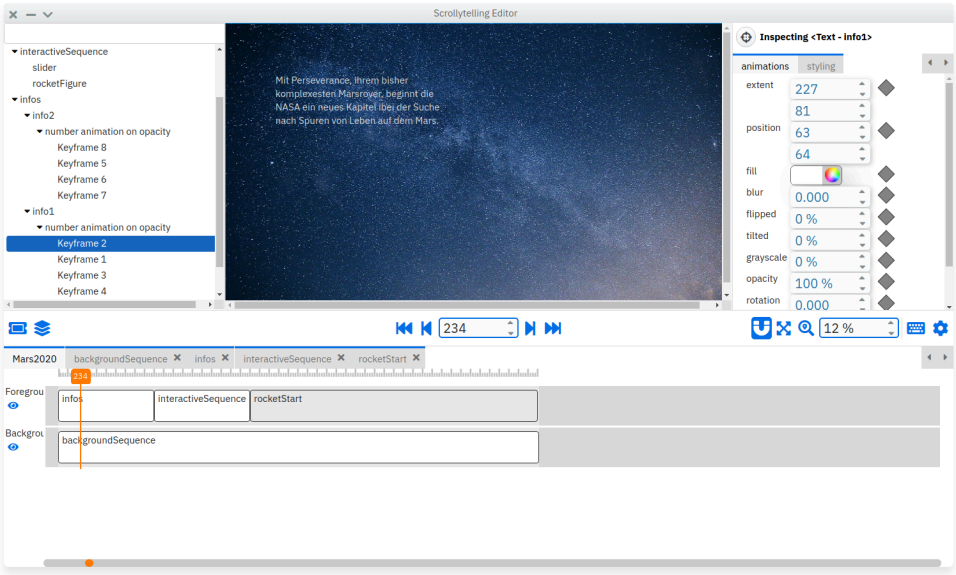


**Figure 1.4:** The editor with an example scrollytelling

changes. Developers then alter the web page and only then get feedback from the content designers again. Developers cannot always do those changes independently since the vision of the scrollytelling was created by the content designers, and the developers may lack the required domain knowledge. A more detailed look into the process and the problems of the process of our project partner is given in subsection 2.1.2.

To address these issues, we built an editor where content designers can create the main story of a scrollytelling and later in the process can change most of the objects and animations themselves. We want to shortly introduce the editor, which is explained in more detail in chapter 3.

## 1.5 Our Solution: qinoq

There are different approaches on how to solve the problem of creating interactive web content, many being time-consuming and expensive. Often it is custom code written by developers. One method is using Idyll [9], a markup language that provides elements to make the creation of interactive web pages more accessible. It is invented to help content creators to build interactive web content on their own. Nevertheless, it is still a written language in which content designers need to express the content. However, as we have seen, scrollytellings are a visual experience. Therefore, Idyll offers no help to visualize what authors have in mind and does not support them in thinking about how interactivity would be the most consumer-engaging.

We opted to build the project qinoq, including an editor to help our project partner and enhance their development cycle. The editor aims to help content designers create basic scrollytellings from scratch without writing code. Developers only need to enrich the scrollytellings with elements that cannot be created using our editor.

To assess the requirements for such an editor, we first thought about a general structure of scrollytellings. Our scrollytellings consist of sequences and layers. Sequences are compositions of individual elements, *morphs*, as described in subsection 3.1.1, that belong together. The sequences are arranged in layers to visually stack them on top of each other.

This common structure allowed us to create the editor shown in Figure 1.4. We developed qinoq in the lively.next[12] environment, since our project partner, was using it before. The environment lively.next is graphics-based while at the same time allowing developers to change the system itself and all of its elements. We will have a closer look at the advantages of lively.next in section 3.1.

With the help of our editor, users can create scrollytellings and scroll-based animations without the need to write code using a UI.

This results in the new workflow shown in Figure 1.5. We aimed to make the interaction of the content designers and developers shorter. Thus, content designers still need to work on the story with their clients, as denoted by (1). Afterward, the editor enables them to build those ideas into a first scrollytelling (2). They can try what works and feels suitable, especially for the essential scroll-based animations (see subsection 1.3.2). As an optional step (3), developers can create missing elements like time-based animations or elements with which consumers can interact in another way than scrolling.

Thus, resulting in a new workflow where content designers and developers work in the same environment and do not need to explain abstract content. Additionally, content designers can try out different possible designs and animations without explaining them beforehand.

The evaluation of whether the editor enables sufficient possibilities to content designers, while at the same time making sure that developers can add elements with code can be found in chapter 5.

## 1.6 Summary

Interactive media and especially interactive articles on the web are getting more and more common. Scrollytellings are an important part of this trend.

We have seen that it is challenging to create interactive media. Authors of interactive media need to consider different aspects like how to make consumers aware of the interaction possibilities. Also, they often need to work in a multi-disciplinary team to create interactive media. Since the creation of many of those media requires the knowledge of programming and needs to be developed quickly

[12] https://github.com/LivelyKernel/lively.next (last accessed on 2021-07-28).

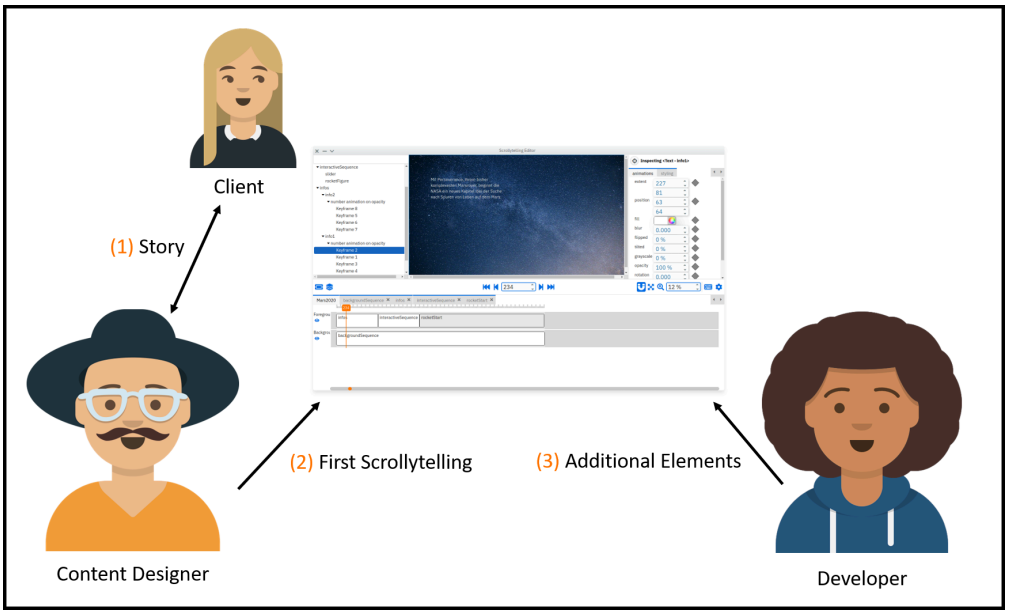


**Figure 1.5:** New workflow of our project partner Typeshift

due to the fast-changing nature of web content, it stands as a challenge to create good interactive media. However, consisting of different modalities makes the creation of interactive media more complex and therefore additionally time-consuming and expensive.

We focused on scrollytellings due to their relevance for our project partner. Scrollytellings are digital stories combined with, for example, scroll-based animations and videos, which create an interactive experience. Being interactive media, they suffer from the same challenges interactive media consisting of different modalities does.

To solve the problem of many iteration loops for our project partner and enable visual expression, we built an editor that allows users to think more visually about their scrollytelling. As we will see in the following chapters, the editor enables authors to build scrollytellings without writing code.

However, our editor does not solve all aspects of the problem. Especially game-like simulations or time-based animations still need to be added by developers. While the editor enables content designers to create first scrollytellings, the environment lively.next makes it possible for developers to add custom behavior on any object. Thus, with our editor, we give content designers and developers the chance to create customized scrollytellings in a single environment reducing transfer processes of the vision of scrollytellings.

# 2 Design Constraints and Requirements for Scrollytelling Creation Tools

The main focus of qinoq is developing a tool that supports the authoring of scrollytellings by collaborating content designers and developers. We already discussed the differences and similarities of scrollytellings and other forms of interactive media like books, movies, and computer games. This chapter looks at existing interactive content creation applications to see what we can learn from them. Which traits are suitable for our scrollytelling editor? What needs to be different?

We will approach these questions the following way: At first, we will understand the environment in which our editor will be used. Therefore, we need to develop the different roles involved in the creation process of a scrollytelling, which we already introduced in section 1.4. Here, we will describe their abilities, needs, and interactions. This is necessary to understand the basic possibilities our editor needs to provide and the constraints that it needs to meet. After looking at the editor's environment, we define the feature space. This is a collection of high-level usage scenarios occurring when creating interactive content with software. Following, we use these features to analyze a selection of existing interactive content creation applications. These applications are used to create interactive media other than scrollytellings, such as animations, websites, or movies. Due to the analysis, we may find possibly suitable applications for the scrollytelling creation. Anyhow, we can derive best practices applicable to our scrollytelling editor and identify issues that have yet to be solved.

Eventually, we will discuss the fundamental characteristics of our editor, thereby drafting an editor concept.

## 2.1 Editor Environment

In this section, we describe the environment in which our editor will be used. The environment comprises the domain and the users. Understanding the environment enables us to formulate the requirements and constraints of our editor. Based on these, we can later formulate the feature space for the analysis and determine suitable implementations of features to draft an editor concept.

### 2.1.1 Involved Roles

As in many projects, the creation of a scrollytelling involves several people with different roles. Each role has its own set of abilities and needs that the editor must support if they interact with it.

**Consumer** A person visiting a published scrollytelling website is a consumer. Consumers usually interact with scrollytellings through web browsers on, for example, desktop computers or mobile devices. They may seek entertainment or information.

**Client** Clients commission scrollytellings, as conventional (interactive) websites do not meet their needs. There might be a multitude of reasons, for example, because they need to communicate a conceptually, spatially, or temporally complex topic or want to stand out with their web page for advertising purposes. They do not communicate with the team behind the creation of a scrollytelling directly. Instead, they are in contact with the content designers.

Clients may differ in the clearness of their visions for the scrollytelling: from rough ideas on how to achieve their goals up to specific requirements that should be met. Usually, clients do not want to be involved in every little decision of the process. Once a clear vision was developed, reviews after significant changes are sufficient. During these reviews, they want to experience the scrollytelling as a consumer would. They may have changing requirements or slight changes in their vision. Clients may not have programming or design skills.

**Content Designer** Content designers play a central role in the creation process. They communicate with the client, develop the scrollytellings' story, draft the scrollytelling in terms of behavior and content, and transfer these ideas to the developers and graphic designers.

Content designers may not have programming skills, nor need to have the skills to create graphics, fonts, or complex animations. They are the creative minds behind a scrollytelling. Together with the client, they develop a vision and enable the other roles to submit work that conforms to that vision.

**Developer** Developers are responsible for the technical realization of the scrollytelling. That is: writing the code necessary to realize the content designers' vision, delivering the scrollytelling, and administrating the underlying infrastructure.

Developers might not have the skills to develop scrollytelling visions or design content. They communicate closely with the content designers.

**Graphic Designers** Graphic designers provide the graphics and animations by the content designers' vision. They may use their professional tools to realize them. They communicate with the content designers.

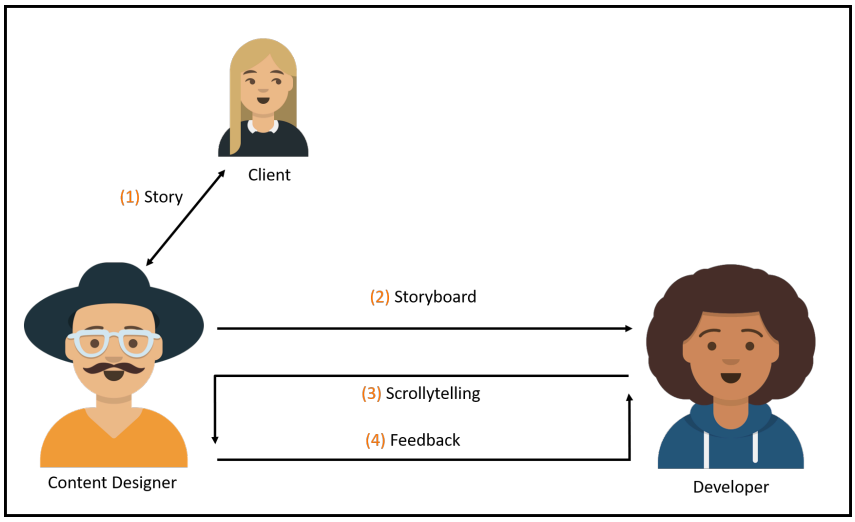


**Figure 2.1:** Scrollytelling creation process (simplified)

At our project partner Typeshift, an agency for digital content, graphic designers are external contributors. Often, they are asked to create Lottie animations. See subsection 4.3.5 for details on this animation type.

### 2.1.2 Creation Process of a Scrollytelling

In this section, we describe the lifecycle of a scrollytelling. By outlining this process, we can later identify significant issues and specify our editor's area of impact, from which we may derive important requirements and constraints for our editor.

Figure 2.1 depicts the scrollytelling creation workflow on a high level. We describe the process in more detail in the following.

**Commission and Vision Creation** The client contacts the team, and the content designers get in touch with the client. Together, they develop a vision for the scrollytelling. This is an iterative process, which results in a *storyboard* that the client signs off. The storyboard is a table describing key moments – we refer to them as *scenes* – of the scrollytelling in chronological order. Each scene description consists of the respective story, the visuals, and the interactions. The storyboard forms the basis for the vision transfer between the content designers, the developers, and the graphic designers.

**Scrollytelling Creation** Once the client has signed off the storyboard, the creation of the scrollytelling begins. The content designers commission the graphic designers with the artwork for the scrollytelling. This includes drafts of the individual scenes, all independent graphics needed to recreate the draft, and all animations. During the draft development, the content designers mediate between graphic designers and the client. The client signs off the drafts, too.

Once signed off, the drafts replace the visuals' descriptions in the storyboard. The content designers pass all artwork alongside the storyboard to the developers, who then create a first interactive version of the scrollytelling in code. This version is uploaded on a company-internal web server so the content designers can verify the result.

In the vast majority of cases, the scrollytelling needs adjustments. May it be due to simple typos, elements that are not optimally positioned, animations whose starts or durations need to change, or parts that do not work in reality as envisioned. The content designers need to write down their feedback, for example via e-mail. They have to describe where the developers need to make which changes. The developers then incorporate the feedback in the code base and update the internally available scrollytelling version. This process typically repeats multiple times and takes up much time for the scrollytelling creation. Many necessary adjustments are subjective and difficult to describe precisely. It is not uncommon to iterate over the same issue more than once.

In the unlikely event that extensive decisions need to be made during the iterations, the content designers may speak back to the client. Apart from these cases, the iterations occur between content designers and developers only.

**Delivery** Once the content designers are satisfied with the scrollytelling, the client receives access to the still internal scrollytelling to experience it first-hand. The content designers transfer the client's feedback to the developers who address it in code. They repeat this process until the client is satisfied. Usually, it does not require a significant number of repetitions.

The final scrollytelling is then either published as a stand-alone website or integrated into existing web projects. Consumers can now interact with the scrollytelling. They may use different mobile and desktop devices. Scrollytellings do not require regular maintenance. If in need, clients can contact the content designers, which forward the request to the developers.

### 2.1.3 Evaluation

The expensive part of the scrollytelling creation process is the repeated iteration between content designers and developers. Content designers do not have the tools to fine-tune scrollytellings to their visions on their own. Using means external to the domain, like e-mails, they have to explain to developers how to adjust details elaborately. If content designers had the tools to reiterate these details independently, content designers and developers could focus on what they can do best. The work would be better divided and the iterations more effective, resulting in faster production of the scrollytelling.

| Category | Feature | Description |
|---|---|---|
| Project Management | Administration and Overview | Means to have an overview of and administrate projects |
| | Opening Projects | Implementation of opening one or multiple projects |
| | Saving Projects | Means to save a project |
| Navigation | Layout and Editor Components | Basic layout and key components of the application |
| | Zoom and Movement | Means to zoom and move within certain components |
| | Capsules Navigation[13] | Means to enter and leave capsules which contain project elements[13] |
| | Programming Interface | Integration of scripting |
| Composition | Content Creation | Means to create new elements in the project |
| | Position (2D) | Means to position elements |
| | Size | ... set the size of elements |
| | Rotation | ... rotate elements |
| | Order (level) | ... determine which elements overlap others |
| | Grouping | ... treat multiple elements as one element |
| | Reusability | ... use multiple instances of one element in a project |
| Configuration | Numbers | Means to set numeric element properties |
| | Discrete Values | ... set discrete element properties |
| | Commands | ... execute actions on elements, like *mirror horizontally* |
| | Colors | ... set color properties |
| Animation | Numbers | Means to animate numeric element properties |
| | Discrete Values | ... discrete element properties |
| | Colors | ... colors |
| | Motion | ... animations of elements' positions |
| | Easing | Means to adjust the easing of animations |
| | Transitions | Means to animate transitions between capsules[13] |
| Production | Preview and Testing | Means to preview project during editing |
| | Final Export | Means to export the final product out of the editor |

**Table 2.1:** Common features of interactive content creation applications. These are investigated in the software analysis, see section 2.4.

[13]Capsule: self-contained abstraction of elements, see subsection 2.4.1.

## 2.2 Feature Space

Features in the context of our analysis are high-level usage scenarios of interactive content creation applications. The entirety of features we want to analyze concerning these applications is our *feature space*.

Table 2.1 shows typical features for interactive content creation applications. Our editor may need to implement these as well. Features may be implemented differently across the various applications. We want to collect these implementations in our analysis to find the most suitable implementation for our scrollytelling editor.

## 2.3 Software Selection

As diverse as the kinds of interactive media, so broad the variety of applications which exist to author them. Our analysis focuses on four applications: HyperCard, Macromedia Flash, Microsoft PowerPoint, and DaVinci Resolve. However, many other applications might be suitable or might have inspiring implementations as well. In the following, we outline prerequisites for our selection and give a short introduction to the selected applications.

### 2.3.1 Prerequisites

Our analysis is limited to software available free of charge for the time of the analysis: software with trial versions, open-source software, and software officially available for free. We target a balance between relevance for the scrollytelling creation and diversity of applications' target media. A mix of historical and actively developed software is desired to identify potential unsuitable or very suitable but forgotten feature implementations. Lastly, the applications should have or should have had a significant distribution in their respective fields. Our goal is the identification of appropriate feature implementations. The applicability of implementations of little-known applications would demand additional verification. A shortlist of applications that fulfill the prerequisites reasonably can be found in section A.1.

### 2.3.2 HyperCard

*HyperCard* was originally released by Apple in 1987 for the *Mac OS 9* [39][34]. It is used to create and edit *stacks* of *cards* [23]. A card represents an index card and is by default a white plane that can be painted and populated with objects like buttons, lists, and text fields. A card always belongs to one stack, which is the window in which the card lives. A stack usually consists of multiple cards. Stacks can be thought of as file folders containing index cards. Only one card is visible at a time. Users can navigate between cards in different ways. For example, they can click the navigation buttons, use the arrow keys, and interact with objects linking to other

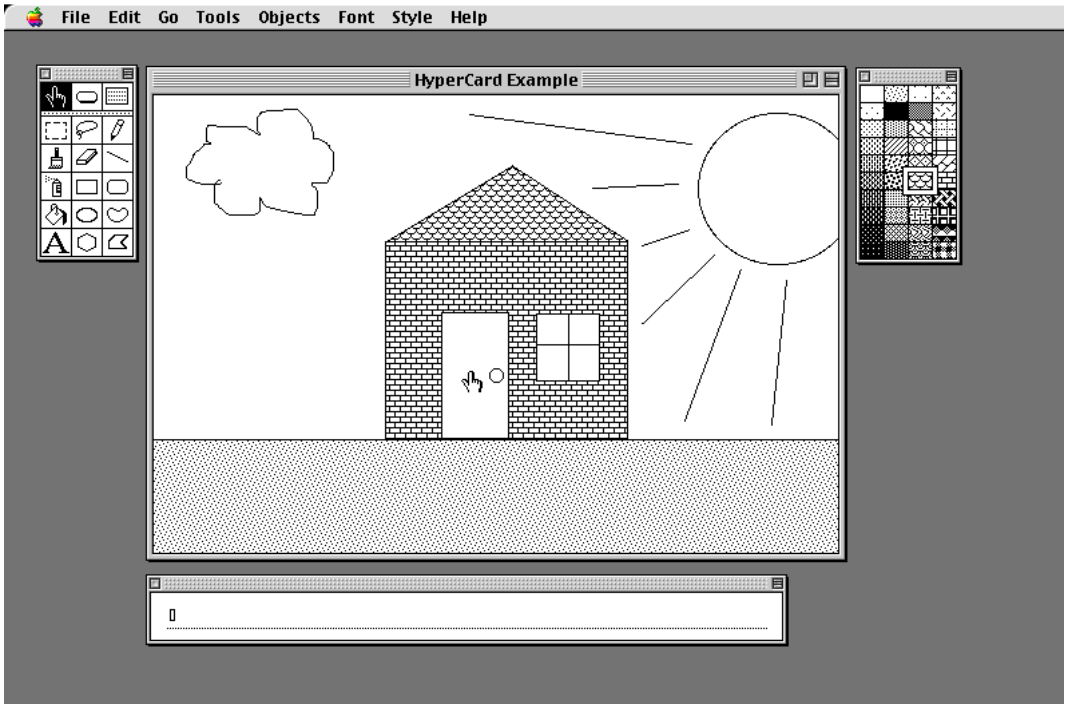


**Figure 2.2:** User interface of HyperCard 2.4.1

cards. Moreover, a card can be introduced with a transition. Objects can likewise link to other stacks, which always open in a new window.

Elements can not only function as hyperlinks to other cards. Scripts in the programming language *HyperTalk* can be attached to buttons, fields, cards, and stacks. That way, HyperCard can be used to create interactive documents with text, graphics, and possibly interactive animations, presentation slides, or simple applications. *HyperCard Help* and *HyperTalk Reference*, which introduce the application features, are HyperCard stacks themselves. HyperCard only supports black and white in its last official release, HyperCard 2.4.1 from 1989, which we will analyze.

### 2.3.3 Macromedia Flash

Macromedia Flash was a popular software for the professional authoring of interactive content.[14] It is available as a free and a professional version. The application mainly focuses on creating flash documents that can be played using the Macromedia Flash Player, later the Adobe Flash Player [42]. Flash documents can be animations, widgets, or applications. They support graphics, sound, video, or certain special effects like particle systems. Additionally, the professional version of Macromedia Flash supports the creation of slide-based presentations.

[14] https://www.adobe.com/support/documentation/en/flash/fl8/releasenotes.html (last accessed on 2021-07-28).

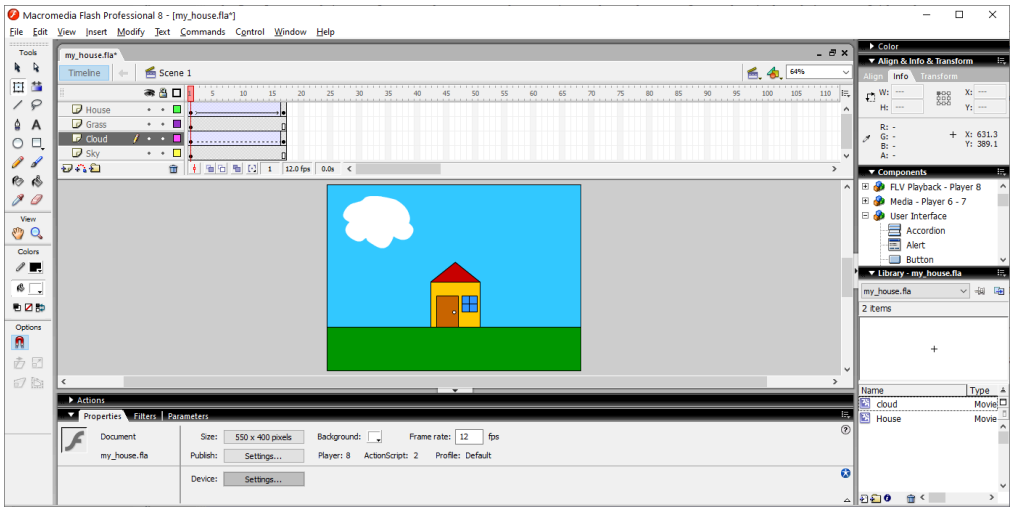


**Figure 2.3:** User interface of Macromedia Flash Professional 8

In Macromedia Flash, flash documents are called *applications*. Macromedia Flash uses a central area to arrange content spatially. This area is called *stage*. Content can be arranged on a timeline both temporally and spatially. Every application has a *frame rate*, determining how many frames are played within a second. At each point in time, only one frame is visible and displayed on the stage. To extend the possibilities of the large variety of settings and interaction and animation types, *ActionScript* can be attached to various objects like frames and their content.

Flash reached its end of lifetime in 2020.[15] Thus, the official support of all Flash-related products, including Macromedia Flash, has ended. In our analysis, we investigate the 30 days trial version of Macromedia Flash Professional 8 from 2005, the last release before Macromedia Flash was replaced with Adobe Flash CS3.[16][17]

### 2.3.4 Microsoft PowerPoint

Microsoft PowerPoint is used to create slide-based presentations. An animation consists of one or multiple sides [46]. We refer to the entirety of a presentation's slides as *slide deck*. These work almost identical to HyperCard. Microsoft PowerPoint focuses on being intuitive and easy to learn. Like Macromedia Flash, it features an area to arrange content that we will refer to as *stage*. Most of the controls are situated in the *ribbon menu*,[18] which is accompanied by a *sidebar* for further options.

[15] https://www.adobe.com/products/flashplayer/end-of-life-alternative.html (last accessed on 2021-07-28).

[16] https://community.adobe.com/t5/flash-player/macromedia-flash-8/m-p/10213404 (last accessed on 2021-07-28).

[17] https://helpx.adobe.com/animate/animate-releasenotes.html (last accessed on 2021-07-28).

[18] Details: https://web.archive.org/web/20080104234859/http://office.microsoft.com/en-us/products/HA101679411033.aspx (last accessed on 2021-07-28).

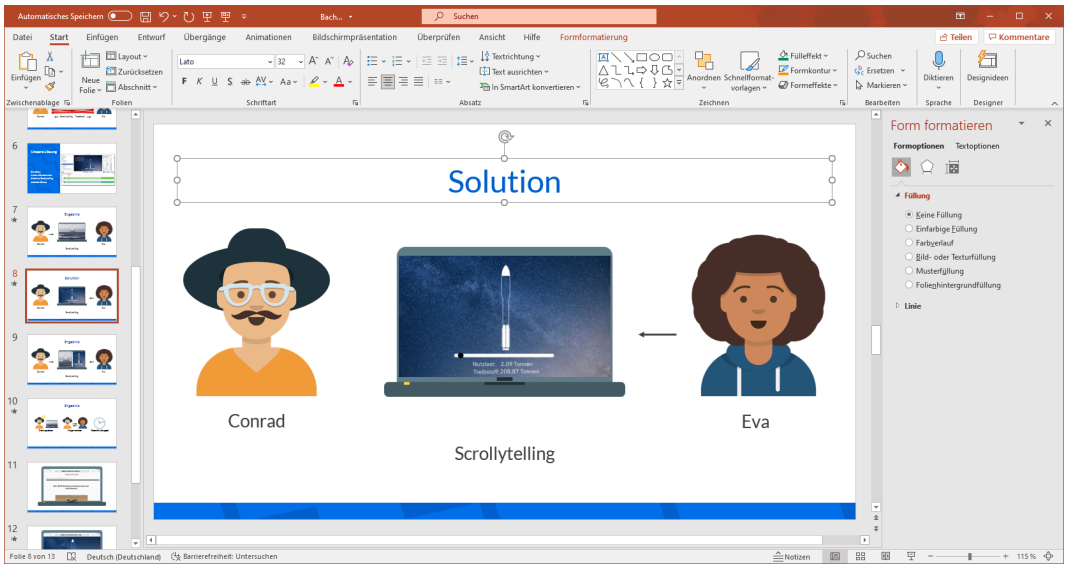


**Figure 2.4:** User interface of Microsoft PowerPoint

While it is easy to get started with Microsoft PowerPoint, it offers many options to customize presentations. It supports various content types, like shapes, images, videos, sound, or diagrams. Content can be animated, slides can transition in different styles, and external content can be referenced. Multiple users can work on the same project at the same time by sharing it online. Microsoft PowerPoint supports scripting via VBA script to influence the application's behavior or alter content programmatically.

Microsoft PowerPoint is in active development. We analyze Microsoft PowerPoint for Microsoft 365 in version 16. Due to an educational license, it is available without additional cost to students of the University of Potsdam. Microsoft PowerPoint Online offers similar features and is available for free after logging in with a Microsoft account.[19]

### 2.3.5 DaVinci Resolve

DaVinci Resolve is an application for video editing, compositing, color correction, audio production, and finishing [51]. It is developed by Blackmagic and aimed at professionals.

The user interface divides into seven pages. Each page supports a different task or step in the production process. They all accommodate a set of panels with tools according to the page's purpose. Some panels are common on most pages: The inspector, the viewer, the media pool, metadata, audiometer, and the timeline. We may have a closer look at individual panels later. As an overview: Similarly to Macromedia Flash's timeline, DaVinci Resolve's timeline arranges content spatially

[19] https://powerpoint.office.com (last accessed on 2021-07-28).

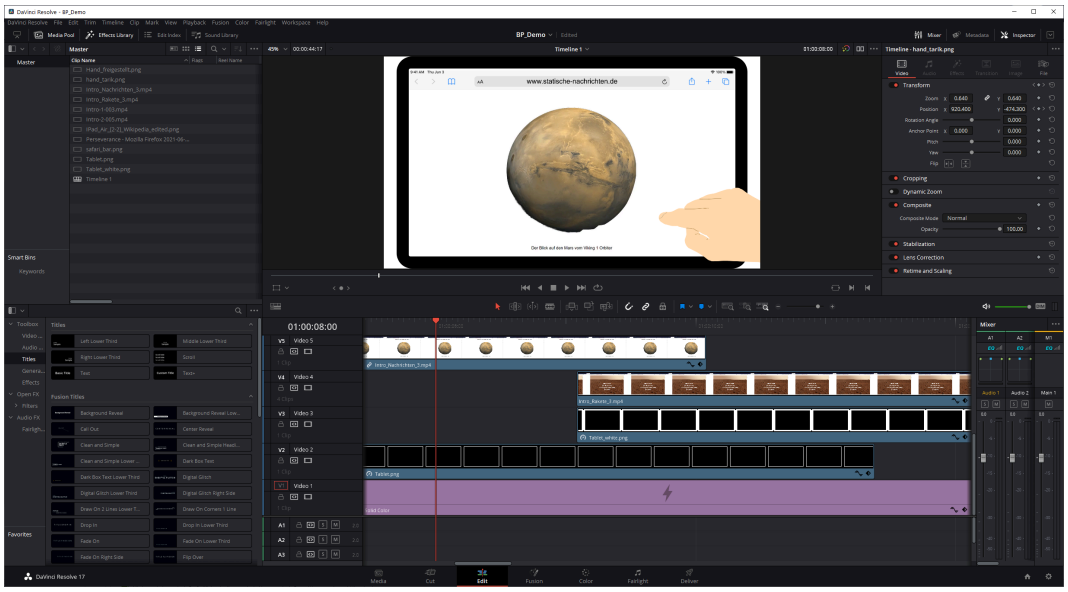

**Figure 2.5:** User interface of DaVinci Resolve 17

in a level-wise order and temporally. The *viewer* displays the content of the currently selected frame. The *fusion* page introduces compositing and 2D or 3D animation features, similar to Blender and Godot. DaVinci Resolve supports scripting to automate repetitive or complex tasks, customize application behavior, extent fusion's functionality, and exchange data with external programs.

DaVinci Resolve is in active development. We analyze the latest release, DaVinci Resolve 17, in its free version. The paid version, DaVinci Resolve Studio, offers more filters and effects and some additional tools and plugins.[20]

## 2.4 Software Analysis

The comparison in subsection 1.3.4 shows that scrollytellings share many features with other interactive media. Hence, it might be valuable to analyze software designed for authoring other interactive media to our editor's feature space.[21] By doing so, we might find an appropriate application for authoring scrollytellings or learn what needs to be different in a self-developed editor.

Our analysis considers the applications HyperCard, Macromedia Flash, Microsoft PowerPoint, and DaVinci Resolve. They have been analyzed based on the support material provided by the applications and hands-on tests [23][42][46][51]. This

[20] https://www.blackmagicdesign.com/products/davinciresolve/studio (last accessed on 2021-07-28).

[21] See section 2.2.

section presents only the key findings of the analysis. A detailed summary can be found in section A.2.

### 2.4.1 Terminology

**Project** Of course, all applications are used to create and edit different entities of workpieces. HyperCard, for example, works on stacks, Macromedia Flash on applications. To simplify, we call the entities opened and edited with the applications *projects*.

**Stage** All applications feature an area where they allow previewing and arranging the elements seen in the later product. We refer to this area as *stage*.

**Capsule** The applications also provide means to abstract multiple elements into some kind of black box. We call those *capsules*. Capsules provide a context for the contained elements. This context may be of a different kind than the context in which the capsules live. In Microsoft PowerPoint, a slide is a capsule. In Macromedia Flash, symbols work as capsules.

### 2.4.2 HyperCard

HyperCard uses cards to capsule and organize content. The users can draw directly onto the cards and place objects like buttons and fields on them. A card has a fore- and a background. The foreground is unique to every card; backgrounds can be used for multiple cards. A menu bar button toggles between fore- and background editing mode. There are several tools available to create and alter the content. New objects like buttons and fields are created via the menu. From then on, they have to be selected and edited using their specialized tools. The other available tools are for drawing. When drawing, the existing drawing is directly mutated. Objects, on the other hand, can be repositioned and resized later on. Their properties can be changed via *info boxes* (see Figure 2.6), which open after a double-click on the object or via the menu. Any change of a card or stack is directly saved in the original file. It is only possible to undo the very last chance. Thus, the support material encourages users to often create safety copies of the stacks at appropriate moments. All this makes changes expensive and iterations time-consuming.

Elements cannot be animated directly. Instead, multiple cards with minor changes can be shown in rapid succession to give the impression of an animation, similar to stop motion films.

Several components of HyperCard are implemented as different independent windows, for example, the stage, the toolbox, or the message box, which can execute textual commands.

HyperCard does not provide an overview of cards and their order. Thus, it relies heavily on the author's knowledge of the project and complicates the navigation within the project. This makes the collaboration of multiple people difficult.

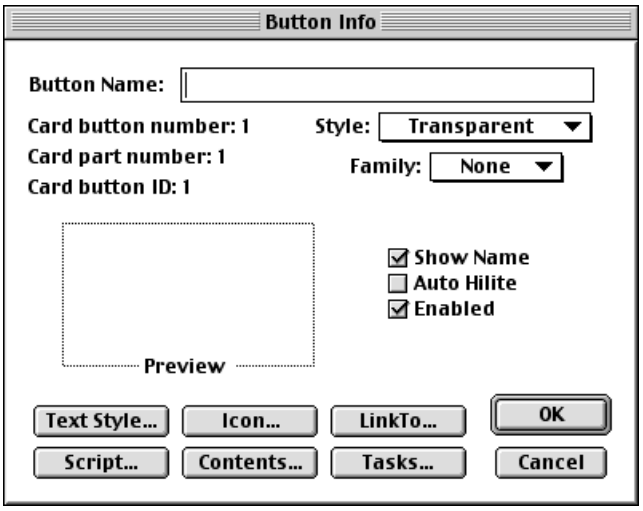


**Figure 2.6:** Button info box: Info boxes like this one are used to adjust object properties in HyperCard.

### 2.4.3 Macromedia Flash

Macromedia Flash uses a timeline to control spatially and temporally, as depicted in Figure 2.7. The timeline is horizontally divided into frames and vertically into layers. The contents of the upper layers overlay the contents of the lower layers. The stage only displays one frame's content at a time.

Elements usually live for multiple frames. They can be created via respective tools or imported into the application. Macromedia Flash offers many options to customize elements. The options are accessed via specialized tools and various panels, each designed for other tasks. These panels can be displayed and arranged flexibly. Some panels and tools have overlapping functionality. If a panel has options that do not apply to the current selection, the respective controls are disabled. Some options on elements are already accessible on selection. However, these options are inconsistent across different element types. Panels and inconsistent behavior require the users to guess or know where to find options.

Elements do not directly mutate the frame. Elements can be altered, for example, rearranged, at any time without influencing other elements. It is possible to undo and redo multiple changes. This all facilitates later changes and iterations.

While the stage displays the content of the currently selected frame, a project, like an animation, can only be previewed in the actual time by opening a secondary window that plays through the timeline. Alternatively, it is possible to drag the timeline's playhead, which controls the currently selected frame, through the timeline. This changes the stage's content accordingly. The drag speed determines the speed of the playback. This makes it cumbersome to fine-tune scroll-based animations, as the users would need to switch between timeline and preview window all the time.

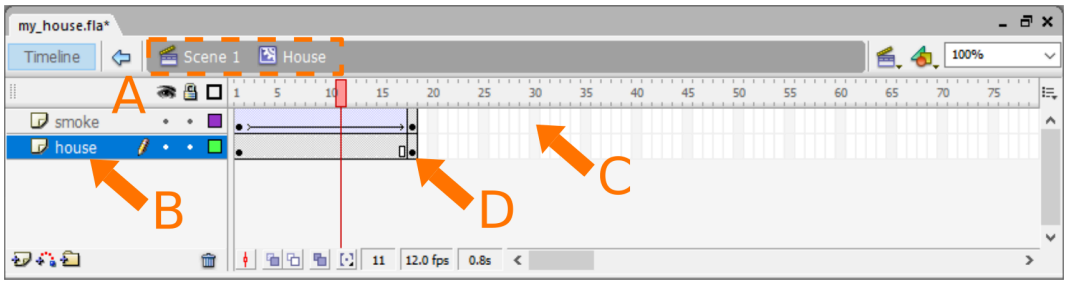


**Figure 2.7:** Timeline of a symbol in Macromedia Flash: The timeline is divided into frames like (A) and layers like (B). The breadcrumb menu (C) allows to navigate between the symbol and the global timeline (Scene 1). Keyframes like (D) are used to control animations.

Content can be capsuled in *symbols*. These provide an internal timeline that works like the global timeline described above. They can be entered with a double-click. Their timeline opens where the global timeline was previously. The navigation between symbol and global timelines uses the breadcrumb method [35].

To animate elements, frames can be converted to keyframes. The keyframes now work like snapshots of the contained elements and their properties. For the frames between two keyframes, the software interpolates the element property values so that all property values of the first keyframe gradually change to the values of the second keyframe. The easing of animations can be customized using Bézier curves. By default, an animation either affects all properties concerning motion or all properties concerning the element's shape. It is also possible to limit an animation to one property.

Scripts can be attached to any element. This works via a specialized panel. It features AssistScript, which should help people without significant coding skills to write their code.

### 2.4.4 Microsoft PowerPoint

The basic concept of Microsoft PowerPoint is very similar to HyperCard. Instead of cards, it uses distinct slides to capsule content. That being said, it improves usability compared to HyperCard and Macromedia Flash. It simplifies the user interface and does not need special tools except for creating elements like shapes or text fields. It combines the transform and selection tool of Macromedia Flash while also making the selection behavior consistent among element types.

Generally, customizations are accessible in two ways: either via the sidebar where all properties are listed or via a *formatting tab* of the ribbon menu. These two components in combination are used instead of Macromedia Flash's panel system. The most used options for each task are accessible in the ribbon menu, which has different tabs for different tasks. Further options can be reached by opening the sidebar out of the respective ribbon menu tab. This makes the options more discoverable. It is also possible to open the sidebar via the context menu

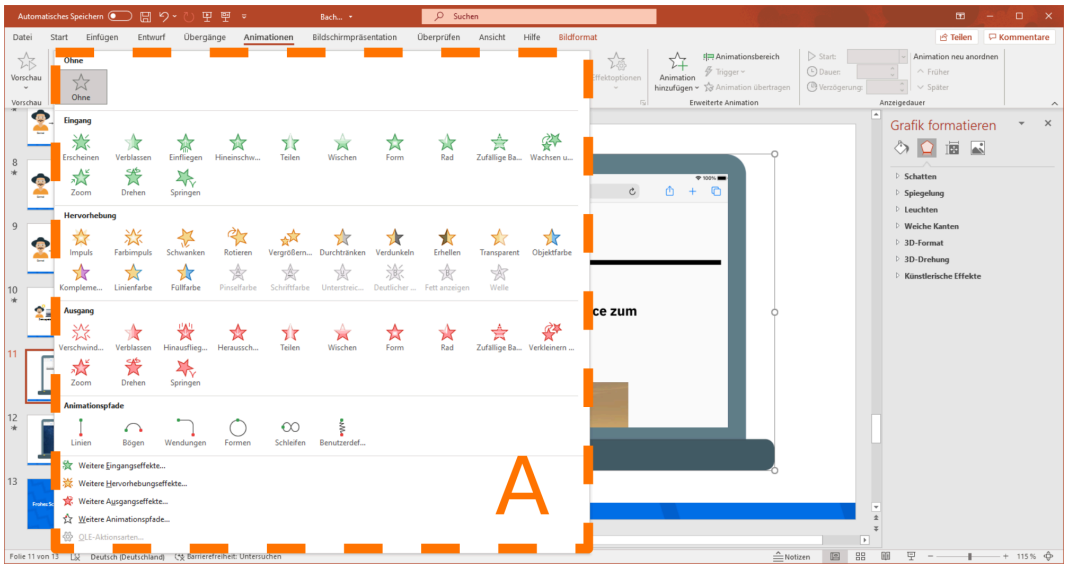


**Figure 2.8:** Available animation types (A) in Microsoft PowerPoint: They can be attached to the selected object and customized via type-specific parameters.

of the element in question. While the sidebar has similarities to panels, it works differently. Instead of fix panel controls, it adapts the controls to the current selection, overcoming not applicable and thus disabled option controls.

Instead of using keyframes, elements can be animated by selecting an animation from a multitude of presets and attaching this to an element, as depicted in Figure 2.8. The animation can be customized via options depending on the animation type. This method is not well suited for complex and precise animations. It lacks customizability and a consistent interface to edit details of multiple animations at once.

