TU WIEN Informatics

Animating IPE-Presentations

A roundtrip-capable implementation based on manim

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Anna Henriksson
Registration Number 11905162

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Martin Nöllenburg
Assistance: Dr. Simon Dominik Fink

Vienna, March 8, 2025

Anna Henriksson | Martin Nöllenburg

Erklärung zur Verfassung der Arbeit

Anna Henriksson

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 8. März 2025

Anna Henriksson

Abstract

Presentations are a fundamental part of communicating work progress and results to an audience. This significance can be seen in the vast amount of different programs that can be used to create them. One feature that is present in one of the most generic programs for creating presentations (PowerPoint) is the ability to animate presentations. More specialized programs, such as IPE, do not necessarily provide this feature. Animations do, however, have a number of merits that can help in conveying the content of said presentation. This is why it is worth considering what options there are to add animations to programs that do not inherently support them.
This project is based on the premise of supplementing IPE with animations. The here proposed tool `ipe_animations` provides a means to generate animations for an IPE presentation. This is done by combining the two programs IPE and Manim.
This thesis documents the creation of this tool. This begins with an analysis of what its requirements would be. The proposed tool is a combination of two different programs. This means an implementation needs to find or create common ground between those programs. This common ground can be found in the theory of the building blocks of the graphics. Subsequently, an implementation of these requirements is introduced from both a user's perspective as well as its implementation process. Completing the picture of `ipe_animations` a short outlook on future work is given.

Contents

CHAPTER 1

Introduction

Successfully holding a presentation depends on multiple factors. A large part of it is the speaker's performance and the overall content of their monologue. Another important part is the materials that are used to accompany what the speaker says. Although they can take on many shapes, there is an established standard format for these supplementary materials. Oftentimes this format is what the term *presentation* refers to.

Creating presentations: There are many ways to create presentations. The only real requirement is that the result can be projected in a predefined order. This can be achieved with many different programs. These do not need to have the express purpose of creating presentations either. It does help when they do, however. Even within the scope of programs aimed at compiling presentations, there is a high variance in specialization. On one end of the spectrum are tools like PowerPoint. It is one of the most commonly known and used programs for this purpose. Being that, it falls under the category of a very generic program used for creating presentations. These programs tend to provide the most commonly used features. On the other end of this distribution there are programs that are more specialized. This specialization speaks to specific target audiences. The amount of available features does, however, tend to decrease with a heightened degree of specialization. Take IPE as an example. It targets an audience that appreciates being able to easily and strictly specify presentation objects. In exchange for this specificity, some cherished features of generic programs, for example animation, are sidelined. The degree, as well as the type of specialization can differ a lot. The most basic features of creating presentable objects and exporting to a format used for presenting will be found in each program. The rest comes down to the specific presentation and how the program can help with creating the content for that presentation. Add to that that the presenter may be familiar with one tool over another, and this choice becomes personal.

Generic programs, like PowerPoint are usually intended for a very broad audience. This type of program provides a large amount of features. Due to this large amount of features, the User Interfaces can become convoluted. Very few users will regularly use the entire range of available features. Still, because of the wide usage of this program, users will be using most features to a similar extent. All users will still experience the drawbacks of there being too many features.

Specialized programs, on the other hand, cater to a more specific target audience, sacrificing this wider array of features in favour of focusing more on other features. IPE for example resembles this second approach much more. The program's features take into account that the intended user base is settled in STEM fields. The user is expected to have a basic understanding of, for example, typesetting text with LaTeX. User interfaces of more specialized programs tend to be less convoluted as there is no need to have that many features.

Making the case for animation: Some disregarded features might, however, be relevant enough that users will actively choose a program they might otherwise not favour. One of these may be the availability of animations. One of the most important tasks of creating a presentation is to find a way to convey all of the relevant information. Carefully structuring data will not always be enough to effectively bring in that information. Animations can add context, without the explicit need to clarify it. Certain types of information, for example continuous processes, can be shown. Following this train of thought, a method to extend using programs' capability for animation may be worth considering.

Proof of concept: Is it possible to combine animation with a more specialized tool? A combination of different programs, one specialized for animation and another tool of choice, could act as a proof of concept. The choice of these tools should fall to programs that have openly available code interfaces, making open source projects the preferred choice.

Setting the premise: For this project, IPE[Otf24] will be used as an example. IPE is popular in scientific circles for its focus on precision drawing as well as the utilization of well-established tools such as LaTeX. That IPE is open source and adaptable by users themselves is just an added bonus. As it is a rather minimalistic and PDF-based tool, however, there is only the slightest hint of support for animations. Even this is a stretch of the term, as what IPE can provide is page-transitions and there are only so many meaningful effects a skilled application of fades can accomplish. If getting the point of the presentation across is strongly helped by adding animations, this might be a reason to create said presentation in PowerPoint in the first place, even if the presenter would prefer to use IPE. What this project aims to do is supplementing IPE with a utility that allows users to create animated presentations. This supplementation is based on

Manim[The24]. The tool developed here should enable users to create a presentation within IPE from which an animated presentation can be generated.

1.1 Terms

In this paper, concepts that can be defined in a quite broad way are discussed. Here, the most relevant terminology is clarified and narrowed down.

Images come in many formats. However, when the term image is used in this thesis, it refers to raster images.

Presentations can, in the broadest sense, be defined as any structured display designed to help convey information on a topic. Usually, the intent is to familiarize a group of people with the topic at hand. This definition based on function is far too broad to be practical in the scope of this thesis. Instead, let the term be defined by form. In this case, let it describe the standardized format that people usually refer to when they discuss presentations. For now, let us call them generic presentations. Generic presentations are separated into a number of self-contained pages (usually referred to as slides) that are intended to be projected and presented. This projection is shown in consecutive order. In addition to defining the concept of presentations in the context of this work, a clarification of what constitutes a singular presentation is necessary. A presentation is assumed to be stored in a single file. Any embedded content (such as videos or images) must be part of a self-contained slide and stored within the file. A presentation is expected to be presented and created in one tool. A format conversion of the same presentation used for further processing is considered a copy.

Animations, as far as relevant here are a type of local change applied to what a viewer would perceive as a singular object or a group of objects. These changes can take on different forms. They are, however, continuous, and at least temporarily change the visible state of the selected object. Animations will mainly be applied to vector-based objects (and texts) rather than image-based objects. These changes can be introductory, changing of an object's state, or remove said object. While there is some divergence from this, the main type of animation will not be based on how vector-based objects are reshaped and moved. Parts of this would be applicable to images. However, there are limitations to image based objects, simply because vector and raster-based objects differ a lot. As a consequence they will usually have different technical implementations in the programs used. The focus in this thesis lies on vectorized objects, because they are expected to bring more utility to the presenter. Another type of animations on images is changing an image in accordance to a pattern. This concept will be referred to as *Effects* simply to differentiate between animations that are based on information on an object, rather than an image. Effects *can* be used to mimic animations. Options to create meaningful effects are, however, largely reduced. Effects will usually be used purely to raise attention.