Well supported is the collaboration of multiple people. It is possible to open a project on different computers at once and synchronize the changes live. A second sidebar offers a good overview of the whole slide deck, which makes navigation easy. It is likewise easy to create multiple versions and iterate since slides can be copied and rearranged easily, and elements are always mutable. Applied effects do not alter the original.

While Microsoft PowerPoint's scripting feature is not easy to access nor well integrated into the application, it offers an environment suited for scripting by opening it in a dedicated editor.

### 2.4.5 DaVinci Resolve

DaVinci has parallels to Macromedia Flash. For example, it uses a timeline to arrange content spatially and temporally. Animations are likewise created using keyframes, although a keyframe in DaVinci Resolve can solely apply to one property.

Despite the similarities, DaVinci Resolve streamlines the user interface through several means. Firstly, it provides an inspector, as seen in Figure 2.9. This panel

offers all properties which could be adjusted on the selected element. The properties are grouped by task (A) and have a consistent interface. For example, all numeric properties have a slider and a text field for the precise input of a number (B). The control also supports dragging the field to increase or decrease the value. Next to such a property value control, the button (C) to create, overwrite, and delete a keyframe is integrated into the inspector. Eventually, it also features arrows to jump to the next or previous keyframe of this property (D). Secondly, the application uses a consistent combination of keys and mouse scrolling to zoom in/out and move the stage and timeline, making it easy to navigate the content without having to use scrollbars. The *cut page* provides a smaller overview timeline, offering a good overview, especially in large projects, and allows for quick navigation. For quick alignment, content and capsules like *compound clips* can snap to each other when being moved or resized. This facilitates timing. Thanks to a tab system, it is simple to navigate between multiple opened capsules and the global timeline.

A *media pool* gives quick access to and administrates all media used within the project.

As seen in Figure 2.10, the easing of animations can be highly customized using Bézier curves in value-time diagrams (A). Yet, the diagram's scale is relatively dense, and it can only show and edit one property at a time. This is not a simple and precise means to adjust easings. The *fusion page* offers a keyframe panel with color-

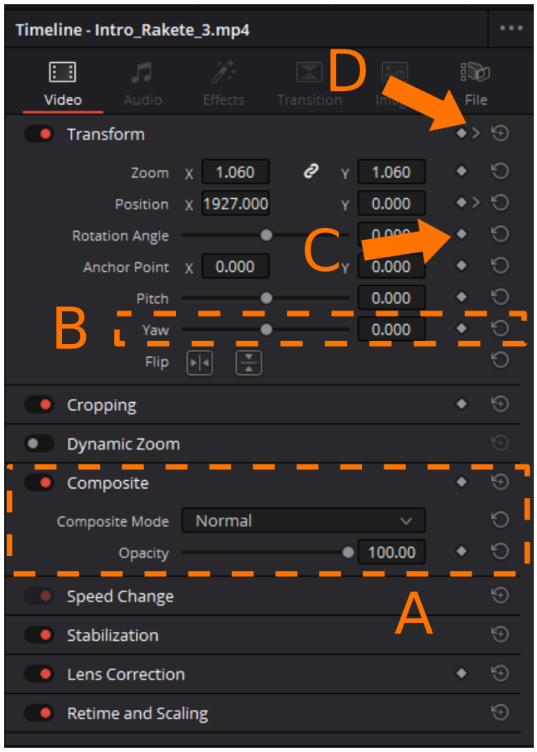


**Figure 2.9:** Inspector in DaVinci Resolve with element group (A), property control (B), keyframe button (C), and keyframe jumper (D)

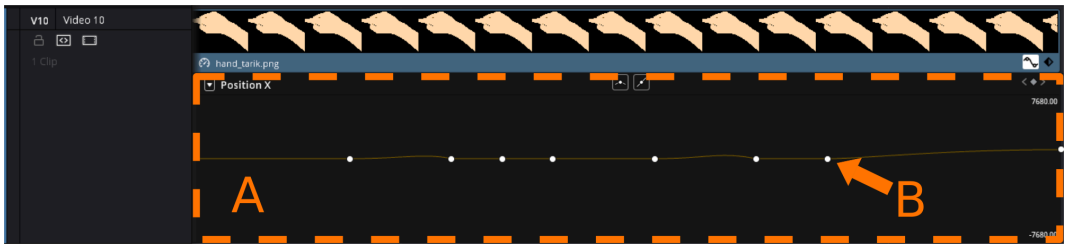


**Figure 2.10:** DaVinci Resolve integrates value-time diagrams like (A) into the timeline to edit keyframes like (B) and animation easings using Bézier curves.

coding, which improves clarity. Elements with animated positions can display their motion path on stage. This path is also directly editable, which facilitates this kind of animation a lot.

Projects are stored in an internal database. This introduces extra steps if projects should be shared for collaboration, except if an according syncing infrastructure exists.

## 2.5 Editor Concept

Our analysis gathered various possible implementations for the feature space outlined in section 2.2. We now want to discuss which implementations are most appropriate to author scrollytellings. These fundamental design decisions enable us to develop our editor which fits the requirements properly. Additionally to the implementations of our analyzed applications, we may incorporate learnings and ideas from applications of the shortlist in section A.1.

The editor is situated in the live-programming environment *lively.next*. This is a requirement of our project partner Typeshift, an agency for digital content which also creates scrollytellings, as they already work with this system.

### 2.5.1 Design Goals

As a preparation for the feature implementation discussion, we want to summarize the design goals for our editor.

The editor is expected to streamline the collaboration between content designers and developers. Content designers should be enabled to compose the artwork contributed by the graphic designers into a scrollytelling on their own. It should also be possible for content designers to animate the elements precisely, at least scroll-based animations. All this should be possible without writing code, as content designers cannot be expected to have that skill.

The editor should support the authoring of highly individualized scrollytellings. Hence, the editor should impose as few restrictions on scrollytellings as possible, compared to the scrollytelling creation by developers only. If a feature is not supported in the editor, it should be possible to implement it directly on the

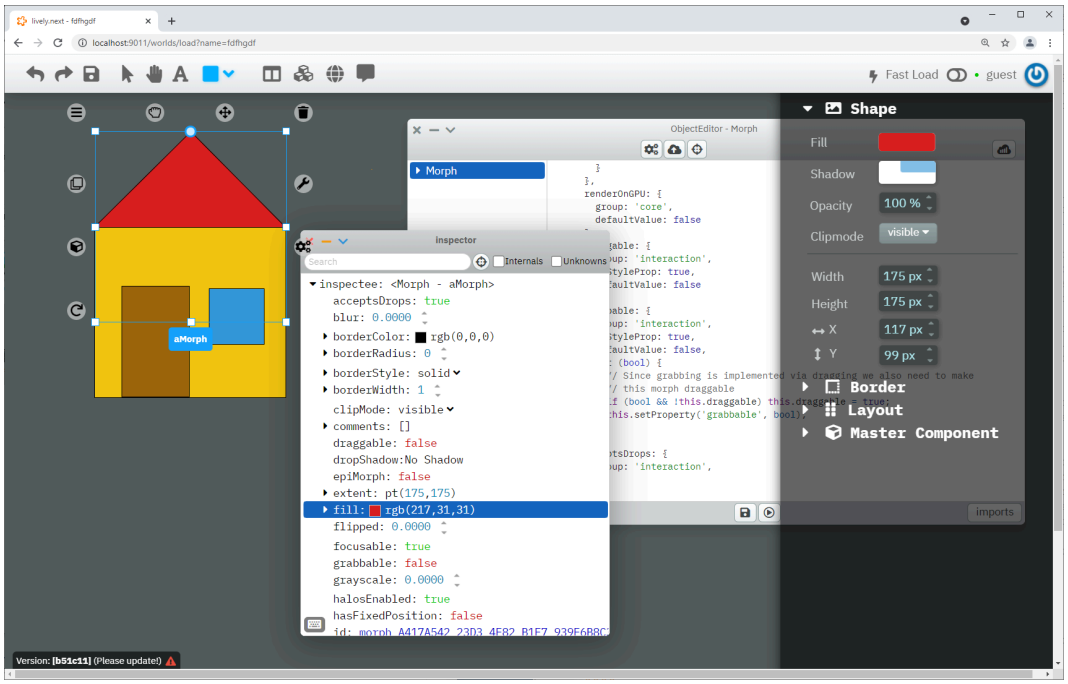


**Figure 2.11:** Content and various tools in an opened lively.next world

scrollytelling without causing the editor to malfunction. However, it is not expected that changes in the scrollytelling's code base occur while the scrollytelling is opened within the editor.

Content designers and developers are expected to hand over the scrollytellings multiple times. The editor should make it easy for the users to familiarize themselves with the scrollytelling and quickly navigate to points of interest.

While the editor should be situated in lively.next and integrate well into the system, it should not be an integral part. As lively.next is an open-source project for various purposes, this specialized scrollytelling editor should only be an optional module for it.

Eventually, our editor does not need to be intuitive for beginners without a prior introduction. It is a tool for professionals. We still aim for a moderate learning curve and high discoverability but do not sacrifice features and efficiency for usability.

### 2.5.2 lively.next

Later design decisions and explanations may be based on traits of lively.next. Thus, we give a short introduction to this system. An in-depth examination of lively.next will be conducted in section 3.1.

The lively.next system is usually used through a web browser and is developed in JavaScript. In this system, every visual element is a *morph*. That means that user-created content and system components like windows base on the same class and share a common feature set and properties. Additionally, all code in the system can

be altered. The lively.next system is designed to provide a single environment in which designers and developers can work on projects collaboratively. Hence, it offers a graphical user interface with features for designers, such as a *top bar* providing tools for creating objects or a *style palette* to format objects. Nevertheless, it also provides many features for developers, first of all, the live programming environment to alter any code, but also tools like a workspace for executing code, a code browser to view and program code, or an inspector to investigate an object's properties.

The basis for every lively.next project is the *world*. It contains the system components, like the top bar and windows, and the actual content created within the system. There are different means to exchange worlds between users. It is also possible to run a shared lively.next instance on a server to have access to the same worlds. It is currently impossible to work simultaneously in one world without overwriting changes made in the other world instance.

### 2.5.3 Design Decisions

Developing the editor requires us to make several design decisions, which we discuss in this section.

**Editor Window** The most basic decision is whether the editor should mainly consist of one window, like Macromedia Flash, Microsoft PowerPoint, and DaVinci Resolve, or multiple freely arrangeable windows like HyperCard. We looked at how the developers of our project partner Typeshift work. They tend to use a lot of different windows to manage the code base and work with the content. Adding multiple windows for our editor decreases the available screen space for their other windows. Also, switching between our editor and other lively.next tools would be cumbersome since each window has to be opened individually. Generally, a single-window application takes the task of organizing windows off the users to the editor's developers. Moreover, it adds to the usability as users can discover and access all features from within one window.

That is why the editor consists of one window and some modals.

**Panel System** A panel system divides input controls into different semantic groups and allows the users to display, hide, and arrange the individual panels freely. On the one hand, users can thereby customize the editor to increase their efficiency. On the other hand, they need to put effort into organizing their workplace. They are also required to guess which hidden panel now contains the desired function. Panel systems may be useful for applications with a wide range of supported content types, as the multitude of features can be split up into logical groups, only showing the currently necessary ones. Our editor is specialized in scrollytellings and does not have as many features as, for example, Macromedia Flash. A panel system would increase the complexity of the user interface without a significant benefit.

That is why our editor uses a fixed structure optimized for the scrollytelling creation by content designers and developers instead.

**Scrollytelling as a Morph** Every analyzed application has an area which we called *stage*. There, the content can be arranged and previewed. We likewise consider this an important feature and incorporate such a stage into our editor because it allows for an intuitive arrangement of scrollytelling elements and gives immediate visual feedback to the users.

In most of the analyzed applications, the content shown on the stage is a visual representation of the data object that describes the project. However, the morphs in lively.next are an integrated whole of state and behavior and are directly visualized in the world. We have to decide whether we want our scrollytelling to be a morph or a data object.

Using an own data structure and visualization technique may increase the scrollytelling's performance, as we describe in section 3.2. On the other hand, realizing the scrollytelling with a morph gives us many advantages. We do not have to implement the scrollytelling's visual representation in the editor. The lively.next system takes care of this. The editor can treat the scrollytelling as a black box, accessing only supported features. Thus, any behavior and feature can be added to the scrollytelling while it still looks and behaves in the editor as it would do outside of it. Moreover, the developers can use all the conventional lively.next tools which they are used to when working they are with the scrollytelling.

Implementing the scrollytelling as a morph integrates it well into the lively.next system and places no significant restrictions on the scrollytelling. Performance issues may be addressed as they occur. For now, we implement scrollytellings as morphs. In section 3.2, we discuss this topic more elaborately.

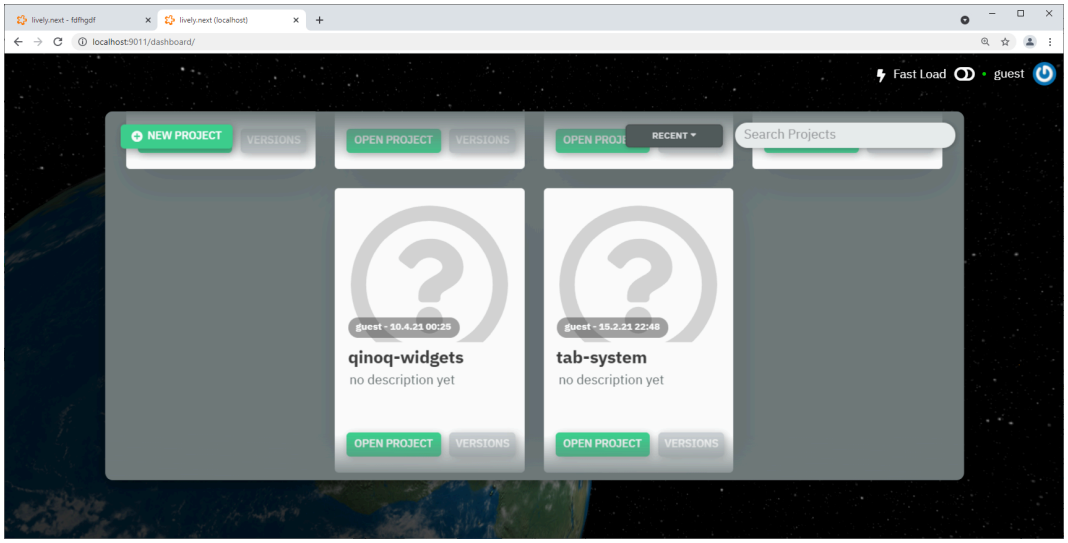


**Figure 2.12:** World overview of lively.next: Used to open and administrate worlds

**Project and Content Administration** In lively.next, every scrollytelling project usually lives in its own world, opened in a browser window. The world state, such as the opened windows and their content, are saved together with the world's content and is restored upon reopening the world. Both enable multiple open worlds at once and quick switches between worlds. The lively.next system already offers a world administration, as seen in Figure 2.12. It is unlikely that users need to work on multiple scrollytelling projects at once. If they must, they can open each of them in their world. If multiple scrollytellings must be opened within the same world, they can be copied and pasted across worlds. Finally, it is unnecessary to provide a scrollytelling share across worlds within the editor.

The situation for scrollytelling elements is similar. Since scrollytellings are usually highly individualized, an exchange of content across scrollytellings is unlikely. If necessary, it is possible to copy elements across worlds and thus scrollytellings. A *media pool* like in DaVinci Resolve, just for elements across scrollytellings, may technically be possible by using the lively.next feature *master components* but would require the editor to have assumptions about certain worlds of the lively.next instance. In conclusion, such a media pool would provide little benefit for a higher coupling of our editor with the system.

Nevertheless, it might be valuable to create a scrollytelling-specific media pool for content external to lively.next, like graphics and Lottie animations. To integrate those into a scrollytelling, users must create respective morphs and set the morphs' data URLs to the content files. This requires multiple steps per element and relies on external tools, may it be to find out the content file location or preview the file to selected the correct one. Moreover, previewing Lottie animations is not well supported on most systems. This kind of media pool could enable users to load elements once at the beginning of the project, maybe via a directory import. Furthermore, it could provide preview facilities for the common scrollytelling elements. Elements could be dragged out of the media pool into the scrollytelling without creating the respective morph first. On the other hand, it would require the scrollytelling to store objects which might not be used within the scrollytelling. Due to time limitations, the media pool does not have a priority in our project.

**Sequences** Our analysis shows two concepts to capsule elements: distinct visual entities like Microsoft PowerPoint's slides or DaVinci Resolve's compound clips. We already discussed that movies and scrollytellings have a lot in common, see subsection 1.3.4. Like movies, scrollytellings are planned using storyboards, which describe visual key moments of the medium. That is why we abstract the capsuling strategies with the terminology of the film industry: *scenes* and *sequences*. According to the German Film Academy, a scene "consists of one or more camera angles unified by setting and action".[22] A sequence is "a part of a movie", which is related "temporally, thematically, [or] spatially" and a "relatively autonomous, self-

[22]Glossary of the German Film Academy: https://www.vierundzwanzig.de/en/glossary/show/1354/detail/ (last accessed on 2021-07-28).

contained unit or phase".[23] A scene is limited to one setting; it is defined by a time and place. This likewise applies to the elements of slides: If two elements occurred at different times or in different settings, they would undoubtedly belong to different slides. Sequences are looser; their elements can merely have a common theme. Much like compound clips, which often feature elements from different points in time or places but form a semantic entity. Hence, we refer to concepts like slides as *scenes* and concepts like compound clips as *sequences*. Thus, we allow multiple sequences to be composed together at once, while there is only one scene visible at a time.

When developing our editor, we need to decide which concept works better for scrollytellings: sequences or scenes. The scrollytelling's storyboard usually visualizes separate scenes or moments of scenes. So do movie storyboards. Yet, movies have continuous progress and are thus edited in timelines as sequences. Using scenes, we focus on the static arrangement of elements. For example, in Microsoft PowerPoint, users define what the slide should look like once it is fully built up. Only then can animations be added. These may lead to this state or disperse it. A scrollytelling often does not have such a "final" state. It is in continuous progress. There are certainly workarounds to give the impression of continuous progress using scenes, yet this is not a desirable workflow. Likewise, there can be scrollytellings that rely on scenes only, but they are not the convention. In conclusion, scrollytellings have a continuous story like movies. That is why we use the sequence method. Our analysis shows that timelines tend to work for this method. This decision fits our project partner's requirements, who wants the opportunity to have overlapping content capsules, for example, to realize a constant background with changing foreground.

For our editor, we want sequences to have a duration and an internal timeline relative to the duration. This makes it easy to change the animation speed of all encapsulated elements when the scrollytelling is fine-tuned.

In DaVinci Resolve, sequences can be reused. This has the advantage that a change in the original sequence propagates to all instances. Created sequences could be stored in a sequence pool and dragged into the timeline when needed. Likewise, currently unused sequences, for example, alternatives to other sequences, could be found there. However, sequences are hardly reused in scrollytellings, at least not without minor adjustments between instances. Unused sequences are also unlikely due to the previously developed storyboard. If in need, the timeline could allow lanes and their contained sequences to be hidden. Sequences could be duplicatable. This does not propagate changes between the duplicates but allows for changes in the duplicate to quickly create a similar sequence.

**Animation** In Microsoft PowerPoint, animations are selected from type presets and attached to elements. Macromedia Flash and DaVinci Resolve use keyframe animations: The values of element properties can be fixed at specific points in time, the transition between two keyframes is interpolated. Attaching animations is simple and quick but lacks a consistent user interface and limits the customization to the

[23] Glossary of the German Film Academy: https://www.vierundzwanzig.de/en/glossary/show/1352/detail/ (last accessed on 2021-07-28).

available animation types and their parameters. Keyframe animations, depending on their complexity, can be cumbersome. On the other hand, they allow for precise control of animations and offer extensive customizability, especially when the easings between keyframes can be adjusted with Bézier curves. The uniform data structure and interface of keyframe animations are beneficial for developers interacting with animations via code. Eventually, keyframe animations can be used as the basis for higher-level animation controls. For example, it is possible to implement transitions for elements. Given two snapshots of an element's state, the editor sets the keyframes necessary to transition from the earlier to the later state (like Macromedia Flash's default animation behavior). Another valuable feature may be the motion path of DaVinci Resolve: the movement of an animated element is visualized as a path on the stage. The path can be edited, resulting in respective changes of the keyframes; an intuitive means to create motion.

That is why we use keyframe animations in our editor.

**Inspector** One design goal is a streamlined user interface. An aspect of this can be a panel that centralizes all element settings. An example of this is the inspector in DaVinci Resolve.

The lively.next environment already offers an inspector which displays all properties of a selected object. This may include internal properties that need to be handled with care. The tool may be valuable for developers but is too powerful for efficient and safe usage by content designers who simply want to access all properties relevant for composing and animating the scrollytelling. Additionally, the lively.next inspector is not integrated into the editor. That is why we prefer a specialized inspector with integrated keyframe controls inside the editor. By default, it may display selected relevant properties only. To keep the inspector flexible, developers should be able to define properties that should also be displayed in the inspector.

There are operations for elements that combine multiple properties and call functions to achieve the desired effect, for example, *center vertically*. These operations cannot be displayed in the canonical inspector. Instead, the inspector could be split into two tabs: One for the properties, the other for these higher-level operations. The operations could be displayed similar to the sidebar content of Microsoft PowerPoint: they are semantically grouped in panels. A panel is displayed if its operations apply to the currently selected element.

**Layouting** Scrollytellings run in the web browser, sometimes as elements of websites, more commonly as stand-alone web pages. Accordingly, they need to support a multitude of screen sizes.

As of now, scrollytellings are sized in pixels. Their elements are likewise positioned with absolute pixel values. To make scrollytellings responsive, developers rearrange elements using code and the somewhat limited lively.next layouts. Animations also have to be readjusted. Due to the significant differences between mobile and desktop devices, this method is similar to creating two different scrollytellings. As such, it is a major pain point in the development process. Furthermore, the editor should enable

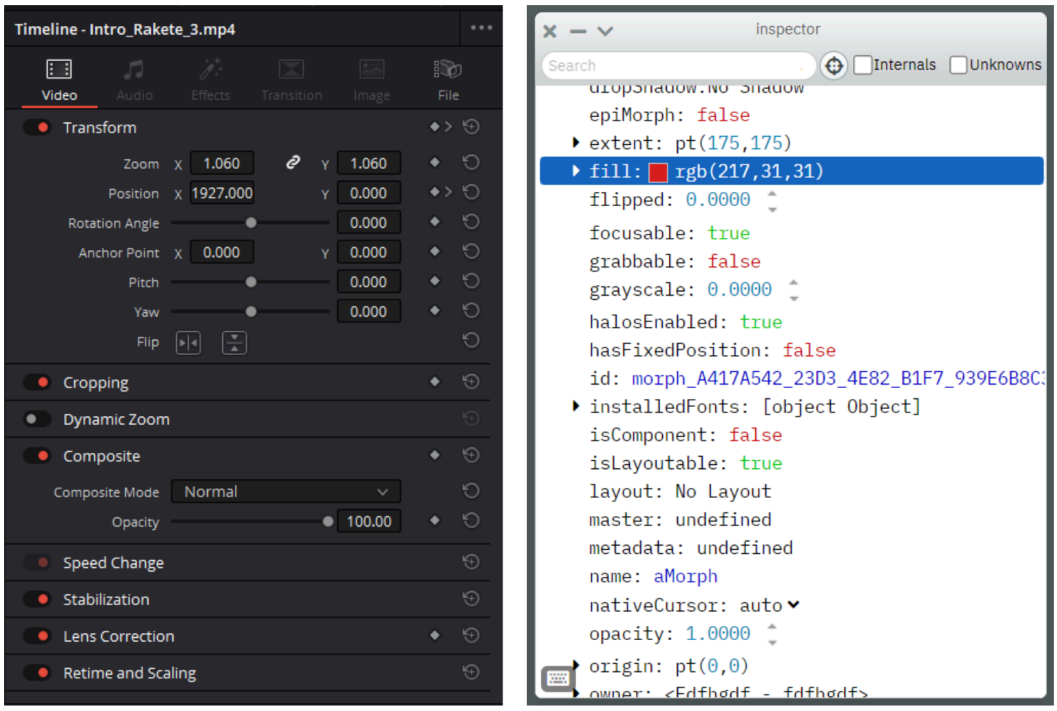


**Figure 2.13:** Inspector (A) of DaVinci Resolve compared to inspector (B) of lively.next: (A) groups settings by task and shows properties relevant for compositing elements. (B) also shows internal properties which should not be accessible to content designers.

content designers to arrange and animate scrollytellings. This includes making the scrollytellings responsive.

There are several ways to address this issue. CSS provides media queries to change the style depending on the screen type and size.[24] However, they are rather difficult to understand and set up. A more intuitive solution seems to be the *ConstraintLayout* of Android,[25] as depicted in Figure 2.14. It defines different kinds of dependencies – the constraints – between elements. The downside: The development of such a layout system is expensive.

That is why we focus our work on the development of an editor with absolute positioning. The editor may still be an improvement compared to the prior workflow. The ConstraintLayout may be valuable future work.

[24] https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries (last accessed on 2021-07-28).

[25] https://developer.android.com/training/constraint-layout (last accessed on 2021-07-28).

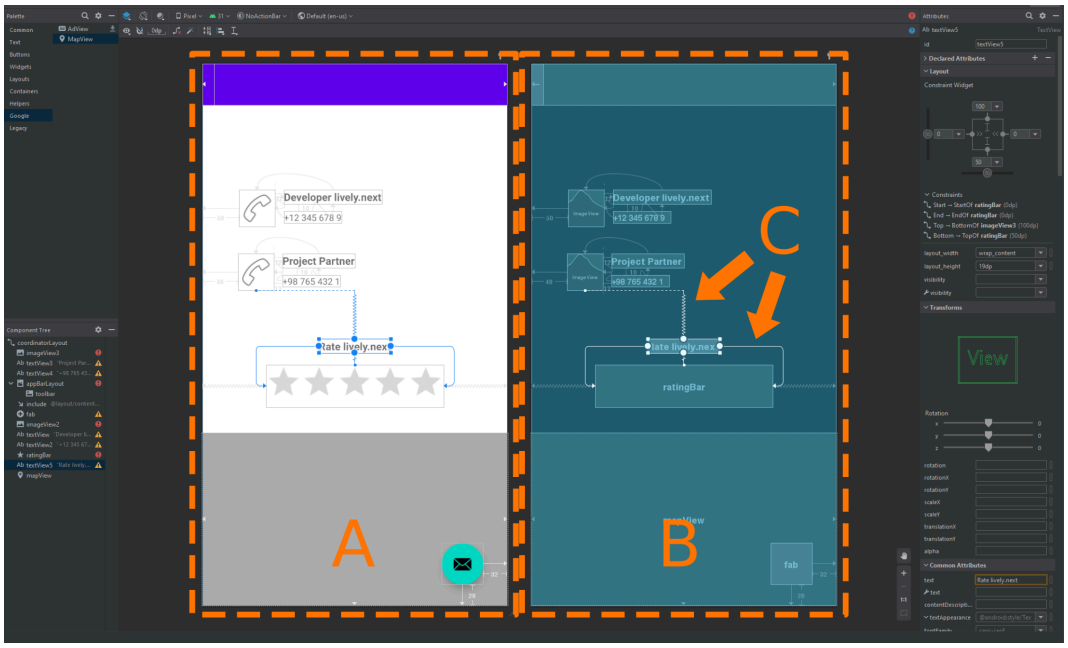


**Figure 2.14:** Example application layouted with ContraintLayout: Blueprint (B) shows all constraints, like constraints (C), of view (A).

## 2.6 Summary

Our analysis shows that none of the investigated applications fit the particular requirements of scrollytelling creation. Still, we learn valuable lessons which we can rely on when developing our editor. The content can be administered best using a timeline. Keyframe-based animations offer great precision and high customizability while being a basis for future abstractions in specialized scenarios. The lively.next system provides multiple features which the editor thus does not need to cover itself. Implementing the scrollytelling as a morph can make use of existing tooling in lively.next and makes the features of the scrollytelling independent from editor support. This results in greater flexibility.

Our discussion of design decisions makes clear that this project has room for improvement. A constraint layout would enable content designers to create responsible scrollytellings. The available animation techniques could be extended, for example with abstractions of keyframe animations. Eventually, a media pool for external elements may facilitate their import.

However, even at the planned state, the editor can be a valuable support for the collaboration of content designers and developers. In chapter 3, we describe the realization of the editor concept and technical backgrounds of the project. In chapter 5 we evaluate how well our editor supports our project partner Typeshift and identify further improvements.

# 3 Design and Implementation of an Editor for Scrollytellings in lively.next

As described in section 1.3, scrollytellings are web pages that combine different modalities, including text and images but also interactive elements into one web page that tells a story that users control via scrolling. An editor for creating scrollytellings needs to accommodate content designers and developers alike and assist them both in doing what they do best to create scrollytellings. The lively.next system[26] is a platform that enables the creation of web pages, and by extension of scrollytellings. We developed the scrollytelling editor with lively.next and it is used in lively.next. That is why we will start by looking at what lively.next is and how the morphic graphics system can support the creation of scrollytellings.

## 3.1 lively.next

Being an advanced version of the Lively Kernel,[27] *lively.next* is a live object system[24] and integrated programming environment. Usage of lively.next (see Figure 3.1) happens primarily through a web browser. It is designed to be used by developers and non-developers, offering a rich selection of tools to support, among other things, the creation of web pages.

Applications and web pages are created in lively.next as a set of runtime objects. Users can create stand-alone web pages within lively.next by bundling (see subsection 3.4.5). Since it is running on the web, the mapping between the development environment and the resulting web page is direct. For example, JavaScript code may be written in lively.next which is used in the finished web page, as well as in lively.next. These web pages can be opened in an arbitrary browser and only use HTML, CSS, and client-side JavaScript.

The usage of external modules is supported, allowing lively.next developers to use a wide range of JavaScript libraries available. We have utilized this to allow for the integration of Lottie Animations[28] within lively.next. This type of animation is further explained in subsection 4.3.5.

While the lively.next instance is used in the browser, a lively.next server runs in the background. The server, based on NodeJS,[29] is responsible for providing files (for

[26]https://lively-next.org/ (last accessed on 2021-07-28).
[27]https://lively-kernel.org/ (last accessed on 2021-07-28).
[28]https://airbnb.design/lottie/ (last accessed on 2021-07-28).
[29]https://nodejs.org/en/ (last accessed on 2021-07-28).

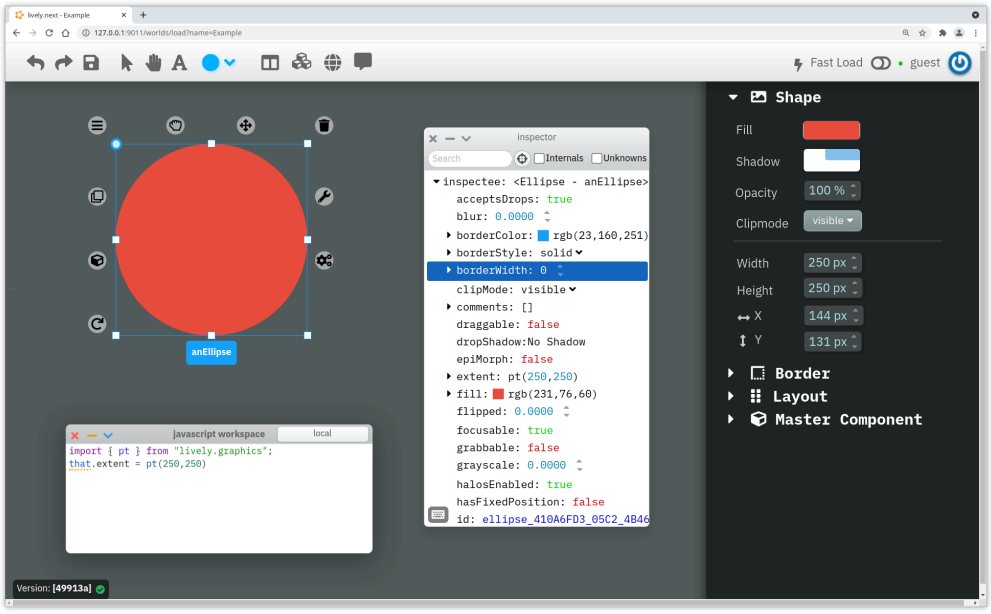


**Figure 3.1:** A chromium browser with a lively.next instance opened

example, JavaScript modules) to the browser via HTTP or WebSocket. This process can run on a remote server or a local machine.

### 3.1.1 Morphs in lively.next

The lively.next system uses the *morphic* graphics system.[30] Morphic is a graphic system first introduced in Self 4.0[44] and used in Smalltalk and other Lively distributions. Every visible object in lively.next is a *morph*, with common morphic properties, behavior, and rendering mechanics. A morph combines data, behavior, and visual interface into one object. Morphs are arranged in a hierarchy that can be traversed with the `submorph` and `owner` properties. All visible morphs are descendants of the `world`, which is a morph itself, through the submorph hierarchy. Thus graphical elements, such as the GUI, within lively.next are constructed using a composition of morphs. Morphs are rendered using a virtual document object model (VDOM) applied to the document object model (DOM) in batches. In the resulting DOM, every morph corresponds to an HTML DOM node.

Since morphs make up all graphical elements, for the creation of web pages and scrollytellings morphs need to be composed into complex arrangements of morph hierarchies as well as changed visually. Properties on morphs determine their visuals.

Interaction with morphs is often done with halo menus, as shown in Figure 3.2. These interface elements can be activated for each morph individually by clicking on a morph while pressing the control key. Halo menus include options such as moving

[30] https://github.com/LivelyKernel/lively.next/tree/master/lively.morphic (last accessed on 2021-07-28).

a morph, grabbing it (removing it from its owner and subsequently adding it to a new owner), resizing it, and opening the lively.next inspector or object editor, tools described in subsection 3.1.2.

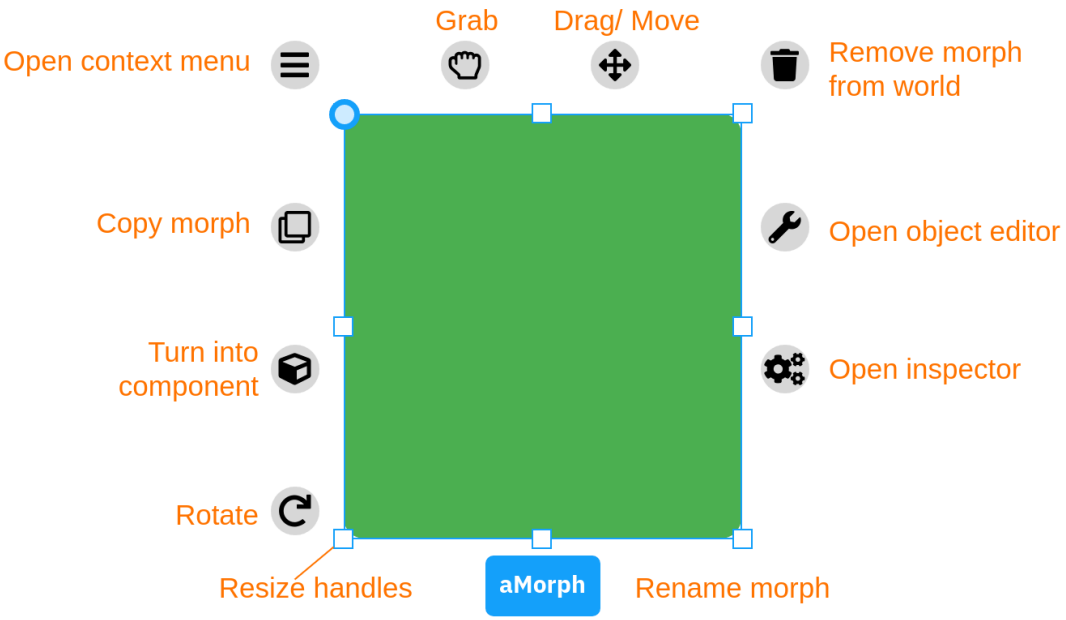


**Figure 3.2:** The halo menu opened on a green rectangular morph

Some of the most important morphic properties are listed in Table 3.1. Properties can be overwritten and extended in subclasses. A property declaration for a custom class may look like Listing 3.1.

**Listing 3.1:** The declaration of properties, as done in morph classes. The property 'caption' is displayed, with a setter and a default value.

```
static get properties () {
    return {
        caption: {
            defaultValue: 'No caption specified!',
            set (caption) {
                this.setProperty("caption", caption)
                this.getSubmorphNamed("label").textString = caption
            }
        },
        //...
    }
}
```

Usually, morphs are subclassed for specific new behavior. The standard morph constructor is used to initialize morphs. It has a parameter for properties, which

| Property Name | Purpose | Type | Notes |
|---|---|---|---|
| name | Identification for developers/ lively.next users | string | Used in the user interface (shown and editable in halo menus) as well as in methods (for example `getSubmorphsNamed(name)`). |
| submorphs | Children in morph hierarchy, positioned relative to this morph, rendered as children in the DOM node[31] | array | |
| _owner | Morph that has this morph as submorph | morph | The _owner property is set by the owner when the morph is added as a submorph. |
| halosEnabled | Enable/ disable the halo interface on a morph | boolean | When this is false, halo menus may only be opened via code, not via pressing control and clicking. |
| position | Positioning of morph relative to origin of owner | point | The values are in pixels. |
| extent | Width and height of morph | point | The values are in pixels. |
| fill | Filled color on screen | color | |
| opacity | Transparency of morph | number | Values may range from 0 to 1. |

**Table 3.1:** A dictionary on the morph's class defines the *morphic properties*.

can be filled with a dictionary of properties that is applied to the new morph. This means morph subclasses typically do not implement a constructor method, as commonly done to specify class-specific initialization behavior using a method with the `constructor()` signature in JavaScript.[32] Morphic properties also support custom getters, setters, and methods for initialization as well as default values. Properties are also crucial for serialization, as only morphic properties are serialized on morphs, not arbitrarily defined keys.

[32] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor (last accessed on 2021-07-28).

To give a brief overview of the morphic interface, here are some important methods. The method `remove()` removes a morph as a submorph from its owner, causing it not to be displayed anymore. The method `addMorph(submorph)` adds the morph in the argument to the submorph hierarchy of the morph the method is called on. When the morph is already a submorph, it is placed at the end of the submorph list, causing it to be rendered in front of all other submorphs. These two methods are crucial for working with morphs; we describe a specific scenario in subsection 3.2.3.

DOM events are handled by lively.next and translated to lively.next internal events. Various event callbacks are implemented on morphs such as `onMouseDown(evt)`, `onDrag(evt)` or `onKeyDown(evt)` for translated DOM events, as well as for lively.next internal events such as `onOwnerChanged(newOwner)`.

Worlds in lively.next are morphs themselves and work similarly to the desktop in an operating system GUI. In it, morphs can be created and placed on the screen and windows (again morphs) with tools opened. Worlds may be saved and loaded on the server, where they are stored as object graphs in the JSON format. Serialization of worlds is described further in subsection 3.4.2. The world can be reached in code with the `$world` shorthand.

### 3.1.2 Tooling in lively.next

Tools provided by lively.next include a browser for source code editing on files, an object editor for source code editing on objects, a JavaScript workspace, an inspector, and a code search tool. These tools support developers in exploratory programming within lively.next [60, 53]. This can be used for quick prototyping to create interactive elements in scrollytellings.

There are many places in lively.next where developers can execute and write code, such as the JavaScript workspace, a simple window with a text input. In every such code environment, options are available to make programming more comfortable. Code is automatically highlighted, checked for syntax errors, and formatted. References to other modules can be automatically resolved by adding missing import statements. When some text is selected, it can be evaluated (doit), and the result can be printed or inspected, opening the *inspector*. Morphs in the world can be selected by clicking while holding the Alt key and are then accessible in code environments with the `that` variable. This enables developers to manipulate morphs in code quickly.

As well as from a code environment, the inspector can be opened with a button in the halo menu. The inspector can be opened targeting any JavaScript object but is primarily built for morphs. When opened on a morph, a filterable list of properties with values is shown as seen in Figure 3.3. These values can be easily edited with widgets or changed in code with an integrated panel. The inspector is the fastest way to change specific properties on morphs and to experiment on them.

Just like the inspector, the *object editor* is opened in a window with a button in the halo menu. It contains the code of the class that this morph belongs to, which can be seen in Figure 3.4. Changes can be performed by changing the code and then saving, which directly influences the behavior of that morph and all others of the same class.

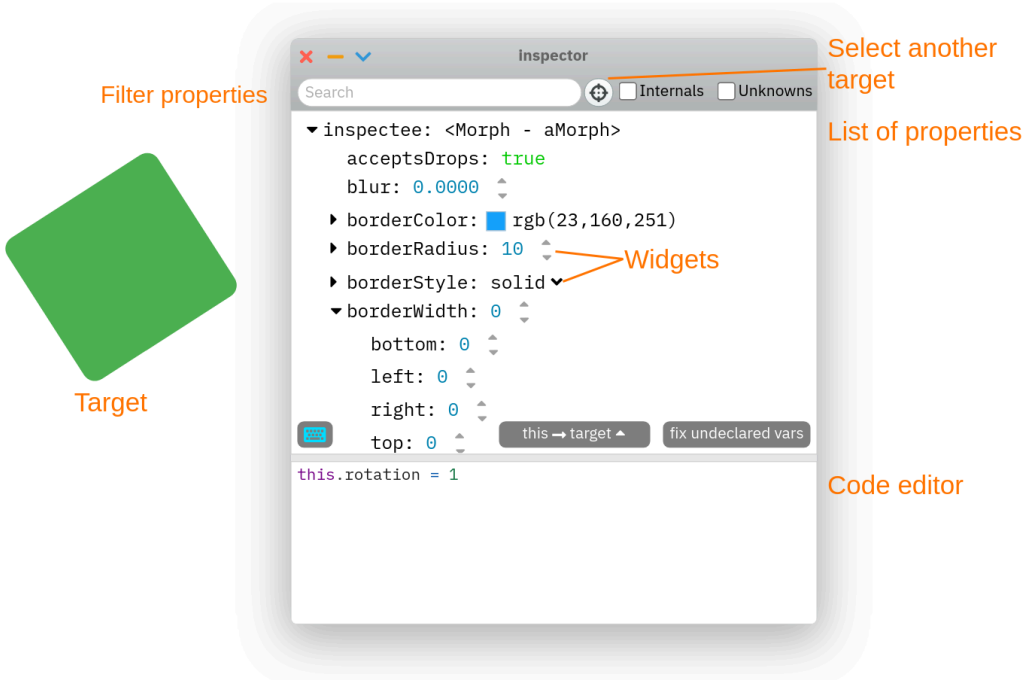


**Figure 3.3:** An opened inspector targeting a green rectangular morph

Additionally, a button allows creating a new subclass of the morph's current class, which the selected morph then belongs to. This allows for the easy editing of morphs in a world without changing important system-wide classes. Bundling of a morph can also be achieved with a button in the object editor (see subsection 3.4.5).

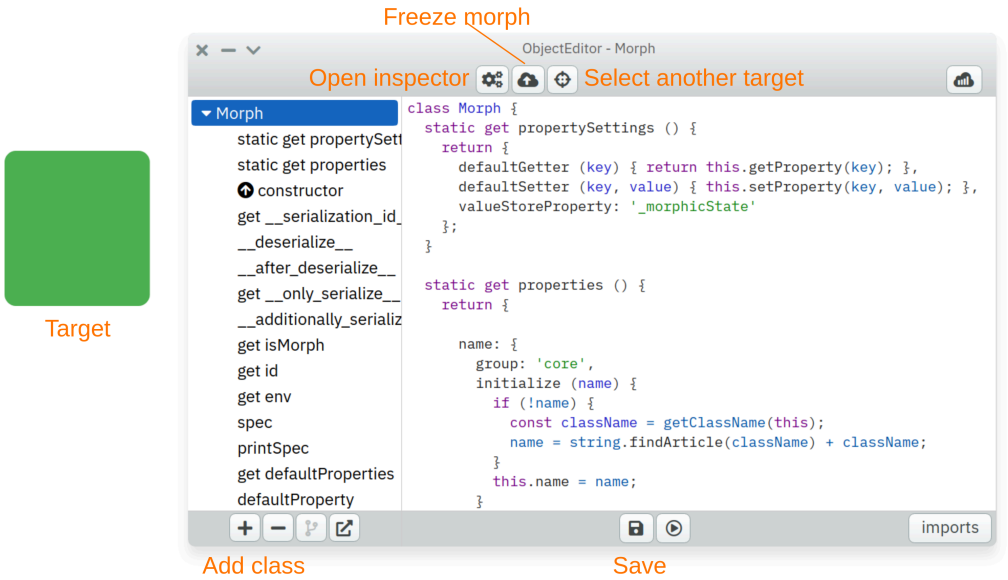


**Figure 3.4:** An opened object editor on a green rectangular morph

The lively.next world also contains the *top bar*, seen in Figure 3.5. It is a user interface element that allows creating certain types of new morphs, changes mouse modes (interaction mode for general interaction such as pressing buttons and halo mode to open halo menus when clicking on morphs), and other options. The top bar is the primary way new morphs are created, but morphs can also be created in code and added to the world.

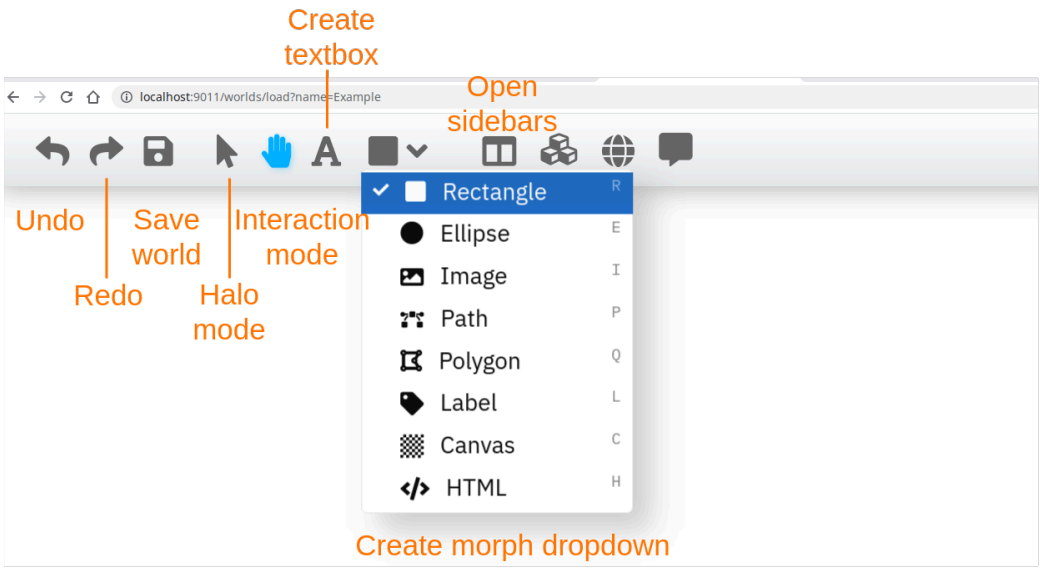


**Figure 3.5:** Left side of lively.next top bar in interaction mode

There is also a *styling palette* to style morphs, a sidebar separate from the inspector, which allows changing properties such as border colors, extent, and various rich text settings. It opens as a side panel and can be seen in Figure 3.6 on the right side. It is explicitly designed to be easy to use and does not offer up code, instead offering styling options common to tools such as presentation programs.

Non-developers can use the styling palette, the morphic halo menu, top bar, or the inspector to an extent to quickly create visual morph hierarchies and fine-tune their appearance. Additionally, developers can use the inspector, the object editor, and other code editing environments to create and alter morphs but also add behavior.

### 3.1.3 Connections

A particular property on morphs is the array of *attribute connections*. While being a property on morphs, it is not a morphic property, and developers do not interact with the array directly. Instead the functions `connect(sourceObj, attrName, targetObj, targetMethodName, specOrConverter)` and `disconnect(sourceObj, attrName, targetObj, targetMethodName)` are used, as demonstrated in Listing 3.2. The array contains attribute connections, which are objects that store a source object,

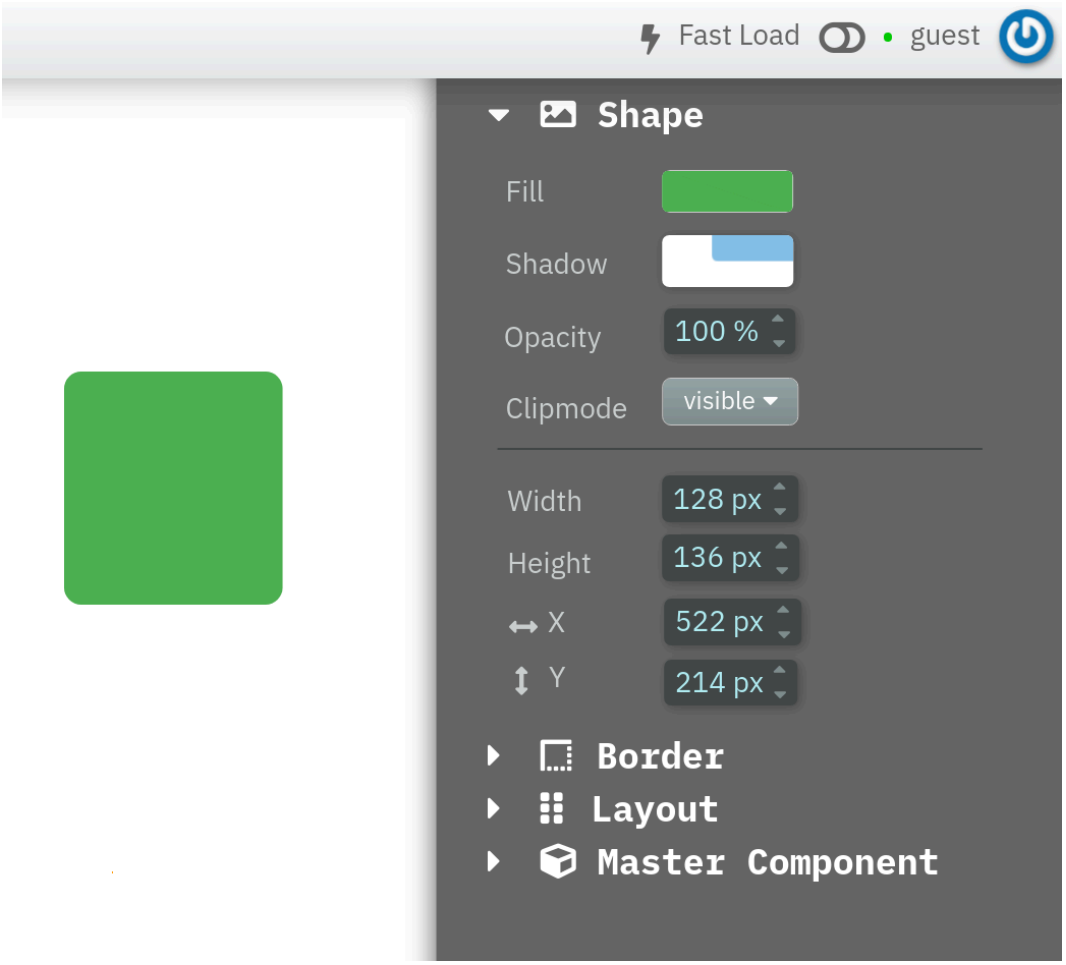


**Figure 3.6:** The styling palette side bar selected on a green morph

a source attribute name, a target object, and a target method name.[33] Whenever a property on a morph is changed, all attribute connections stored in that morph are filtered to have that morph as the source object and that property as the source attribute name. Every matching connection is triggered and may call callback functions or change attributes on other morphs (with the target object and target method name). Additional logic can be implemented with anonymous functions that convert inputs or manipulate other properties, shown in line 7 of Listing 3.2.

**Listing 3.2:** Demonstration of connections on two morphs

```
1 // Given morph a and morph b
2 connect(a, "position", b, "position");
3 // Such a connection results in morph a moving whenever morph b
4 // is moved (both move to the same position)
```

[33] https://github.com/LivelyKernel/lively.next/blob/master/lively.bindings/index.js (last accessed on 2021-07-28).

```
// This can be undone with the disconnect function.
disconnect(a, "position", b, "position");
// Using a converter function, we can fix morph b
// at a distance to morph a.
connect(a, "position", b, "position", {
  converter: (value) => value.addXY(100,100)});
// Connections can also be triggered by method calls.
connect(a, "onMouseDown", b, "triggerButtonAction");
```

Connections are widely used in lively.next. A common use case is to have a control for a property on a morph to be connected with that morph. Connections can be used in both directions, thus changing the control changes the property, and changing the property changes the control. The connection system detects circles that are created through such a double connection and only runs them once.

## 3.2 Scrollytellings in lively.next

The lively.next system supplies capabilities for visual storytelling with morphs and offers tools to edit them, as described above. Additionally, behavior can be added to morphs, and external JavaScript dependencies can be leveraged, allowing for flexibility. Morphs can be exported as web pages through bundling (see subsection 3.4.5). This makes it a suitable environment for the creation of scrollytellings.

As all graphical objects are composed of morphs in lively.next, creating a scrollytelling in lively.next is achieved through a composition of morphs as well. Even with this requirement, there are still different ways in which morphs can be utilized to create a scrollytelling. We now want to look at a few different options of creating scrollytellings in lively.next.

Let us consider a web page that tells the story about the start of a rocket to Mars, as seen in Figure 3.7. It should include a background that does not scroll to contrast with the scrolling content (A). Information concerning the rocket start should be displayed in a text (B). An interactive slider should show how much fuel is needed to transport some payload into space and to Mars (C). We want a video of the rocket start to be controlled via scrolling (D).

Scrollytellings may be created using plain HTML, which would be supported through the use of HTML morphs in lively.next. These are morphs that can contain arbitrary HTML and render that HTML directly to the web page into the bounds of a morph. Creating a scrollytelling in such a way would not open up the rich morphic interface for elements in the scrollytelling, such as images or text. With this approach, we would lose almost all the tooling in lively.next.

Another option for the creation of scrollytellings in lively.next makes use of an HTML5 canvas. A reason to do this would be to achieve a better performance than through plain HTML[8]. While canvas morphs that support HTML5 canvas rendering are supported in lively.next, the canvas morph would encapsulate all scrollytelling elements, thus hiding the interface needed for editing in an editor.

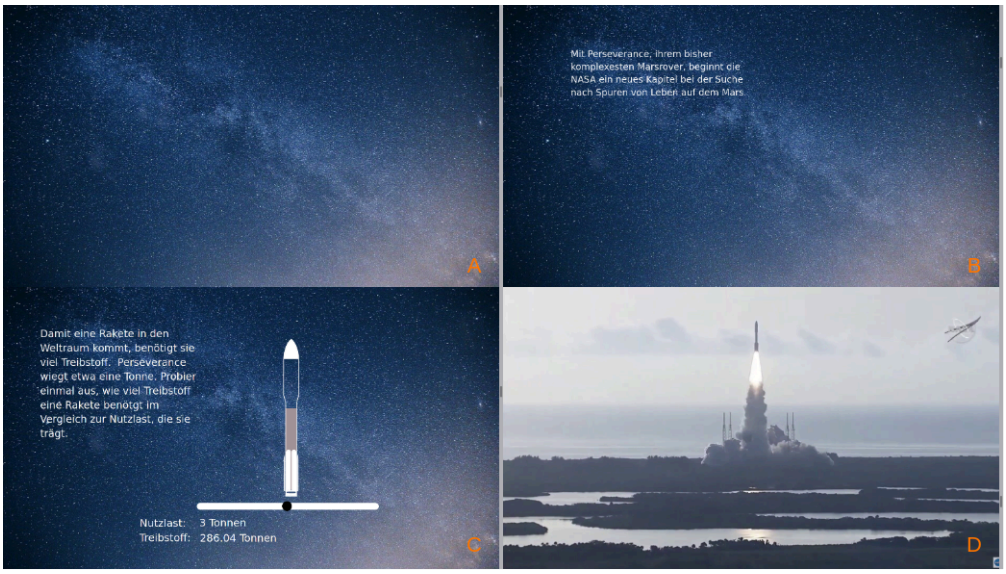


**Figure 3.7:** Four screenshots from the example scrollytelling we want to create

Scrollytelling elements would have to be drawn using the canvas API, and an additional abstraction would need to be created on top of it, which the lively.next morphic system already offers. Again tooling would not be available, making the adjustment cumbersome. Changing specific scrollytelling elements would not be able for non-developers.

Scrollytellings can be created *ad hoc* by combining morphs and adding logic. This approach allows rapid prototyping of scrollytellings without the use of an editor. Tooling in lively.next can be used, such as the styling palette or the inspector to make quick changes to some components. Adjusting animations, such as the rocket's ascension, would need to be programmed through code and thus be unavailable to non-developers. A visual editor is required to enable content designers to work on the scrollytelling. However, scrollytellings and their submorphs must define a clear interface for such an editor, thus rendering ad hoc solutions unsuitable. This was the workflow of our partner Typeshift before the editor was developed. We also describe this in subsection 2.1.2.

That is why we create scrollytellings and their components in a fixed hierarchy of morphs of specific classes.

### 3.2.1 Structure of Scrollytellings in qinoq

It is useful for scrollytellings to expose their elements through an interface to be editable within an editor. This interface should allow operations such as editing and inspecting morphic properties of morphs within. Scrollytellings use a composite pattern as follows from them being composed of morphs. Thus the primary interface used for interacting with them is the morphic interface. The structure must be flexible enough to allow for many different kinds of scrollytellings.

To show our structure, we will take the example of the scrollytelling about a starting rocket. A simplified structure can be seen in Figure 3.8. We will now examine the parts that make up a scrollytelling.

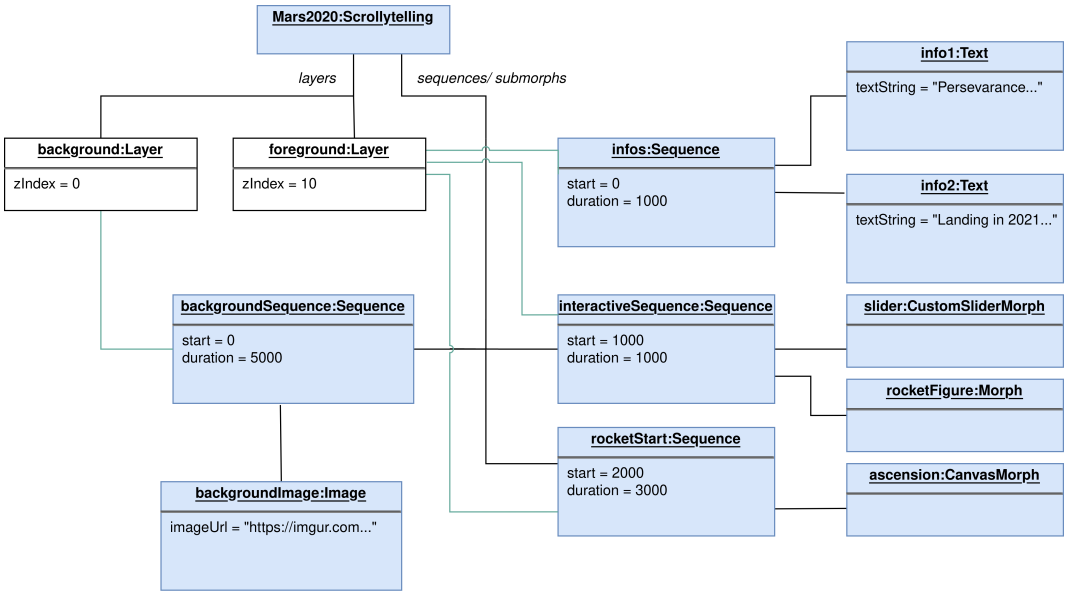


**Figure 3.8:** The structure of an example scrollytelling as an object diagram: Blue objects are morphs.

**Morphs in a Scrollytelling** Morphs are the atomic units of a scrollytelling. Everything that can be seen is a morph. Different types of morphs are provided by lively.next, such as simple forms like rectangles and ellipses, labels and text areas, and more flexible types such as HTML morphs. In our example, the background may be an image morph, a text offering information may be a text morph, and the slider may be a morph explicitly created for this purpose. We will call morphs used in such a way *generic morphs*.

Generic morphs of custom classes (custom morphs) are the interface in which developers can add custom functionality such as games, interactive content beyond scrolling or state machines. Further details are described in subsection 3.3.4.

**Sequences** Generic morphs in a scrollytelling are combined to form *sequences*. These take on the role of semantic entities as they group elements that are related. In our example, all morphs that make up the fuel interaction may be part of the same sequence. This means they are displayed together and enter and leave the scrollytelling together. Sequences are morphs, and the generic morphs they contain are their submorphs.

Generic morphs may only belong to a single sequence, which is enforced through the strictly hierarchical submorph-owner-tree structure. Sequences only contain generic morphs and may not contain other sequences or scrollytellings as submorphs.

Sequences have a start and a duration (which also defines an end). These properties directly map to the vertical scroll position at which the sequence is shown while scrolling through the scrollytelling. Sequences contain data relevant to the morphs in it, such as an array of animations.

**Layers** *Layers* are used to organize sequences, wherein every sequence is assigned to a layer. A layer may hold different sequences that do not overlap (Two sequences overlap if they are in the same layer and one sequence's start value is smaller than the other sequence's end, while its end is greater than the other sequence's start). Layers are not morphs but rather simple objects, so they are outside the submorph hierarchy. Layers have a zIndex[34] that is used to determine which sequence is shown in front of another sequence.

In our example, we can imagine a background layer that holds a sequence for a background and a layer for the foreground that contains elements such as informative text boxes.

**Animations** Generic morphs in a scrollytelling may be animated using *qinoq animations*, as described in detail in section 4.3. The animations are stored in an array at the sequence. Every animation has a target morph, a morphic property to animate, and a list of keyframes used for interpolation. There may only be one animation for every morph property combination, and each animation is only responsible for one property on one morph.

Animations are stored in sequences. Storing them in the morphs would require us to store them away somewhere else for saving since only properties are serialized. We cannot add an animation property to morphs added to the scrollytelling since property definitions cannot be changed for single morphs.

Simple animations used frequently in our example are animations for the opacity property on morphs. Going from 0 to 1 at the beginning of the sequence and then again from 1 to 0 at the end makes elements appear smoother, as they do not suddenly pop out of nowhere. These can be used for the text boxes or the slider, for example.

### 3.2.2 Scrolling in Scrollytellings

Implementing scrolling in scrollytellings is not trivial. Scrolling needs to support navigating the page with the arrow keys, the mouse cursor, and touch interaction on mobile devices. An additional requirement for mobile devices is the support of inertial/ momentum scrolling, where scrolling starts with a flick and continues without additional input. To achieve these goals, we decided to use the built-

[34] The name is taken from the CSS property `z-index`, which orders DOM elements. See also https://developer.mozilla.org/en-US/docs/Web/CSS/z-index (last accessed on 2021-07-28).

in browser scrolling behavior. Browsers already support the different ways of navigation depending on which platform they are running on. An element has to be clipped in a container to show a scrollbar and enable scrolling.

With each scrollytelling, a special morph, called *scroll overlay*, is placed in the world. This scroll overlay is always on top of the scrollytelling but has a transparent fill. Inside that morph, another morph is placed, the *scrollable content*, which is an invisible morph with a height that is greater than the height of the scroll overlay, as seen in Figure 3.9. This leads the scrollable content to be clipped. The morphic property `clipmode` maps to the CSS overflow property.[35] With `clipmode` set to auto for the scroll overlay, scroll bars are shown, and scrolling is possible with the native browser scrolling. The height of the scrollable content is dependent on the length of the scrollytelling (achieved through a connection).

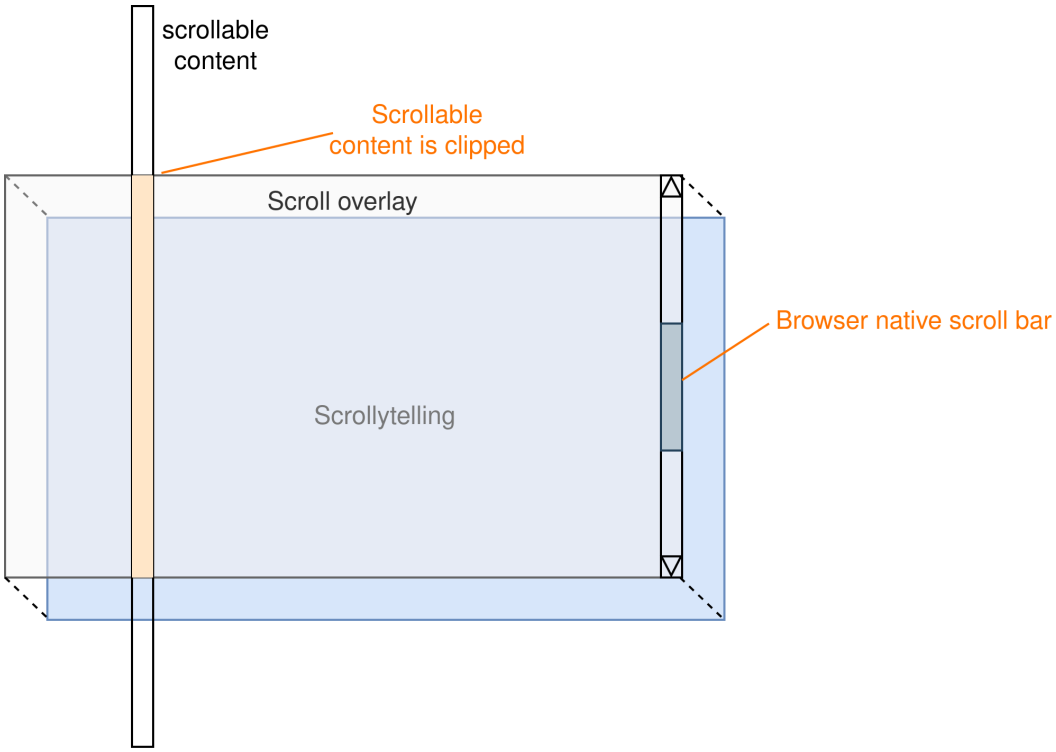


**Figure 3.9:** Visualization of the clipping of the scrollable content in the scroll overlay: The scrollytelling is the blue pane in the background, the scroll overlay the semi-transparent pane on top.

This solution requires the scroll overlay to propagate all manner of events to morphs that lay behind it. This is facilitated by overwriting the callback methods implemented on morphs in lively.next, such as `onMouseDown(evt)` and calling the

[35]https://developer.mozilla.org/en-US/docs/Web/CSS/overflow (last accessed on 2021-07-28).

same method on the morph that lies beneath the scroll overlay in the display order of morphs.

### 3.2.3 Drawing of Scrollytellings

When the scroll overlay is scrolled, the method `redraw()` is called on the scrollytelling. For every sequence in the scrollytelling first the method `updateProgress(scrollPosition)` is called, which allows animations to be performed, as described in subsection 4.3.6. It is then decided if the sequence is to be displayed. This is determined with its start and end and additional flags such as if the sequence's layer is marked as hidden. Afterward, the sequence is either added to the scrollytelling with `addMorph(submorph)` or removed with the `remove()` call. These are both implemented by the morph class and thus available to all morphs, as described in subsection 3.1.1.

We make sure to sort the sequences array of the scrollytelling whenever a sequence is added, or a layer moved. That is why using `addMorph(submorph)` in such a way causes the sequences, and thus also the submorphs of the sequences, to be rendered in the correct display order.

## 3.3 A Scrollytelling Editor in lively.next with qinoq

Creating scrollytellings within lively.next in the structure mentioned above can be done with code. However, our project's goal is to enable content designers to create scrollytellings; thus, a visual editor is needed. Since scrollytellings are created in lively.next, the editor must also interface with lively.next. As we have seen, lively.next as an environment offers a lot of tooling, which is also built within lively.next. Therefore, we have also built the editor in lively.next, and users will work with it within lively.next. The editor is composed of morphs that make up the visual interface and handle behavior. The editor follows the design decisions we have outlined in subsection 2.5.3. The combination of the editor and the scrollytellings created with it is called **qinoq**.

### 3.3.1 Editor Structure

The editor is opened in a window and can be resized, minimized, and closed as any other window in lively.next. It consists of different panels. The main panels of the editor are, going clockwise and starting on the upper left side, the tree, the holder, the inspector, the menu bar, and the timeline. The editor in its entirety can be seen in Figure 3.10. We will now take a look at the different panels and their functions.

**Tree** The *tree* is a visual representation of the structure of the scrollytelling. As seen in Figure 3.11, it shows a hierarchical graph of the scrollytelling, displaying sequences, submorphs of sequences, animations, and keyframes. It can be filtered

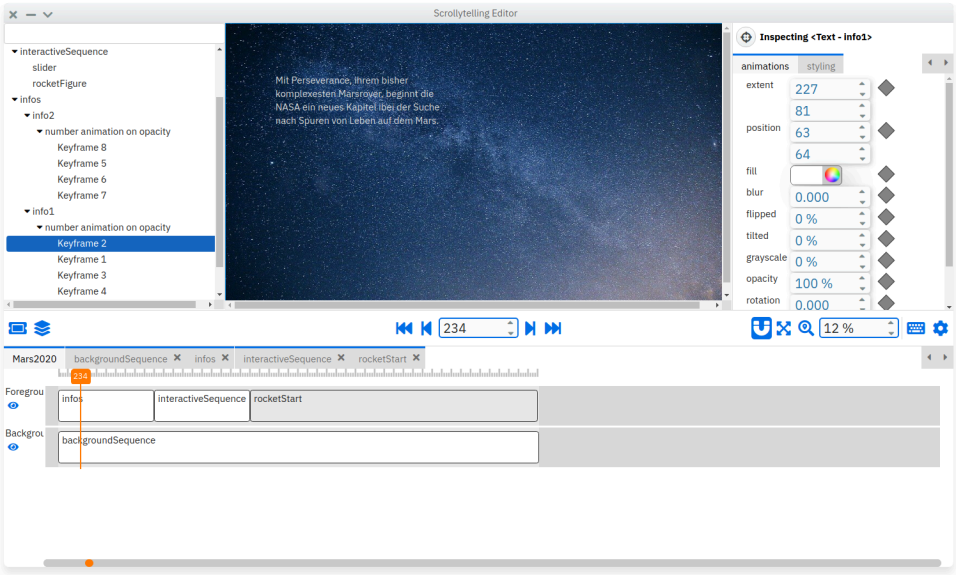


**Figure 3.10:** An opened editor with an example scrollytelling with global timeline

to search for specific items, which is supported by the ability to name sequences, keyframes, and morphs in the editor. Clicking on items allows the editor to jump to an appropriate view of the item. This allows users to quickly move around to views on different elements of the scrollytelling. For a sequence, a tab on the sequence with the sequence timeline is opened; when jumping to a keyframe, it is marked with a red border in its property lane.

While animations are not children of their target morphs for the reasons described above, it is shown in the graph in such a way because every animation has only one target morph, and the animations are grouped by morphs in the sequence timeline.

**Holder** The *holder* is a morph that contains the currently edited scrollytelling. When no scrollytelling is loaded in the editor, it can be created with a button on the holder. An existing scrollytelling can be grabbed using the halo menu and dropped on the empty holder, thus loading the scrollytelling into the editor. The process of creating a scrollytelling is described in detail in subsection 5.2.2

Since the holder contains the scrollytelling, it also allows for scrolling, as described in subsection 3.2.2. When a scrollytelling is in the holder, the scroll overlay is a submorph of the holder. The holder contains the actual scrollytelling, not a preview of it. This is in contrast with previews in video editing software.

**Views**

While by default, the entire scrollytelling is shown, the scrollytelling in the editor has a special view, the *sequence view*, in which only morphs of a sequence are shown in color. In contrast, all other morphs are shown in black and white with reduced opacity. This allows users to see what morphs belong to the sequence while still

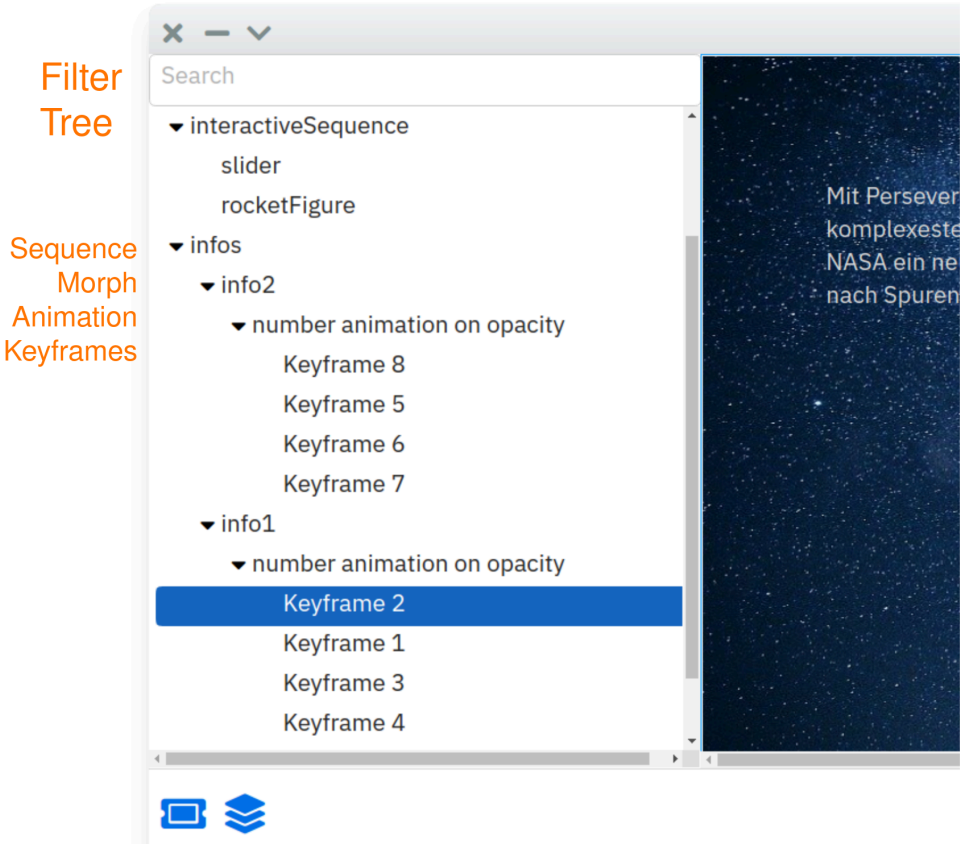


**Figure 3.11:** Tree in editor with example scrollytelling

being able to match morphs and animations of that sequence with morphs from other sequences.

Only in the sequence view morphs may be added to the scrollytelling. Adding morphs can be achieved by grabbing and dropping them into the holder using the halo menu or using the top bar that is integrated in lively.next and explained in subsection 3.1.2, which allows the creation of morphs by dragging in the world. When the sequence view is visible, users can draw morphs on the scrollytelling, thus creating them and adding them to the sequence.

**Inspector in the Editor** The *inspector* can target a chosen morph in the scrollytelling. It allows for the inspection and adjustment of various morphic properties on the target, such as changing the position or filled color. As seen in Figure 3.12 for the `info1` text morph that contains information on the rocket start, properties are shown in a list with associated widgets. These properties can also be animated in the inspector with the keyframe buttons associated with each property. It takes inspiration from the integrated lively.next inspector and the styling palette, described in subsection 3.1.2. Modeling this tool after the styling palette should make it easier to understand for users that are already familiar with that tool.

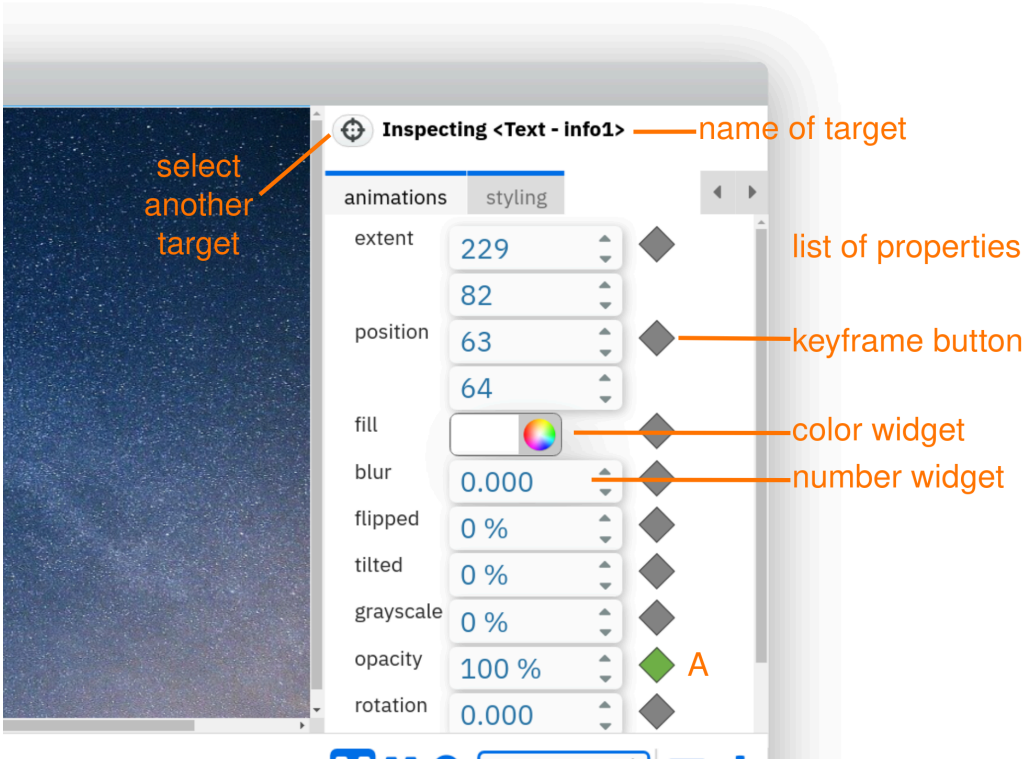


**Figure 3.12:** Inspector in an editor with example scrollytelling targeting a text named 'info1'

Properties that can be animated on a morph are determined in a two-step process. First, a prepared dictionary of morphic properties with their types (such as point, number, and color) is compared with the properties that exist on that morph. This dictionary contains the most important properties to animate scrollytelling elements such as opacity, extent, position, filled color, scale, and rotation. In a second step, additional properties are added that are defined in the class of the morph. A morph's property declaration can contain the `animateAs:` attribute, which also specifies the property type. Developers can create custom morphs with behavior determined by properties and mark these properties with `animateAs:` to allow content designers to create animations. An example for this can be found in subsection 4.3.2.

To animate a property with the inspector, the value widget of the property is of interest. This value widget can be a color picker, a text field, or number input fields depending on the property type. The value in the widget and thus the value in the morph may be changed with the widget directly or in any other way. For example, a morph's position may also be changed with the halo menu. Once the value is satisfactory, a keyframe can be set at the current scroll position of the scrollytelling with the keyframe button, creating an animation if there is none for that morph and property in the sequence. Additional keyframes can be set at other scroll positions, causing interpolation by setting different values. When scrolling again to a scroll position that already has a keyframe on the inspected morph, this is indicated by a

colored keyframe button, as can be seen at (A) in Figure 3.12 on the opacity property. Setting a keyframe on the same property at the same scroll position overwrites that keyframe's value.

Additionally, the inspector can be used to modify the morph while not animating it. This is done in the styling tab. The inspector shows different options for different types of morphs in that section. For example, for social media button morphs, which are a particular type of morph we added that allows consumers to share the finished scrollytelling with others on social media platforms, the inspector allows the setting of a URL and the text of the message.

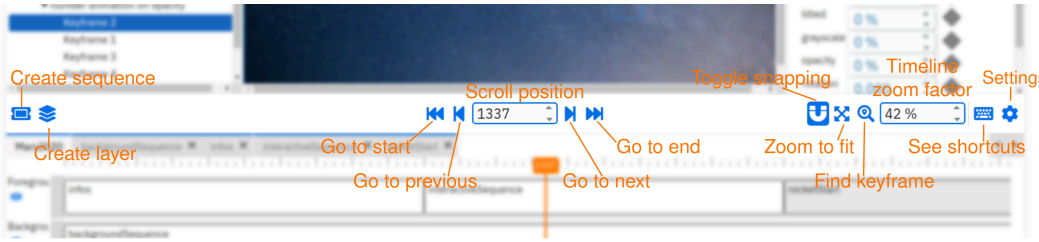


**Figure 3.13:** Menu bar in global view

**Menu bar** The *menu bar* is a control panel at the center of the editor. It is displayed in Figure 3.13. On the left side, new elements can be added to the timeline, namely sequences and layers. These options are disabled in the sequence timeline. In the center, the scroll position can be adjusted. This is facilitated through a number widget that displays the current scroll position and can be edited. Buttons around it allow jumping to different scroll positions in the scrollytelling. Depending on which view the scrollytelling is in, the buttons have different functions, for example, jumping between keyframes or jumping between sequences. There are additional options on the right side, mainly targeting the timeline, such as a timeline zoom control.

**Timeline** The *timeline* is a central component of our editor. It can be in two modes, which allow manipulation of different kinds of elements of a scrollytelling. The timeline always allows zooming and scaling, elements are arranged in lanes, and the vertical direction is the scrollytelling's scroll position. In both views, elements can be ordered horizontally (by scroll position, sequentially) and vertically (by display order, the frontmost element is on the uppermost lane). Additionally, a vertical line shows the current scroll position of the scrollytelling within the timeline. This line is called the cursor.

In the global timeline, visible in Figure 3.14, which is opened by default when creating a new scrollytelling or loading one, the items are sequences, and the timeline lanes correspond with layers of the scrollytelling. Sequences can be moved within a layer or to different layers, resized, and removed. Layers can be hidden and rearranged. Context menus allow additional options such as renaming or copying

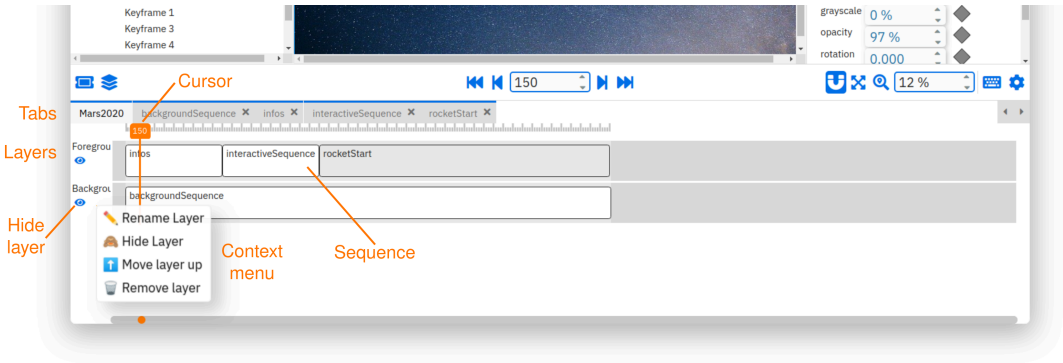


**Figure 3.14:** Global timeline in example scrollytelling

of sequences. Here we create the sequences that will make up our scrollytelling and adjust their length and layers. To fill them with content, the sequence timeline is used. Double-clicking a sequence opens a new tab and a new timeline for that sequence, a sequence timeline.

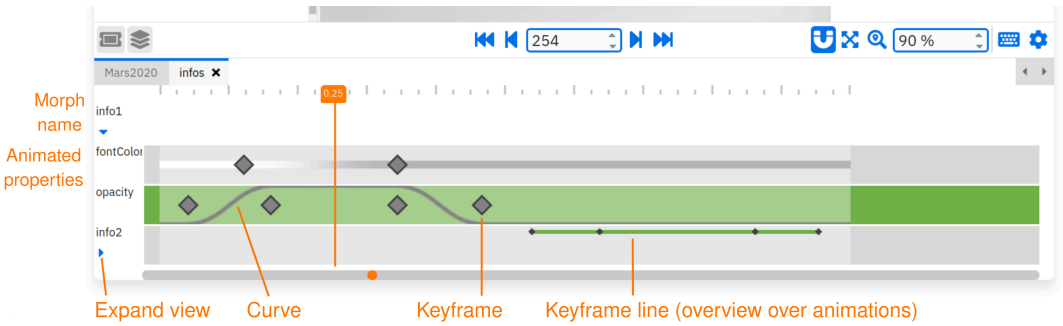


**Figure 3.15:** Sequence timeline in example scrollytelling, opened on the 'info' sequence

In the sequence timeline, shown in Figure 3.15, which corresponds with the sequence view on the scrollytelling, the items in the timeline are keyframes. There are two different kinds of lanes. Morph lanes give an overview of the animations on the morph with keyframe lines and can be expanded to show property lanes. These lanes show the individual keyframes for one animation and are labeled with the animated property. Here keyframes can be arranged or removed, as well as edited by dragging them or with the context menu. To show and differentiate animations, every property has a color assigned to it. We hope that content designers will grow accustomed to the colors and eventually identify properties very quickly. Property lanes have curves to allow users to understand the animation at a glance and compare different easings on keyframes. In sequence timelines, the cursor does not show the

absolute scroll position but rather the relative scroll position in the sequence. This makes it more in line with keyframes which also use relative values.