Tool: For convenience's sake the term *tool* will be a shorthand referencing any dedicated program to create a presentation. This means programs whose main purpose is creating presentations in some shape. These programs will support exports in either common presentation creation files, video-formats or PDF-formats.

Context: The term context is another very broad term, that will from Chapter 4 onward be referring to the current state of the developed tool's runtime environment. More specifically it refers to what is currently contained within Manim's environment.

1.2 Thesis structure

As an initial introduction to the topic an analysis of both the current workflow as well as a summarized evaluation of a number of presentations from conferences will be presented in Chapter 2. It will focus first on introducing different tools to create presentations, considering them in both their merits and drawbacks. This is followed up by analyzing a sample of presentations for both included animations, as well as their potential for animations that could improve the presentation itself. Then different options for combining multiple tools to supplement animations into formats that have no or only indirect support for animations are considered. One of these approaches is then examined more closely. This examination is then used to propose requirements for a supplementary tool to animate IPE. Chapter 3 will put an emphasis on concepts such a tool would need to be based on. The chapter can somewhat be likened to introducing the ingredients of a recipe. Initially both IPE and Manim, the tools the solution is based on, are introduced. This includes a brief overview of work-relevant details. Another category of "ingredients" that will be introduced is an exploration of the underlying theory. Chapter 4 acts as a manual that summarize and explain the usage of the core functionalities of the developed solution. It views the developed tool, `ipe_animations`, from a user's perspective. The following chapter, 5, contrasts this by considering a development view instead. It reflects on parts of the development process and lays out aspects of the implementation. As a conclusion to the analysis of `ipe_animations` omissions of the IPE-specification as well as corrections of inaccuracies will be discussed in chapter 6. A few approaches that can be taken to resolve these are described there. Finally, Chapter 7 gives a short summary of the work's final results.

CHAPTER 2

Assessment of Requirements

Before starting to develop something new it is important to understand what a user-base will find use for. The target audience considered in this thesis consists of people whose work has a heavy focus on precise mathematical expression. They are expected to prefer tools that allow them to express exactly what they want, even if these tools are less intuitive to the uninitiated. Figuring out what they need means understanding what features are necessary or desirable to them. Some questions that must be considered here are the following: What features do the tools used to create presentations provide? In what capacity are these features present? What are the limitations with those features? Figuring out what the features are can be done by taking a number of tools that have a precendent in being used by the intended target audience as examples. This will be focused on in Section 2.1. Another part of this is assessing what people might use animations for; (see Section 2.2). A sample of presentations used in conferences is taken and analyzed for included or potential animations. This analysis will be used to get an idea about what types of animation could be considered useful. Following this train of thought is the question of what combination of tools can give the best set of features. Currently available methods to expand the list of available features are described. These suggestions are then broadly outlined in their methodology and their issues. This will be the theme of this chapter's ultimate section, Section 2.3. Finally, a proposal for a more versatile and practical approach is given.

2.1 Understanding the requirements

Understanding the current state of things is one of the first steps in understanding where improvements can be made. Initially, an analysis of the features that could be perceived as useful by the user base is provided. To get a better understanding of what current tools are capable of, a selection of them is set side by side to the aforementioned features.

A summary of this is also given in Table 2.1. The relevant features are further described in the following paragraph.

2.1.1 Features

Certain features are necessary or at least strongly desirable for the creation of presentations. Some of these features are more important to the intended target audience in STEM fields specifically.

GUI to create and modify objects: While it is possible to create presentations with text-based and other user interfaces, a graphic user interface is a far more intuitive method to create, modify and remove objects. As an example for this compare creating shapes with the graphics library of a programming language with creating graphics in a dedicated drawing program such as Inkscape[Ink]. Color picking and changing any other property is a lot easier with the "What you see is what you get" paradigm most GUIs are based on. GUIs usually embed all relevant functionality in some form of context menus. The way these menus are accessed and structured reveal a lot on what the tool was developed for. The GUI will very strongly influence the users' experience using the program. A program with many features may suffer from an overly loaded and confusing GUI. Simpler programs may have certain limitations because an overly simplified GUI that does not allow access to show every feature. Seeing that different target audiences have different preferences and preemptive knowledge of tools, a simpler GUI that requires more external input and is less loaded may be preferable to some users. Ideally, a GUI has easily accessible grids for precise drawings. Most programs to create presentations are GUI-based. However, there are tools that are not primarily GUI focused, rather text-based, be that tools like LaTeX or libraries for coding. These non-GUI based tools may be more versatile and precise in certain aspects, but do not provide immediate feedback on changes.

PDF-export: PDF is a popular file-formats. It is legible on most machines. It is regarded as a stable format and is therefore often used for finalized documents. Exporting to PDF ensures that a final version can be presented on almost any machine. The format is less volatile than working formats and supports index-able text. Additionally animations are only supported in a limited capacity. Not all viewers are able to present them however.

Video-export: Videos are in their most common realizations an even more broadly displayable format than PDF is. They are not easily editable, and do not generally allow for indexed text. They can however store animations very well, as they may be equated to a recording of a screen. Displaying animations as videos decouples the presentation from specific programs such as PowerPoint.

Animations: The inclusion of animations can be attention-grabbing as well as useful for easing the flow of a presentation. Changes that happen over time can also be modeled more accurately if animation is an available feature. Aside from these reasons there are many reasons one may choose to animate presentations. It may just be a desire to create a more aesthetic slide transition. Preferably animations could be played in a way that is decoupled from timings in videos.

LaTeX-support: LaTeX is a common way to format text in the targeted user base. The user is given a lot of control on how exactly the content is to look. Additionally LaTeX is very versatile yet specific in how a document is generated. This is very useful for rendering mathematical expressions. Another use case for it is specifying combinations of characters. This is something linguists like, particularly when they are dealing in phonology or obscure languages. Additionally LaTeX allows for relatively simple inclusion of user-created resources.

Open source: When using open source tools there is a set certainty that usage conditions will not change. This ensures everyone is able to freely access the content placed under the open source terms. Additionally, an open source tool has an increased likelihood of there being access points to the program itself. There is also increased ability to have an insight into the tool's inner workings, as the source code is openly available. Oftentimes open source projects permit, often even encourage users to create their own extensions to the tool.

2.1.2 Programs and their features

This subsection is an overview of the features offered by commonly used tools. A summary is given in Table 2.1. A more detailed description of the shortcomings and other relevant qualities of each tool follows. This is separated into different paragraphs that describe one tool each. IPE and Manim are emphasized, as these two tools will be the ones that form the basis of `ipe_animations`.

PowerPoint and its various clones are probably the most commonplace and widely used programs to create presentations with. PowerPoint excels in providing a very broad and yet generalized set of features. Its target audience is anyone who wants to create presentations. PowerPoint is a GUI-based application. This GUI is, however, rather feature-laden and can seem overly complex. PowerPoint supports most of the features listed above. LaTeX remains unsupported. There exist options to add formulaic expressions. These options are less flexible and user-controlled than LaTeX-code. Although there are open-source alternatives to the software, PowerPoint itself is proprietary software created by Microsoft.

IPE is a program that caters to a more specific audience than PowerPoint. This audience places a heavier emphasis on the precise control of a few features rather than

program	Paradigm	GUI	LaTeX	PDF	Video	Animations	open source
PowerPoint	GUI-based	✓	-	✓	✓	✓	-
IPE	GUI-based	✓	✓	✓	-	-	✓
Manim	code-based	~	✓	-	✓	✓	✓
Beamer(LaTeX)	Text-based	-	✓	✓	-	~	✓
RevealJs	code based	-	~	✓	-	✓	✓

Table 2.1: An overview of available features in different tools. IPE and Manim are highlighted to draw attention to how they complement the other. Additionally the implementation introduced in this project is based on adapting these two tools.
Legend:
✓: feature fully present
~: partially present or easily implementable,
- : feature absent

having an overly broad pallette of features to choose from. The user needs to have a more advanced technical understanding as not all features can be directly accessed in the GUI. Instead, there is a reliance on XML-based files to style objects. All text is based on LaTeX source-code. Animations can only be created in a very limited capacity, that being PDF-slide transitions. IPE is described in further detail in Subsection 3.1.1.

Manim is a program that generates videos from code defined by the user. It is a code-based framework. As such, it is very versatile in usage. The user declares objects and uses them as animatable objects. There are presentation frameworks for Manim. The Manim Editor [1] or its predecessor Manim Web Viewer[2] are examples of this. Manim has no inherent support for PDF. Looking at Manim's internal structure, however, implementing support for PDF would be a larger project. There are experimental GUIs for Manim.[3] More details on Manim are explained in Subsection 3.1.2.

Beamer is a combination of an extension to LaTeX, and a presentation tool. As a LaTeX-library it provides additional functionality that will provide a PDF-file that contains the given animations. It is not possible to explicitly add complex animations, however, embedding externally created animations is possible.

RevealJs is a library for JavaScript, that acts as a framework for creating presentations in HTML-based pages. Additionally, ir allows for animation. It does not directly enable LaTeX-based text, it is an extension to JavaScript. That means it will predominantly – if not exclusively – be used by people who already use JavaScript and know how to code.

[1] https://github.com/ManimCommunity/Manim_editor, last accessed: 08.03.2025
[2] https://github.com/christopher-besch/Manim_web_presenter?tab=readme-ov-file, last accessed: 08.03.2025
[3] https://www.reddit.com/r/3Blue1Brown/comments/qg7u6l/Manim_gui_desktop_application, last accessed: 08.03.2025

Combining RevealJs and a library that converts LaTeX-code to code-parsed objects that can then be used in RevealJS would be one way to introduce LaTeX into this framework.

Reflection As seen in Table 2.1, no one single program covers all desired features. Many of these programs do, however, provide means to combine them in one way or another. If one looks at IPE and Manim in particular, it is evident that a combination could cover all of these points. This will be elaborated on in 2.3.

2.2 Evaluation of conference presentations

Additionally to determining technical requirements, it is necessary to understand what the target audience would be presenting. For this, understanding the content and context of the presentations is consequential.

Data acquisition: The main user group will be people in scientific fields holding presentations on their projects. Evaluating conference slides of these will show what users would actively use. The following conferences have been evaluated:

- 40^{th} European Workshop on Computational Geometry (EUROCG2024)
- The 31^{st} International Symposium on Graph Drawing and Network Visualization (GD2023)
- The 32^{nd} International Symposium on Graph Drawing and Network Visualization (GD2024)

First, the data set was cleaned of duplicate data and organizational content. Then a relatively simple script was run over the datasets to tally each instance by the tool they were last edited by. This script would extract meta-information from a file and bin them into the category they fit in best. The order of precedence in deciding what tool was used to create the file was first the file extension, as the two extensions were `-ppt(x)` and `.pdf`. Following this, the entries for the first the generating software, then the producer, and at last the creator was queried. The first hit would then be the category to which the presentation was counted. This was repeated for each conference.

Data analysis: Table 2.2 provides a fist overview of what tools presenters use to create the slides they submitted. There is a degree of uncertainty about what the actual creation of the presentation looked like. The metadata that was taken into account can only represent an image of the latest changes. The true number of presentations that have utilized any of these tools at some point in their creation may very well be higher. However, even disregarding this, it is already obvious that the presentations had in large part been created with IPE. Complex animations that can change individual objects can only be found in the PowerPoint files. As discussed in Section 2.1, this is a limitation of

tool	EuroGC24	GD23	GD24	TOTAL
Adobe	2	0	1	3
IPE	22	21	22	65
PPTX	16	8	5	29
LaTeX	29	8	10	47
Other	1	1	1	3
TOTAL	70	38	39	147

Table 2.2: A tally of presentations by the last known creation program, grouped by event

Effect	EuroGC	Gd2023	GD2024
General (Fades, transformations, ...)	6	4	2
Graph-manipulation	3	0	1
Video	0	3	2
Linked animations	3	0	0
None	4	2	2

Table 2.3: Found animation effects grouped by event and animation type.
Categories
General: This category summarizes operations commonly found in transitions and object transformations. Recoloring operations are also categorized here
Graph-Manipulation: Similar to the above category this category includes more semantic value, that being for example a graphs edges getting colored in or a vertex and its connected edges being dragged.
Video: This is a category simply denoting embedded video files.
Linked Animations: This category describes a focus on ensuring two separated parts of the slide move as if dependent from one-another.
None: Here there is no animation encoded in the PowerPoint file

not only the used tools, but of the PDF-file format as well. A second analysis provided insight into the found categories of animations. Here, only `.pptx` files are considered to be animated. This is not entirely accurate, as there is some support for the concept within the `.pdf` format. This is also visible in an outlier in the data created with *Beamer* that has an embedded animation. That animation is a video detailing a graph generation. The results of this analysis are given in Table 2.3.

These numbers are not overly large. However, only 29 of the 147 presentations were created with PowerPoint, the program that features the intuitive creation of animations. It should be noted that some presentations are counted in multiple categories, as a single presentation can be counted in multiple categories. Even so, the reason for having chosen PowerPoint may also lie in a personal preference of the author. After all, there is a large number of presentations that do not feature effects at all.