Timelines are organized in tabs, and every timeline is opened in a separate tab named after the scrollytelling or the sequence (for the global timeline and sequence timelines, respectively). This is intended to allow for quick navigation between timelines. Selecting a sequence in the tree also automatically opens the tab with the corresponding sequence timeline.

Most of the time for creating a scrollytelling is spent in the sequence timeline, as this is where the particularly time-consuming fine-tuning takes place.

### 3.3.2 Editor Scrollytelling Interaction

The editor is used to display the properties of a scrollytelling and react to changes. In the other direction, the editor also enables changing the properties of a scrollytelling with user input.

When a scrollytelling is loaded into the editor, connections (see subsection 3.1.3) are placed in different parts of the scrollytelling, such as a connection from the property "name" on a sequence to the name of a tab that corresponds to that sequence.

Thus morphs in the scrollytelling are already edited when it is loaded in the editor. We must make sure to remove those connections correctly when the scrollytelling is removed from the editor. Otherwise, the connections would still reference the editor, causing problems when the editor should be closed and data consistency is no longer guaranteed, and when serializing the scrollytelling. This is done by going through the entire submorph hierarchy of the editor and removing every connection from a morph in the editor that leads to a morph in the interactive once it is removed from the editor.

Most editor components are qinoq morphs. These particular types of morphs are aware of deserialization (see subsection 3.4.4) and may store a reference to the editor they are part of. The editor itself has a reference to the scrollytelling that is currently loaded. That way, every qinoq morph object also references the scrollytelling, and changes can be directly passed to the scrollytelling component when changes are triggered in the editor. Some morphs in the editor have a more direct reference to their associated scrollytelling morphs. For example, when a sequence on the timeline is moved, it will change the start property on its associated sequence in the scrollytelling.

This may lead to circles of property propagation. When a property in the editor is changed, which leads to a change in the scrollytelling, it triggers a connection or a property setter that leads to a change in the editor. We cannot use the direct circle prevention built in the connection system, as it only accounts for connections that directly trigger other connections. When a method is instead called by a connection (or a property setter) and then sets a property that would trigger a connection (or a property setter) and thus a circle, this cannot be prevented automatically. To stop these circles from fomenting, flags are placed to indicate where the change originated. These are checked before changes are applied.

### 3.3.3 Interaction with the Editor

The editor is built to be a tool for professional usage and requires some practice to be used efficiently. The editor is built in lively.next and thus follows some conventions of interaction within lively.next. The primary interaction method is the mouse, using clicking, scrolling, and dragging, such as clicking on buttons, scrolling through the inspector's property list, or dragging sequences to move them around on the timeline. On selected editor elements, context menus can be opened by right-clicking.

Some operations can be performed with the keyboard, such as moving selected sequences, selecting all sequences, or deleting selected sequences. However, most operations are not available through the keyboard. Keyboard interactions often suffer from their relative lack of discoverability. We tried mitigating this by adding a list of shortcuts that can be accessed from the menu bar.

Elements in the scrollytelling, such as morphs, do not have to be edited by the tools supplied by the editor. Instead, it can be helpful to use the integrated tooling of lively.next. Sometimes this is more convenient, such as when positioning a morph and using the halo menu rather than the inspector, but for custom behavior, it is essential to use lively.next tools such as the object editor.

While lively.next emphasizes the ability of users to change the behavior of all parts, this can confuse some users. For example, the halo menu available on all morphs allows changing of morph hierarchy or removing morphs. When this is done to some components of the editor, unexpected behavior may occur, or specific actions will no longer be available. This is why morphs that make up the GUI of the editor do not allow halo menus to be opened on them by default. If users want to change editor behavior, setting the `debug` property of the editor to `true` enables halo menus again.

Since content designers are expected to start the creation of scrollytellings, they can start them by themselves. Opening the editor can be achieved with code, which is not accessible for non-developers. A solution to this problem lies in the `localconfig.js` file located in lively.next's root directory, to which code can be added that is executed when a world is loaded. Using this file, the developer can prepare a shared lively.next server for the content designer by starting the editor on the first loading of a world. A possible configuration can be found in Listing B.1.

**Undo and Redo** The ability to undo and redo actions performed within the editor is beneficial for increasing the speed at which scrollytellings can be created and edited, allowing users to fix mistakes quickly.

Within lively.next undo and redo operations are supported for morphs. The lively.next undo manager[36] can record at most one undo operation at a time. An undo operation consists of one or more target morphs and a changeset of the properties on the target morphs that changed during the undo operation. When the operation is undone, the changeset is applied in reverse. This system does not support the

[36] https://github.com/LivelyKernel/lively.next/blob/master/lively.morphic/undo.js (last accessed on 2021-07-28).

creation or removal of morphs, as only existing morph property changes may be recorded.

This makes it impossible to capture all operations that are possible within the editor with undo operations. Only certain operations are covered, which only affect morph properties such as moving or resizing of sequences. As the undo operations can only be recorded on morphs, operations in our editor that work on properties of other objects, such as changing a keyframe's easing function, cannot be recorded.

As shown, the lively.next undo implementation is not ideal. The usability of our editor would benefit if we were to use a separate undo manager with undo operations based on the command pattern and not only property changes. Alternatively, the lively.next undo manager could be updated to solve the outlined problems.

### 3.3.4 Working with Scrollytellings beyond the Editor

Since all elements of a scrollytelling are morphs, they can be accessed natively within lively.next. This allows inspection of morphic properties and changing of the behavior of morphs or sequences by creating a new class and assigning a selected morph to that class, which can be achieved with the object editor within lively.next.

Developers can edit the behavior of morphs within a scrollytelling such as implementing simple state machines or games, which the editor does not directly support. Developers have large control over the generic morphs of a scrollytelling, thus providing flexibility.

Hooks are implemented to simplify writing custom behavior in morphs, namely `onInteractiveScrollChange(scrollPosition)`, `onSequenceEnter()` and `onSequenceLeave()`. The sequence calls these on their submorphs.

Developers can also implement custom animations. These can be scroll-based and be supported by the editor through properties (with custom properties supported through the `animateAs:`) setting or be time-based, which makes them inaccessible to edit for content designers.

In our example, we only want one interactive component, a slider, to see what payload requires how much fuel at a glance. This means we can build a slider morph, open it in the object editor, and subclass it to specify its custom behavior. The slider can then interact with the other morphs in the sequence via names and the submorph hierarchy of its owner, the sequence.

## 3.4 Serialization and Deserialization

Serialization and deserialization are essential for multiple aspects. Serialization of morphs allows copying of morphs, which can be used to speed up the scrollytelling creation. Morphs need to be serialized to be bundled, the process to create stand-alone web pages used for the distribution of scrollytellings. Saving worlds and storing them on the lively.next server is not only essential for backups and picking up work later but also for collaboration between content designers and developers within the same world.

Most objects in lively.next may be serialized.[37] This serialization captures the properties of the object in a serialized format, an object that may be stored in a JSON file. Every object that is to be serialized is assigned a temporary ID. The serialization contains the ID of the object that was serialized and an object table, which is an array of all objects that have been serialized. Since objects may reference other objects, references to objects are stored by saving the object's ID. This prevents circular dependencies.

This serialization procedure[38] has its origin in Lively Kernel[38], a direct predecessor of lively.next.

Given an object *a* defined in such a way:

```
let a = { b: {c: 1}, d: 2}
```

The serialization of the object *a* may then look like Listing 3.3.

**Listing 3.3:** Snapshot of *a* as a JSON object

```
{
  "id": "DCB9E734-2B73-417B-9B34-67E2E993CDD4",
  "snapshot": {
    "DCB9E734-2B73-417B-9B34-67E2E993CDD4": {
      "rev": 0,
      "props": {
        "b": {
          "value": {
            "__ref__": true,
            "id": "D1285944-25DB-49C8-821E-951C6F3572B3",
            "rev": 0
          }
        },
        "d": { "value": 2 }
      }
    },
    "D1285944-25DB-49C8-821E-951C6F3572B3": {
      "rev": 0,
      "props": {
        "c": { "value": 1 }
      }
    }
  },
  "requiredVersion": ">=0.1"
}
```

Serialization of morphs works similarly, with only morphic properties being serialized, not arbitrary properties, facilitated through the `__only_serialize__ ()` method. Only properties that deviate from default values are serialized. Furthermore,

[37] Functions cannot be serialized because of their closures, which is a problem we will face in subsection 3.4.4.

[38] https://github.com/LivelyKernel/lively.next/tree/master/lively.serializer2 (last accessed on 2021-07-28).

morphs have the callback method `__additionally_serialize__ (snapshot, ref, pool, addFn)` to add additional data to snapshots.

Serialization of morphs is used when saving, bundling, and copying morphs and can be manually triggered with the `serialize(obj, options)` function.

### 3.4.1 Morph Deserialization

When morphs are deserialized, a new morph of that type is created with the default properties. Afterward, the properties from the snapshot are applied to the morph. While doing this, setters for properties are run. Additional behavior on deserialization may be specified with the `__after_deserialize__ (snapshot, ref, pool)` callback method.

Deserialization of non-morphs is more complex, especially for morphs from external modules. That is why the usage of objects from external modules should be coupled with morphs that save the possible state of these objects and then restore that state with the `__after_deserialize__ (snapshot, ref, pool)` call.

### 3.4.2 World Serialization

To save the lively.next world, it can be serialized like any other morph. After serializing the world successfully, it is then again deserialized to ensure that it can be loaded later. Only when both checks are successful is the world saved.

World snapshots are saved as JSON files on the server. Some meta information is saved with a world, allowing for different versions of the same world to be saved on the server.

Loading a world deserializes the snapshot, first recreating the morphs from the object graph. Afterward, the world morph is selected and rendered, causing all submorphs to be rendered with it.

While synchronous collaboration on the same server in the same world is not possible, it is possible to share the URL of a world on a server, which is structured like this:
**http://www.somelivelyserver.com:9011/worlds/load?name=ALivelyWorld**

This then allows asynchronous collaboration on the same world, which is the envisioned collaboration process between developers and content designers for work on a scrollytelling.

### 3.4.3 Scrollytelling Serialization

Since scrollytellings are a hierarchical collection of morphs, serialization is already supported out of the box. Some considerations have to be taken, however. The scrollytelling must not have references to the editor it was created in, lest the snapshot would also contain the editor, which is harder to serialize. This also includes connections, which are removed as described in subsection 3.3.2.

Ensuring that the editor would not be included in the snapshot was a problem that occurred numerous times during our development, which sometimes prevented saving.

### 3.4.4 Editor Serialization

Saving the editor was a much more complex challenge. While serialization did work, deserialization often faced problems because of the way we designed the editor. While it is possible to work productively without saving the editor, grabbing the scrollytelling out of the holder, closing the editor, and then saving, this is not ideal.

One problem with deserialization is that the lively.next deserialization mechanic cannot guarantee *when* any specific morph is deserialized. This runs into problems when methods are called that assume a correct morph hierarchy. When deserializing, setters are called that affect other morphs. For example, when a timeline sequence, a morph representing a sequence on the timeline, is deserialized, its extent is set. Setting the extent triggers the setter, which changes the length of the represented sequence, a mechanic usually used to resize sequences in the timeline. This calculation requires the timeline as well, as the timeline may be zoomed, thus leading to a different length. This would then lead to an error, as the timeline may not yet be deserialized.

The solution to this problem is deserialization-aware morphs.

**Qinoq Morphs for Serialization** Qinoq morphs are a subclass of morphs that are aware of deserialization. This means the methods seen in Listing 3.4 is implemented:

**Listing 3.4:** Methods implemented in qinoq morphs, making them deserialization aware

```
__deserialize__ (snapshot, objRef, serializedMap, pool) {
    this._deserializing = true;
    super.__deserialize__(snapshot, objRef, serializedMap, pool);
}

__after_deserialize__ (snapshot, ref, pool) {
    delete this._deserializing;
    super.__after_deserialize__(snapshot, ref, pool);
}
```

Qinoq morphs have a flag set whenever they are in the process of deserialization.

Qinoq morphs were used frequently within the editor and are the superclass of almost all custom morphs in the editor.

With the usage of qinoq morphs, setters for qinoq morph simply check if they are currently in the process of deserialization. If so, they only set the property and do not do other actions that may involve other morphs that may not have been deserialized yet. So a simplified extent property for a timeline sequence would look like Listing 3.5

**Listing 3.5:** Simplified extent property on timeline sequence

```
extent: {
    set (extent) {
        this.setProperty('extent', extent);
        if (!this._deserializing) {
            this.updateSequenceAfterArrangement(); // This method
                changes the sequence
        }
    }
},
```

**Qinoq Buttons** In our editor, buttons are often used as user interface elements. The buttons' implementation was initially usually done with label morphs, which had functions defined by the arrow function expression assigned to a callback method such as `onMouseUp`.

```
settingsButton.onMouseUp = () => this.openSettingsMenu();
```

This ran into problems when serializing the editor since these functions cannot be serialized.

We made some inquiries into replacing them with lively.next closures. These are objects that store the function as a string and a variable mapping object that is applied when the function is executed. This allows serialization, but the closures are harder to maintain as they are stored as strings, where problems may only be encountered at runtime.

We opted for a custom class of buttons, called qinoq button, that allowed us great flexibility with uniform styling and solved the serialization issue. A qinoq button has properties for target, command, or action. Commands are another lively.next feature: Morphs may define a method `get commands()`, which returns an array of command objects, each identified by a name and including an exec parameter that is a function. With commands, the button's functionality is not on the button but at the target of the button, which can be seen in Listing 3.6. The action property on qinoq buttons allows further flexibility when commands are not used.

**Listing 3.6:** onMouseUp method as implemented on qinoq buttons

```
onMouseUp () {
    this.command ? this.target.execCommand(this.command) : this.
        target[this.action]();
}
```

### 3.4.5 Bundling

Creating scrollytellings with the editor would not be very useful if they could not be experienced by consumers. Fortunately, lively.next can be used to create web pages. To do this, we use a process called *bundling*. Bundling is not a lively.next specific term, as it usually used to mean combining multiple JavaScript source

files into one[14](pp. 1104-1107) while minifying and optimizing it. This increases performance and reduces loading times, and results in fewer HTTP requests. The terms bundling and freezing are used interchangeably within lively.next. First, a morph is selected with the object editor, where a button for freezing is available. The dialogue shown in Figure 3.16 is then opened.

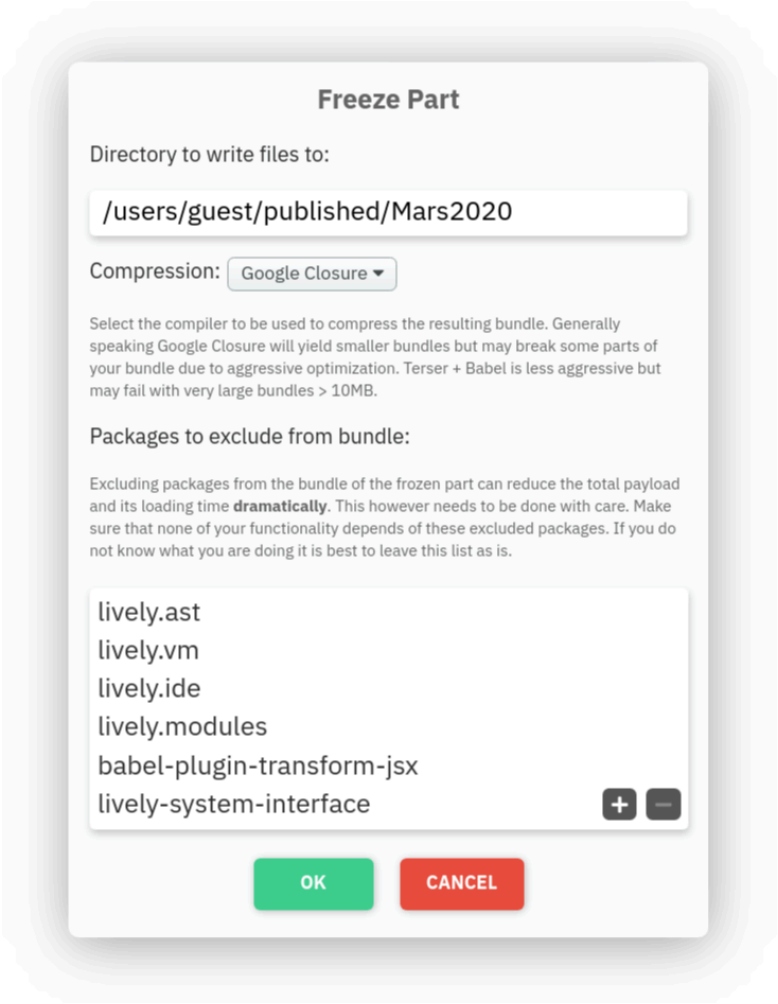


**Figure 3.16:** Freeze part dialogue used for bundling opened on example scrollytelling.

After selecting a target directory, the morph is subsequently serialized into a snapshot. All classes and files used for the morph, as well as dependencies, are bundled together into a single file, which is then minimized and optimized. The selected directory will then contain an `index.html` file, which references a `load.js` file. This file contains the snapshot, the deserializer, the morphic renderer, and any

other classes used by the snapshot. It is run through babel[39] to enable older browsers to display it correctly. When modules are loaded asynchronously, the file can be split up into separate files to reduce their size. Assets are stored in subdirectories. The resulting web page can be opened with the `index.html` file. In it, lively.next tooling such as the top bar or morphic halo menus is no longer available. This makes the result suitable for delivery to consumers.

While bundling works for simple morphs, the compiler can be very opinionated about what is accepted. We used the Google Closure Compiler.[40] For example, it did not accept regular expressions in the literal notation. They had to be converted to the constructor form.[41]

**Dependencies and Modules** The module system of lively.next, while compatible with ECMAScript 6 (ES6), is based on SystemJS.[42] This allows the module system to also work with older browsers that do not support ES6.[43]

To make bundling of the scrollytelling possible, we had to make sure to only include necessary dependencies in the modules that make up the scrollytelling. The scrollytelling modules have no reference to the editor. Dependencies that are not necessarily needed in a bundled scrollytelling are not loaded with the ES6 module mechanic but asynchronously when needed with the `System.import(filepath)` function provided by SystemJS.

## 3.5 Summary

The environment of lively.next and the morphic graphic system offer tooling for developers and non-developers to combine morphs to create visual elements and enrich them with behavior. Through bundling, which creates websites from morphs, lively.next is a suitable environment for creating scrollytellings. Creating scrollytellings with a fixed structure allows us to develop a stable interface for them while retaining flexibility through the many possibilities of morphs. This in turn makes it possible to create an editor for these scrollytellings in the same environment. The editor allows content designers to create and alter scrollytellings, while the scrollytelling structure enables scrollytellings to be edited beyond it. Thus it plays to the strengths of content designers and developers. We have seen technical challenges for the editor, especially regarding serialization. These challenges could however be solved, so the editor is usable. Special considerations were taken for animations, which is the subject of chapter 4. We test the actual usability of the editor in chapter 5.

[39] https://babeljs.io/ (last accessed on 2021-07-28).
[40] https://github.com/google/closure-compiler (last accessed on 2021-07-28).
[41] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions (last accessed on 2021-07-28).
[42] https://github.com/systemjs/systemjs (last accessed on 2021-07-28).
[43] https://caniuse.com/mdn-javascript_statements_import (last accessed on 2021-07-28).

# 4 Animating Content in qinoq Scrollytellings

Our editor and the overall qinoq system allow for the creation of scrollytellings inside of lively.next. A major part of our contribution is that the editor does not require any programming skills. It enables content designers to take on large parts of the creation process of scrollytellings, as described in subsection 2.1.2, by themselves.

Creating scrollytellings in our editor does not only entail destructuring the content into separate sequences based on a storyboard but also animating content depending on the current scroll position in the scrollytelling. Previously, those tasks were not feasible for content designers alone but needed to be done by or in collaboration with developers.

Scrollytellings, as a form of interactive content on the web, utilize animations for various purposes, as described in section 1.3. For example, animations in scrollytellings can:

- indicate possible interactions to users,
- create a rich atmosphere for users,
- depict parts of the entailed story.

Hence, animations are of high importance in scrollytellings. We will take a closer look at them throughout this chapter.

We begin by describing the animation implementation of qinoq in-depth.

We will consider how externally created Lottie animations by graphic designers can be integrated, thus honoring the multi-stakeholder creation process of scrollytellings.

Afterward, we briefly examine the mechanism necessary to render content in lively.next and thus qinoq animations.

We will then look at a possible performance optimization for animations in qinoq. With the rise of increasingly capable graphics systems in modern digital devices, animations have become an integral part of many kinds of software applications and a staple in modern web design [59]. Thus, there are efforts to optimize the performance of animated content on the web. One result of these efforts is the Web Animations API. We will consider how browsers render content and how the Web Animations API optimizes the rendering of animated content. We will present a proof-of-concept integration of the Web Animation API in qinoq and evaluate its advantages and disadvantages.

## 4.1 Animations

The term *animation* entails different visual phenomenons and will trigger different associations for different people. In its original meaning, the word describes the usage of multiple static pictures that, when displayed consecutively in a fast fashion, create the impression of movement [2]. These pictures exist as separate images first and are only later used as building blocks of a dynamic animation [43].

In the context of the following chapter, we understand animations to be any visual effect that results in observable changes on displayed elements that occur over some duration [62]. Therefore, examples for animations in the context of scrollytellings are:

- Snowflakes falling in the background of a scrollytelling to create an atmosphere.
- A new element appearing from the screen's side, moving over parts of the screen and finally being positioned in the middle of the screen.
- A button having a pulsing, colored border to indicate the possibility of interaction to consumers.

A new element appearing instantly, for example, because it was recently added to the currently displayed sequence of a scrollytelling (see subsection 3.2.1), will not be considered an animation.

## 4.2 Keyframe Animations

The animation implementation of qinoq is based on the keyframe technique. Before taking a detailed look at the implementation, we will introduce the underlying idea of keyframe animations. We begin with the base case of linear animations. Afterward, we will consider how to improve those animations using easing functions.

### 4.2.1 Linear Keyframe Based Animations

Keyframe animations originate from animation films in which leading artists define and draw some crucial moments of the film [33]. Those frames, acting as the framework for the complete animations, are called *keyframes*. They depict key moments of the film or scene at hand. Based on these keyframes, other artists can then fill in the remaining frames between them. Since these appear between keyframes, they are also known as *tweens* [50]. This animation technique is used today in game engines, for example Godot,[44] 3D creation software like Blender[45] and design tools such as Adobe After Effects.[46]

[44]https://godotengine.org/ (last accessed on 2021-07-28).
[45]https://www.blender.org/ (last accessed on 2021-07-28).
[46]https://www.adobe.com/products/aftereffects.html (last accessed on 2021-07-28).

Web Technologies like CSS Animations and the JavaScript Web Animations API also use this approach.

Digital keyframe animations function by defining distinct moments at which an animated property or object should have a specific value or state.[47]

The scale on which those moments are defined is dependent on the context of the animation. For animations defined in terms of time, such a moment could for example be 0.5 seconds after the start of an animation. For animations progressing with a changing scroll position on a site, such as the scrollytelling animations covered in this chapter, these moments are defined by a specific scroll position or progress throughout the page.

We call the unit in which the progress of an animation is defined the *driver* of that animation. We will call the moment associated with a keyframe in accordance to the driver of its corresponding animation the *position* of the keyframe.

Defining multiple keyframes at different positions animates content by interpolating the value of the animated property between the values of the defined keyframes as the driver of the animation progresses between the positions of the keyframes. Therefore, one needs to specify at least two keyframes for an animation to be well defined[48].

Figure 4.1 shows an example of how the vertical position of a circle can be animated with two keyframes over three seconds.

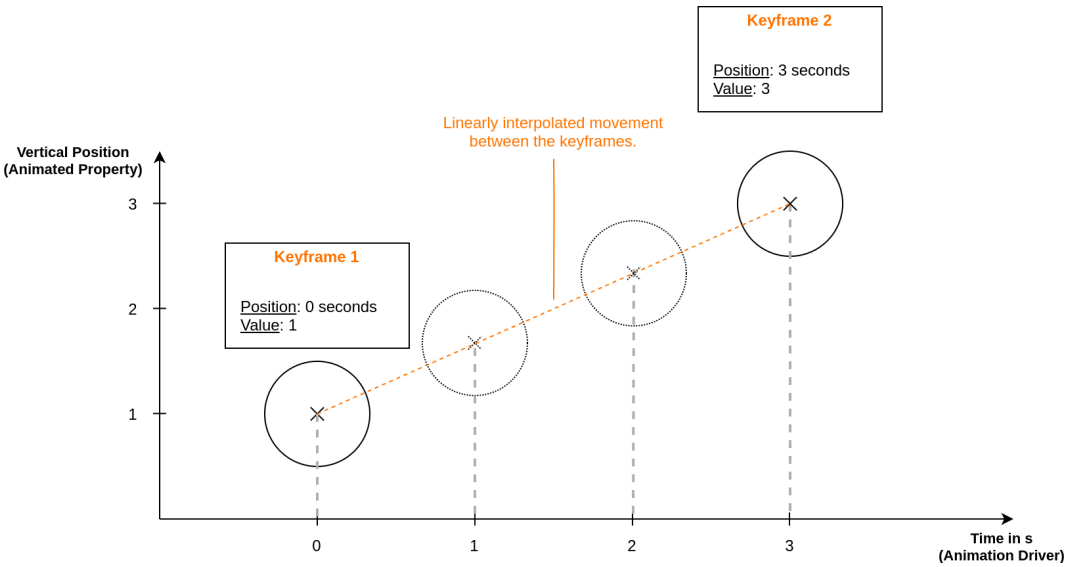


**Figure 4.1:** The vertical position of a circle being animated with two keyframes over a three-second duration

[47] Exemplary: https://www.adobe.com/creativecloud/video/discover/keyframing.html (last accessed on 2021-07-28).

[48] Some tools also allow instantaneous changes of values at the position of a keyframe. In those cases, a single keyframe suffices for an effect. However, instantaneous changes are explicitly not covered by the term animation as it is used in this chapter.

As can be seen in Figure 4.1, the interpolation between the keyframes is done via a linear function. Therefore, the animated property in all calculated tweens will lie on a straight line between the values given by the two defining keyframes. The animated property changes at the same rate as the driver of the animation.

A constant velocity of the animation driver will always lead to a constant rate of change of the animated property.

### 4.2.2 Easing Functions

The velocity at which objects change their properties is rarely linear in the real world. Instead, the velocity of change on an object increases and decreases. For this reason, animations with a constant velocity often look unnatural [6, 50]. To mitigate this and create natural-looking digital animations, *easing functions*, or *easings*, are used.

That natural change is mostly non-linear is best illustrated with a moving object in the real world. Because of its inertia, an object cannot immediately reach its final velocity when moving. Instead, it will increase in velocity when first starting to move. An easing that achieves this effect is known as *ease-in*. At the end of the object's movement, it will also not stop instantly but decrease in velocity until it reaches a velocity of zero. This is also known as *ease out*.

Formally, an easing function is a mathematical function used as an indirection between the animation driver changing and the linear interpolation between the two keyframe's values. It is usually defined on the interval $[0, 1]$ and used with the current progress throughout the animation as the input. Its output is again a value on the same interval $[0, 1]$.

Using $[0, 1]$ as domain and range allows to abstract from the concrete duration of a given animation. It also allows to implement a comprehensive set of functions computationally efficient [13].

Calculating the animated property at a given moment in an animation with an applied easing is done via functional composition [50]. In the above-described base case, the value of the animated property was directly dependent on the animation driver's value. With easing functions, the animated property depends on the output of the easing function, which directly depends on the current value of the animation driver.

Figure 4.2 depicts the operating principle of easing functions graphically.

An easing (A) is applied to the animation defined by the keyframes (B) and (C). At the midpoint of the animation, its progress (1) is the input for the easing function. The functions output value (2) then becomes the input for the linear interpolation (3) calculating the value of the animated property (4).

In the default case described in the previous subsection, the velocity at which the property changed its value was constant. After 50% of the animation duration had lapsed, the animated property held exactly the value of $value\ of\ Keyframe1 + (value\ of\ Keyframe2 - value\ of\ Keyframe1) * 0.5$. With the exemplary easing function considered in Figure 4.2, it is clear that the velocity with which the animated property changes is not constant.

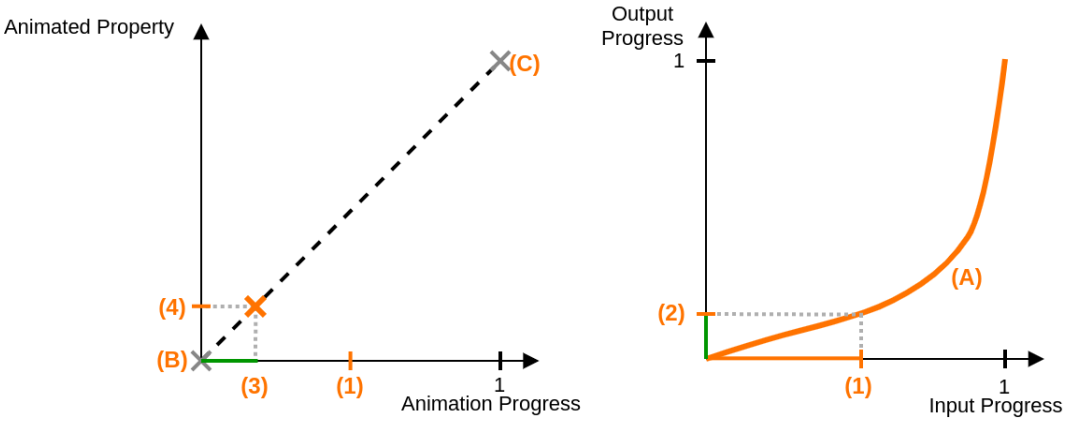


**Figure 4.2:** The operating principle of easing functions: The graph of an easing function (right) and how it is applied to an animation (left)

Using easing functions, even if the driver of an animation changes at a constant velocity, accelerations, and decelerations can be achieved in digital animations resulting in a more natural look[49].

Figure 4.3 combines our understanding of keyframe-based animations and easings, showing how the animation depicted in Figure 4.1 looks like with an ease-in applied.

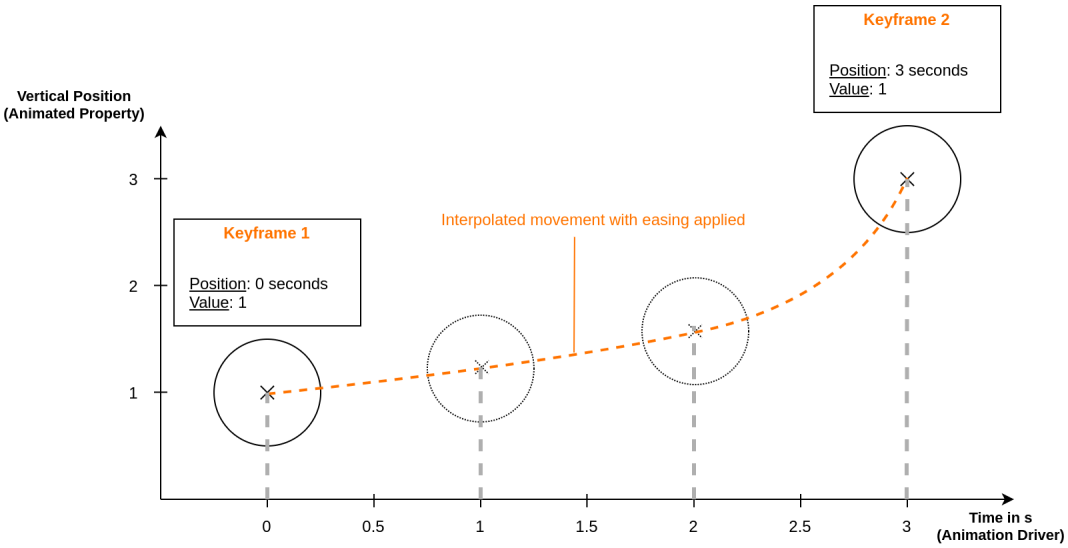


**Figure 4.3:** The vertical position of a circle animated with two keyframes over a three-second duration with ease-in applied

[49] easings.net (https://easings.net/ last accessed on 2021-07-28) provides interactive demonstrations of Penner's easing functions in which their effect on the velocity of change can be explored.

While one common set of easing functions is widespread[50], any arbitrary mathematical function can be used as an easing function. By using non-monotone easing functions effects such as bouncing can be achieved.

One popular approach to define easing functions is using cubic bezier curves [29]. They allow for a wide variety of functions to be defined [27]. Tools like Blender, Adobe After Effects, and Web Technologies like CSS all allow defining easing functions this way. Often, graphical tooling to manipulate the control points exists[51]. Placing some restrictions on the placement of the control points can ensure that the set of all possible bezier curves only contains mathematical functions [27]. Restricting them to cubic bezier curves with fixed end-points in $P(0,0)$ and $P(1,1)$ still allows to approximate the original Penner functions reasonably well [28]. With only the control points changeable, they are also often easier to manipulate with graphical tools.

## 4.3 qinoq's Animation Implementation

As seen in chapter 1, scrollytellings are websites where the main consumer interaction happens through scrolling.

As opposed to traditional, mainly text-based, and static web pages, scrolling on scrollytelling pages does not simply shift the viewport of the page but allows consumers to interact with the page in a more involved way, as described in section 1.3. Movement and thus animated content are a necessity for scrollytellings. Since the exact timing of animations heavily influences how consumers perceive a scrollytelling [61, 33] and therefore decides if the desired immersion in the story succeeds, qinoq and its editor need to allow for the fine-granular creation and manipulation of animations. This also becomes clear in the design goals of our editor as defined in section 2.5: Allow for the animation of morphic properties, in a graphical way (meaning without the necessity to write code) that is fast to change and provides immediate feedback in order to allow for rapid iterating.

As we have seen in subsection 3.2.1, qinoq implements these goals by defining keyframe-based animations on morphic properties. The editor acts as a graphical frontend to manipulate these animations and other parts of qinoq scrollytellings. Although our implementation could be extended to allow for time-driven animations, how to semantically include them in our model of scrollytellings and incorporate them in the editor is non-trivial. Particularly, since the editor's interface needs to stay tidy. We will not cover these questions in this chapter and focus solely on animations driven by scroll position.

[50] First proposed by Robert Penner [50], these functions are also known as Penner's easing functions. Their function graphs are depicted in Figure C.1.

[51] Examples can bee seen at https://docs.blender.org/manual/en/latest/editors/graph_editor/fcurves/introduction.html (Blender), https://developer.mozilla.org/en-US/docs/Web/CSS/easing-function (CSS), and https://helpx.adobe.com/after-effects/using/speed.html (After Effects), all last accessed on 2021-07-28.

Keyframe-based animations are well suited to reach our stated design goals, as they are well-suited to a workflow based on a graphical user interface [58].

In the following section, we will take a look at the underlying implementation of animations in qinoq. They consist of the following fundamental pieces:

1. A target morph with an animated property.
2. A property that the animation changes on this target.
3. A collection of keyframes. The minimum number of keyframes that are necessary for a working animation is two, as discussed in section 4.2.

Keyframes and the animations they belong to are implemented as separate classes as part of qinoq. `Animation` objects hold a collection of `Keyframe` objects. We will consider the concerns of both of these classes. Afterward, we will see how to combine them to achieve animated content in qinoq scrollytellings.

### 4.3.1 Keyframes

Keyframe objects hold a position and a value. The animated property will adopt the value at the keyframe's position during the scrollytelling. Positions of keyframes are specified as the *progress* relative to the sequence that contains the animation. A position of 0 refers to the first scroll position of a sequence. In contrast, a position of 1 refers to the last scroll position at which a given sequence is visible in the scrollytelling. Positioning animations relative to their sequence allows changing already created animations proportionally when changing the extent of the containing sequence later. Hence, if content designers already fine-tuned animations in a sequence and later decide that the particular sequence needs to be longer or shorter, the relative positions of the keyframes in the sequence and thus the timing of the animations they defined stays consistent.

Keyframes also store the name of the easing function to be applied when animating towards this keyframe.

### 4.3.2 Animations

An animation object holds a collection of keyframes, its target morph, and the morphic property it animates.

Since an animation object contains a collection of keyframes, we have to differentiate going forward: *Animation* can mean an actual `Animation` object or a *visual animation*. A visual animation is a gradual, visual change to an object caused by a value difference of two consecutive keyframes targeting the same property. Since qinoq groups all keyframes targeting the same property on a given morph inside the same animation object, this single object can result in multiple visual animations.

Because keyframe animations function by defining the value a property should hold at specific positions and then interpolating between those values, the data type of a property is essential for animating it via keyframes. For each animation type (that means for each data type that animated properties can have), there exists a respective subclass of `Animation` in qinoq that implements the appropriate interpolation.

Table 4.1 shows the currently supported types as well as some exemplary morphic properties that have this type. Morphic's properties have been described more in-depth in subsection 3.1.1.

| Data type | Exemplary morphic properties |
|---|---|
| Number | opacity/scale/... |
| Point | position/extent/... |
| Color | fill |
| String | textString on Labels |

**Table 4.1:** Supported Data types to animate with qinoq and exemplary morphic properties of that type. Number and String are JavaScript types while Color and Point are provided by lively.next.

Our animation implementation linearly interpolates between the values in the keyframe pair for points, colors, and numbers. The interpolation is calculated separately per dimension for points and colors, which consist of multiple scalar values.

Users of qinoq can animate strings with a *Typewriter Animation*. This animation type interpolates between two strings, where one is a real prefix of the other. During such an animation, the letters missing to create the other string from the prefix are progressively added.

As depicted in Figure 3.12, our editor allows users to graphically create and change animations on a set of predefined properties. We provide an interface to expand the set of animatable properties. The interface allows animating arbitrary properties on morphs, as long as the property has one of the data types specified above. Figure 4.4 shows the necessary code to define such animatable custom properties in morphic (A) and how the inspector of qinoq's editor reflects them (B).

As we have seen in subsection 3.1.1, morphs are the visual building blocks of lively.next. Therefore, our animation implementation allows animating every visual object inside of scrollytellings by targeting morphs.

### 4.3.3 Easings

Easings (see subsection 4.2.2) can be applied to all animations. Easings are stored in keyframes. For each visual animation defined by an animation object and its keyframes, the easing stored in the second keyframe of a pair is applied when animating between this pair of keyframes. This allows using a separate easing for each visual animation.

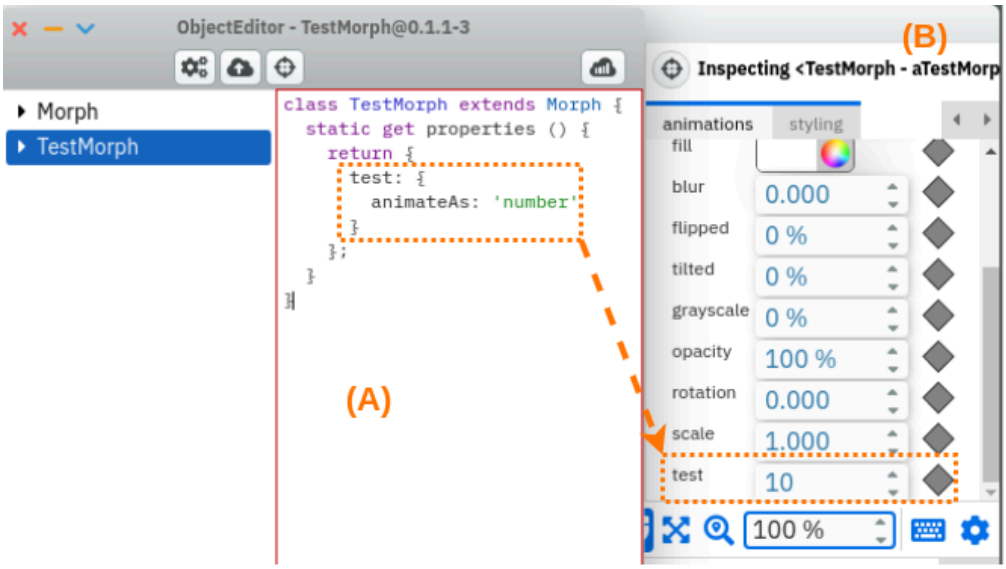


**Figure 4.4:** Animatable custom property defined on a subclass of `Morph`.

Our implementation comes with a predefined set of 25 easings implemented in lively.next[52] based on code from the Bourbon library.[53] They are defined as cubic bezier curves. Developers can specify new easing functions in the same way. Users of our editor can change the easing applied to an animation but currently cannot define new easing functions themselves.

By default, all keyframes in qinoq have an *inOutSine*[54] easing set to facilitate animations with a natural feel. It accelerates and decelerates an animation at its beginning and end loosely based on the sinus function.

### 4.3.4 Defining Animations Programmatically

Users of qinoq can create, change, and delete the keyframes that define visual animations and hence whole animations by using the editor. However, as lively.next allows to make changes to nearly every object at run-time, developers might want to manipulate qinoq animations programmatically.

Manipulating animations programmatically for customized experiences, for example, by changing an animation based on a previous interaction a consumer has made in a scrollytelling[55]. Limitations of the editor might also require developers to interact with animations. This is covered in more depth in subsection 5.2.2.

[52] https://github.com/LivelyKernel/lively.next/blob/48345c5b04e26b2d15cf9841e5b5ecb82c2fbea0/lively.morphic/rendering/animations.js#L12-L38 (last accessed on 2021-07-28).
[53] https://www.bourbon.io/ (last accessed on 2021-07-28).
[54] One of Penner's functions. See Figure C.1.
[55] The Typeshift scrollytelling "Über Flockenbau und Sturzflüge" (https://typeshift.io/snowflakes/ last accessed on 2021-07-28) shows a simple example for this. At the end of the scrollytelling, the falling snowflakes are the ones consumers created themselves beforehand.

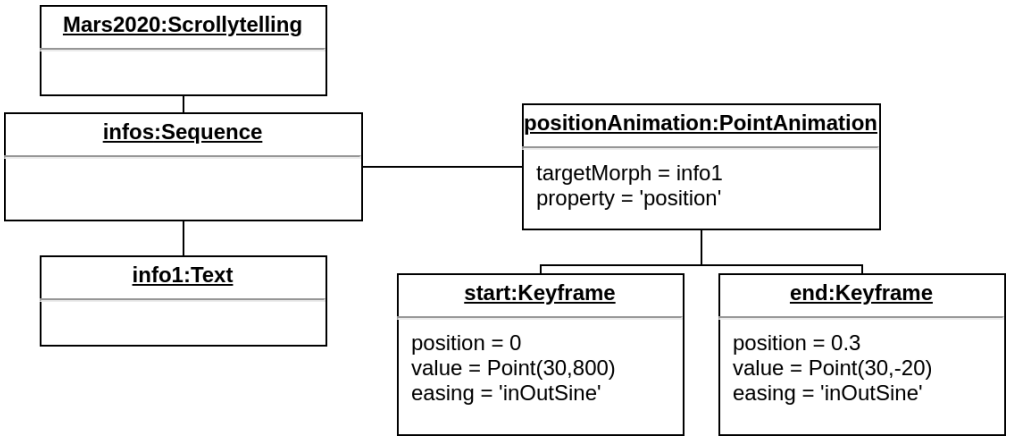


**Figure 4.5:** Object Diagram of an exemplary qinoq animation

Developers can use the same functions the editor utilizes internally to create and manipulate animations. Listing 4.1 shows how an animation that changes the position of a text morph named *info1* over the first 30% of its sequence *info* can be defined programmatically.

First, the individual `Keyframe` and `PointAnimation` objects are created. They are then combined with the `addKeyframes()` method of the `Animation` class and added to the scrollytelling using the `addAnimation()` method of `Sequence`.

**Listing 4.1:** Programmatic creation of a qinoq animation.

```
1 const start = new Keyframe(0, Point(30,800), { name: 'start' });
2 const end = new Keyframe(0.3, Point(30,-20), { name: 'daylight' });
3 const positionAnimation = new PointAnimation(info1, 'position');
4 positionAnimation.addKeyframes([start, end]);
5 info.addAnimation(positionAnimation);
```

Although methods like `removeKeyframe()` for animations exist, developers can also choose to directly manipulate the relevant objects of animations. The object diagram depicted in Figure 4.5 shows the object structure of the above defined animation.

This structure is consistent for all animations created within qinoq. When it comes to later altering defined animations in the editor or displaying them in scrollytellings, there is no difference between creating the displayed object structure programmatically or through our graphical editor.

### 4.3.5 Integration of Lottie Animations in qinoq

Because graphics are an essential part of scrollytellings, as we have seen in section 1.3, they need to be of high quality. Therefore, they are often created by professional graphic designers. A frequently used tool is Adobe After Effects. In the creative

industry, the Creative Suite by Adobe has become the de-facto standard for working with graphics and film[56].

Using animations created within Adobe After Effects directly inside of web applications was not possible in the past without costly recreation by engineers. Lottie[57] is a library designed to bridge this gap.

The library is available, among other versions, as a JavaScript library to use Lottie animations natively in web applications. Animations created in After Effects can be exported in the JSON[58] format via a plugin[59] to be consumed by the Lottie library. The library comes with a player that takes JSON data and renders the animation on a specified document object model (DOM) element. Inside of lively.next, a Lottie morph acts as a morphic wrapper around Lottie animations. It is implemented with embedding Lottie animations in scrollytellings in mind.

Lottie morphs are a subclass of the HTML morph. The DOM node of the HTML Morph acts as the target for the Lottie player to render the animation.

The player provided by Lottie focuses on time-driven animations and supplies an API to `play()`, `pause()`, and `loop()` animations. However, using the `renderFrame()` method on a Lottie animations renderer also allows setting an animation to an arbitrary frame. The Lottie morph uses this possibility to couple the state of Lottie animations to the scroll position of a page. For this reason, it provides a `progress` property that will calculate which frame to render each time its value is set. Sequences and animations have the `progress` property as well, thus providing a simple, shared interface. This is beneficial for developers when adding programmatically controlled elements and animations to scrollytellings, as we will see in subsection 5.3.4.

By using a Lottie morph, simply providing JSON data that defines a Lottie animation is sufficient to insert and play this animation inside of lively.next. Animating the morph or the progress of the contained Lottie animation is possible in the same way one would define animations on other morphic properties in our editor. Switching the displayed Lottie animation is possible through a dialogue that can be opened via the Lottie morph's halo menu.

There is no way to edit a Lottie animation as it was exported from After Effects in qinoq.

Lottie animations introduce a third layer of ambiguity to the term animation. It is essential to distinguish between the *Lottie* animation, which is the sequence of pictures and their relation defined and exported in After Effects, and the *qinoq* animation controlling how these pictures are rendered in relation to the current scroll position.

[56] Illustrated by market-share statistics such as https://enlyft.com/tech/products/adobe-creative-suite (last accessed on 2021-07-28).
[57] http://airbnb.io/lottie/# (last accessed on 2021-07-28).
[58] The JavaScript Object Notation, https://www.json.org/json-en.html (last accessed on 2021-07-28).
[59] https://aescripts.com/bodymovin/ (last accessed on 2021-07-28).

### 4.3.6 Applying Defined Animations

In subsection 3.2.3, we have seen how changes to the scroll position propagate through the scrollytelling. We will now add to this understanding and describe how the animations defined by `Animation` objects and their `Keyframes` are executed as the scroll position changes. Figure 4.6 illustrates the described message flow for the application of animations.

Upon each change of the scroll position in a scrollytelling, its `redraw()` method is called. This will trigger a call to `updateProgress()` on each sequence existing in the scrollytelling. Each sequence will then propagate the newly set `progress` to all of its animations. Setting the progress of an animation will cause a calculation of the value of its target property at this moment. The target property of the targeted morph will be set to this value.

For the calculation of the target property, the two keyframes of the animation that come closest to the current progress in either direction will be found with a linear search (if they exist). If less than two keyframes are found, either no valid visual animation is defined, or the current scroll position is smaller than its first keyframe, or respectively larger than its last one. In this case, the value saved in one of the found keyframes is assumed. We refer to the closest keyframe before the current progress as the start of an animation and to the closest keyframe following the current progress as its end.

The relative position of the current progress to the positions of start and end will be calculated with the `lerp()`[60] function shown in lines 2-4 of Listing 4.2. Thus, the function calculates which portion of a visual animation has already been performed at the current scroll position.

The specific linear interpolation to calculate the new property value depends on the type of the animation ( `NumberAnimation`, `ColorAnimation`, `PointAnimation`, or `TypewriterAnimation`, as described in subsection 4.3.2). We will illustrate the fundamental mechanism with the implementation of `NumberAnimation` in Listing 4.2. The easing of the end keyframe will be applied to the relative progress of the animation (line 6). The resulting value then is applied as a factor to scale the underlying linear interpolation between the property values of the start and end keyframe as described in subsection 4.2.2.

[60] In computer graphics, *lerping* is often used as shorthand for linear interpolation [20]. As we have seen in subsection 4.2.1, this is the foundation of keyframe animations.

**Listing 4.2:** Calculating the property value of a `NumberAnimation`.

```
// implementation of the lerp function is the same for all
   subclasses of Animation
function lerp (start, end, t) {
    return (t - start.position) / (end.position - start.position);
}

// implementation of the interpolate function inside the class
   NumberAnimation
function interpolate (progress, start, end) {
    const factor = end.easing(this.lerp(start, end, progress));
    return start.value + (end.value - start.value) * factor;
}
```

With conformance to their respective data type, the implementation for the other subclasses of `Animation` are equivalent.

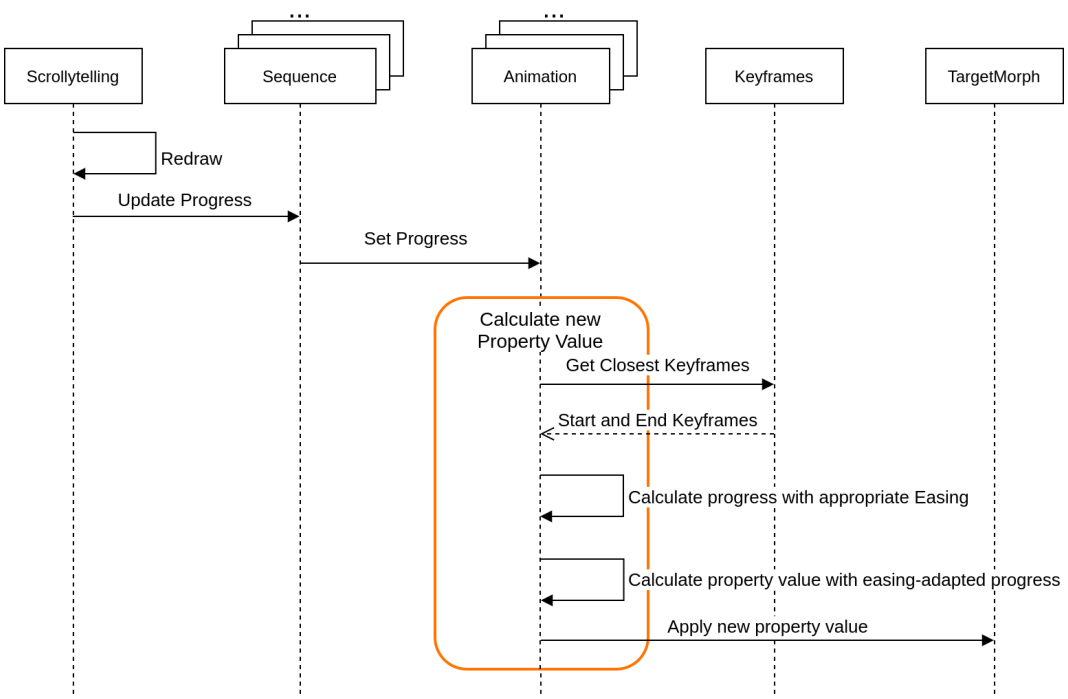


**Figure 4.6:** Data and message flow in qinoq animations: Each arrow represents one function call. All arrows enclosed in the orange outline represent calls made during execution of the same function.

**Rendering Morphs** Updating the visual presentation of a morph upon changing a morphic property relevant to its styling is performed by the *renderer* of the morphic system. Thus, this process is transparent to developers.

The morphic renderer[61] runs in a loop using the `requestAnimationFrame()`[62] method of the browser. The `requestAnimationFrame()` method allows developers to provide a callback parameter that gets executed before the next repaint operation of the browser is performed (the repaint process will be covered in more detail in the next section). DOM manipulations require a complete rerun of the browser's critical rendering path, that will be explained in more depth in subsection 4.4.1. The lively.next system utilizes a so called *virtual DOM* (VDOM) to batch DOM manipulations [1, 7]. It is based on the `virtual-dom` JS library.[63] A data-only replication of the DOM is constantly maintained and reconciled[64] with the real DOM each time the rendering loop of lively.next runs. Therefore, changing a style property on a morph will be translated into updated CSS, which propagates through the VDOM and results in a DOM update.

## 4.4 Browser-Side Performance Optimizations for Animated Content

The mechanism that qinoq uses to apply animations was written specifically for qinoq and relies on the rendering mechanisms of morphic. Subsequently, for the actual display of animation effects on web pages, it relies on the standard rendering of the DOM by browsers.

With the *Web Animations API* (WAAPI), there exist a W3C Draft Standard [4] for an API to animate content on the web using JavaScript.

In the following, we will consider the rendering process of web browsers and its meaning for the previously described animation implementation. We will then introduce the WAAPI and evaluate it in terms of performance benefits.

### 4.4.1 The Browser Rendering Process and its Performance Implications

The *rendering engine* is the part of the browser that handles the display of web pages. Since animations require fast-paced changes to what is visible on a page, the rendering engine is heavily involved in animating content. We will take a look at the general workings of such engines in order to derive possible performance optimizations for qinoq animations.

[61] https://github.com/LivelyKernel/lively.next/blob/48345c5b04e26b2d15cf9841e5b5ecb82c2fbea0/lively.morphic/rendering/renderer.js#L86-L91 (last accessed on 2021-07-28).

[62] https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (last accessed on 2021-07-28).

[63] https://github.com/Matt-Esch/virtual-dom (last accessed on 2021-07-28).

[64] https://github.com/LivelyKernel/lively.next/blob/c3d8a4e4cec4a260fdcb668dc41bba65d7997006/lively.morphic/rendering/morphic-default.js#L514-L533 (last accessed on 2021-07-28).

There are different rendering engines in widespread use today as the major browsers (Mozilla Firefox, Google Chrome, and Safari) each have their own rendering engines Gecko,[65] Blink,[66] and WebKit.[67]

We will not focus on one specific implementation, but will consider a general view on the rendering process of browsers.

**The Critical Rendering Path** The rendering process of browsers is also known as the *critical rendering path* (CRP).[68]

On a high level, the process [31, 32] is as follows:

1. After requesting and parsing the HTML, and subsequently CSS and JavaScript, code for the site, the DOM is constructed. Parsing the CSS yields the CSS object model (CSSOM).
2. DOM and CSSOM are then combined into the *render tree*, which captures all visible content. For every DOM node, CSS rules that need to be applied are attached. This step is often referred to as *style*.[69]
3. The render tree is used to determine the layout of the page to be displayed. For each node, its position and dimensions are determined. The result is the *layout tree*.
4. The layout tree is traversed to generate *paint records*. A paint record describes instructions on how to draw the elements to be displayed. It is important because the order in which elements need to be painted can be different from the order in which they are contained in the DOM and thus in the layout tree. For example, an element might overlap its parent element due to its `zIndex`. Additionally, the *layer tree* is generated, based on the layout tree. Layers contain different sets of elements that the browser will later draw together. Developers can also indicate that they want specific elements placed onto their layers, although browsers may override such decisions.
5. When the layer tree has been computed, the resulting information is sent to the *compositor thread*.

These operations are all handled by the *main thread*[70] of the browser. Thus, while executing them, the browser cannot execute other actions, such as handling user interactions or running site-specific JavaScript [32]. This also holds in the other direction: If the main thread executes JavaScript code of a web page, it cannot run the CRP until execution has finished.

Additionally, the steps of the CRP rely on each other. For example, a DOM change will trigger a rebuild of the render tree and subsequently trigger all following steps.

[65] https://developer.mozilla.org/en-US/docs/Mozilla/Gecko (last accessed on 2021-07-28).
[66] https://www.chromium.org/blink (last accessed on 2021-07-28).
[67] https://webkit.org (last accessed on 2021-07-28).
[68] https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path (last accessed on 2021-07-28).
[69] https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model (last accessed on 2021-07-28). The naming stems from the fact that for each visible elements, its styling properties get computed.
[70] https://developer.mozilla.org/en-US/docs/Glossary/Main_thread (last accessed on 2021-07-28).

**Rasterization and Composition** When the layer tree has been submitted, the compositor thread will translate the layer tree and paint records into pixels displayable on a screen. This process is called *rasterization* [32]. The compositor will rasterize each layer individually. For large layers, it might spawn multiple raster threads. The individual layers are then composited into a single *compositor view* which can be rendered on the screen by the systems graphics card (GPU).

Although initially rasterizing each layer separately and keeping them individually in-memory sounds like an overhead, browsers can perform a beneficial optimization:

If an action only changes the relation of objects to each other or shifts the viewport of the page, such as users scrolling a page, the main thread does not need to be involved in updating the displayed site. The already rasterized layers can be newly composited and a new compositor frame will be created and displayed by the GPU. The main thread does not need to be involved. Therefore, it is not necessary to wait for script execution or garbage collection to render the new frame.

**Implications of the Critical Rendering Path for qinoq Animations** Although the morphic renderer already utilizes the `requestAnimationFrame()` method, which is especially beneficial in the context of animations since it allows to schedule necessary calculations for the progress of an animation in correspondence to the browser's repainting cycle [26, 10], there is potential for further optimization for qinoq animations:

Visual changes to morphs always require a complete rerun of the CRP as at least the CSSOM will change, as seen above. Most screens today have a refresh rate of 60 Hz. Therefore, browsers try to provide 60 frames per second. Failing to do so leads to visible frame drops.[71] Assuming, for example, 6ms for the execution of the CRP, this leaves 10ms for all other calculations running on the main thread [36, 57]. In lively.next this includes the reconciliation of the VDOM and DOM. In qinoq, all computations for currently displayed visual animations also need to finish in this time frame to prevent not providing a newly rendered frame in time.

Staying in the window of 10ms is the only way to guarantee a fixed framerate for a website. Although fewer frames per second are sufficient to achieve motion perceived as smooth, higher framerates are perceived as better [19]. Additionally, an inconsistent framerate is noticeable and perceived negatively [3, 61].

The exemplary 10ms time box for main-thread activity is thus essential to achieve *jank-free*[72] animations [36, 57].

Since this problem is only amplified by the main thread frequently being occupied with other responsibilities such as garbage collection, one promising approach to improve website performance is moving load off the main thread [57].

[71] https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (last accessed on 2021-07-28).

[72] "Jank refers to sluggishness in a user interface, usually caused by executing long tasks on the main thread" according to https://developer.mozilla.org/en-US/docs/Glossary/Jank (last accessed on 2021-07-28).

**Animations Off the Main Thread** Established technologies for animating content in the web are CSS Animations[73] and CSS Transitions.[74] They allow developers to define keyframe-based animations declaratively and have broad browser support.[75] The browser natively renders these animations. The cost of animating varies by property. Most notably, animations on `transform` and `opacity` can be performed only using the compositor thread and thus without involving the main thread.[76] However, the declarative style of CSS Animations is not a good fit for animations that depend on interactions or other scripts, usually implemented using an imperative approach.

In contrast, the WAAPI allows developers to define and interact with keyframe-based animations via JavaScript. It provides an API to unify the conceptual models behind CSS Animations, CSS Transitions, and other existing animation technologies. JavaScript animations defined with the WAAPI run using the exact browser mechanisms that render CSS Animations and Transitions [52, 3].

Using the WAAPI comes with several performance benefits:

Computations that are necessary for rendering animations, such as calculating which element resides at which position during an animation, are handled directly by the browser, eliminating the need for custom scripting and allowing for native optimizations.

Most importantly, DOM elements animated with a WAAPI animation will usually get their layer, and the animation will run solely in the compositor thread whenever possible [3].

The WAAPI is supported in all recent versions of major browsers.[77] Additionally, there exists a polyfill[78] that utilizes the `requestAnimationFrame()` method.[79]

## 4.5 A Proof-of-Concept Integration of Web Animations in qinoq

Using Web Animations allows developers to utilize the compositor thread of the browser for animations and therefore shift load off the main thread. In the following, we look at a proof-of-concept implementation (POC) that implements qinoq animations using the Web Animations API. We will describe the workings

[73] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations (last accessed on 2021-07-28).

[74] https://developer.mozilla.org/en-US/docs/Web/CSS/transition (last accessed on 2021-07-28).

[75] https://caniuse.com/css-animation and https://caniuse.com/css-transitions, both last accessed on 2021-07-28.

[76] https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Animating_CSS_properties (last accessed on 2021-07-28).

[77] https://caniuse.com/web-animation (last accessed on 2021-07-28).

[78] A polyfill refers to a JavaScript library that provides functionality not natively implemented in older browsers. See https://developer.mozilla.org/en-US/docs/Glossary/Polyfill (last accessed at 2021-07-28).

[79] https://github.com/web-animations/web-animations-js (last accessed on 2021-07-28).

of the implementation, evaluate its advantages and disadvantages and consider its possible extensibility.

The POC based on Web Animations should expose the same interface to developers as the original animation implementation of qinoq so that they are easily interchangeable. The POC will support a selected set of morphic properties to be animated, namely `position`, `scale`, and `fill`, as they were frequently used in the scrollytellings we looked at during our project.

### 4.5.1 Implementation

The heart of the WAAPI is the `element.animate()` method. It creates an `Animation`[80] object and starts the animation. The method takes an array of keyframes as well as options as parameters to define animations. Most of these options are so-called timing options since the WAAPI currently only supports time-based animations natively. Examples include the total duration of an animation and how often it loops.[81]

Therefore, the implemented wrapper primarily needs to achieve two goals:

1. Translating qinoq keyframes into ones consumable by the WAAPI. This requires a mapping from the morphic properties qinoq uses to CSS attributes that are valid in WAAPI keyframes.
2. Bridging the temporal nature of the WAAPI and qinoq's scroll position-driven animations. This requires a mapping between those two and a mechanism to modify the progress of WAAPI animations by scrolling.

We define a new class called `WebAnimation` in qinoq. It can be used interchangeably with qinoq's default `Animation` objects but uses the WAAPI instead of changing morphic properties. We have to implement three methods in order to be conformant with the qinoq API. The resulting skeleton code can be seen in Listing 4.3.

- The constructor needs to take a target morph and a property to animate as parameters (line 3).
- An `addKeyframes()` method that takes an array of qinoq keyframes as parameters (lines 12-15). For the POC, we will only cover the case where two keyframes are provided and their order is correct. Therefore, the keyframe defining the target properties value at the beginning of the animation should be the first element in the array.
- A setter for `progress` on the animation, as described in subsection 4.3.6. Changing this value should visually change the animated morph according to the defined animation. Thus, if the progress is set to 1, the value of the animated property on the targeted morph should be the one specified for the end of the animation.

[80] The WAAPI and qinoq both have their own concept of Animation objects and Keyframes objects. Although they have similarities, it is essential to distinguish between them.

[81] https://developer.mozilla.org/en-US/docs/Web/API/EffectTiming (last accessed on 2021-07-28).

**Listing 4.3:** A WAAPI wrapper conforming to the qinoq API for animations.

```
export class WebAnimation {

  constructor (targetMorph, property) {
    this.target = targetMorph;
    this.property = property;
    this.keyframes = [];
    this.webAnimation = null;
  }

  // Accepts EXACTLY two keyframes, one for the beginning and one
     for the end of the animation.
  // Provide Keyframes in the correct order.
  addKeyframes (keyframes) {
    this.keyframes = keyframes;
    this._keyframes = this.generateCSSKeyframes();
  }

  generateCSSKeyframes(){
    //  ...
  }

  set progress(progress){
    // ...
  }

}
```

In the next two segments, we will cover how the POC solves the two above-mentioned key challenges, translating the keyframes of qinoq to CSS and bridging the gap between qinoq's scroll-position-driven animations and the time-driven animations of the WAAPI.

**Translating Keyframe Definitions** The keyframes of qinoq, as described in subsection 4.3.1, consist of a position and a value that the animated property should assume when the animation driver reaches the position of the keyframe. The position is defined on the interval $[0, 1]$, equivalent to the progress of the sequence to which the target morph belongs.

The keyframe objects that the WAAPI expects are reminiscent of CSS Keyframes.[82] They are JavaScript objects that map the CSS property to change onto the value that should be assumed[83]. By default, Web Animations spaces these keyframes evenly throughout the animation.

[82] https://developer.mozilla.org/en-US/docs/Web/CSS/@keyframes (last accessed on 2021-07-28).

[83] Due to the workings of CSS, for some properties, this might not be an absolute value but rather an effect to apply. For example, the CSS rule `translate()` (https://developer.mozilla.org/en-US/docs/Web/CSS/transform-function/translate() last accessed on 2021-07-28) takes a point and uses it as the vector for the desired movement of an element.

If this is undesired, one has the option to specify the offsets of keyframes explicitly. Offsets are defined on the interval $[0, 1]$ as the percentage of the already elapsed animation, similar to qinoq's progress property. For example, the value of a keyframe with an offset of 0.3 is assumed after the animation has been 30% completed.

If the last keyframe of a Web Animation has an explicit offset different than 1, an implicit keyframe with an offset of 1 is inserted. It holds the so-called *neutral value*[84] of the animations effect. The CSS rule resulting from this keyframe is such that it does not change the element's state as it exists at the beginning of the animation when being applied. Hence, its value is called the neutral value. However, depending on the animated CSS property, applying the effect in the state the element is in with the last explicit keyframe applied might cause observable changes.

There is no comparable concept in qinoq. For a progress value that is equal or greater than the last keyframe of an animation, the value of the animated property is always equal to the one held by the last keyframe. The duration of a visual animation in qinoq is defined implicitly by the offset of two keyframe's positions. The WAAPI, on the other hand, has an explicit notion of animation duration.

Therefore, translating the keyframes from qinoq to the WAAPI requires inserting an explicit keyframe at the end of the animation if the position of the end keyframe in this animation is less than 1 (Listing C.1, lines 25-29). This leads to the definitions of an animation's duration and effect being equivalent between the WAAPI and qinoq.

Additionally, the creation of mappings from morphic properties onto CSS properties is necessary. They need to produce CSS rules according to the values specified in the qinoq keyframes (Listing C.1, lines 32-59).

**Setting the Animation Progress** The WAAPI is currently only designed for time-driven animations, although there are discussions about defining a scroll position-based timing model.[85] Therefore, we need to map scroll progress onto the time-based timeline of web animations. There are two parts to how we achieve this:

First, we immediately call the `pause()` function on the animation object[86] created by `element.animate()` (Listing C.1, line 77). The animation is created exactly once when the progress of the animation is first set (Listing C.1, lines 64-76). This happens when the sequence it belongs to gets displayed for the first time. That way, time is effectively disabled as a factor changing the animation, as we want to control the continuation of the animation through scroll progress.

The creation-time of the WAAPI animation object returned by `element.animate()` is essential. Since the WAAPI is provided to us by the browser, it has no concept of morphs and works exclusively with DOM elements. Therefore, although the morph might already exist, we can only create the animation when its DOM node is created (see subsection 4.3.6). The morphic renderer allows us to access the DOM node that

[84] https://drafts.csswg.org/web-animations-1/#neutral-value-for-composition (last accessed on 2021-07-28).

[85] https://drafts.csswg.org/web-animations-1/#time-value-section (last accessed on 2021-07-28).

[86] Since it is creates by `element.animate()`, this is an WAAPI `Animation` object.

represents a morph by using its `getNodeForMorph()`[87] method (Listing C.1, line 62). Trying to create the Web Animation before the morph is rendered would error since only rendered morphs have a corresponding DOM node.

The second part to using a scroll progress mapping is using a duration of 100[88] for all animations (Listing C.1, line 67). This would mean a duration of 100 milliseconds when playing the animation based on time. However, we can also set the time of a Web Animation. This is possible by setting the `currentTime` property[89] on an Animation object. By setting the duration of the overall animation to 100, we can set 100 distinct values for `currentTime`, which maps to the progress concept of qinoq. Each time the progress of the animation is set in qinoq (see subsection 4.3.6), we use this progress value to set the new time of the Web Animation (Listing C.1, line 79).

The complete source code of the `WebAnimation` class in qinoq can be found in Listing C.1.

### 4.5.2 Evaluation of a Web Animations Integration in qinoq

We saw how changes to morphic styling properties get translated into DOM and CSSOM changes by the morphic renderer and subsequently alter the displayed page. We also considered how the WAAPI can utilize the compositor thread of browsers to animate some properties without involving the browser's main thread and how to integrate Web Animations into qinoq. We will now test and evaluate the implemented integration. During the evaluation, our focus will be if the animation of content solely on the compositor thread is now possible in qinoq.

**Demonstration Setup** For our evaluation, we create and compare example instances of animations using both the original qinoq implementation and the new implementation based on the WAAPI.

We will create a scrollytelling with only one sequence. The sequence contains only one square morph. Three properties of the morph will be animated over different durations throughout the sequence: Its color, its extent, and its position.

The code to create both of these demos can be found in the Appendix in Listing C.2 and Listing C.3. Switching between both implementations is effortless when creating scrollytellings programmatically. Since the WAAPI implementation is conforming to the interface for Animations and Keyframes defined by qinoq and takes care of translating the keyframes, switching requires only exchanging the `Animation` constructor of qinoq against its `WebAnimation` constructor. Thus the additional work

[87] https://github.com/LivelyKernel/lively.next/blob/c3d8a4e4cec4a260fdcb668dc41bba65d7997006/lively.morphic/rendering/renderer.js#L119-L135 (last accessed on 2021-07-28).

[88] Choosing 100 is the easiest form of mapping and uses a percentage semantic. Since progress in qinoq is represented as float, a higher number could be chosen in the future to achieve animations in higher "resolution".

[89] https://developer.mozilla.org/en-US/docs/Web/API/Animation/currentTime (last accessed on 2021-07-28).

necessary for the integration of the WAAPI based animation implementation in qinoq's editor is limited.

The scrollytelling has the following content:

1. During its first half, the square's color changes from yellow to red.
2. Beginning at the midpoint of the scrollytelling, its scale increases to five times its initial size. This point is reached when the scrollytelling is 70% completed.
3. During the second half of the scrollytelling, the square gets moved 80 pixels both to the right and bottom.

**Practical Evaluation** When scrolling, the qinoq animation and the one utilizing the WAAPI behave the same throughout their respective scrollytelling[90]. Figure C.2 shows side-by-side pictures of both animations at different scroll positions in their scrollytelling.

We will now use the *Chrome Developer Tools*[91] to investigate how the browser renders both of these animations.

Using the Developers Tools, it is possible to investigate a broad set of website characteristics. The Developer Tools contain a Layers tab[92] that allows seeing

- the layers a displayed page is of,
- how these layers are arranged, and
- how many times a given layer was repainted.

To test the rendering behavior of the browser, we open two lively.next worlds that each contain one of the previously defined demonstrations. With the Layer tab of the Developers Tools open, we manually scroll through each of the scrollytellings.

Figure 4.7 shows the layer tab for the scrollytelling using qinoq animations. The scrollytelling as a whole is part of the document layer (A). Using the Rotate Mode[93] (C) we can also visually confirm this. When scrolling through the scrollytelling, the paint count of this layer is steadily increasing with each change in scroll progress. The paint count states how often a layer already was repainted by the browser during its existence [25].

Figure 4.8 shows the same setup for the scrollytelling using Web Animations. Here, the animated Morph is positioned on its own layer (A), sitting above the one containing the scrollytelling (B). It has been positioned on its own layer due to an ongoing transform animation (C). Although the paint count of this layer does not increase with each scroll change, it does increase as long as the animation changing the color of the square is running (D). However, scaling and moving the square does not increase the paint count of its layer.

Every qinoq animation requires work by the main thread to calculate the animation and for a rerun of the whole CRP. In contrast, when using Web Animations, animating

[90] This test was conducted with lively.next on commit `148ecf` in `Chromium 91.0.4472.101 Built on Ubuntu, running on Ubuntu 18.04`. The same is true for all following evaluations.

[91] https://developer.chrome.com/docs/devtools/ (last accessed on 2021-07-28).

[92] https://developer.chrome.com/docs/devtools/evaluate-performance/reference/#layers (last accessed on 2021-07-28).

[93] https://developer.chrome.com/docs/devtools/evaluate-performance/reference/ (last accessed on 2021-07-28).

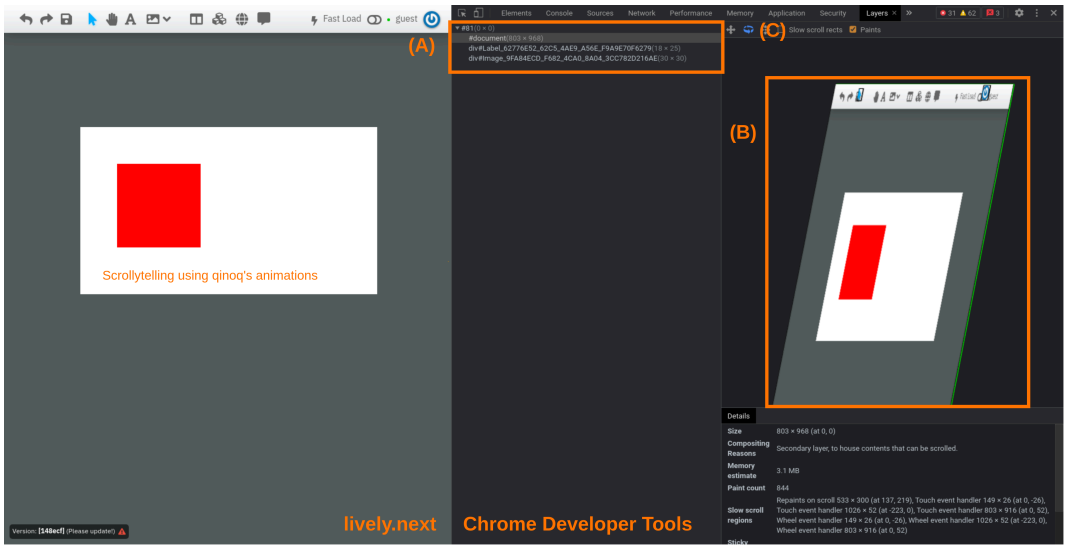


**Figure 4.7:** A scrollytelling utilizing qinoq's animation implementation in lively.next with open layer tab of the Chrome Developer Tools.

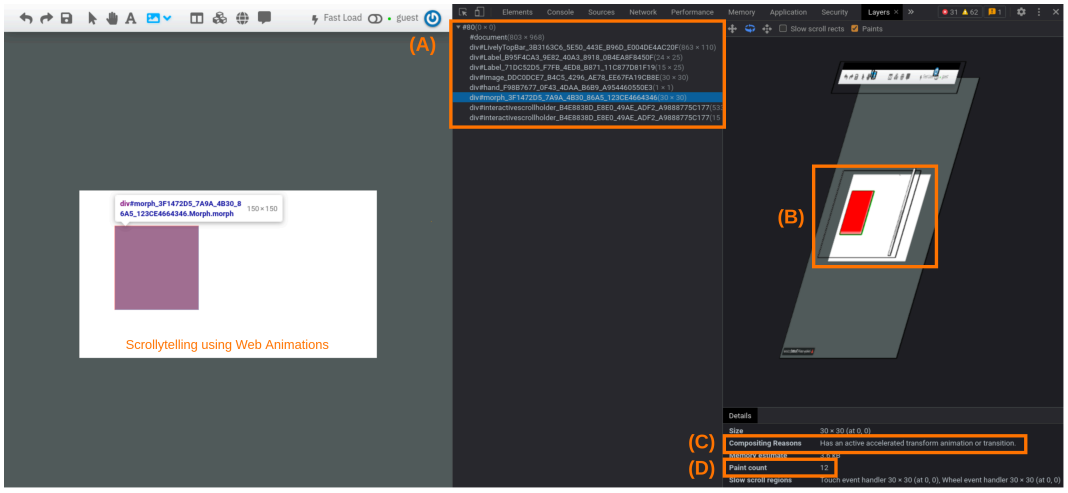


**Figure 4.8:** A scrollytelling utilizing Web Animations inside qinoq in lively.next with open layer tab of the Chrome Developer Tools.

the scale and position of the square can be handled entirely by the compositor thread. The different rendering paths are also depicted in Figure 4.9.

Animating the color still requires work by the main thread, indicated by the increasing paint count due to a (partial) rerun of the CRP.

**Benefits and Limitations** As we have seen, using the WAAPI in qinoq allows us to delegate the computations necessary for animating content to native browser APIs. The presented WAAPI wrapper does not contain code to calculate new property values, as opposed to the original qinoq animation implementation since the browser automatically takes care of this.

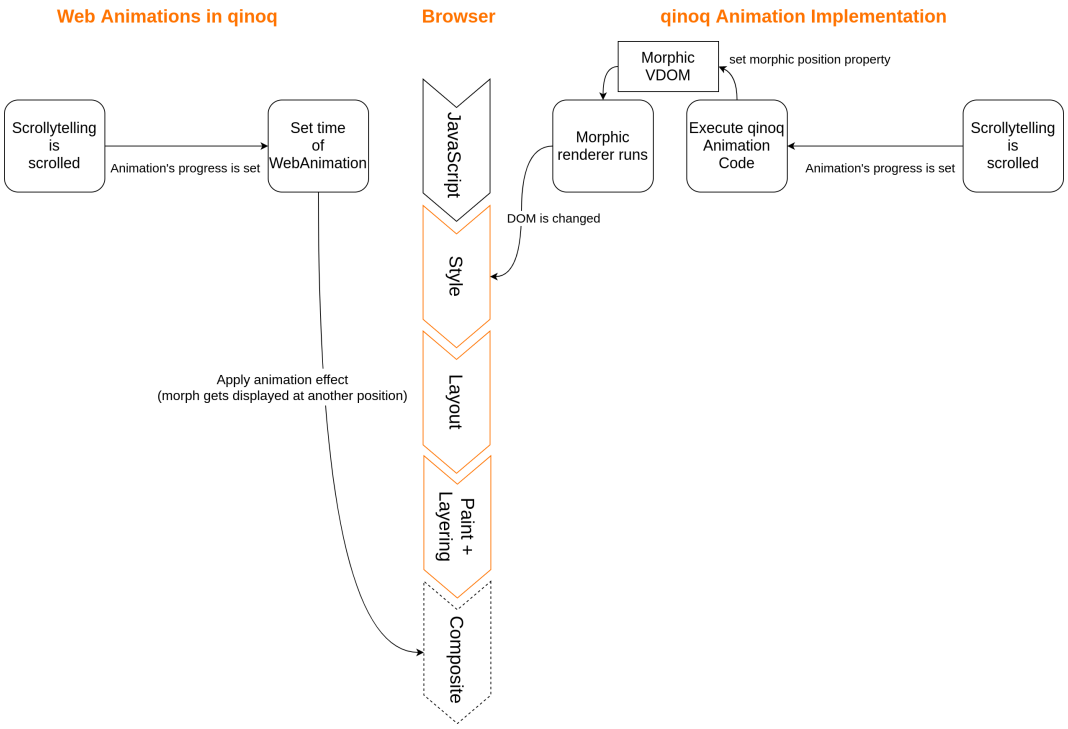


**Figure 4.9:** Visualization of how WAAPI animations in qinoq and original qinoq animations are applied and rendered by the browser: An animation on the position of a morph is assumed. Using the WAAPI, there is no integration with morphic. However, the CRP (in orange) can be skipped entirely and only a new composited frame is required. The main thread (solid lines) is not involved. Round edges indicate JavaScript execution. Sharp edges represent data objects.

Although this leads to less code in the implementation of animations, it comes with the downside of less flexibility. Developers and designers are limited by the options browsers provide. Consequently, it is only possible to animate the visual properties of elements since the WAAPI relies on CSS. Animating custom properties of morphs remains only possible with our custom animation implementation.

During the evaluation, we have seen how the usage of Web Animations allows us to benefit from compositing when animating. Some animations can be executed without any involvement of the main thread. Web Animations changing `transform` and `opacity` can be run by the compositor alone. This is not always possible. Some CSS properties require changes at earlier steps of the CRP.[94] As we have seen, changing the color of an element is an example of this, forcing the related layer to be repainted. Nevertheless, a smaller area to paint due to the separation of elements onto their layers provides a performance benefit [37].

[94] https://csstriggers.com/ (last accessed on 2021-07-28) provides an overview over which rules require which steps to run upon changes.

However, maintaining a large number of separate layers may result in high memory usage and the individual rasterization of all of them can also become computationally costly.

Using the WAAPI, developers and designers of scrollytellings can rely on the browser to optimize such decisions and profit from a more slender main thread.

This is especially beneficial for consumers on mobile devices or weaker hardware [57]. Since the Web Animation standard is also relatively young, additional future optimizations in browsers are expected [3].

The browser-side handling of the animation comes with another significant downside: Morphic and lively.next do not usually expect external DOM changes. The reconciliation of morphic's VDOM and the actual DOM described in subsection 4.3.6 only changes the DOM in accordance to morphic's VDOM, not the other way around. This leads to morphic properties holding incorrect values after applying a Web Animation on a morph. As a consequence of this, the lively.next workflow during development is impaired. An example of this can be seen in Figure 4.10, where the halo menu's position and scale are off after animating the morph using WAAPI.

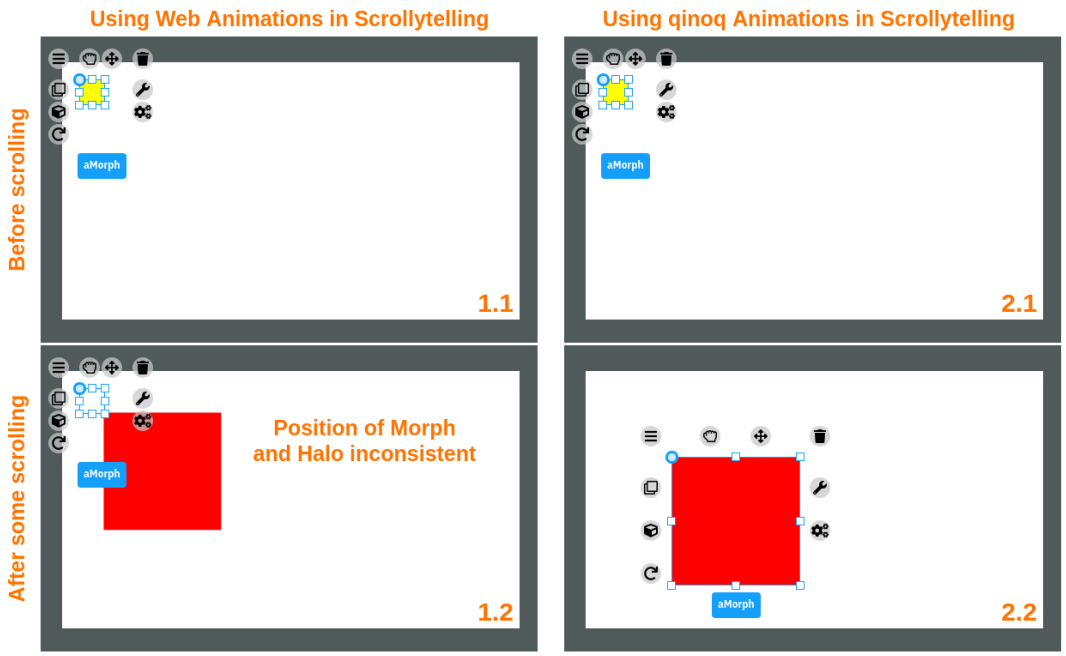


**Figure 4.10:** A morph's position and the placement of its halo menu is inconsistent after animating the morph using Web Animations.

### 4.5.3 Future Work

It has become clear that adopting Web Animations in qinoq is a worthwhile endeavor.

Simultaneously enabling support for WAAPI and custom animations would allow qinoq's animations to benefit from increased performance due to browser-side optimizations where possible while keeping a great deal of flexibility where necessary.