Expanding the list by presumption Additionally to considering animations that could be found in PowerPoint files, a superficial look at what may be useful in the context of the presentation was taken. For this, all presentations were taken into account. They were analyzed for their content with regard to animations that could be the most useful. The presumptions made were then abstracted into a number of categories. Figure 2.1 summarizes the types of animation that were assumed to be useful. As these conferences had a rather strong focus on concepts involving graphs, it is to be expected that many of the given presentations may benefit from animating graphs. These graphs can however be quite diverse in their realizations and focal points. As oftentimes individual parts of graphs are changed, and the types of graphs vary in usage and partially structure it may be easier for the user to manually decide on what parts of the object is to be changed. This allows for more specific control of the object. An expansion related to graph-manipulation, as briefly considered in Section 6.2, for common operations is worth to keep in mind for future development, however.

2.3 Proposing an alternative

Now that common use cases as well as the current approach to this intent have been discussed, the task of defining requirements for an alternative method of animation can be tackled. This section builds towards the idea of creating an effective and easily applied workflow that is further described in section 2.4. There is a large selection of tools available to create animated presentations. Subsection 2.1.2 mentions only a small portion of them. Combining multiple tools can appease shortcomings of any individual tool. A few options to do this are discussed here. Particular attention will be given to using PowerPoint and IPE, as this would be the current process one would use when animating IPE-presentations.

Screen recording One of the most intuitive ways to create a video file that can convey a presentation in the way intended is to have a practice-run of the final presentation and to screen-record it. One drawback of this is controlling the timing of transitions will have to be done manually. Another is that unexpected interruptions during the presentation may result in mismatched timings.

Inserting Animations into Beamer Beamer enables one to insert animations created in external programs into the LaTeX- file. This means one could for example render a video in a presentation tool or any other program, say Manim, and then embed it. This is not a very versatile approach. Issues here may be that changes in the animation must first be re-rendered and then re-inserted and then recompiled.

Converting IPE to PowerPoint files There is a direct conversion tool from IPE files to PowerPoint called `poweripe`[4]. It enables users to (lossily) convert back and

[4]https://github.com/otfried/ipe-tools/tree/master/poweripe,
last accessed: 08.03.2025

- General:
  This category of animations is the most generally applicable, and holds effects such as general transformations, changes of colors and other changes of single objects. It forms the basis of all other categories.
- Graph-related operations:
  As graphs are a good way to portray relations between objects, be that networks or an underlying structure in data, there is reason to include animations that are based on graphs. This could be the generation of a graph in a certain pattern, a partial focus on the graph by fading out other parts and even linking the movement of adjacent vertices and edges. Depending on the specific use-case there is a wide variety of options to view graphs.
- Clone and Manipulate:
  Explaining stepwise processes can profit from copying the original object and then changing the copy.
- Text-content:
  Another thing that can be useful is manipulating texts within the presentation. This could mean replacing text, crossing it out, or highlighting it.
- Zoom in/out:
  Zooming in and out of objects can be an easy way to show that something is a part of a whole. Additionally it could be useful to reduce noise by removing non-relevant parts of the objects.
- Link two different parts of slide together:
  Two different objects in a slide could be linked in their movements. This could be used to explain correlating processes. Take as a generic example an animation that visualizes trigonometric functions and a unit circle.
- Progress bar or Timeline:
  For presentations that detail processes or chronological events it could be useful to present what state of the process the presentation is showing. Animating this could more easily imply a continuous progression. Additionally the change rate of the process could be conveyed.

Figure 2.1: Overview of potential animation categories

forth in between IPE and PowerPoint. The conversion to PowerPoint is direct. The conversion back requires a PDF-export that is then converted back. This comes with losing all non-translatable information. Seeing that animations as they are understood in this thesis are exclusive to `.ppt(x)` files this means losing any animations. After the file is converted to `.pptx`, there exist two separate versions of the file that are for now equivalent. Every change, be that added animations in PowerPoint or anything else further distances them. Taking this approach to animating presentations works best when the presentation is finalized by the time animations are added. A core issue one needs to consider here is that it may be quite late into the process where changes are made. The necessity to change could for example come after the user has determined their presentation is done, and a fair amount of animations have been added. This is the process that is for the purposes of this thesis assumed to be the standard workflow for animating IPE. To make the users' perspectives and struggles clearer, two scenarios are outlined below.

The ideal scenario: Allie has created a presentation with IPE. She already has a very clear vision what everything will look like and is certain that she will not be making any major changes. She is sure any changes made will only be relevant for this next presentation. Allie conscientiously prepares her presentation so that she will only need to add animations. Then she proceeds to use the tool `poweripe` to convert her presentation to a PowerPoint presentation. She keeps the IPE-file, that she will for now leave to be. Allie now also has a copy of her presentation converted to PowerPoint's file format. Since she has so perfectly planned out everything she simply adds in her animations. Her IPE file and her PowerPoint file are baring the effects she added close to identical. She expects to be done with the slides after she has presented them has no intention for now to alter things.

Another scenario: Unlike Allie, Bobby prepares a midpoint presentation of his project. He wants to add animations to his presentation, because he is convinced they will really add to what he wants to say. He is working on a project with a heavy focus on timed processes and would like that to be properly portrayed. Because he is only partway into his project, he is not entirely sure how he wants to present things, what parts he should really focus on to achieve the best effect. He knows he will often have to change the contents of his file. He plans to use this presentation of his as a working file for future presentations as well. If he animates and changes his slides now, he may very well end up with very different IPE and PowerPoint files. And if he wants to work on his IPE-file and later convert it to PowerPoint again he will have to add all of his animations again. Because there is no equivalent concept to animations in IPE, a lot of his work will get lost.

Combining code-based and GUI-based tools A more advanced and work-intensive way to resolve the drawbacks of individual programs would be to use code-based frameworks and import the content of the other program's files for further processing. Given

that the program's functions can be accessed externally, these can be used and further manipulated in a code-based tool. This comes with the comparatively high effort of building necessary data structures and processes to export and import the data. To completely disregard this approach would, however, disregard the fact that putting the necessary work into this is a one-time investment. It would ignore the fact that, if properly designed, an implementation of this approach can be used to animate any type of animation in the appropriate format. It may consequently be a good idea not to view this as a solution for individual presentations but to create a re-usable framework instead. This idea forms the basis for the implementation suggested in this project.