We have only worked with scrollytellings containing Web Animations by means of code. However, the main benefit of qinoq lies in its graphical editor to author scrollytellings including their animations. For the integration of a WAAPI based animation implementation into our editor, the difference mentioned above between morphic properties and the current DOM values resulting from Web Animations is the most pressing problem. As all of lively.next, our editor heavily relies upon the usage of halo menus and the manipulation of morphs utilizing the inspector (see subsection 3.3.1). For this, consistent values between the morphic properties and the visual presentation of a morph are essential. The presented implementation needs to be adapted to update the morphic property specified for an animation after each change in progress. Further investigation into the implications this has for rendering and thus performance is necessary.

Additional implementation is also necessary to support more morphic properties, as the POC only supports three properties. However, the POC already shows that it is possible to animate different morphic properties that map onto the same CSS rule simultaneously (Listing C.1, lines 69-72).[95]

Another limitation in comparison to qinoq's animation implementation is the limitation to two keyframes per animation. However, the above-described approach to map qinoq's progress to WAAPI offsets can easily be extended to more than two keyframes.

The wrapper currently also lacks support for easing functions. The `element.animate()` function already accepts an `easing` parameter as part of its timing options.[96] The easing functions can be defined by cubic bezier curves, the same way as in lively.next (see subsection 4.3.3). A caveat might be that each visual animation in qinoq can have its own easing. To achieve this using the WAAPI, it might be necessary to create a separate animation for each visual animation.

## 4.6 Summary

Keyframe animations are a technique well-suited to animate content in scrollytellings due to their expressiveness when animating, their approachability through graphical interfaces, and their prevalence in other creative software. Throughout this chapter,

[95] https://developer.mozilla.org/en-US/docs/Web/API/KeyframeEffect/composite (last accessed on 2021-07-28).

[96] https://drafts.csswg.org/web-animations-1/#time-transformations (last accessed on 2021-07-28).

we have seen how qinoq implements visual animations on morphic properties driven by scroll-position using the keyframe technique. Animations in qinoq use linear interpolation to animate between keyframes and can apply easing functions to create natural-looking animations. Due to a uniform interface, the animations easily integrate into qinoq's scrollytellings and, most importantly, its editor.

We considered that the current implementation of animations in qinoq requires a complete rerun of the browser's critical rendering path. Due to the risk of frame drops, which negatively impact the perception of animations, this is undesirable. A promising approach is to move computation off the main thread. We have seen how the Web Animations API allows us to achieve this by utilizing the layering system of browsers and their compositor thread as much as possible. The presented prototypical Web Animation integration into qinoq realizes these benefits, as we have seen in a demonstration. As the Web Animation API acts on DOM elements while lively.web revolves around the concept of morphs, further work is necessary for successful integration into our editor.

# 5 Evaluating qinoq Regarding the Creation of Scrollytellings on an Example

With qinoq, we designed software that pushes the boundaries of scrollytelling creation towards a process without writing code. This chapter evaluates our approach through testing the creation process with qinoq based on an example scrollytelling and empirical evaluation.

In the beginning, we take a brief look at the scrollytelling creation process of our project partner Typeshift. To explain the creation process of a scrollytelling, we use an example to show what qinoq's advantages and disadvantages are and where it has its limits. This highlights the functionalities the editor has and missing features. But also how scrollytellings are structured and how the editor supports content designers in creating such scrollytellings. In addition, we take a look at the programming interface with which developers can implement animations and behavior not creatable with the editor.

Afterward, there is the empirical evaluation, where our project partner tests qinoq and creates the same scrollytelling used for the walk-through of the creation process. Here we see whether qinoq matches our project partner's conceptual model of how scrollytellings are structured. First, a content designer creates a basic scrollytelling. Afterward, a developer will take this scrollytelling and add custom behavior. Then we take a look at how well content designers and developers can work together with qinoq. In the end, we evaluate and compare how qinoq improves the process of our project partner and how it solved the problems they initially faced.

## 5.1 Typeshift's Workflow without qinoq

First, we recap the current workflow of our project partner as described in subsection 2.1.2. Firstly the content designer talks to the client. The content designer discusses the structure and design of the scrollytelling briefly. The content designer then creates a storyboard. The texts and images are listed in the storyboard, showing exactly how the scrollytelling should look like. Then the content designer iterates over the storyboard with the client until both are satisfied with the result. The content designer may request custom Lottie animations or images from an external graphics designer. The developer later uses those animations and images in the scrollytelling. Afterward, the content designer talks to the developer to implement the scrollytelling with the help of the storyboard. The developer implements it and sends a finished

scrollytelling to the content designer. Now the content designer can scroll through the scrollytelling. Due to misunderstandings, the developer might implement ideas differently than the content designer imagined them, or they might not fit in the scrollytelling. So the content designer and developer iterate over this scrollytelling until they are satisfied with the result. This process may take much time. As soon as the developer finishes the scrollytelling, it is sent to the client. The client can now make remarks. The developer implements those remarks by iterating over the scrollytelling again to finish it.

## 5.2 Creation Process of Scrollytellings with qinoq

In this section, we will take a look at the creation process of a scrollytelling with qinoq. We will create an example scrollytelling based on an already existing scrollytelling created by Typeshift. Our project partner created it for the MediaTech Hub Potsdam (MTH)[97] with the help of external graphic designers. This scrollytelling will serve as a reference for the evaluation of qinoq. Within the scrollytelling there are:

1. property and Lottie animations,
2. custom animations programmed by developers, this includes scroll- and time-based animations, and
3. static content like images or text.

What the scrollytelling lacks is a complex way of interacting with it. We will create one later to test our tool, even if this is not part of the original scrollytelling.

The goal of the evaluation is to test the affordance of qinoq. With an already existing scrollytelling, it is clear to the participants how the scrollytelling should look. With this scrollytelling, we do not test how well the vision can be transferred. However, with the goal of qinoq being that content designers and developers can create a scrollytelling in one tool, the creation process is the main focus of the evaluation.

We want to evaluate how well content designers and developers can work with qinoq. Based on the example scrollytelling, we will look at some more complicated tasks to perform when creating scrollytellings, which problems occur, the key functionality the editor offers, and how convenient this is. We will not consider potential misunderstandings of the conceptual model or any problems that may occur unrelated to the limitations or workflow of qinoq.

### 5.2.1 Example Scrollytelling

The scrollytelling we take as an example is a scrollytelling[98] created by our project partner Typeshift[99] for the client MTH. The MTH is an association consisting of media and technology companies and start-ups. The scrollytelling is a review and summary

[97] www.mth-potsdam.de (last accessed on 2021-07-28).
[98] www.mth-potsdam.de/mth-jahresrueckblick-2020 (last accessed on 2021-07-28).
[99] typeshift.io (last accessed on 2021-07-28).

of the year 2020 for the association. It was purely implemented by a developer, with a content designer creating the storyboard.

Within the scrollytelling, there are some complex animations, which we try to recreate with qinoq. When describing the scrollytelling, there are different scenes. Those differ from sequences, which are described in subsection 3.2.1. A sequence encapsulates and groups morphs. On the other hand, a scene describes a separate section of the scrollytelling and can consist of several sequences. However, qinoq uses sequences to structure the content.

The first scene of the scrollytelling is a custom animation. This time-based animation consists of an image in the background and waves in the foreground as seen in Figure 5.1. Those waves (A) move within a defined space with a randomized movement.

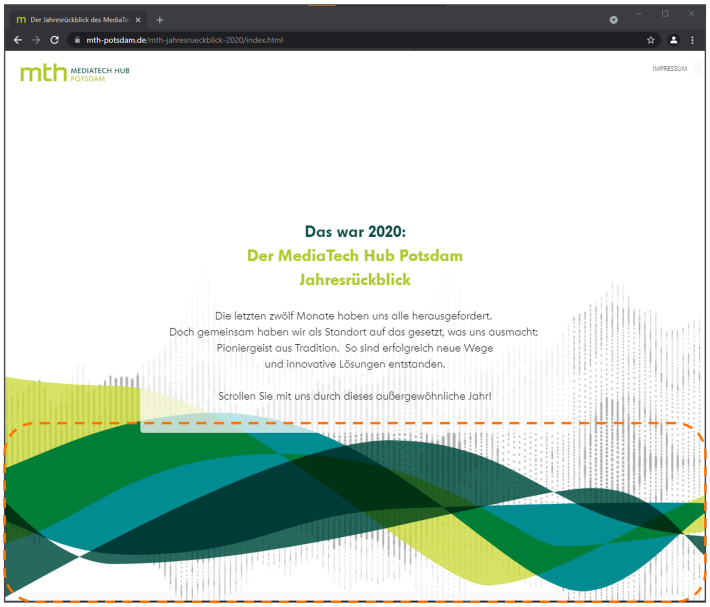


**Figure 5.1:** MTH Scrollytelling start scene

After this, some scenes consist of Lottie animations created by a graphics designer and enriched with custom animated morphs by developers, as described in subsection 4.3.4.

An example of a Lottie animation can be seen in Figure 5.2, the second scene. The desk with the laptop appears with a caption underneath it. When it has fully appeared, it zooms onto the laptop screen.

The scrollytelling also contains completely custom animations, which a developer implemented. We will take a closer look at them when recreating.

At the end of the scrollytelling, as seen in Figure 5.3, there is static content like text and images (A) but also some buttons for subscribing to the newsletter (B) and social media share buttons (C) to share the scrollytelling with others via social media. When recreating, we will also include such buttons.

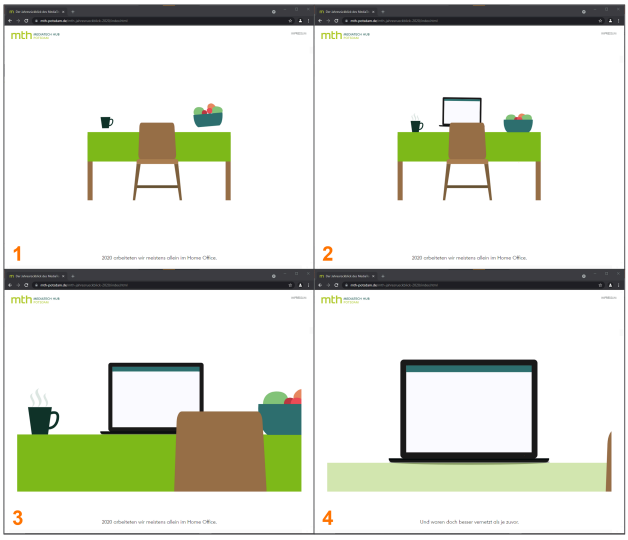


**Figure 5.2:** MTH Scrollytelling second scene

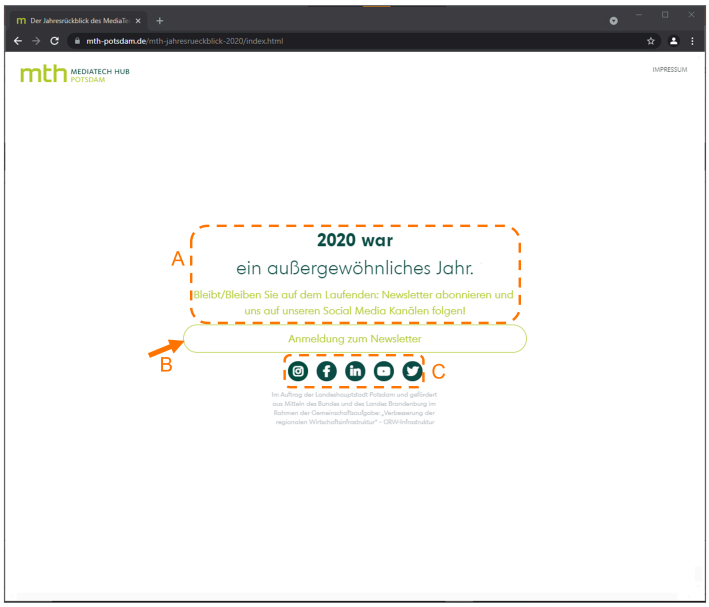


**Figure 5.3:** MTH Scrollytelling end scene

The rest of the scrollytelling consists of Lottie animations, custom scroll-based animations created by the developer, and static content like texts.

### 5.2.2 Creation Process

Now we will take a look at the creation process of a scrollytelling with qinoq and its editor. At first, we start with simple animations, which get more and more complex

in the course of this test. With that, we show what is possible with qinoq and how the single user interface components described in subsection 3.3.1 work together.

We start with what can be done with the editor, limiting ourselves to what a content designer can create without the help of a developer.

**Creating a New Scrollytelling** When the editor is opened, no scrollytelling is loaded as seen in Figure 5.4. A user then can create a new scrollytelling by clicking the button (A) or by loading an already existing scrollytelling via drag-and-drop (B) on the holder, which is described in subsection 3.3.1, into the editor.

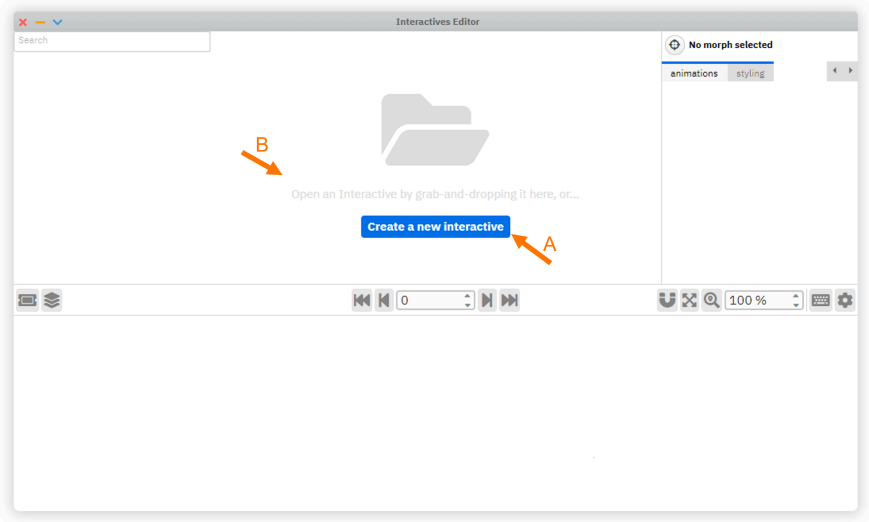


**Figure 5.4:** An empty editor newly created

When creating a new scrollytelling, as seen in Figure 5.5, there is a default sequence (A) in the scrollytelling. It allows an easier and faster start when creating new scrollytellings.

**Creating the Second Scene** After that, the user can start creating the main scrollytelling. We will recreate the example scrollytelling described above.

To create the scrollytelling, we will start with some simple animations. The start scene of the scrollytelling is more complex and will be covered later. In the second scene of the scrollytelling, there appears a desk with a laptop on it. An external graphics designer created this Lottie animation.

To create the animation, we will use a Lottie morph. In our example, we double-click on the first sequence to open the sequence view. To create a new Lottie morph, we select the Lottie animation morph from the top bar, as described in subsection 3.1.2. Now we can create a new Lottie morph via drag-and-drop in the holder. After that, we can change the animation URL of the Lottie morph to the correct path, where we saved the Lottie animation. The editor automatically creates

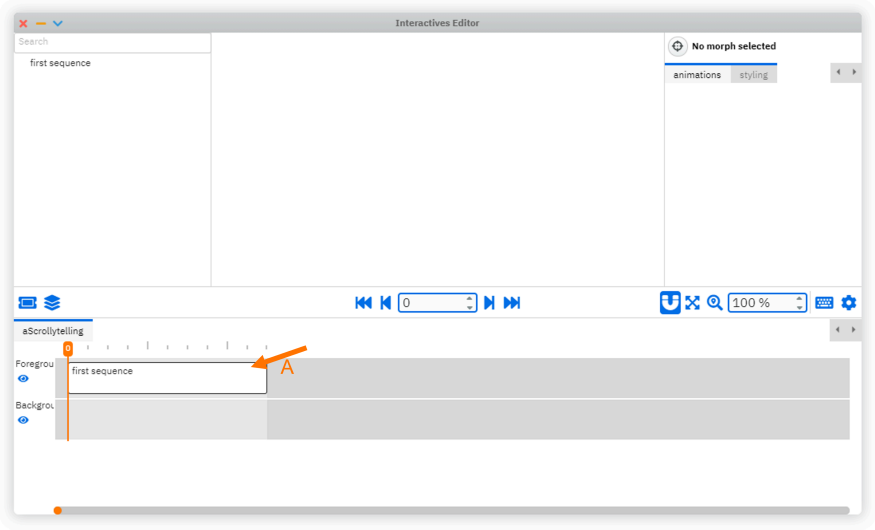


**Figure 5.5:** An editor with a newly created scrollytelling

an animation for the `progress` property from 0 to 1 as seen in Figure 5.6 (A). The animation will begin at the start of the sequence and conclude with the end of the sequence.

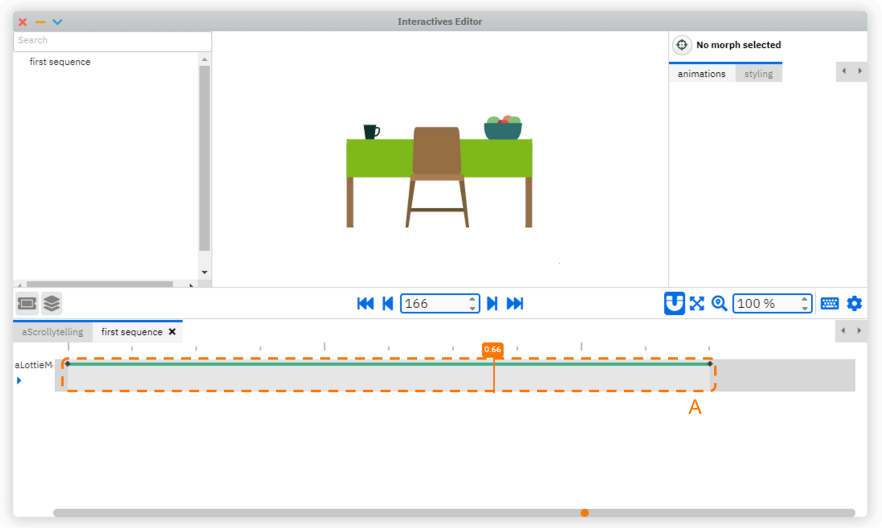


**Figure 5.6:** Lottie morph with animation in scrollytelling

Now that we have created the first animation, we can align and resize the morph. For the currently inspected morph, we can change the styling and properties as seen in Figure 5.7 with the inspector described in subsection 3.3.1. With properties, we

can define animations, which we will cover later in detail. We can align the morph to the center of the scrollytelling via the styling tab (A) with a single button press on the alignment option (B).

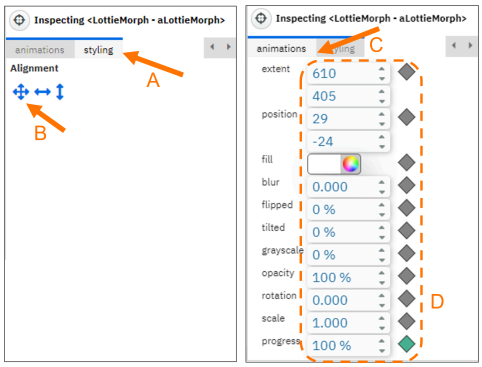


**Figure 5.7:** Styling (left) and animation (right) tab of the inspector for lottie animation

To finish this scene, we create a new label underneath the Lottie animation and set the correct text. Figure 5.8 displays the finished scene.

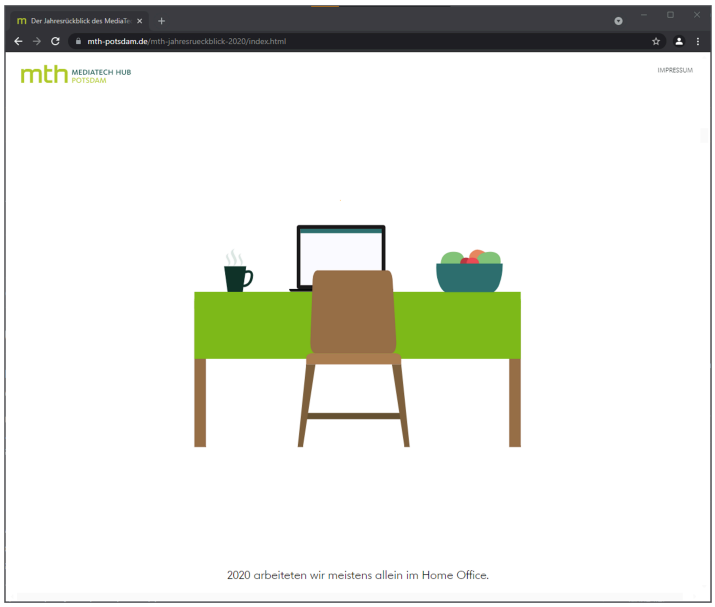


**Figure 5.8:** Second scene finished

Within the animations tab seen in Figure 5.7 in the upper right corner (C), the properties of the morph can be adjusted. There is a list of every property (D) which can be animated. This list does not contain every property that a morph has. In the

above example, we used a Lottie morph. Lottie animations already come with a predefined animation when being in the editor, thanks to the `progress` property as described in subsection 4.3.5. For every other morph without this property, the user must define the animation by hand.

For this, the content designer inspects the morph with the animations inspector. When we inspect the morph, there is the list of properties as seen in Figure 5.9. With the keyframe button (1.) next to the property, we can create a keyframe. When we click the button, this will create a keyframe (2.) with the value shown in the property field at the current scroll position.

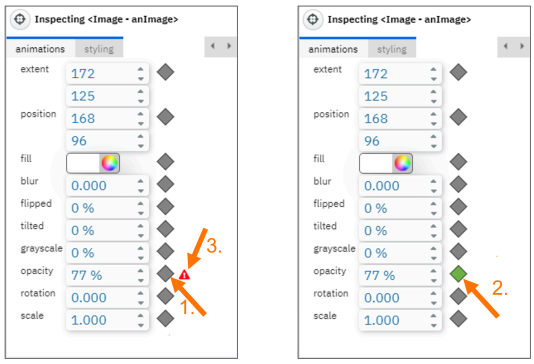


**Figure 5.9:** Keyframe at current scroll position set in animations inspector

The lane of the corresponding morph displays the newly created keyframe. Notice the red warning (3.), which indicates that changes made to the property can only be saved when creating a keyframe at this scroll position. This warning only shows when another keyframe for this property of the corresponding morph exists. When we create two keyframes for the same property, there will be an interpolation between those two keyframes. If there is no other keyframe, this property is constant throughout the sequence. Until now, qinoq supports only linear interpolation. For the interpolation, the user can select an easing.

**Labeling Created Items** Such scrollytellings can become very large quickly, therefore everything can be renamed for a better orientation with all items listed in the tree, which is described in subsection 3.3.1, as can be seen in Figure 5.10. We can rename the scrollytelling or sequences by double-clicking on the corresponding tab and entering a name. For example, for the scrollytelling itself, it would be the tab for the global timeline (A). For sequences, it would be the corresponding sequence tab (B).

In tree (C), everything currently part of the scrollytelling is listed. There is a search field, which filters the entries. With a double-click on a specific entry, the editor will focus on the chosen entry. With this users get the advantage to navigate through the project efficiently. The tree contains every sequence, including its corresponding morphs and animations, as its entries. The tree displays those, as the name suggests,

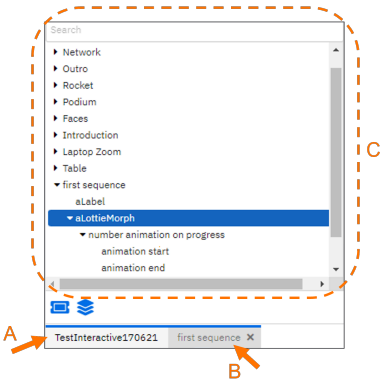


**Figure 5.10:** Timeline tabs with tree above

in a tree-like structure. Users can also rename keyframes. This is done by right-clicking on them in the timeline and then selecting "rename keyframe". Just like keyframes, sequences or layers can also be renamed by right-clicking on the elements.

**Creating the Last Sequence** With the second scene finished, we want to create the end of the scrollytelling. It has some static text, which fades in when scrolling to it. We can do this with the animations inspector seen in Figure 5.7 with an animated `opacity` property (D). Additionally, there are some special buttons, the social media share buttons (A) as displayed in Figure 5.11. Those allow a consumer to share the scrollytelling with others with a single click.

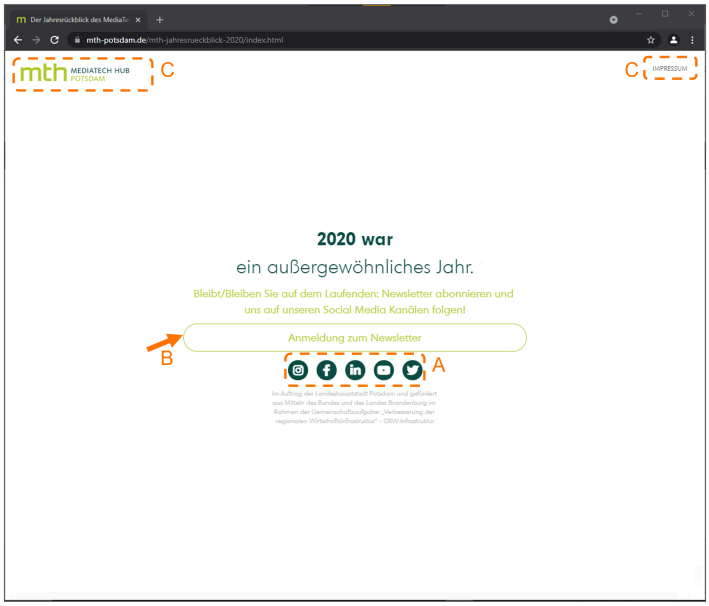


**Figure 5.11:** Last sequence of the example Scrollytelling

We added the social media buttons as special buttons to the top bar. When selected, they can be created by clicking in the holder. Once created, we can adjust them over the styling tab of the inspector as seen in Figure 5.12. There is a selection (A) to choose from various social media platforms to share the scrollytelling. Depending on the selection, the content designer can add different things, like the subject for an email (B) or the message for a tweet (C). When changing the platform, the icon changes as well. The buttons are still completely customizable. Users can change the size, color, and all other properties with the animations inspector.

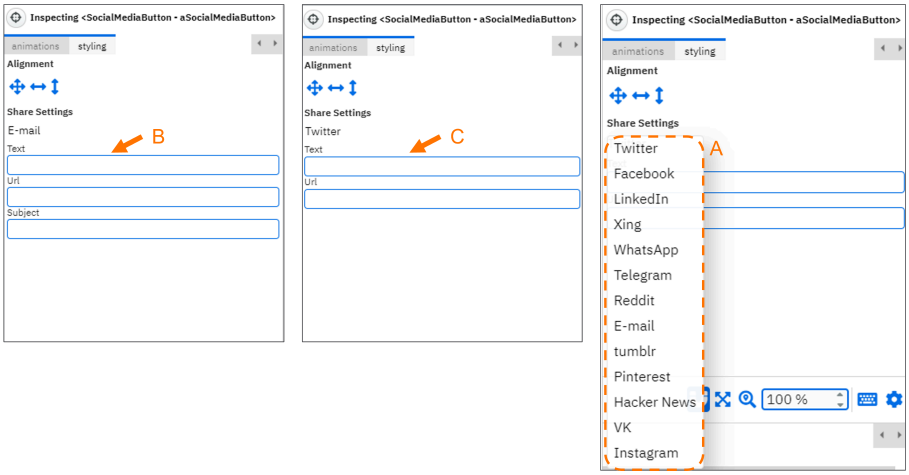


**Figure 5.12:** Styling tab for social media button with presets email, Twitter and selection dialog

At the end of the scrollytelling Figure 5.11, there is also the newsletter button (B). Users cannot create custom buttons with the editor. Instead, a developer must implement it through code. This is because it has custom behavior, which may differ for each scrollytelling. Thus no general solution can be provided.

**Creating a New Layer** When creating scrollytellings, there is often a background and foreground or elements which are persistent over the whole scrollytelling. These would be the buttons in the top left and top right corner in the example scrollytelling seen in Figure 5.11 (C). To create background layers, qinoq provides a layering system as seen in Figure 5.13.

Here we can create a sequence with the button (1.) located in the menu bar, which is described in subsection 3.3.1, and then drop it in a new layer (2.). We can create new layers with the corresponding button (3.). Layers structure the scrollytelling but also give users the possibility to layer sequences. If a sequence is in a layer above another, the first sequence will overlap all sequences in a layer below it. With this system, things like background layers can be created.

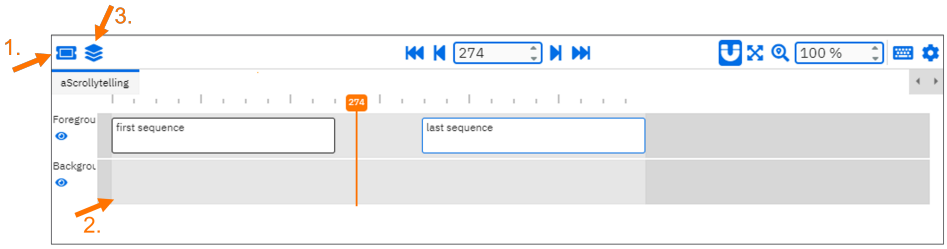


**Figure 5.13:** Creating a new sequence, which then can be dropped in a new layer

**Adding Custom Buttons** All the things mentioned above could be done purely with the editor without the need to write code. With the editor, we can animate and design large parts of the scrollytelling.

However, not everything is possible purely with the editor. Many times there are buttons in scrollytellings with custom behavior. Frequently used buttons are available as morphs for content designers to use. An example would be the social media buttons described above. However, custom buttons must be created by developers, like the newsletter button mentioned earlier.

Most commonly, images should be clickable, for example, to open a link. In that way, they behave like buttons. One example would be the MTH logo in the top left corner seen in Figure 5.11 (C). Those buttons cannot yet be created with the editor. Therefore developers must create them through code. A content designer can create a new morph, for example, an image morph. The content designer can then fine-tune the image with the animations inspector to adjust the properties. A developer can later make this image clickable. For that, we open the object editor described in subsection 3.1.2 and then create a new subclass. Within this subclass, we define one method which defines the behavior when clicking on the image. We can define the `onMouseDown()` method and, in this example, use the `window.open()` call to open a link in a new browser tab. With this simple code, the custom button is already implemented and works as expected. Additionally, we can set the `nativeCursor` property of this morph to change the cursor's appearance when hovering over the image. The changed cursor is an indicator for consumers that they can interact with this element. We can create buttons with arbitrary behavior this way.

**Creating Scroll-Based Animations** We cannot reproduce some other animations in the example, like the map animation seen in Figure 5.14 with the editor.

Developers have to create those animations through code. For such scenarios, developers can create the animation as they would without qinoq by adding the desired behavior to the morph, with the object editor described in subsection 3.1.2. Once they finish the animation, they can drag the morph like any other morph into the editor. Then the scroll change of the scrollytelling must be connected with the progress of the animation. For this, there are multiple possible ways. The easiest is

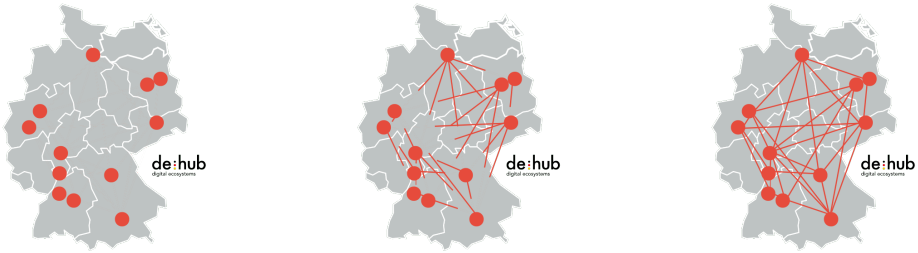


**Figure 5.14:** Start, middle and end frame of the map animation

defining the `progress` property on the custom animation morph. When the `progress` property is updated, developers must ensure the animation also updates itself. Then content designers can animate the progress animation like any other animation. We also provide a small interface for developers. This consists of several methods. For example, whenever a sequence is entered or left, there is a notification. Developers can implement a hook to this notification in order to add the custom behavior they want. When the progress is updated, the `onInteractiveScrollChange()` method is called on every sequence in the scrollytelling. Developers can use this method interchangeably with the `progress` property. All custom scroll-based animations can be added to the scrollytelling without problems. Content designers can then tweak those animations based on the keyframes. Developers can also listen to the `onSequenceEnter()` or `onSequenceLeave()` methods if needed. Those are called whenever a sequence is displayed or not displayed anymore.

**Creating Time-Based Animations** A more interesting example is the animation at the start scene of the example scrollytelling, seen in Figure 5.1 (A). The animation has scroll-based and time-based behavior, thus making it harder to recreate.

We use a polygon morph for this animation and define the vertices and edges to create the wave. To create the floating effect, we animate the positions of the vertices based on the current time. When the morph is animated, we can add it to the scrollytelling. This does not change the morph's behavior. It has to be said that content designers cannot fine-tune the timing of time-based animations. This could be added in the future. Such animations are purely accessible through code. In addition, some properties cannot be adjusted through the editor. This is because developers might set some properties programmatically, so with each animation step the code resets the properties to specific values. Besides that it is no problem for developers to add a time-based animation to the scrollytelling. The morph can be created entirely separate and then added via drag-and-drop to the belonging sequence. The boundaries for content designers could be pushed even further, enabling them to create time-based animations in the future.

**Adding Complex Interaction** With complex interaction methods, consumers should be more involved in the story to experience the topic themselves as described in section 1.2. Content designers and developers can accomplish this by creating

interactive elements. With those interactive elements, consumers can interact and change things in the scrollytelling on their own. Depending on the consumer interaction, the scrollytelling should behave differently. An example our project partner created is an interactive element [100] for the science publisher Spektrum.

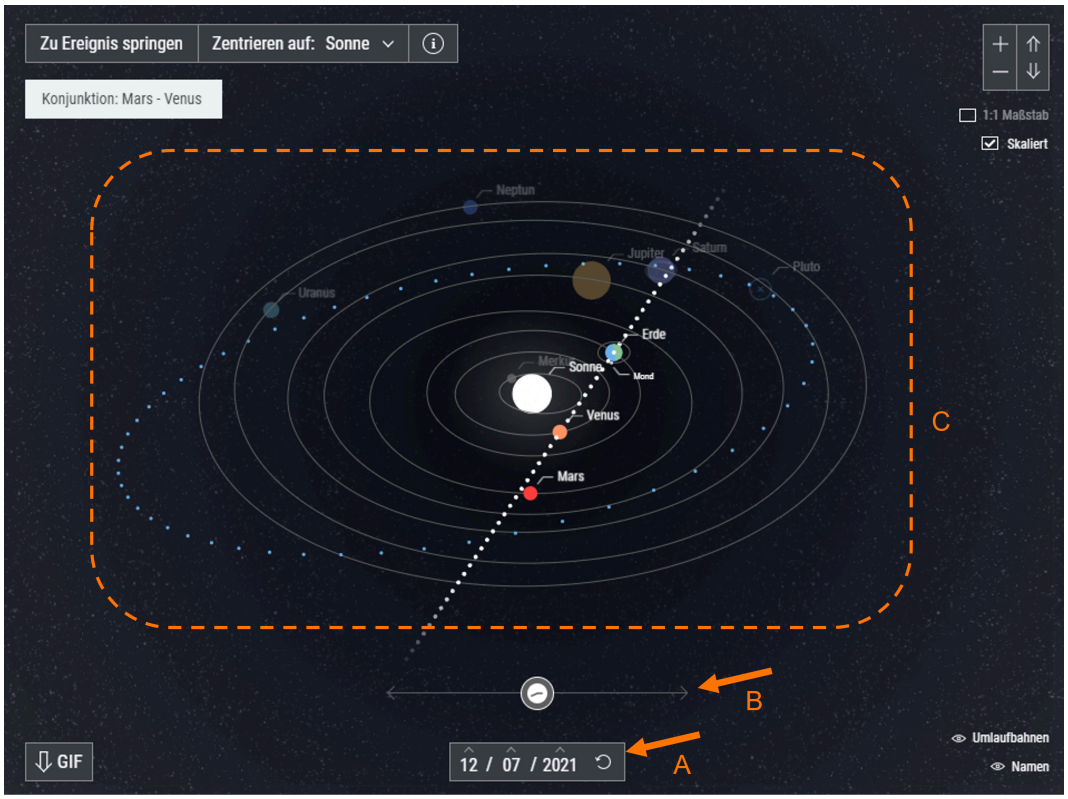


**Figure 5.15:** Interactive element showing our solar system

Figure 5.15 shows this interactive element with our solar system at different dates with the corresponding positioning of the planets. Consumers can interact with the element in a variety of ways:

1. They can set a specific date with an input field (A). This will change the constellation of the planets, depending on how the constellation was or will be at that specific date.
2. With the slider (B), consumers can fast-forward and rewind the time. This will also change the input field and constellation of the planets.
3. They can drag the whole element to change the angle of view (C).

There are many other ways consumers can interact with this element, but they do not differ from a development perspective, so we will only look at those three.

[100] www.spektrum.de/news/interaktive-planetengrafik-action-im-sonnensystem/1891840 (last accessed on 2021-07-28).

We will start looking at how to replicate those interactions and, based on this, show where the limitations for developers working with qinoq are. What we will not cover here is how exactly the solar system is drawn or updated. A morph's behavior, which is not related to events the morph receives, does not change when the morph is part of a scrollytelling.

We start by recreating the input field (A) with a morph, reacting to click events, the same way we created the buttons earlier. This interaction method is not problematic for developers to implement into a scrollytelling created with qinoq.

Both of the two remaining interaction methods utilize the drag behavior. To replicate this, we will create a slider (B). The main component of this slider will be the slider knob. Additionally, there will be a slider holder. This defines the space within which the slider knob can move. Based on this, the current position of the slider knob can be calculated as a percentage relative to the slider holder width. For the knob to work, we have to implement the `onDrag` method on that morph. Within the `onDrag` method, we can change the date based on the position the knob has. The knob receives the drag events, which allow us to implement the desired behavior. When the date changes, it will inform the drawing area to refresh. The same happens if the date is changed via the input field (A).

By this, we created the first two ways consumers can interact with the solar system. The last thing is the interaction method with the solar system itself. We can implement the `onDrag` method to recreate the interaction with drag, as we did earlier for the slider knob. From a programming perspective, those two ways of interacting with the scrollytelling are pretty similar. With the drag event, we now change the angle that the solar system is drawn in instead of the current time as we did for the slider knob.

Just as we covered the three interaction ways here, we can implement the rest of the interaction the same way because they are based on drag or click events. On a larger scale, most interactions in scrollytellings we discovered so far are click- or drag-based.

**Limitations** As we could see, it is possible to recreate complex ways of interacting within the scrollytelling. Now we want to discover what could limit developers when implementing interactive elements in scrollytellings. With qinoq, there is predefined behavior, which means qinoq provides a fixed structure that already takes care of the basic concepts, like event handling or displaying the correct sequences. With the interfaces qinoq provides, developers can implement a variety of animations, as seen above. Nevertheless, it might limit developers in the way consumers can interact with the scrollytelling. With the primary source of interaction being scrolling, there must be a morph receiving those scroll events. In qinoq, this is realized through a scroll overlay as described in subsection 3.2.2. The scroll overlay captures all events and notifies the underlying morphs, that a developer might add to the scrollytelling if an event happens. For the most common events, like mouse-clicking, dragging, or dropping, we delegate the event to the underlying morph at the position the event occurred. Unfortunately, not all events are covered. One example would be the `onScroll` event, which is currently not delegated. When the scroll overlay does not

delegate an event, this is a limitation for developers because they cannot use this specific interaction. A possible way of solving this would be to delegate all events to the underlying morph. We have not implemented all events yet because of missing implementation time, but they could be added in the future. However, it has to be said that this is no significant limitation because we covered almost every way a consumer can interact with a morph.

With a created scrollytelling, developers can now bundle the scrollytelling morph to publish it to a finished website. This can be done with a button in the object editor as described in subsection 3.4.5. Bundled scrollytellings do not have a responsive layout. Therefore the developer must add this later manually.

## 5.3 Empirical Evaluation

In this section, we will evaluate qinoq based on a user study. We will start with a content designer creating a new scrollytelling and recreating a couple of sequences of the example scrollytelling described in subsection 5.2.1. Recreating the scrollytelling will show how well content designers can interact with qinoq and how well qinoq supports content designers with the creation process. After that, a developer will enrich the scrollytelling created by the content designer with custom animations. In the end, there will be a custom scrollytelling that replicates the example scrollytelling. Our participants for the test are from our project partner Typeshift. With qinoq designed to solve their specific problems, it is best to evaluate it from their perspective. The test will show what works for them and what parts of qinoq still have to be refined to support them with their work even better. We will look at the key features already mentioned in subsection 5.2.2 in regards to how the participants use those and whether there are any problems or if it is clear to them. There should be three key takeaways:

1. How intuitive and discoverable are the implemented features?
2. Are there any points where the representation of scrollytellings did not match the conceptual view of the participants?
3. How well can content designers and developers collaborate with the editor?

With the final question, we will look at what could be done in the future to improve the collaboration between content designers and developers.

### 5.3.1 Test Scenario

As a test scenario, we try to recreate the scrollytelling described in subsection 5.2.1. The participants created the original scrollytelling, so the vision of the result is clear for both of them. This has some benefits because the vision must not be transferred to them and between the two, saving time for the participants. With this clear vision, it is also clear how the result should look like, so we have something to compare. This will test how well users can utilize qinoq to create scrollytellings. What this does not test is how well qinoq supports content designers in the creative process. This may seem to be a disadvantage of the scenario, but actually, it is a benefit. Because of

this, content designers and developers will focus on usability instead of the visual appeal of the scrollytelling.

At first, the content designer will start to create a new scrollytelling. Given are all assets and Lottie animations created by external designers. Then some of the sequences from the example scrollytelling will be recreated. Those only include sequences that can be created purely with the editor without the need to write code. This will test how suitable the editor is for content designers and how well qinoq embeds into the lively.next system. The content designer of our project partner used lively.next a couple of times before, but primarily for testing purposes. So the lively.next programming environment is not well known.

Afterward, the unfinished scrollytelling will be passed over to the developer. The developer then will add a custom scroll-based animation via code. We will not test other animations. As seen in subsection 5.2.2, this is not a problem because we can add any morph to the scrollytelling. The goal of this scenario is to test how well the integration of code-based animations works for developers. It should be mentioned that the participant for this test has a deep understanding of lively.next and all its components. With this part, we want to test if the programming interface is straightforward in adding code-based animations. The animation which the developer should recreate is the map animation described in subsection 5.2.1. Again, the test aims to see how well developers can add custom animations with a minor example that tests the support qinoq provides for developers, like the programming interface we supplied to add custom animations. This test should show how accessible and discoverable those interfaces are for developers and if they have to adjust their workflow to use qinoq.