2.4 An IPE-based Animation tool

In the previous sections, the focus was put on the features that are considered important in creating presentations. It was put on the options are given both in individual as well as in the combination of different tools. This meant laying out the features given in each tool and taking a look at how another tool can be used to supplement missing features. Limitations and drawbacks of these have also been mentioned; (see Section 2.1). To support a fundament of what can be regarded as useful for users, an evaluation of a number of presentations was given in Section 2.2. This information is now used to create requirements for how this new suggestion should be implemented.

2.4.1 Concerning the current workflow

To better grasp how a user views this issue another look is taken at the example users Allie and Bobby:

> Allie has presented her slides. She sometimes thinks converting her files is a hassle, and does consider having to change the occasional typo twice a bit of a nuisance. All in all, though, she is perfectly content to go on as is. Bobby, on the other hand, is struggling. This is the fifth slide he has structurally changed, because he figured out a new layout would work out better than what he currently has. For now he continues on to work with PowerPoint. He knows he will convert his file back to IPE at some point, and is really not looking forward to animating all of the animations again. He is even considering dropping the entire idea of animating his presentation. In his frustration he wonders:
>
> *Can I not just render this to a video without any unnecessary conversions?*
>
> Allie understands his frustrations. After all, it is not always she is so certain in what she wants to do either. That and – as it turns out – the project she had just presented has become a part of a bigger project she and a few colleagues of hers will be working on. She will be needing to change her presentation.

Both Allie and Bob would profit from a way to cut out this intermediary role PowerPoint takes. If you ask either of them, they will argue that PowerPoint files may not even be the most ideal format to represent animations. An alternative representation could be preferable.

Video killed the PowerPoint star PowerPoint is a proprietary software whose files are partially legible in other programs as well. This does not mean perfect compatibility in different versions or open-source variants either, as is the experience of many people who use one office alternative or the other. Video, on the other hand, is highly standardized. Video-files are very broadly implemented throughout different device-types and operating systems. Seeing how pervasive video formats are today, this does not come as much of a surprise. Even devices such as modern TVs can display videos in common formats. Baring issues such as incompatible codecs (coder and decoder pairs), there are few issues that come with format compatibility for videos. Even these are usually a quite minor issue as there are a number of programs capable of converting from different formats. This justifies an approach that expects a user to encode their animations in a script. This script would then invoke a library that immediately generates a video instead of requiring users to use PowerPoint.

Create a tool that enables IPE users to animate IPE presentations by forming a bridge between IPE and Manim. It shall decouple animations from an external program's metadata. It should instead be code-based and provide the following key features:

1. render IPE presentations
2. insert animations into specified points of the presentations
3. access objects based on custom property in IPE
4. modify these annotated objects
5. each view-transition can be modified in its own method.
6. changes in IPE files are reflected in later renders of the file.

Figure 2.2: Objectives of the tool developed in this thesis

2.4.2 Combining IPE and Manim into a Framework

A framework that enables users to animate IPE in a useful way should be able to bridge IPE and the other chosen tool. This tool must be compatible enough to allow for relatively easy object imports and still complement IPE in the features IPE does **not** provide. Looking once again at table 2.1, one can see that Manim can handle the features that make IPE preferable to PowerPoint. Additionally, Manim itself is a library used for animating vector-based objects and can create video-files. Creating a framework to animate IPE with Manim would mean a user could create a presentation within IPE, and then in a post-processing step add in animations with a script. The animation script should be as expressive and editable as possible. This would allow for the animations to be stored independently from the animation file. Any change in the IPE file to be reflected in a newly generated presentation. Additionally, metadata would not get lost in conversions, as these are stored separately. This hypothetical program would need to read IPE's content and transcribe them into Manim objects. Subsequently, animations would need to be stored and played when they are meant to. The target audience of both IPE and Manim can be expected to have an understanding, if not a preference of how to create code-scripts. This means that it is reasonable to create this framework for a code-based environment. Ideally, this could be done in Python.
Figure 2.2 summarizes the main objective of the thesis.

CHAPTER 3

Prerequisites

In the previous chapter objectives for integrating animations into IPE have been set. These were based on a feature analysis of different tools and the disection of a sample of presentations. This led to the conclusion that creating a program that bridges IPE and Manim could be a practical way to include all desired features. To accomplish the goals set in Figure 2.2, an interface between two separately maintained tools needs to be created. This presumes an understanding of both the programs that are to be connected, as well as knowledge of what the theoretical basis for this implementation is. Initially, in Section3.1, both tools – IPE and Manim – will be introduced. This is followed by an excursion into underlying theoretical concepts in computer graphics, in Section 3.2, insofar as they are relevant for this project. That section is separated into individual entries, each one of which first explains the mathematical basics and then points out their relevance in creating the proposed tool.

3.1 Taking a closer look at the tools

Effectively combining IPE and Manim requires understanding how the relevant data – in this case, page formats and page content – is structured. Both tools are intended to visualize vector-based objects. There is some overlap between the two programs, e. g. sharing similar libraries (such as Cairo being used as a potential rendering engine in both). As discussed in Subsection 6.1.3, this can potentially be used in future improvements. The main output format of Manim is video. IPE, on the other hand, considers PDF to be its main export format. Because of what the tools focus on (,as detailed in Table 2.1,) certain structures do not transfer from IPE to Manim and vice versa. Notable examples of this are that IPE has no concept of the animations Manim was created for, or the way in which IPE provides options to reuse previously defined values.

3.1.1 IPE

IPE[Otf24] is a tool that is used to create presentations. It is somewhat reminiscent of Inkscape and PowerPoint. One of its merits is that it incorporates LaTeX into its toolset. This subsection explains the underlying data contained in an IPE-file.

Data Structure

IPE stores its data in an XML-based format. This format barely differs from the data structures of the IPE-library. The description is largely based on the library-internal structure, because the implementation of `ipe_animations` loads the IPE-file as an IPE-internal object. The *File* object can be equated to the root node of the XML-file. It stores global properties such as the used LaTeX-compiler as well as whether the pages are numbered or not. This object contains a *Document* object. The *Document* consists of *Pages* and *Stylesheets*. There is a difference between the XML-format and the in-memory object here. IPE cascades the contained *Stylesheets* against one another. Individual *Page* entries contain their own sets of *Views* and *Layers* each. The relations between these three concepts are explained in a dedicated paragraph. They are elements that ensure that the presentation is properly structured. A *Page*'s content is largely separate from this. The content itself is stored in *Objects*. They are strongly linked to the *Stylesheets* of the *Document*.

Pages, Views, and Layers *Views* and *Layers* are parts of a *Page*. *Views* can be regarded as a version of a *Page*. When an IPE file is exported to PDF, or presented, each of the *Views* – presuming it was not set to be skipped over – is shown in consecutive order. Each *Page* can be toggled for visibility. Additionally, each *View* of a *Page* can be toggled for visibility. While *Views* are not usually found in presentation programs, their purpose can very easily be inferred. The concept of *Layers* can, however, not as intuitively be assumed from other programs' functionality. Like in other common programs, *Layers* allow for a grouping of selected *Objects* into a unit that can be collectively transformed or toggled for display. Unlike common programs, however, is that there is no correlation to the front-to-back "layering" of the *Page*'s content, as would usually be expected from the name. *Views* and *Layers* are both owned by the *Page* and share no direct links.

Stylesheets and Attrributes Some properties of *Objects* are expected to be shared between *Objects*. These are encapsulated in *Attributes*. The main purpose of a *Stylesheet* is to be a lookup table used to store various styling properties. *Attributes* only need to be looked up if they are symbolic. In that case, they act as pointers to an entry in the *Stylesheet*. *Attributes* can for example be gradient patterns, fills or opacity-values. An IPE-file can have multiple *Stylesheets*. These are then combined to a *Cascade*. In addition to this main function, the *Stylesheet* is meant to give the option of overriding the standard page format or other global styling properties such as the style of titles and page numbers. IPE implements a top to bottom lookup-scheme, that means values that

have multiple entries only the uppermost entry is considered. It also means that the page format that is found highest up in the document is used.

Objects A *Page* contains a list of *Objects*. These *Objects* are assigned a *Layer* each, so their visibility can be toggled. There are five different types of *Objects* that share properties like their applied transformation.

Paths are vector-based objects. They describe a shape defined in a series of instructions to construct it. *Paths* are the *Objects* that can be given the most *Attributes*. All of these relate to the way this shape is rendered, be that its colors, outline, or even the fill-methods.

Texts are LaTeX-compiled objects. Their *Attributes* set values for the compilation of the LaTeXsource code.

Images are bitmaps that are projected onto an originally rectangular canvas that may, however, be distorted to be any parallelogram. The only applicable *Attribute* for *Images* is their opacity.

Groups are composites of *Objects*. Groups can be masked or given their own background.

whereas

References are pointers to a *Stylesheet*-entry, that is then resolved to an *Object*.

These concrete object-types inherit from the quasi-abstract class *Object* that abstracts their shared properties, such as a user-defined property or the model-matrix by which the object is transformed.

3.1.2 Manim

Manim[The24] is a Python library to create animations that explain mathematics. Its output format is primarily video, though there are use cases in which images are rendered instead. Manim is a code-based tool. While experimental GUIs exist, they cannot be properly utilized for the purposes of this thesis. IPE already defines where the objects of the presentation will be.

Structure In Manim a video is generated by a *Scene* that can be separated into *Sections*. A video is generated by playing individual *Animations* within this scene. These *Animations* take the current state of the *Scene* as their basis and manipulate selected objects within it. The base class for contained objects is the `Mobject`-class. This type of object has several subclasses to which *Animations* can be applied. The most relevant to the thesis is the `VMObject` (vectorized Mobject) and its various subclasses.

`ImageMobject` is used solely for rendering images, though there is potential for them to be substitutes for other badly implementable features, as will be discussed in Chapter 6.

3.2 Theoretical concepts

There are commonalities in the implementation of both IPE and Manim. These occur because both programs use well-established concepts that – if not standardized – share large parts of their theoretical foundation. Only a subset of Computer Graphics (shortened here as CG) theory is relevant in order to understand the data structures that have to be transcribed from one format to the other. The most important aspects of the theory are explained in the following subsections. The main source recited within this section is the book *Fundamentals of Computer Graphics*[MS18].

3.2.1 Coordinate systems and transformations

This subsection summarizes the basics of how objects are situated and scaled within the space they exist in. This can refer to an object on a presentation slide or page. This theory can, however, be expanded to any environment that positions objects in an environment that can be described in vectors. There are two aspects to this concept,

1. the definition of space relative to the object[MS18, Chapter 7]
   and
2. how a conversion between different spaces is defined [MS18, Chapter 6]

Coordinate spaces Objects can be defined in different coordinate spaces. This can be any arbitrary but deterministic definition of space. In practice, Cartesian coordinate systems with two or three dimensions will be the most commonly encountered format. Objects' positions are in this paradigm usually defined as point vectors. Coordinate spaces describe the concept of there being multiple scopes with which an object can be defined. These scopes may have different scales as well as different reference points to one another. This can be compared to how describing the position of a specific place on Earth will differ from describing the position of the sun in relation to the Milky Way. Each of these descriptions is relative to a reference point unique to the object, the scales used are, however, entirely different. The local space of the object describes the points of an object itself relative to a point unique to the object itself. This relative position is not necessarily the center of the object, though it does often coincide with it. As an example for this, the distance for each point on a the outline of a circle to the center being exactly the radius. Under the assumption that the reference point is the center of this circle, all points could be defined as a combination of $\sin(\alpha)$ and $\cos(\alpha)$ pairs.
Another type of coordinate space is world space. This is where in the world an object is positioned. All relative coordinates are transferred over. Imagine putting this circle

somewhere on a surface. Relative to the surface the positions of the defining points of the circle are not these sine and cosine combinations, but rather a combination of $\sin(\alpha) + center_y$ and $\cos(\alpha) + center_x$, where $center_{xy}$ describes the center of the circle. The last scope introduced here is view space. This can roughly be understood as the viewer's lens on the world. Assume that there is a camera somewhere on this surface. That camera is how any viewer would be able to perceive it. The camera is at an arbitrary position in relation to the surface's center and is usually expected to be movable. While in the case of `ipe_animations` this camera position is static, the reference spaces for IPE and Manim do differ in what they consider the origin of their coordinate systems. When rendering there is an order in which these coordinate spaces are applied. They can be considered as nested within each other. Object space is the innermost layer. View space is the outermost one. In this case, the order in which our circle's coordinates need to be converted into the outer layer looks as follows:

$$\text{object space} \rightarrow \text{world space} \rightarrow \text{view space}$$

This is not an exhaustive list of coordinate spaces that are used in rendering pipelines. However, in this project, the amount of relevant spaces is limited to these three. In IPE – unless an object is transformed by the user post-creation – local space and world space will coincide. References, however, are deliberately created to later be moved to their positions. As IPE and Manim do not share the same coordinate space there is a view transformation in between those two.

Transformations is a term that describes conversions between coordinate spaces. It is common practice in applied Computer Graphics to represent transformations of objects as linear equations. These are usually encoded in a square matrix with one more dimension than the target dimension space.

Constraining transformation matrices can become necessary if certain geometric properties of an object should not be changed. IPE differentiates between three different levels of constraining the effect the transformation matrix has on objects. Figure 3.1 explains these constraints.