### 5.3.2 Test Method

When evaluating software, it is essential to understand what barriers the participants face and what their actual problems are. Therefore, we will conduct usability testing with the think-aloud method with our project partner [47, 54]. The strength of the think-aloud method is that only a small group of participants is sufficient to gather substantial feedback. The participants are constantly articulating their goals, how they are trying to achieve these, and other thoughts linked to the process. So we can gather much information with just one participant. If one participant has a problem, it is likely that others will also have the same or at least a similar problem. So using the think-aloud method, many problems can be identified and eventually solved in further iterations of qinoq.

### 5.3.3 Testing with a Content Designer

At first, we will take a look at the testing with the content designer. We will decompose it into a couple of tasks. Just as in the walk-through, the goal is to recreate sequences from the example scrollytelling described in subsection 5.2.1. The focus of the test is to see whether the conceptual model [16] of qinoq matches the expectations of the content designer. Additionally, problems in the usability flow

should be discovered, and we evaluate how well qinoq integrates into the lively.next environment for content designers to use it.

For the test, the content designer was given a running lively.next instance and all graphics and animations which were used for the MTH scrollytelling. Based on this, the content designer created everything else independently.

**Creating a New Scrollytelling** At first, the content designer must create a new lively.next world. There was already an opened editor for the testing environment when creating a new world for a faster start. So creating a new scrollytelling was also just the press of a button. Nevertheless, there was some confusion because creating a new world and creating a new scrollytelling requires the user to put in two names. It was not clear to the participant what the use of the first name, the world name for lively.next, was. The second name is the name of the scrollytelling. Otherwise, there were no significant issues.

**Creating the Second Scene** With the newly created scrollytelling, the task was to create the scene already shown in subsection 5.2.1. This animation was done with a Lottie morph. The participant tried to add everything needed for this scene. This was a Lottie morph and a label. It was not entirely clear where to find those morphs and how to add them to the scrollytelling. This indicates the missing affordance of the lively.next environment, specifically the top bar. We could have implemented our own top bar within the editor with essential morphs. A significant disadvantage would be that this must be updated whenever the top bar in lively.next changes. So we decided to keep the lively.next top bar, because we believe, even if this is a barrier when starting to use the tool, for experienced users, which qinoq is designed for, this will not be a big problem.

When a new scrollytelling is created, users find themselves in the global timeline, and as mentioned earlier, it is not possible to add morphs here. Instead, users must open a sequence timeline to add morphs there. This is not discoverable because there is no indicator when directly trying to create a morph in the holder. However, a warning is shown when dragging morphs into the scrollytelling when the global timeline is opened. We can fix this by also adding a warning when trying to create morphs directly in the holder.

The participant found the text box morph button in the top bar, created a new text box morph in the world, and then tried to drag it onto the holder. Because of the warning shown there, the participant opened the sequence view of the already existing sequence and added the morph. By double-clicking on the text box morph, the text can be edited. When searching the Lottie animation morph, the participant also discovered the label. This was confusing for the participant because the difference between a label and text box morph was unclear. It was explained that labels are for displaying text and text box morphs are for consumers to input text that can be processed later. This is not discoverable and lively.next users have to know this to use them as intended. Like the example with the top bar, this shows that lively.next is in an early development stage. However, with a growing user base, more and more usability problems will be fixed to improve the workflow. In the

end, the participant created and added a label and Lottie animation morph to the sequence. The link to the Lottie animation was adjusted with all needed morphs added to the sequence, and the label's text changed.

Now that every element is in the sequence, the participant adjusted the positioning of the two elements. This was rather easy with the alignment options in the styling inspector, as seen in Figure 5.7. The desk animation consists of a Lottie animation. Because of that the keyframes are already created, and the animation could be viewed by scrolling through the sequence. Now the label must be animated to fade in. This was a challenging step for the participant because the concept of keyframes was not entirely clear.

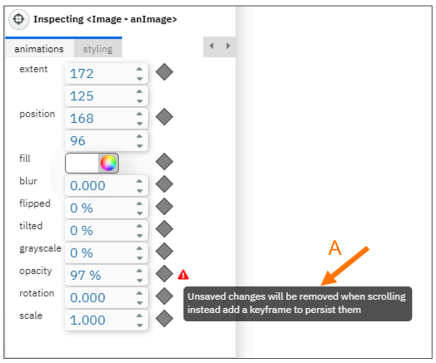


**Figure 5.16:** Warning when creating a keyframe

The participant noticed the warning, seen in Figure 5.16 (A) when creating a new keyframe. This warning indicates that the new value will not be saved when scrolling to another scroll position unless a keyframe is created. Reading the tooltip helped the participant to understand the fundamentals of keyframes, and the tooltip was accepted as a valuable indication. After explaining the concept of keyframes in more detail, creating new animations was relatively easy. Some things did not quite work as expected, for example, some shortcuts for copying morphs, undoing changes, or changing the font of the text label. Those problems were resulting in an unstable environment and some bugs both in lively.next and qinoq. Those bugs could subsequently be fixed, but we will not cover them here because they did not change the result of the test.

Overall the scenario went well. There were some barriers, especially with creating keyframes. Keyframes are one of the fundamental concepts behind qinoq and the editor. In conclusion, qinoq's design and functionalities enable content designers to create scrollytellings and animations. The concept behind keyframes might be hard to understand at first. With qinoq being a tool designed for power users, this is no problem because creating keyframe-based animations is very fast when understood once. It is unnecessary for content designers to understand the mathematical and technical concepts of keyframes, just how to use them. This could also be discovered through an explorative process.

**Creating the Third Scene and Fine-Tuning** After the second scene, the third scene had to be recreated. A couple of faces appear on the laptop screen within this scene, as seen in Figure 5.17.

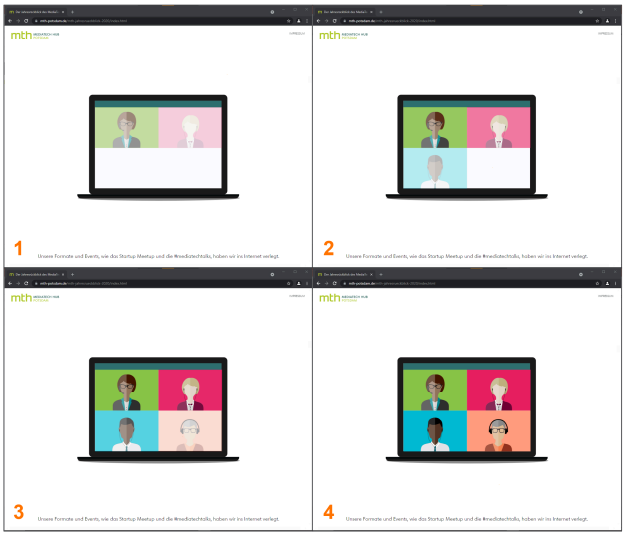


**Figure 5.17:** Third scene with animated images

A Lottie morph and four image morphs are needed to recreate this scene. The participant used the Lottie animation from the previous sequence with the `progress` property set to 100%. With this, there is no animation, just a static image. This way, the transition between the previous sequence and this one is flawless. This is very easy because a user can right-click every morph in a sequence in the timeline to copy and paste it into another sequence. The only thing left to do is remove the keyframes for the `progress` animation and set it to 100%. Now with the Lottie animation finished, it was time to animate the faces. One problem occurred because the layering of the morphs in the sequence was wrong. One of the faces was behind the laptop screen. There is no problem in reordering the layers in the global timeline. It is possible to drag the layer to the proper position. However, lanes in the sequence timeline are not draggable because lively.next already offers a solution for that in the halo menu, which is explained in subsection 3.1.1. This was unexpected for the participant because the interaction method differs from the global timeline for almost the same functionality. We could also implement drag behavior for morphs in sequence timelines to solve this problem. This is a more complex task for lanes because all submorphs must be considered as well. We did implement an option to change it when right-clicking the morph in the timeline. However, once the morphs were in the correct order, animating the faces to appear was no problem. This also showed, once the concept behind keyframes is understood, it enables a speedy workflow for content designers to create animations. After everything was finished,

the participant scrolled through the sequence and noticed that it did not feel natural. This was because of the timing in which the individual faces appeared on the screen. For the fine-tuning, the participant wanted equal spacing between the keyframes to achieve a smooth animation. Just as for sequences, there is also snapping for keyframes, which means if a user drags one keyframe within a small range from another, the keyframe which gets dragged will snap to the same position the other keyframe has. This also works for multiple keyframes. With snapping, a natural feeling spacing between keyframes must only be found once, which then can be transferred to the other keyframes. Also, adjusting the easings of the keyframes is relatively easy. The only limitation here is that there are currently only 25 different easings to chose from, as described in subsection 4.2.2.

**Creating a New Layer** After the first two scenes were finished, one thing that was missing was creating the background layer. This will test whether the concept of having different layers with sequences matches the conceptual model of the participant. So the task was to add the two labels on the top left and right corner as seen in Figure 5.11 (C). The participant intuitively created a new sequence and placed it in a layer below the other sequences. After that, the participant created the two images and arranged those. Finally, the participant changed the length of the sequence to match the length of the scrollytelling. Interestingly, the participant articulated a couple of times that the concept of sequences, which is described in subsection 2.5.3, was not entirely clear while testing. However, in the test, the sequences were used as intended. So even though the participant felt like not knowing how sequences should be used, they were utilized in an intended way. The reason could be that the word "sequence" was previously not used. Our project partner structured the scrollytelling in individual scenes. With qinoq, the representation was changed because something like layering would not be possible if every scene is encapsulated.

**Easing Representation** Now that the two scenes are finished, it is time to recreate the start scene of the example scrollytelling. Content designers and developers must work together to create this scene. The developer needs to add a time-based animation, which will be covered in the next section. The content designer can arrange the text and animate it. It is a simple animation on the `position` and `opacity` property. One exceptional circumstance with this property is that the `position` has an x and y component. For both axes, the animations inspector has a number input field. It is also shown in the sequence timeline in the lane for the morph with two individual curves. Those are for the graphical representation for the property change within one animation, as described in subsection 3.3.1. There must be two curves between the keyframes for the `position` property, one for x and one for y. The representation for each property was confusing for the content designer. The curves show the change to the property mapped to the height of the lane. However, it was even worse with two curves because now the curves in the lanes did not match the images in the easing selection. Because now the curve for y was inverted, which does not correspond to the image in the easing selection and has an inverted gradient.

The concept of having keyframes and easings is fundamental, which the content designer got used to quite fast, but the graphical representation could be clarified for easier understanding. However, finding a way to solve the inverted gradient for the y curve is much more complicated. Another possibility would be not to draw the curves at all, but as an earlier test showed, the curves helped more than they confused, which is why we included them for now. With qinoq being a tool mainly experienced users will use, this should not be a considerable disadvantage because it enables a faster workflow when understood.

### 5.3.4 Testing with a Developer

Now that the basic scrollytelling is finished, the developer can start adding custom animations with code. We will look at a simple custom animation and how the developer interacts with the programming interface qinoq provides. The developer works in the same lively.next world the content designer created earlier.

**Creating custom animations with code** The animation to recreate is the map animation described in subsection 5.2.1. To create this animation, the participant at first subclassed a canvas morph. Because the participant already knows the scrollytelling and animation, it was effortless to recreate. With a canvas morph, the participant can set single images to be rendered. All the images needed for the animation were given. Therefore the task for the participant was mapping all the images to the progress of the sequence. For that the participant used the `progress` property on the custom canvas morph:

```
getImageForProgress (progress) {
  return this._frames[Math.floor(progress / 100 * (this._frames.
     length - 1))];
}

update () {
  const frame = this.getImageForProgress(this.progress);
  this.context.clearRect(0, 0, this.width, this.height);
  this.context.drawImage(frame, 0, 0);
}
```

With the `getImageForProgress()` method the corresponding frame for the current progress is selected from the `_frames` array. In the `update()` method, which is called whenever the `progress` is updated, the correct frame is selected and drawn on the `context` of the canvas morph.

As described in subsection 5.2.2, the editor immediately displays the `progress` property in the animations inspector. The participant then created two keyframes at the correct positions. With the mapping finished and keyframes set, the animation played, but the mapping was incorrect because the range of the `progress` property was wrong. This was not clarified enough in the programming interface and differed from the one seen for Lottie animations. The progress for Lottie animations is mapped between 0 and 1, but in the animations inspector displayed between 0% and

100%. This is for easier accessibility for content designers. By using the `progress` property, content designers can easily adjust the animation by changing the positions of the keyframes. The participant mentioned that the programming interface was not clarified enough but could create the animation without problems. After the test, we clarified the programming interface with the participant's help to match the expectations.

## 5.4 Discussion

We can now evaluate how qinoq improved the workflow and how qinoq helps content designers and developers for scrollytelling creation based on the testing. Later we will discuss the problems that occurred while testing and how those problems can be solved in the future.

### 5.4.1 Change in Workflow

Without the editor, a content designer creates the storyboard. The content designer sends the storyboard to a developer, who implements the scrollytelling. Changing the scrollytelling was not possible for content designers. Creating such individual scrollytellings is a very time-consuming process. There must be much communication to explain the vision to the developer. There may be some misunderstandings, or some ideas might not turn out as expected. So these parts of the scrollytelling must be reworked. The example scrollytelling took our project partner 24 hours of development time. With qinoq, the approximate development time would be around 12 hours. This time is the estimation from the participants based on the progress made while testing and rebuilding the scrollytelling, which was 4 hours, and how long it would take them to finish this project. There might occur some unexpected difficulties, but even then, the development time would be much lower compared to before. It was the first time for the project partner to work with the editor with all its features. So in the future, they might be able to complete it even faster when getting used to qinoq. With the editor, content designers also can help with creating scrollytellings. This enables them to work more freely and experiment with the positioning or timing of the individual elements. Developers can now focus on creating custom animations and interactive elements. With this, content designers and developers focus on their field of expertise. The creation process of scrollytellings with qinoq is much less time-consuming than before, as mentioned above. Content designers can also experiment and realize their vision to see what works and what does not. This facilitates the communication between content designers and developers enormously. With content designers adjusting the scrollytelling, there are much fewer feedback loops. Content designers can now fine-tune animations independently, without the help or need of a developer. Both user groups now focus on their domain and what they are good at, with a lower risk of misunderstandings.

### 5.4.2 Enhancements for Content Designers and Developers

**Content Designers** In the old workflow, content designers relied on developers to implement their vision as described in subsection 2.1.2. With the editor, content designers can now create scrollytellings on their own without writing code. While testing most of the options the editor provides were quickly accessible and discoverable. There are still some problems left that will be worked on to improve the editor even further, which we will cover in the next section. Keeping the overview over large projects is no problem with the tree, naming the individual items, and structuring with layers and sequences. The editor allows content designers to fine-tune keyframe-based animations and directly implement their vision of the scrollytelling. Content designers are now able to create highly customizable scrollytellings without relying on developers to implement their vision. However, developers are still needed to create complex animations and interactive elements.

**Developers** Without qinoq, developers must implement the whole scrollytelling. This iterative process is very time-consuming. Now developers do not have to create the whole scrollytelling but instead focus on unique animations and interactive elements. With this, developers can work much more focused. Another improvement is the given structure, even if it may limit the developer partially when creating specific animations. Thereby developers can directly start to create the animations without the need to bother about event handling or similar problems common to all scrollytellings. This results in a much faster creation process for scrollytellings and custom animations. Developers can now work graphically with the editor. In comparison to before, the developer had to create animations with code and then test them. If they did not look right, the developer had to adjust them again. It is much easier to create those animations with a graphical interface, which displays changes directly.

### 5.4.3 Possible Enhancements

As seen in section 5.3, a couple of problems occurred while testing. These problems and enhancements can be clustered into three categories:

1. Possible enhancements for the lively.next environment
2. Possible enhancements for qinoq
3. Mismatch between the conceptual model of qinoq and the user

We will now look at some of the examples and their impact on the creation of scrollytellings. Each category can have a massive impact to the point that it renders qinoq unusable for content designers in productive use, so it must be carefully considered how to tackle the individual problems.

**Possible Enhancements for the lively.next Environment** We will look at a few examples from the first category, but because we did not design this system, they are not of that much importance for the test of qinoq. It is still worth mentioning those because they can significantly impact how content designers interact with the

system and ultimately with the editor. One problem, for example, was regarding the font of the texts. The participant created a label for the second scene. Changing the label's text was no problem, but the participant did not find the correct option to change the font. This is because there is no menu entry for changing the font of a label when right-clicking it. This option is not yet implemented for labels in lively.next. In lively.next, there are two different text morphs, a label, which the participant used, and a text box. For the text box, there are many more options to customize it, such as changing the text's font or alignment. A text box introduces other problems. One example would be that text box morphs are also an input method for a consumer. This must be disabled if it should only function as a label. Changing fonts might seem like a minor issue, but design, fonts, and alignments of texts have a considerable impact on the immersion, described in subsection 1.2.3, of scrollytellings. Of course, developers can change the font. However, with the editor's goal being to enable content designers to create scrollytellings and push the boundaries of what they can do, this holds them back. The most significant problem is that those issues are not discoverable or documented because lively.next is not widely used. The only way to be aware is to know about the issues. This maps to many problems related to lively.next and can only be changed through better documentation or a more straightforward and self-explanatory user interface. So those problems might be annoying for content designers now, but most of them will probably be fixed in the future. The impact for qinoq is noticeable, but with qinoq designed for experienced users, those do not hold them back from using qinoq.

**Possible Enhancements for qinoq** Within qinoq, some problems arose while testing. One, and probably the most crucial example, would be the visual representation for easings, especially for properties with two dimensions like position or extent. Another example are the shortcuts, which sometimes do not work. Especially the undo and redo features are helpful and should work reliably. Additionally, there are just missing features, which would be very helpful for the creation of scrollytellings. One example would be changing the lane of morphs via drag behavior, just like in the global timeline for layers. Another essential feature is the responsive layout, which allows creating one scrollytelling usable for desktop and mobile. Those minor enhancements are mostly easy to implement but can have a noticeable impact, particularly on the time it takes to perform some tasks. Other issues affect the software in a much more enormous scope, such as the ability to create time-based animations with qinoq itself instead of code. This would push the boundaries for scrollytelling creation, without code, even further. Adding the time-based component to qinoq means introducing another layer of abstraction to the already existing scroll-based timeline. Therefore the user interface for this must be carefully designed to be still concise and easy to use.

**Mismatch between the Conceptual Model of qinoq and the User** The last category of problems is by far the most important. If there is a mismatch between the conceptual model of qinoq and the user, it makes qinoq basically unusable for the user. One example would be how to structure the whole scrollytelling. In the

test, the participant found the concept of layers and sequences not intuitive at first glance. This will be a game-changer if users do not accept this concept. Fortunately, the participant used the concept just as intended and was only confused with the naming of sequences. The confusion may be related to content designers structuring the scrollytelling in individual scenes, differing from sequences.

## 5.5 Summary and Outlook

In this chapter, we started by looking at the project partner's workflow without qinoq. We introduced the example scrollytelling, which we used for evaluating qinoq. We looked at the main functionalities qinoq has and its limits, based on this example scrollytelling. We discussed how qinoq improves the workflow for content designers and developers. Based on the scenarios, we asked a content designer and developer from our project partner Typeshift to recreate the example scrollytelling. We observed how users interact with qinoq, what already worked, and where the remaining problems are. In the end, we categorized the problems into three categories. Based on these categories, we evaluated how well qinoq supports its users.

As we could see, qinoq improves the creation process of scrollytellings for content designers and developers. Nevertheless, there are still some remaining problems that could be solved in the future. The possible enhancements mentioned above for qinoq would allow even faster and more straightforward creation of scrollytellings. With a consistently improving lively.next environment, the problems related to lively.next will also disappear, making qinoq a powerful tool for content designers and developers to create highly customizable scrollytellings. One prominent missing feature is the possibility of creating time-based animations with the editor. With the support for time-based animations, users can create even more animations with the editor. With qinoq, content designers can create and edit scrollytellings without the help of developers. So both content designers and developers can focus on their specific tasks. So even though qinoq achieved its goal, it could be improved in the future to push the boundaries even further while still being easy to use.

# 6 Conclusion

With our project, qinoq, we aimed to help our project partner Typeshift create scrollytellings. Scrollytellings are interactive web pages combining the benefits of movies, pictures, and books.

As scrollytellings are interactive media, they may require authors to have skills in multiple fields like writing, content composition, animation, and programming. Therefore, a multidisciplinary team is required. Currently, many iterations are necessary when creating a scrollytelling. Our editor aimed to reduce iterations by enabling content designers to create and edit partial scrollytellings independently from developers. At the same time, developers add highly specific missing animations or interaction possibilities for consumers.

We have analyzed existing software applications for authoring interactive media. Thereby, we gathered a variety of concepts for implementing features supporting content creation. The discussion of our findings has shown that none of the existing software suits our design goals of streamlining the collaboration between content designers and developers when creating scrollytellings. Hence, we developed a concept for our own editor. It should use a timeline to compose content spatially and temporally and sequences to encapsulate content semantically. Realizing animations using keyframes allows for highly individualized animations while forming a flexible basis for different animation techniques.

The lively.next environment is well-suited for the creation of scrollytellings. Everything visible is made up of morphs, which can be combined to form more complex elements. This is supported by tooling both for developers and non-developers. To create scrollytellings in lively.next, we have created a structure that combines a clear interface with the flexibility of the morphic system. Our editor was also created in lively.next. It enables content designers to create scrollytellings from scratch by combining morphs to sequences, adding animations, and fine-tuning morphic properties. We provide an interface for developers to enrich scrollytellings with custom, interactive elements.

Scrollytellings make heavy use of animations. Animations in qinoq are based on the keyframe technique. We have shown how qinoq implements keyframe animations based on linear interpolation and how easing functions can be applied to achieve natural-looking animations. We have also seen how Lottie animations can be integrated into scrollytellings, utilizing the flexibility of our animation implementation.

We evaluated the features of qinoq based on an example scrollytelling created by our project partner, which includes different types of media. With this example scrollytelling, we showed what the advantages, disadvantages, and limitations of

qinoq are. Finally, we evaluated qinoq regarding usability with a content designer and developer from our project partner.

Our evaluation showed that the functionality the editor presently provides works well and achieves many of our design goals. Building upon it, we see many opportunities to improve qinoq further. Additional features such as creating time-based animations within the editor could push the boundaries of what content designers can create. Currently, time-based and non-linear path animations can only be created programmatically. For content designers to create such animations, a carefully designed user interface is required, allowing precise editing of the animations while still being easy to use. Especially time-based animations could be challenging to integrate into the editor since it currently only revolves around the scrollytelling's scroll position.

Furthermore, we also noticed that the performance of our animation implementation could be improved by using the Web Animations API, where that makes sense. Usage is especially beneficial where it leads to animations that can be executed solely by the browser's compositor thread without work by the browser's main thread. However, this must be carefully considered since it may deteriorate the functioning of lively.next tooling, such as the halo menus.

Some workflows within the editor could be improved to enable even faster scrollytelling creation, such as expanding the set of undoable operations and responsive layouting for the scrollytelling.

Since we implemented qinoq in lively.next, its usability is highly coupled to that of lively.next. Usage of our editor will benefit from future improvements in lively.next, such as further approachability for non-technical users and increased stability.

Nonetheless, even at its current stage, qinoq is a powerful tool for content designers and developers to create highly customized scrollytellings.

With qinoq, we improved the workflow of our project partner Typeshift. With content designers creating and editing scrollytellings directly in lively.next, developers and content designers benefit from a faster workflow and a better separation of tasks.

# Appendices

# A Appendix Chapter 2

## A.1 Application Shortlist

The following interactive content creation applications fulfill the prerequisites[101] for the software analysis in section 2.4 reasonably. They are thereby on the shortlist to be selected as one of the investigated applications.

| Application | Short Description | Decision |
|---|---|---|
| Blender | Open-source 3D creation suite for 2D and 3D animation, modeling, motion graphics, video editing, and post production [102] | Not selected. Many features of minor relevance, has overlaps with DaVinci Resolve |
| DaVinci Resolve | Software solution for video editing, color correction, visual effects, motion graphics, and audio post production [103] | Selected. Movies are closely related to scrollytellings. Provides similar implementations like Blender and Godot. Aims for collaboration between differently skilled people |
| Godot | Open-source game engine for 2D and 3D [104] | Not selected. Many features of minor relevance, similar implementations to DaVinci Resolve |
| HyperCard | Editor and learning environment for interactive content and applications, developed in the 1980s and 1990s for Mac OS 9 [34] | Selected. Historical solution to combine design and scripting |
| Inkscape | Open-source editor for vector graphics [105] | Not selected. Similar to Macromedia Flash and Microsoft PowerPoint, but project is limited to one page |
| klynt | Editor for Interactive Storytelling Online [106] | Not selected. Strong focus on interactive conventional websites. Similar implementations to Microsoft PowerPoint and DaVinci Resolve |

**Table A.1:** Shortlist of applications suitable for software analysis

[101] See subsection 2.3.1 for details.
[102] https://www.blender.org/ (last accessed on 2021-07-28).
[103] https://www.blackmagicdesign.com/products/davinciresolve/ (last accessed on 2021-07-28).
[104] https://godotengine.org/ (last accessed on 2021-07-28).
[105] https://inkscape.org/ (last accessed on 2021-07-28).
[106] https://www.klynt.net/ (last accessed on 2021-07-28).

| Application | Short Description | Decision |
|---|---|---|
| Macromedia Flash | Professional authoring tool for creative content. Actively used in the early 2000s[107] | Selected. Historical. Professionally used to create all kinds of interactive content for the web |
| Microsoft PowerPoint | Popular tool to create slide-based presentations [108] | Selected. Slides offer alternative approach to interactive content. Intuitive user interface |
| Wix | Popular toolkit to build websites [109] | Not selected. Focus on conventional websites and usage of design templates with limited options for customization |

**Table A.2:** Shortlist of applications suitable for software analysis (cont.)

## A.2 Extended Software Analysis

Based on the features outlined in section 2.2, we explored the selected applications from section 2.3 concerning their implementations of these features. The following is a detailed summary of the findings which form the basis for the software analysis in section 2.4. The analysis bases on the support material provided by the applications and hands-on tests [23][42][46][51].

### A.2.1 Setup

The applications were tested on Windows 10. DaVinci Resolve 17 and Microsoft PowerPoint are available for installation online and support this operating system. Macromedia Flash Professional 8 is no longer available at the vendor. Hence, it was downloaded from *Soft32* [12]. The installation can be conducted ordinarily. HyperCard 2.4.1 was released for Mac OS 9 and is not available for sale anymore. It was downloaded from *Macintosh Repository* [22]. As it does not run on the Windows operating system, Mac OS 9 was emulated with the *SheepShaver Emulator* to conduct the analysis on Windows [45].

### A.2.2 Terminology

In the following, we will refer to the workpieces handled by the applications as *projects*. The area that allows previewing and arranging the elements seen in the later product will be called *stage*. A *capsule* is an abstraction of multiple elements into

[107] https://www.adobe.com/support/documentation/en/flash/fl8/releasenotes.html (last accessed on 2021-07-28).
[108] https://www.microsoft.com/en-gb/microsoft-365/powerpoint (last accessed on 2021-07-28).
[109] https://www.klynt.net/ (last accessed on 2021-07-28).

some kind of black box. It provides a context for the contained elements, which may be of a different kind than the capsule's context. Find a more elaborate explanation in subsection 2.4.1.

### A.2.3 Navigation

**Layout and Editor Components** The applications have in common that there is always an area showing a preview of the final product or even being it and giving the opportunity to directly arrange and interact with elements within this area to edit the final product. In the further analysis, we will reference this area as *stage*. The stage is usually the central part of the application and often has a prominent location in the center.

The tools for authoring the final product are implemented differently, however. HyperCard makes extensive use of the menu bar in that regard. Only the *message box*, a command line for the application, opens in its own window. It is also possible to drag the editing tools like selection, pen, and button creation out of the menu bar into a floating window. All extensive settings of elements in the project can be set in dedicated option windows.

Projects in Macromedia Flash and DaVinci Resolve are usually primarily driven by time. That is why a *timeline* next to the stage, is used to control elements of the project in a temporal and spatial dimension, as described in the introductions to 2.3.3 and 2.3.5. In Macromedia Flash, it is situated above the stage, in DaVinci Resolve below, while it can have different layouts as seen in, depending on the context it is used in, see Figure A.1, Figure A.2, and Figure A.3. The other tools are usually organized in *panels*, specialized areas for tasks like transformation, alignment, metadata, or keyframes. Macromedia Flash has a flexible panel system, allowing panels to be displayed and hidden, collapsed, resized, moved to specific regions in the User Interface (UI), and instantiated as own floating window. The panels might have

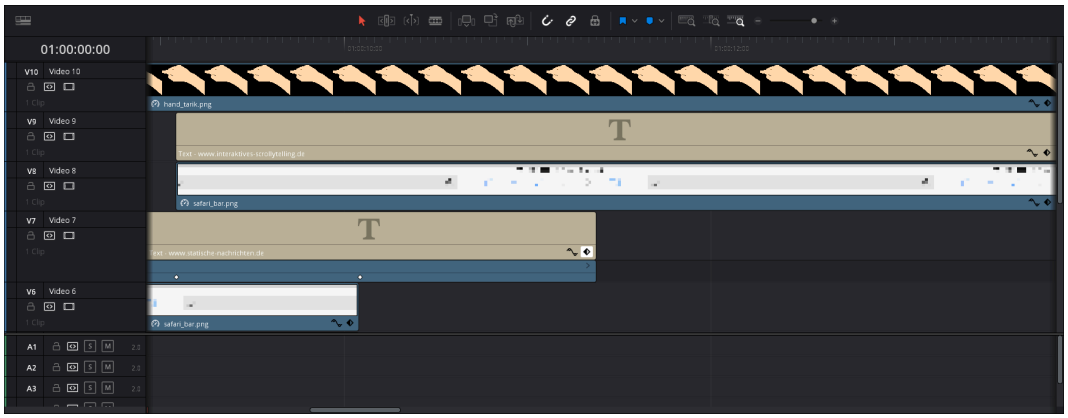


**Figure A.1:** Timeline of the *edit page* in DaVinci Resolve. It focuses on the final editing. Therefore, it displayes media in great detail and offers a large variety of trimming tools.

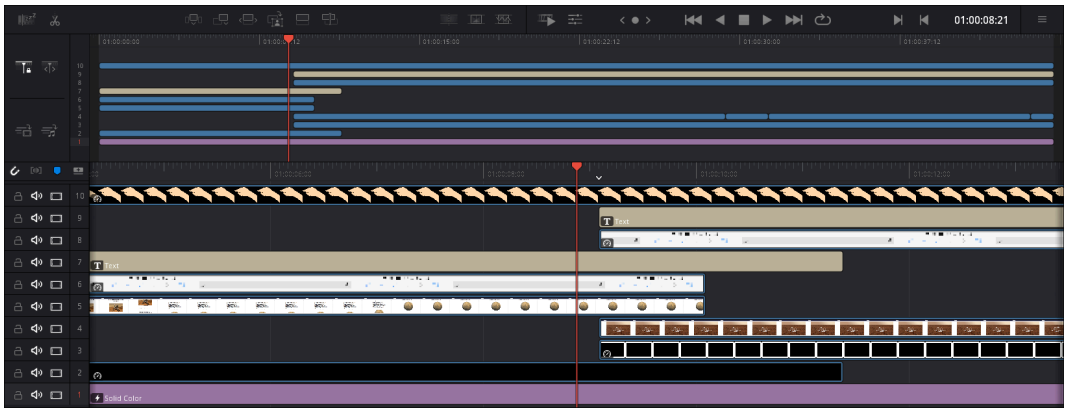


**Figure A.2:** Timeline of the *cut page* in DaVinci Resolve. It focuses on fast editing, for example for having a rough cut. Hence, it provides a good overview with the upper half showing the whole project and the lower half zoomed far out. Diverse cut presets can set automatic cuts, many buttons allow easy adjustments of the playback speed.

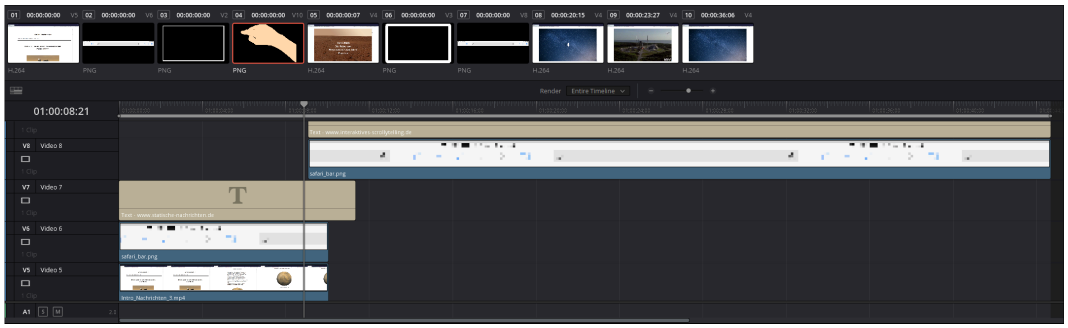


**Figure A.3:** Timeline of the deliver page in DaVinci Resolve. It is used to select the parts which should be rendered. Thus, it provides more details than Figure A.2 and lets users quickly jump to any media by clicking on the respective image at the top border.

a *pop-up* that shows all options in the panel's domain. DaVinci Resolve limits the customization more, mainly to display panels or not and resize them. Both offer an *inspector*, a panel that lists properties of the selected elements and lets the user change these. In both applications, the menu bar only houses general operations like *save* and *open*, controls the visibility of panels, and offers some basic features for authoring. PowerPoint also offers areas specialized in specific tasks but shows them in a *sidebar*. If more multiple areas are open, one of them is shown in the sidebar. The others are grouped in tabs next to it. They can also be opened in a floating window. Some of the areas give access to pop-ups with more options as well. However, most of the features and tools can be accessed via the *ribbon menu*, a tab system grouping

them by task.[110] It replaces the ordinary menu bar. The buttons within the ribbon menu have icons and are usually labeled, too. It also offers *contextual tabs*, which give access to tools depending on the selected element.

**Zoom and Movement** As described, the stage and, to an extent also the timeline are commonly used in the selected applications. Due to their importance for the final product, the users may want to work precisely within these tools. Also, depending on the content may be too large for the available area on the screen.

The applications usually solve the issue of precise work within the stage and timeline by letting the user zoom the areas' contents, much like viewing the content through a camera with a zoom lens. Content outside of the areas' boundaries' is not displayed. Mouse wheel movements can adjust the zoom, often in combination with combinations of key presses, a slider for the amount of zoom, fixed zoom level presets, the input of the exact zoom level, or specific tools which increase or decrease zoom on click or define a rectangle area which should be zoomed to. There is often a *fit to window* button that selects a zoom level that fits all content into the window. Only HyperCard does not zoom the whole content. When in paint mode, it is possible to draw while seeing the region around the pen's tip enlarged in a floating panel, much like drawing with the pen under a magnifying glass. There are also different approaches to dealing with overflowing content, which can occur due to the zoom or a simply too small display. HyperCard offers a small floating window of fixed size depending on the content size. The window represents the entire content. Within this window, a rectangle displays the region which can be seen in the stage. The region can be moved and resized, hence moving the stage's content or resizing the stage window. The other applications let the users grab and move the content directly, for example, by a grab tool that enables click and drag for the whole content or mouse wheel movements, sometimes combined with keypresses to allow for either vertical or horizontal movement. Also commonly used in the other applications: scroll bars indicate the visible part of the content and allow for horizontal and vertical movement by dragging the scroll bars.

**Capsules Navigation** Depending on how exactly the capsules[111] in the applications work, different implementations exist to navigate between the root and the capsules. The roots of HyperCard and PowerPoint organize capsules in a one-dimensional space without nested capsules. Here, the users navigate via arrow keys, buttons, the menu, key combinations, or the mouse wheel, or select from a list of capsules. In Macromedia Flash and DaVinci Resolve, capsules are arranged in two dimensions and might also be nested. Opening their capsules' contexts is usually done via a double click on the capsules. To navigate between them, a breadcrumb navigation or a tab system is used [35].

[110] Details: https://web.archive.org/web/20080104234859/http://office.microsoft.com/en-us/products/HA101679411033.aspx (last accessed on 2021-07-28).

[111] Capsule: self-contained abstraction of elements, see subsection 2.4.1

**Programming Interface** As our editor aims to facilitate the collaboration of content designers and developers, and the selected applications also support scripting, it is interesting to see the programming interfaces of these applications.

HyperCard makes a clear distinction between designing and scripting. Scripts in the scripting language *HyperTalk* can be attached to buttons, fields, cards, backgrounds, and stacks. The respective editors are only accessible via a button in the modal with more options for the element or a combination of keypresses and mouse clicks. Also, the support documents for working with HyperCard and for HyperTalk are separate, the former barely referencing the latter. PowerPoint also rather hides this feature. Macros can be attached to elements via the menu. The script opens in an editor separate from the PowerPoint instance.

Macromedia Flash makes scripting more accessible by referencing it throughout the support documents. Following Macromedia Flash's panel system, there is a panel for scripting the selected element. It includes an *assist mode* to support beginners and content designers at scripting. Some settings on elements are automatically translated to code that can be edited via this panel.

DaVinci Resolve mainly focuses on scripts for automating complex tasks, customization of the application behavior, and data exchange. Only in the fusion page, used to create visual effects, it is thought to extend the features. Hence, the *console*, used for logging and script input, is also only available as a separate window via the menu, apart from an occasional log message in the status bar at the fusion page's bottom.

### A.2.4 Project Management

**Administration and Overview** All selected applications offer a project overview. This is usually a list of recently opened projects. HyperCard offers the option to add directories whose contained projects are then displayed in an overview to be opened right out of the application itself. A similar approach follows DaVinci Resolve, but it manages all projects in an internal database rather than files on the computer's file system. Hence, its project overview is the canonical way to open and administrate projects.

**Opening Projects** All applications support opening multiple projects at a time. In Macromedia Flash and DaVinci Resolve, it is only possible to have one instance of the application running, which houses all opened projects in a tab system, multiple windows, or a drop-down menu to switch between them. PowerPoint opens each project in its own instance. Macromedia Flash, as well as PowerPoint, has the ability to open one project multiple times. Changes in the project will be synced between all instances. The HyperCard editor is mainly integrated into the menu bar of Mac OS 9. Every project opens in its own window, and if this window is focused, the editor options become available through the menu bar. At the same time, a project can only be opened once at a time.

**Saving Projects** In HyperCard, every action directly affects the project file. In a sense, all changes are saved automatically. It is still possible to save a copy of the opened project, which is even the anticipated way of creating working project versions. All the other applications write changes to the file only on manual saves but can automatically save after a specific period or immediately. PowerPoint offers the auto-save only for projects opened from OneDrive or SharePoint, both cloud services of Microsoft, but can create recovery files during the work.

### A.2.5 Composition

**Content Creation** The applications have different approaches when it comes to the creation of content elements. DaVinci Resolve utilizes the *media pool* to import and administrate all media within the project – videos, audios, images. From there, it can be dragged into the timeline of the project. It is also possible to drag media from the file explorer right into the application. Elements created within DaVinci and arranged on the timeline, like texts and some effects, can be dragged from respective panels. Further elements administrated in special panels, like nodes on the fusion page, can be created using dedicated toolbars within these panels.

PowerPoint uses the *insert tab* in the ribbon menu as a central source for all creatable elements. The tab accommodates buttons for the different elements. For drawn content, like shapes, the respective form tool can be selected to draw the shape directly on the stage. Special objects like images, videos, and diagrams are selected and roughly designed in specialized separate windows before insertion and later refinements. Other special objects, for example, forms and comments, are created within a sidebar. Furthermore, specific presets of WordArts and Terms are directly inserted after selection and can then be finetuned. Often used insertion buttons are also available in the *start tab*.

Macromedia Flash's insertion method is not as streamlined. Artwork is imported via the menu and a dialog or created via the drawing and painting tools from the sidebar. The tool sidebar can also be used to add text. Other elements like videos and audio can be imported via the menu bar. Symbols are created via a different place in the menu bar. Moreover, components like buttons, lists, or checkboxes can be dragged into the stage via a panel that has to be displayed first.

HyperCard has two main methods to insert elements. Elements like buttons or fields can be added via the menu bar. The tools for drawable elements – shapes, pens, or text – can be selected either from their menu bar tab or the dedicated tool window and then be drawn onto the stage.

**Position (2D)** Positioning elements work similarly in all applications. They can always be dragged on the stage. The movement can be restricted to one direction – horizontally, vertically, sometimes at an angle – by simultaneously pressing the *Shift*-key. Except for HyperCard, all applications also offered number fields to precisely input a position via x and y coordinates. Macromedia Flash and PowerPoint also offered further transform features for relative alignments, like centering. In DaVinci Resolve, the position is integrated into the *inspector*, a panel for streamlined and

consistent adjustment of different element settings, which offers a central place for settings consistent behavior for the input fields. For example, it is possible to click and drag a number field to increase or decrease the value.

**Size** Like with positioning, all applications allow for resizing via mouse dragging directly within the stage. While selected buttons and fields in HyperCard, selected via the button or field tool, can always be resized by dragging a corner of the selection, Macromedia Flash requires using a special resize tool on some elements and allows resizing on selection for other elements. In contrast to HyperCard, Macromedia Flash shows knobs to indicate when and where resizing is possible. Likewise do PowerPoint and DaVinci Resolve, whereby PowerPoint shows them for every selection like HyperCard, while DaVinci Resolve's stage needs to be in the drag mode, and the selection is made within the elements in the timeline. Macromedia Flash, PowerPoint, and DaVinci Resolve also show number fields to precisely set the selection's size. Macromedia Flash shows these fields in two different locations; PowerPoint and DaVinci Resolve stick to their sidebar, respectively the inspector system.

**Rotation** HyperCard only allows for the rotation of selected drawn content, not of buttons or fields. The rotation mode is activated via the menu bar and shows a grabber at each edge to rotate. Selections are always rotated around their center. PowerPoint limits rotations also to the selection center but shows a dedicated grabber at every selection. Rotating in DaVinci Resolve works like their resizing, and the anchor point for the rotation can be adjusted. Macromedia Flash again has a separate tool for this function, also with an adjustable rotation anchor. All applications except HyperCards allow a precise rotation via number fields.