Affine Transformations as a super set: Affine transformations describe all transformation matrices that ensure that parallel lines always stay parallel. This obviously includes all rigid transformations and pure translations as well. They are always representable as some combination of scaling on different axes, rotation, and translations. Shears can be represented as a specific combination of these three operations as well. They are, however, because of their relatively simple form and common usage, usually included in lists of basic affine functions. Figure 3.3 shows how the corresponding matrices of these transformations. To combine these matrices into a final transformation, they are multiplied with one another. Matrix multiplication is not commutative, the same way in that it is not irrelevant in which order an object is transformed. As an example, take a

1. **Translations**, are the transformation type that have the lowest effect on the object they are applied to. This matrix changes nothing but the position of the object. Translation is the reason an additional dimension is usually added to a transformation matrix. With the additional dimension it is ensured that only the relevant coordinate point is affected by the transformation.
2. **Rigid transformations** are defined by preserving the object's size and shape. This means that in addition to moving the object around, the matrix can now rotate the object or flip it on its axes.
3. **Affine transformations** have the highest influence on how an object is changed by the matrix. Usually model-matrices are assumed to be affine. Affine transformations still guarantee that parallel lines remain parallel. This allows for objects to be sheared and scaled.

Figure 3.1: Constraint-levels for object transformations. Higher levels are contain lower levels.

$$\begin{pmatrix} a_{00} & a_{01} & x \\ a_{10} & a_{11} & y \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3.2: a typical affine matrix with arbitrary values.
Notice that the last row consists of constant values

scale	rotate	translate	shear
$\begin{pmatrix} scale_x & 0 & 0 \\ 0 & scale_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} shear_x & 1 & 0 \\ 1 & shear_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Figure 3.3: Transformation matrices forming the basis for affine transformations.
Legend:
x is the factor on which this change is applied on the x-axis
y is the factor on which this change is applied on the y-axis
α is a clockwise angle by which the object is rotated

square that is scaled and rotated, depending on the order the transformed object results in either a rotated rectangle or an arbitrary parallelogram. Due to the rules of matrix multiplication, there is no combination of factors able to influence the bottom row, and it always retains the values of an identity matrix.

Details to consider in IPE and Manim IPE's representation of matrices is intended to save as much space as possible, which means that the redundant bottom row of the matrix is omitted. This is because any transformation of an object must be at least affine. As explored previously, there is no combination of affine matrices that can result in any other values in the bottom row than those grayed out in Figure 3.2. The constraints for transformations (translation, rigid, or affine) are independently editable in IPE. The visualization of the object changes, yet, the matrix remains the same. Manim does not natively differentiate between these constraints. That means, it is necessary to reduce these matrices to have a lesser effect. An interesting side-effect of Manim is that applying a transformation automatically resets the object to the origin of its coordinate system. That means, objects need to be placed back to their position again. This is not a particularly complex operation, it only requires the position be stored, transformed and then reapplied. This resetting of positions is one of the possible causes of an error mentioned in Subsection 6.1.1. The other possible source could be a missing transformation matrix that needs to be applied.

3.2.2 Curve representation

There are different types of objects in computer graphics based on different generating objects, as is explained in Chapter 15 of the book[MS18, Chapter 15]. In a lot of libraries the base assumption is that shapes will be based on polygons (usually triangles). Additionally to polygons there are point-cloud based and equation based objects. Equation based objects can have different realizations as well. Of these splines and bézier curves are the most commonly encountered variations. Both of these are similar in that they are based on interpolating between anchor points, but their implementations somewhat differ. For the purposes of this thesis, however, only bézier curves are relevant. A bézier curve is a polynomial interpolation between a set of anchor points. A curve of the n^{th} degree has $n+1$ anchors. The curve starts and ends at the first and last points respectively, the other anchors define the curve's convex hull. There are algorithms to both subdivide and elevate the degree of the curve's equation. Both IPE and Manim consider cubic bézier curves as one of their basic primitives, if not their most important one.

Most curves IPE will directly be represented as cubic béziers. If not internally so, then generally they are converted to it on some occasion. Manim stores all of its vector-based objects as a series of cubic bézier curves.

There are two exceptions to IPE translating to béziers directly:

Circular and/or elliptic paths are by nature non-polynomial. Approximations need to be made to represent them as cubic curves. There are approximations with

relatively low error.[Ri6] Internally, IPE does this when rendering to PDF, as the format does not support circular path generation. Because Manim can generate circular objects, it is more practical to append an arc or ellipse to a vector-object, and have Manim handle that translation instead.

Straight lines can always be represented as higher-level polynomials. It is relatively easy to convert a line to a cubic bézier curve. However, using the function that adds a line to a vector-object in Manim directly, comes with the issue that the starting point cannot be defined. This is an issue with disjoint lines, as they do not link to the last element of the group. Instead of this a curve with a convex hull that equals the line can be added. The easiest way to accomplish this is to pair the first and second, as well as the third and fourth anchor points, and to assign each group the same value of the start point for that first group or end point.

3.2.3 Boolean operators on shapes

Another common technique for generating and modifying objects is applying boolean operators to a shape. Boolean operators allow for shapes to be combined with or removed from each other or for objects to be masks for other objects as well. These operations can be chained by taking the result of one operation and then applying a new operation on it. The basic principles of this method are similar to set theory. Any point inside that exists within the shape is tested to see if the condition defined applies to it and handled accordingly. [MS18, Section 13.3] Analogously to set theory the following operations are commonly found with combining shapes like this:

- Union
- Difference
- Intersection
- Exclusion

A feature in IPE that can be approximated with this is clipping or masking groups with a shape. Details on this can be found in Subsection 5.3.4. Cropping a page as briefly mentioned in Subsection 6.1.1 could be done using this approach. Manim supports boolean operators on vector-based objects, however there are a few aspects in this implementation that need additional handling to minimize visual discrepancies. A few differences remain as trade-offs for more noticeable errors.

3.2.4 Fill-Rules

A fill rule defines what parts of an object are inside or outside of it. This is particularly interesting in objects with holes or self-intersecting objects, as in this case the two common methods to determine this can conflict. There are, in general, two fill-rules implemented in various libraries:

Non-zero-winding decides what lies inside or outside of an object based on the winding of the current point.[HA14, Chapter 7] That means that the direction in which each bézier curve segment is drawn is what is relevant to this fill-rule.