**Order (level)** HyperCard has a background and a foreground. Within each level, there is a paint layer for drawings always behind an object layer for elements like buttons and fields. The objects can also be arranged relative to each other within the object layer. Macromedia Flash uses the layers in the timeline for ordering. There can be multiple elements within one layer. These can also be ordered relative to each other within the layer. DaVinci Resolve only allows at each point on the timeline only one element, making the layer order in the timeline the only way to order elements. The only exception is fusion with special nodes like the *merge node*, which can mix two elements to one, thereby putting one element behind the other. PowerPoint has one space for all elements within the slide. They are all ordered relative to each other. Additionally to the typical menu items to bring elements closer or to send them farther, it offers a list of all slide's elements, whose order determines the order within the slide.

**Grouping** Grouping makes multiple elements act like one, especially when it comes to selections and transforming. While HyperCard does not provide such a feature, Macromedia Flash and PowerPoint do. A group there is created by transforming currently selected elements into a group. The group can be entered and

disbanded likewise. DaVinci Resolve allows for linking multiple elements, resulting in the simultaneous selection of linked elements if the respective mode is activated, and thereby allowing for transformations like dragging in the timeline at the same time.

**Reusability** Some applications offer the repetitive usage of a created or imported element. A change in one instance applies to all other instances, too. Macromedia Flash's symbol system supports this. A *Flash Symbol* is an artwork converted to a reusable asset. DaVinci Resolve allows for the repeated usage of any media of the media pool. To also reuse media combined with specific effects active, like cropping, the media can be wrapped in a nested timeline or compound clip, some kinds of capsules, which in turn can be reused. PowerPoint does not support the reuse of elements but allows for reusing slide backgrounds and layouts via *master layouts* for slides. To still quickly apply changes to multiple spots, it allows settings to be applied to all via a button press in selected cases, like transitions.

### A.2.6 Configuration

The configuration of elements in HyperCard is done via *info dialogs* specifically designed for every element type. Commands for drawing, like reverting colors, are available via the menu bar. The same applies to font-related styling of elements, but these are also accessible via the info dialogs. As HyperCard, Macromedia Flash does not offer a streamlined interface for element configuration. It uses use-case-related panels instead. A panel enables all its input boxes that are applicable for the selected elements and disables the others. Some panels show parameters dynamically depending on the elements. The panels offer buttons for some commands; others are located in the menu bar. PowerPoint allows the configuration via the sidebar or windows specialized on certain element types. The displayed settings in the sidebar are adapted to the selected element; no disabled input boxes are visible. The ribbon menu offers the buttons for commands like mirroring. As mentioned, DaVinci Resolve relies heavily on the inspector, which displays all available settings of the selected elements. These are grouped in tabs by element type, like video, audio, or transition. Within the tabs, they are ordered in use-case-related panels. The input boxes for the settings are consistent for the settings type. Numeric values, for example, always have a slider next to the input field with the precise numeric value, which increases or decreases the value on mouse drag. The panels also offer command buttons and other inputs.

### A.2.7 Animation

There are different approaches when it comes to animations. Note, animations in this context are property value changes of an element over time. They may not be confused with animations. Based on this definition, HyperCard does not support animations. Macromedia Flash uses *keyframes* and *tweens* to control animations. A keyframe is a snapshot of all elements in the keyframe's layer and their settings.

To animate *in-between* these snapshots, tweens can be attached to the keyframes. Depending on what the animation should focus on, there are *shape tweens* and *motion tweens*. Motion tweens cannot be applied to keyframes that contain multiple elements. By default, tweens are linear and apply to all properties of the element. Nevertheless, it is possible to ease-in or ease-out the tween by an absolute period. It is also possible to set a custom ease function relative to the tween duration and limit the tween to one property. The tween directly reflects property changes of elements within a keyframe. If the change occurs between two tweened keyframes, a new keyframe is directly inserted. During a motion tween, elements can also follow a *motion path*, basically any line drawn in a linked guide layer. PowerPoint does not offer animation via keyframes. Instead, specific animations, like *fade*, *jump*, or *move out*, are selected from presets in the ribbon menu and attached to elements, and customized via the ribbon menu. Path animations, like *line*, *curve*, or *loop*, can be edited directly on the stage. The animations can have different triggers and be executed in a fixed order. Both can be administered per element via the ribbon menu or for all animations on the slide via the sidebar. An animation dialog contains all available features, for example more triggers. In DaVinci Resolve, almost every parameter is animatable. This is done with keyframes, similar to Macromedia Flash. But these keyframes only apply to one parameter; they define the value of one parameter at a specific frame, the values in between are computed. Being related so much to the parameters, the keyframe controls are directly integrated next to each parameter setting in the inspector. One button to set the keyframe, which also indicates via color via a keyframe, was already set at this frame, and two arrows letting the users jump to the closest keyframe before and after. It is also possible to select an easing preset via a context menu. To have complete control over the easing of keyframes, the timeline offers a view in which the easing between keyframes can be edited with bezier curves in absolute values. Once the value of a parameter at a keyframe changes, the keyframe uses this new value, but this does not create new keyframes between existing keyframes, unlike Macromedia Flash. If the position of an element is animated, it is possible to show and edit the path of the element on the stage.

The fusion page of DaVinci Resolve also offers an entirely different way to animate. Fusion works a lot with nodes that apply specific atomic effects on the affected media, creating custom effects by connecting nodes with each other. This page offers a keyframe editor for animating the parameters affected by the nodes. The editor shows one layer per node and effect, color-coded according to the node color. Keyframes are represented by vertical lines, an animation by a translucent container. If the layer of a node is collapsed, the keyframes are superimposed onto the layer. Otherwise, they are shown in their respective parameter layer indented below the node layer. With these keyframes, the timing of the animation can be finetuned. A spreadsheet offers another view of the keyframes, allowing for changes of the time and property value.

### A.2.8 Transition

Transitions are changeovers from one capsule to another, depending on the application also from one element to another. Macromedia Flash does not support transitions between capsules, apart from when working with screens in the professional version which may not be of greater importance in this context and are comparable to HyperCards and PowerPoint. These two applications, however, rely heavily on transitions. By default, their capsules, being cards in HyperCard and slides in PowerPoint, are instant. Still, both offer a multitude of different transition presets, whereby PowerPoint outnumbers HyperCard. While PowerPoint applies the transition to a slide and offers transition parameters via the ribbon menu as well as a preview, transitions in HyperCard are attached to the button that links to the next slide, making it possible to enter a card with different transitions. The only parameter for HyperCard transitions is the selection of five speed categories. DaVinci Resolve does support transitions between all kinds of media. They can be selected from a wide variety of panels, previewed by hovering and dragged, and dropped onto the elements. Their duration can be edited in the timeline like other media durations and otherwise customized via the inspector. It is also possible to customize the transition progress via bezier curves. While it supports the creation of custom presets, PowerPoint offers the *apply to all* button to speed up the customization.

# B Appendix Chapter 3

## B.1 Code

**Listing B.1:** Configuration that automatically starts an editor in the world when none is opened.

```
import { promise } from 'lively.lang';

(async () => {
  await promise.waitFor(30000, () => $world.get('lively top bar'))
  let qinoq = await System.import('qinoq');
    if(!($world.get('interactives editor')))
        await new qinoq.InteractivesEditor().initialize();
})();
```

## B.2 Figures

Here are some screenshots of tools mentioned in subsection 3.1.2 which were not further explained there.

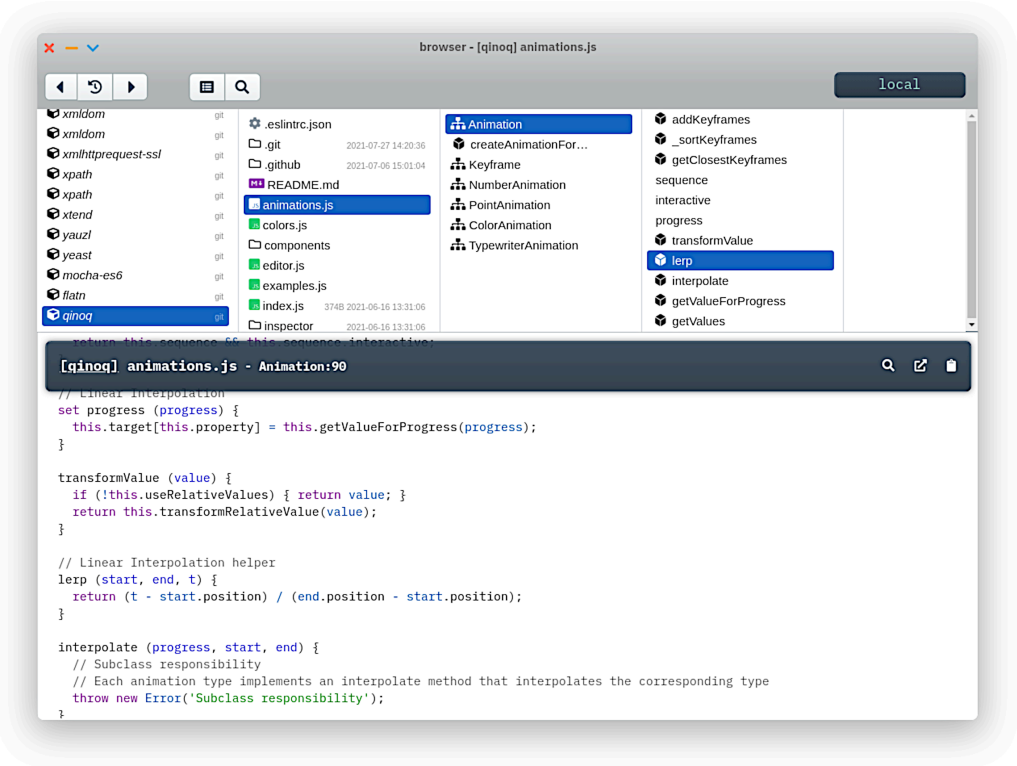


**Figure B.1:** Browser opened on the animation.js file in the qinoq package.

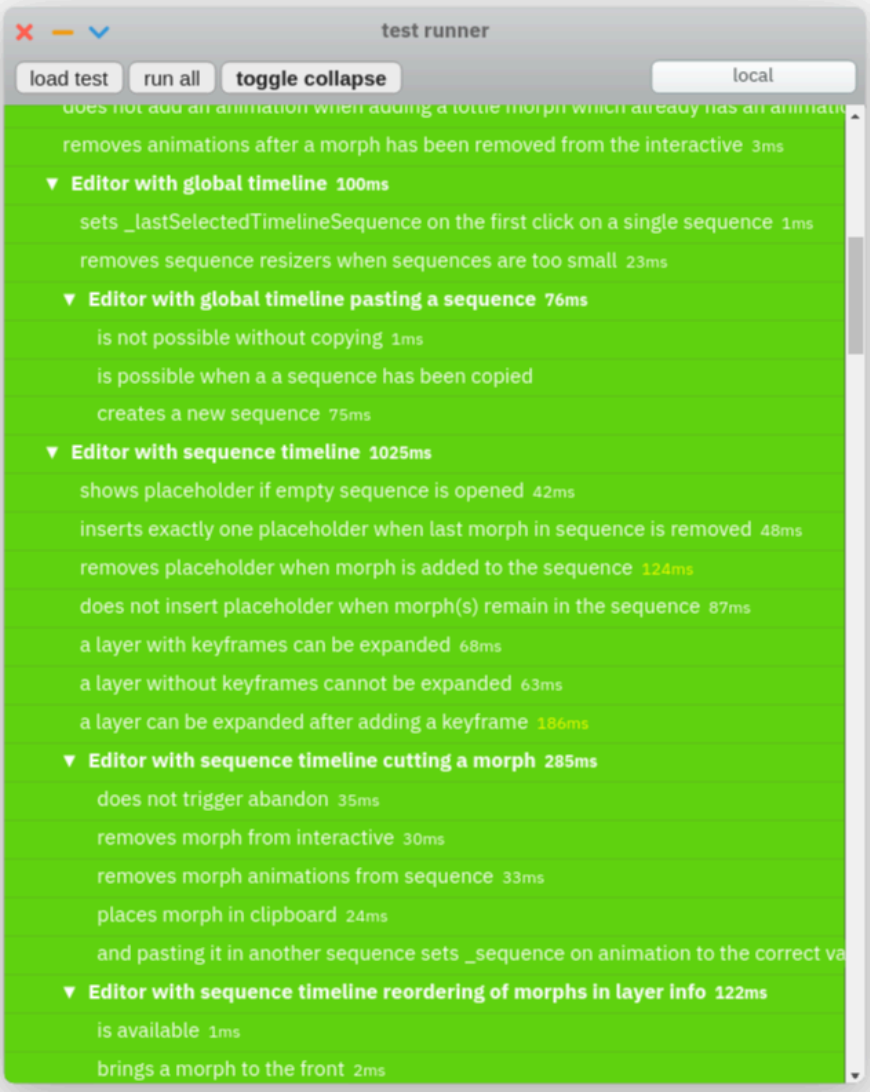


**Figure B.2:** Test runner after a successful run through qinoq's test suite.

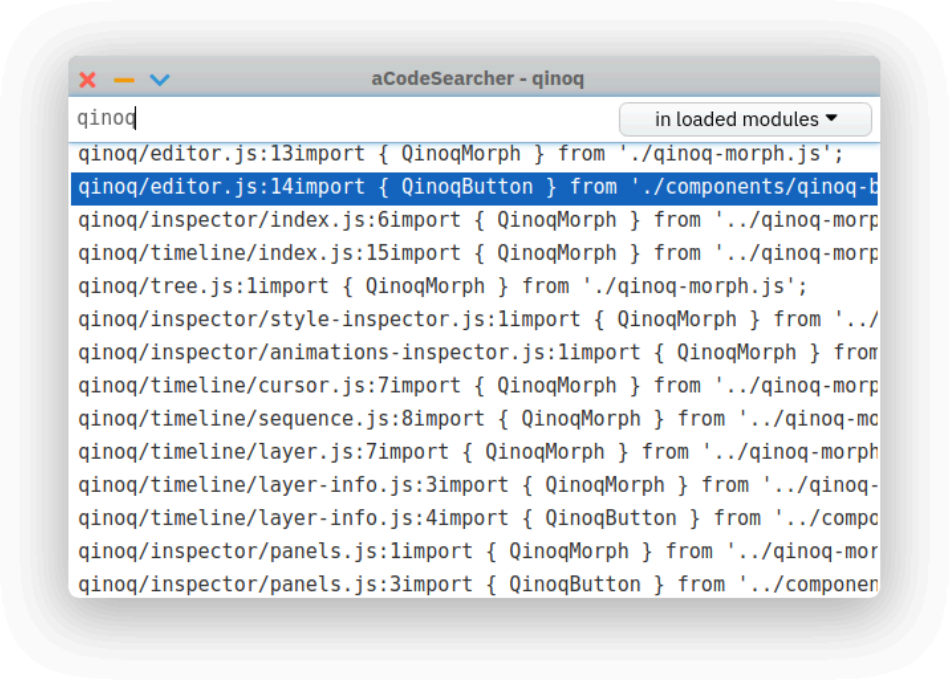


**Figure B.3:** Code search utility showing results after running a search for ”qinoq”.

# C Appendix Chapter 4

## C.1 Code

The code of the prototypical Web Animations integration in qinoq can also be found in the qinoq repository[112].

**Listing C.1:** Web Animations API based animation implementation conforming to the qinoq interface for animations.

```
export class WebAnimation {
  static usesTransform (prop) {
    return [
      'position',
      'scale'
    ].includes(prop);
  }
  constructor (targetMorph, property) {
    this.target = targetMorph;
    this.property = property;
    this.keyframes = [];
    this.webAnimation = null;
  }
  // Accepts EXACTLY two keyframes, one for the beginning and one
  // for the end of the animation.
  // Provide Keyframes in the correct order.
  // https://drafts.csswg.org/web-animations-1/#keyframes-section
  addKeyframes (keyframes) {
    this.keyframes = keyframes;
    this._keyframes = this.generateCSSKeyframes();
    this._keyframes.forEach((kf, i) => {
      kf.offset = this.keyframes[i].position;
    });
    // Make explicit duration and duration implied through offset
    // of keyframe position equivalent
    if (this.keyframes[1].position != 1) {
      this._keyframes.push(JSON.parse(JSON.stringify(this.
          _keyframes[1])));
      this._keyframes[2].offset = 1;
    }
  }

```

[112]https://github.com/hpi-swa-lab/qinoq/tree/spike/web-animations (last accessed on 2021-07-28).

```
  generateCSSKeyframes () {
    switch (this.property) {
      case 'position':
        const xOffset = this.keyframes[1].value.x
          - this.keyframes[0].value.x;
        const yOffset = this.keyframes[1].value.y
          - this.keyframes[0].value.y;
        return [
          { transform: 'translate(0px,0px)' },
          { transform: `translate(${xOffset}px,${yOffset}px)` }
        ];
      case 'scale':
        return [
          { transform: `scale(${this.keyframes[0].value})` },
          { transform: `scale(${this.keyframes[1].value})` }
        ];
      case 'fill':
        const c1 = this.keyframes[0].value;
        const c2 = this.keyframes[1].value;
        return [
          { backgroundColor: `${this.keyframes[0].value}` },
          { backgroundColor: `${this.keyframes[1].value}` }
        ];
      default:
        throw 'Not yet implemented.';
    }
  }

  set progress (progress) {
    this.targetNode =
      this.target.env.renderer.getNodeForMorph(this.target);
    if (!this.targetNode) return;
    if (!this.webAnimation) {
      const timingOptions = {
        fill: 'forwards',
        duration: 100
      };
      if (WebAnimation.usesTransform(this.property)) {
        // combine effects that rely on the same CSS property
        timingOptions.composite = 'add';
      }
      this.webAnimation = this.targetNode.animate(
        this._keyframes,
        timingOptions
      );
      this.webAnimation.pause();
    }
    this.webAnimation.currentTime = progress * 100;
  }
}
```

**Listing C.2:** A square with animated fill scale and position utilizing WebAnimations in qinoq.

```
import { Scrollytelling, Keyframe, Sequence, Layer } from "qinoq";
import { Morph } from "lively.morphic";
import { Color, pt } from "lively.graphics";
import { WebAnimation } from "qinoq/web-animations.js";

const scrollytelling = new Scrollytelling()
const exampleLayer = new Layer()
const exampleSequence = new Sequence({
  name: 'example', start: 0, duration: 500})
const demoMorph = new Morph({fill: Color.rgbHex('FFFF00'),
  extent: pt(30,30), position: pt(20,20)})
exampleSequence.addMorph(demoMorph)
exampleSequence.layer = exampleLayer
scrollytelling.addLayer(exampleLayer)
scrollytelling.addSequence(exampleSequence)

let kf1, kf2
const demoPositionAnimation =
  new WebAnimation(demoMorph, 'position')
kf1 = new Keyframe(0.5, pt(20,20))
kf2 = new Keyframe(1, pt(100,100))
demoPositionAnimation.addKeyframes([kf1, kf2])
exampleSequence.addAnimation(demoPositionAnimation)

const demoFillAnimation = new WebAnimation(demoMorph, 'fill')
kf1 = new Keyframe(0, '#FFFF00')
kf2 = new Keyframe(0.5, '#ff0000')
demoFillAnimation.addKeyframes([kf1, kf2])
exampleSequence.addAnimation(demoFillAnimation)

const demoScaleAnimation = new WebAnimation(demoMorph, 'scale')
kf1 = new Keyframe(0.5, 1)
kf2 = new Keyframe(0.7, 5)
demoScaleAnimation.addKeyframes([kf1, kf2])
exampleSequence.addAnimation(demoScaleAnimation)

scrollytelling.openInWorld()
```

**Listing C.3:** A square with animated fill scale and position utilizing qinoq animations.

```
import { Scrollytelling, NumberAnimation, ColorAnimation,
   PointAnimation, Keyframe, Sequence, Layer } from "qinoq";
import { Morph } from "lively.morphic";
import { Color, pt } from "lively.graphics";
const scrollytelling = new Scrollytelling()

const exampleLayer = new Layer()

const exampleSequence = new Sequence({name: 'example', start: 0,
   duration: 500})

const demoMorph = new Morph({fill: Color.rgbHex('FFFF00'), extent:
   pt(30,30), position: pt(20,20)})
exampleSequence.addMorph(demoMorph)
exampleSequence.layer = exampleLayer
scrollytelling.addLayer(exampleLayer)
scrollytelling.addSequence(exampleSequence)

let kf1, kf2
const demoPositionAnimation = new PointAnimation(demoMorph, '
   position')
kf1 = new Keyframe(0.5, pt(20,20), {easing: 'linear'})
kf2 = new Keyframe(1, pt(100,100), {easing: 'linear'})
demoPositionAnimation.addKeyframes([kf1, kf2])
exampleSequence.addAnimation(demoPositionAnimation)

const demoFillAnimation = new ColorAnimation(demoMorph, 'fill')
kf1 = new Keyframe(0, "#FFFF00", {easing: 'linear'})
kf2 = new Keyframe(0.5, '#ff0000', {easing: 'linear'})
demoFillAnimation.addKeyframes([kf1, kf2])
exampleSequence.addAnimation(demoFillAnimation)

const demoScaleAnimation = new NumberAnimation(demoMorph, 'scale')
kf1 = new Keyframe(0.5, 1, {easing: 'linear'})
kf2 = new Keyframe(0.7, 5, {easing: 'linear'})
demoScaleAnimation.addKeyframes([kf1, kf2])
exampleSequence.addAnimation(demoScaleAnimation)

scrollytelling.openInWorld()
```

## C.2 Figures

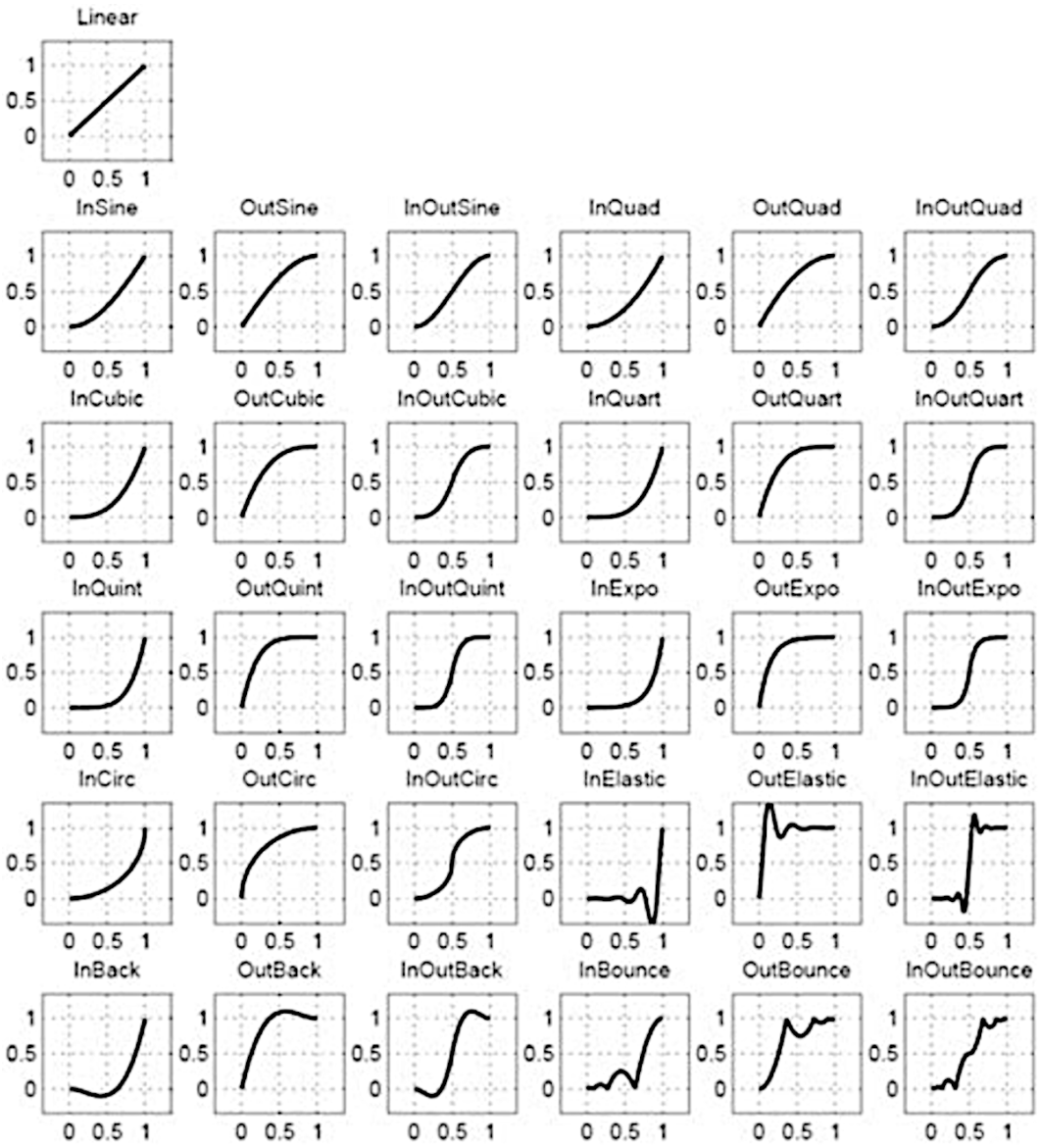


**Figure C.1:** Penner's easing functions [50]. Figure from [28].

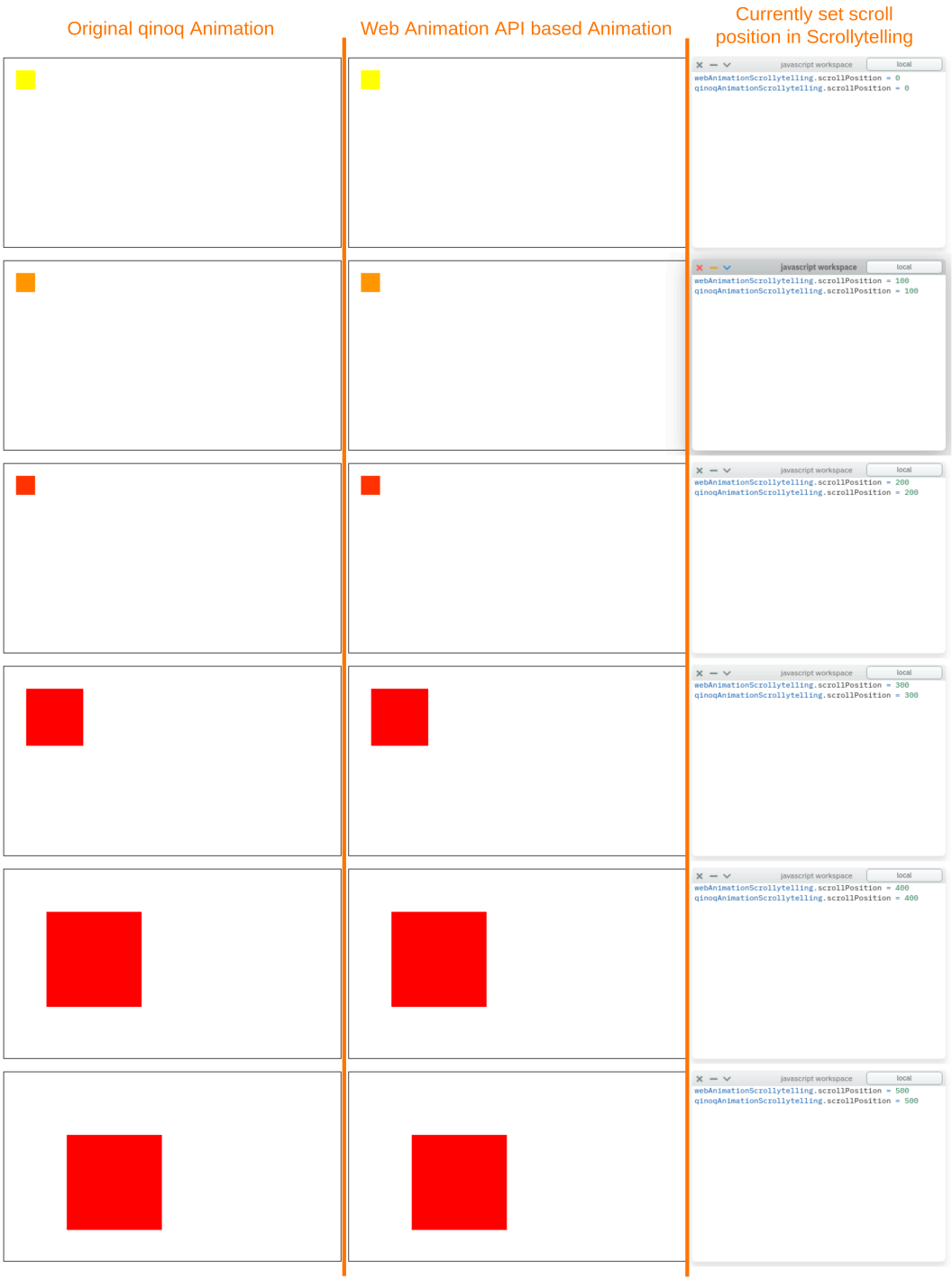


**Figure C.2:** Comparison between a demo animation utilizing the default qinoq animation implementation and one using the qinoq WAAPI wrapper at different points during the animations.

# Bibliography


[1] I. Aderinokun. *Understanding the Virtual DOM*. 2018. URL: https://bitsofco.de/understanding-the-virtual-dom/ (visited on 2021-07-22).

[2] R. M. Baecker. "Picture-Driven Animation". In: *AFIPS '69: Proceedings of the Spring Joint Computer Conference*. ACM, 1969.

[3] B. Birtles. *Animating Like you Just Don't Care with Element.animate – Mozilla Hacks - the Web Developer Blog*. 2016. URL: https://hacks.mozilla.org/2016/08/animating-like-you-just-dont-care-with-element-animate (visited on 2021-06-30).

[4] B. Birtles, R. Flack, S. McGruer, and A. Quint. *Web Animations – Editor's Draft*. 2021. URL: https://drafts.csswg.org/web-animations-1/ (visited on 2021-07-18).

[5] J. Branch. *Snow Fall: The Avalanche at Tunnel Creek*. Dec. 2012. URL: https://www.nytimes.com/projects/2012/snow-fall/ (visited on 2021-06-25).

[6] B.-W. Chang and D. Ungar. "Animation: from Cartoons to the User Interface". In: *Proceedings of the 6th annual ACM symposium on User interface software and technology – UIST '93*. ACM, 1993.

[7] D. Chęć and Z. Nowak. "The Performance Analysis of Web Applications Based on Virtual DOM and Reactive User Interfaces". In: *Engineering Software Systems: Research and Praxis*. Edited by P. Kosiuczenko and Z. Zieliński. Advances in Intelligent Systems and Computing. Springer, 2019.

[8] K. Chinnathambi. *DOM vs. Canvas*. Oct. 28, 2015. URL: https://www.kirupa.com/html5/dom_vs_canvas.htm (visited on 2021-07-23).

[9] M. Conlen and J. Heer. "Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web". In: *UIST '18: Symposium on User Interface Software and Technology*. ACM.

[10] F. Copes. *The requestAnimationFrame() Guide*. 2018. URL: https://flaviocopes.com/requestanimationframe/ (visited on 2021-07-16).

[11] D. Dowling and T. Vogan. "Can We 'Snowfall' This?" In: *Digital Journalism* 3.2 (Mar. 4, 2015). eprint: https://doi.org/10.1080/21670811.2014.930250.

[12] *Download Macromedia Flash 8 8.0*. Repository for Macromedia Flash 8. 2013. (Visited on 2021-07-11).

[13] S. Eiserloh. *Math for Game Programmers: Fast and Funky 1D Nonlinear Transformations*. 2018. URL: https://www.youtube.com/watch?v=mr5xkf6zSzk (visited on 2021-06-27).

[14] D. Flanagan. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language*. O'Reilly Media, 2020.

[15] T. Garrand. *Writing for Multimedia and the Web: A Practical Guide to Content Development for Interactive Media*. CRC Press, Oct. 14, 2020.

[16] I. M. Greca and M. A. Moreira. "Mental Models, Conceptual models, and Modelling". In: *International Journal of Science Education* 22.1 (2000).

[17] L. Green. *Communication, Technology and Society*. NSW: Allen & Unwin, 2002.

[18] M. C. Green, J. J. Strange, and T. C. Brock. *Narrative Impact: Social and Cognitive Foundations*. Taylor & Francis, 2003.

[19] R. Hagström. "Frames That Matter". Bachelor's Thesis. Uppsala Universitet, 2015.

[20] P. Hanrahan. *Interpolation and Basis Fns. CS 148 Lecture 7*. Stanford University, 2009. URL: http://graphics.stanford.edu/courses/cs148-09/lectures/interpolation.pdf (visited on 2021-07-24).

[21] F. Hohman, M. Conlen, J. Heer, and D. H. P. Chau. "Communicating with Interactive Articles". In: *Distill* 5.9 (2020). URL: https://distill.pub/2020/communicating-with-interactive-articles.

[22] *HyperCard 2.4 – Macintosh Repository*. Repository for HyperCard 2.4.1. 2018. URL: https://www.macintoshrepository.org/2632-hypercard-2-4 (visited on 2021-07-11).

[23] *HyperCard 2.4.1*. Support material: HyperCard Help and HyperTalk Reference. 1998. (Visited on 2021-07-11).

[24] D. Ingalls, T. Felgentreff, R. Hirschfeld, R. Krahn, J. Lincke, M. Röder, A. Taivalsaari, and T. Mikkonen. "A World of Active Objects for Work and Play: The First Ten Years of Lively". In: *Onward! 2016: International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2016.

[25] P. Irish. *Profiling Long Paint Times with DevTools' Continuous Painting Mode*. 2020. URL: https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode (visited on 2021-07-25).

[26] P. Irish. *requestAnimationFrame for Smart Animating*. 2011. URL: https://www.paulirish.com/2011/requestanimationframe-for-smart-animating/ (visited on 2021-07-20).

[27] Ł. Izdebski, R. Kopiecki, and D. Sawicki. "Bézier Curve as a Generalization of the Easing Function in Computer Animation". In: *Advances in Computer Graphics*. Edited by N. Magnenat-Thalmann, C. Stephanidis, E. Wu, D. Thalmann, B. Sheng, J. Kim, G. Papagiannakis, and M. Gavrilova. Lecture Notes in Computer Science. Springer, 2020.

[28] Ł. Izdebski and D. Sawicki. "Easing Functions in the New Form Based on Bézier Curves". In: *Computer Vision and Graphics*. Edited by L. J. Chmielewski, A. Datta, R. Kozera, and K. Wojciechowski. Lecture Notes in Computer Science. Springer, 2016.

[29] M. Kamermans. *A Primer on Bézier Curves*. 2013. URL: https://pomax.github.io/bezierinfo (visited on 2021-06-27).

[30] N. Kitroeff, M. Abi-Habib, J. Glanz, O. Lopez, W. Cai, E. Grothjan, M. Peyton, and A. Cegarra. *Why the Mexico City Metro Collapsed*. June 12, 2021. URL: https://www.nytimes.com/interactive/2021/06/12/world/americas/mexico-city-train-crash.html (visited on 2021-06-25).

[31] S. Kobes. *Life of a Pixel*. 2019. URL: https://www.youtube.com/watch?v=m-J-tbAlFic (visited on 2021-07-07).

[32] M. Kosaka. *Inside look at modern web browser (part 3)*. 2020. URL: https://developers.google.com/web/updates/2018/09/inside-browser-part3 (visited on 2021-07-07).

[33] J. Lasseter. "Principles of Traditional Animation Applied to 3D Computer Animation". In: *SIGGRAPH Comput. Graph.* 21.4 (1987).

[34] K. Leander. *HyperCard Forgotten, but Not Gone*. 2008. (Visited on 2021-07-19).

[35] M. Levene. *An Introduction to Search Engines and Web Navigation*. John Wiley & Sons, 2011.

[36] P. Lewis. *Rendering Performance*. 2019. URL: https://developers.google.com/web/fundamentals/performance/rendering (visited on 2021-07-16).

[37] P. Lewis. *Simplify Paint Complexity and Reduce Paint Areas | Web Fundamentals*. 2019. URL: https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas (visited on 2021-07-23).

[38] J. Lincke. "Evolving Tools in a Collaborative Self-supporting Development Environment". PhD thesis. University of Potsdam, Germany, 2014. URL: http://d-nb.info/1112206698.

[39] K. Lischka. "Hypercard: Apples Offline-Browser wird 25". In: *Der Spiegel* (2012).

[40] M. Lombard and T. Ditton. "At the Heart of It All: The Concept of Presence". In: *Journal of Computer-Mediated Communication* 3.2 (Sept. 1, 1997). URL: https://doi.org/10.1111/j.1083-6101.1997.tb00072.x.

[41] M. Lombard, T. B. Ditton, and L. Weinstein. "Measuring Presence: The Temple Presence Inventory". In: Proceedings of the International Society for Presence Research Annual Conference, Los Angeles, California, USA (Nov. 2009).

[42] *Macromedia Flash Professional 8*. Support material: Flash Help dialog. 2005. (Visited on 2021-07-11).

[43] A. Maio. *What is Animation? Definition and Types of Animation*. 2020. URL: https://www.studiobinder.com/blog/what-is-animation-definition/ (visited on 2021-07-18).

[44] J. Maloney. *Morphic: The Self User Interface Framework*. Sun Microsystems, Inc. and Stanford University, 1995. URL: https://ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf.

[45] E. Mendelson. *Mac OS 9 for Windows*. Instructions to run Mac OS 9 in Windows using the SheepShaver Emulator. (Visited on 2021-07-11).

[46] *Microsoft PowerPoint für Microsoft 365 MSO*. Support material: integrated help sidebar which uses online documentation. (Visited on 2021-05-22).

[47] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools". In: *Computer* 49.7 (2016).

[48] C. Nuernbergk and C. Neuberger, editors. *Journalismus im Internet: Profession – Partizipation – Technisierung*. Springer, 2018.

[49] O. Özcan and L. Akarun. "Teaching Interactive Media Design". In: *International Journal of Technology and Design Education* 12.2 (May 1, 2002).

[50] R. Penner. *Robert Penner's Programming Macromedia Flash MX*. McGraw-Hill/OsborneMedia, 2002.

[51] G. Petty. *DaVinci Resolve 17 Reference Manual*. Download at software installation, accessible from within the software. 2021. (Visited on 2021-07-15).

[52] A. Quint. *Web Animations in WebKit*. 2018. URL: https://webkit.org/blog/8343/web-animations-in-webkit/ (visited on 2021-07-18).

[53] D. W. Sandberg. "Smalltalk and Exploratory Programming". In: *ACM SIGPLAN Notices* 23.10 (1988).

[54] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-computer Interaction*. Addison-Wesley, 2010.

[55] M. Slater. "Measuring Presence: A Response to the Witmer and Singer Presence Questionnaire". In: *Presence: Teleoperators and Virtual Environments* 8.5 (Oct. 1999).

[56] S. S. Sundar. "Social Psychology of Interactivity in Human-website Interaction". In: *Oxford Handbook of Internet Psychology* (Sept. 18, 2012). Publisher: Oxford University Press.

[57] D. Surma. *The Main Thread is Overworked & Underpaid*. 2019. URL: https://www.youtube.com/watch?v=7Rrv9qFMWNM (visited on 2021-07-07).

[58] S. C. L. Terra and R. A. Metoyer. "Performance Timing for Keyframe Animation". In: *SCA '04: SIGGRAPH/Eurographics symposium on Computer animation*. ACM, 2004.

[59] L. Thorlacius. "The Role of Aesthetics in Web Design". In: *Nordicom Review* 28.1 (2007).

[60] J. Trenouth. "A Survey of Exploratory Software Development". In: *The Computer Journal* 34.2 (1991).

[61] J. Verwey and E. Blake. "The Influence of Lip Animation on the Perception of Speech in Virtual Environments". In: International Society for Presence Research, 2005.

[62] B. Wells. "Frame of Reference: Toward a Definition of Animation". In: *Animation Practice, Process & Production* 1.1 (2011).

[63] B. G. Witmer and M. J. Singer. "Measuring Presence in Virtual Environments: A Presence Questionnaire". In: *Presence: Teleoperators and Virtual Environments* 7.3 (June 1998).

[64] C. Wolf and A. Godulla. "Potentials of Digital Longforms in Journalism. A Survey among Mobile Internet Users about the Relevance of Online Devices, Internet-specific Qualities, and Modes of Payment". In: *Journal of Media Business Studies* 4 (2016).

# List of Figures

# Aktuelle Technische Berichte des Hasso-Plattner-Instituts

| Band | ISBN | Titel | Autoren/Redaktion |
|---|---|---|---|
| 140 | 978-3-86956-517-0 | **Probabilistic metric temporal graph logic** | Sven Schneider, Maria Maximova, Holger Giese |
| 139 | 978-3-86956-514-9 | **Deep learning for computer vision in the art domain : proceedings of the master seminar on practical introduction to deep learning for computer vision, HPI WS 20/21** | Christian Bartz, Ralf Krestel |
| 138 | 978-3-86956-513-2 | **Proceedings of the HPI research school on service-oriented systems engineering 2020 Fall Retreat** | Christoph Meinel, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Erwin Böttinger, Christoph Lippert, Christian Dörr, Anja Lehmann, Bernhard Renard, Tilmann Rabl, Falk Uebernickel, Bert Arnrich, Katharina Hölzle |
| 137 | 978-3-86956-505-7 | **Language and tool support for 3D crochet patterns : virtual crochet with a graph structure** | Klara Seitz, Jens Lincke, Patrick Rein, Robert Hirschfeld |
| 136 | 978-3-86956-504-0 | **An individual-centered approach to visualize people's opinions and demographic information** | Wanda Baltzer, Theresa Hradilak, Lara Pfennigschmidt, Luc Maurice Prestin, Moritz Spranger, Simon Stadlinger, Leo Wendt, Jens Lincke, Patrick Rein, Luke Church, Robert Hirschfeld |
| 135 | 978-3-86956-503-3 | **Fast packrat parsing in a live programming environment : improving left-recursion in parsing expression grammars** | Friedrich Schöne, Patrick Rein, Robert Hirschfeld |
| 134 | 978-3-86956-502-6 | **Interval probabilistic timed graph transformation systems** | Maria Maximova, Sven Schneider, Holger Giese |
| 133 | 978-3-86956-501-9 | **Compositional analysis of probabilistic timed graph transformation systems** | Maria Maximova, Sven Schneider, Holger Giese |

ISBN 978-3-86956-521-7
ISSN 1613-5652