Even-odd partitions the object by intersecting it with a ray and counting how often the edge was crossed. If the number is even, the part of the object is considered to be outside, otherwise it is inside.[MS18, Chapter 4] IPE supports both of these rules. Manim implements only the winding rule. Converting these is not simple as the even-odd rule is implemented with a ray-tracing methodology, whereas the winding rule takes the object around it and the direction the line is wound in as its reference. As this is something that is relatively complex to implement and will for most common objects not be relevant, this feature was disregarded. A potential approach to implementing it can be found in 6.1.3

3.2.5 Images

Raster-based images are – as far as this project is concerned – relatively simple. In their simplest form, they are 2-dimensional arrays storing individual color values. [MS18, Chapter 3]. The colors are in this scope, described in a four-channel format using an additive color model combining red, green and blue light. The fourth channel defines an α-component that describes the pixel's opacity. That is not to say that there are no complicating aspects to them, there are. Internally this is the format images are converted to, the way they are stored can differ. Some common formats, such as jpeg do not even support transparency for example. Those aspects are, however, either confined to image formats that are not supported in IPE, or they are implementation details that are already handled in standard libraries. An example of the latter is image compression and encoding. This would be of some importance if this project were parsing IPE files directly. As this is not the case, reading images falls to the IPE library.

Images in Manim Manim on its own provides a lot of functionality that cover most of the functionality necessary for rendering an accurate IPE-presentation. For images, however, some additional functionality is necessary to add. That is because some issues and discrepancies do occur with images. Currently there is an open issue regarding transformations that seemingly applies transformations seemingly mirrored.[1] Another difficulty is the image transparency. That could be solved by adding an additional method with relative ease. For now, there is a workaround for both of these issues using a rendering library. These two issues are minor and require only minor intervention.

Images as Ressources There are certain use cases where using Images as an additional resource for computation can help in converting IPE-objects to Manim-objects. Masking images in Manim does not work the same way intersecting vector-based objects works.

[1] https://github.com/ManimCommunity/Manim/issues/2412
last accessed: 08.03.2025

This is because Manim requires a vector-object to be passed for an intersection object. Resolving this is possible by using another compatible type of mask to combine objects instead. It is possible to generate such a mask by rendering only the object and subsequently extracting a frame from that render. Another use case of images is mimicking features that are not so easily transferrable to Manim. The train of thought that leads to this approach is that any program will on some level be displayed as an image. A solution that lies at hand is embedding separately rendered partial images. A graspable example of how this can work is found in video games. They often use such images as textures for objects. This is how reflective surfaces are often rendered. To separate the concept of generic and these pre-rendered embedded images the latter are referred to as textures. Generating and using textures is a technique that is often used as an intermediary step. [HA14, Chapter 20]
This can be used to portray attributes that are not directly supported in Manim. The idea is to create a (partially transparent) image that visually mimics objects. That image is then inserted into the world. Using this approach comes with information loss as this does partially convert the object to an image type. That only becomes relevant when further processing of these objects is necessary. Textures can also be used as resources for further processing of an object or image. Image processing based techniques have various uses, a commonly used example outside of the scope of this project being edge-detection. To bring this back to the project, fill-rules can as described in Subsection 6.1.3, for example be implemented using a technique like this. Depending on what is required this texture is either externally generated and then applied as a secondary texture or it is an intermediary point used in a current render.

CHAPTER 4

An Introduction to ipe_animations

The tool `ipe_animations` is a Python-based framework that brings together IPE and Manim. The underlying idea is to keep the creation of the presentation within IPE. The workflow of creating a presentation in IPE would so remain the same. This means the user does not have to bother with any other presentation tool and they can add animations without needing to convert any presentation files. `Ipe_animations` accomplishes this by converting the contents of an IPE-file into structures that can be animated when `ipe_animations` is run. This imported data is then animated using Manim. A summary of the most important features is given in Figure 4.1.

`Ipe_animations` enables IPE users to animate IPE presentations by forming a bridge between IPE and Manim. It decouples animations from an external program's format-specific data. It is, like Manim, based on Python code and provides the following features:

1. render IPE presentations as video
2. insert animations into specified points of the presentations
3. access objects based on custom property in IPE
4. animate accessed objects in a Manim-framework using Manim
5. separate scopes for each page/view-transition

Figure 4.1: Realization of core features specified in Figure 2.2

Input The input required to render with `ipe_animations` is the data contained within the given IPE file as well as a representation of what the contained animations would look like. The IPE file contains all the information on the presentation itself. This includes individual objects with custom properties that can be used for object differentiation within the program. This property is used mainly as a selector, though other uses could be considered. To store the animations, another file, a Python file, sharing the name of the IPE-file is used. This Python file contains a number of decorated functions. These decorators act as a layer of abstraction to make `ipe_animations` more usable. Their usage is explained in Section 4.3. Both the IPE file as well as the Python script are to be kept in the same directory.

Output The output of `ipe_animations` is video.

For now the output is in the media folder, located in the `root/media` directory. This can be changed in the Manim configuration. For reference see Section 4.5. The best way to present the result is to use the Manim Editor. It structures all relevant files and separates sections so each transition happen on user input.

Brief process summary The tool works by first reading the IPE file - importing and converting all relevant data into a Manim context. After this the corresponding Python file is dynamically loaded. All defined methods decorated using the tool-specific decorators are then extracted from the file and stored. As a final step all pages and views are cycled through, initializing the current frame and then running the specified function. This function selects any objects that are annotated with specified properties and introduces them into the context of the method. There the user can manipulate the objects to their liking. Further details on this are given in 4.4.

Example-code While the sections below do explain the functionality in more detail, there is a demo-presentation provided with the source-code to better show the functionality of the project, as well as to act as a test-suite for `ipe_animations`. This can be found under the `ipe_files` subdirectory of the project folder

Description of Manual First in Section 4.1 installation details are explained. This is followed by a more detailed summary of the expected workflow in Section 4.2. Section 4.3 goes into describing the unique method-decorators that are used to make the tool work. In conclusion, in Section 4.5 some miscellaneous tips are given.

4.1 Setup

`Ipe_animations` is a tool that is built on two tools, and requires their installation to work. Depending on the system, this installation can differ in complexity. Full functionality is easiest achieved with Linux-systems. To aid the installation process this

section mentions a few aspects that could become relevant. The following programs are required to use `ipe_animations`:

- Python 3.10 +
  - cppyy
- ipe (linkable libraries)
- Manim version 0.18
- LaTeX-packages as required
- + manedit (as presentation tool)

To use `ipe_animations` linkable library files for `ipe` are required. This is because the program accesses the IPE library written in c++ to read the file contents. These libraries are not provided for all operating systems and may require a manual installation from source. It may become necessary to additionally install qtwidgets and qt respectively. The current Windows release of IPE for example is given as downloadable binaries only. Because of this and how IPE's installation files are set up is it is easiest to set it up on UNIX-like systems. When working with Windows systems it is far more convenient to emulate, as installation requires translating Unix-based make files into a format that windows can handle. This emulation is perhaps easiest done with `WSL`, though any virtualization of a UNIX-system should work. Any references to the IPE library are to be changed in the file `./ipemodule.py`. Ideally, when working with a UNIX-setup, the Manim Editor (`manedit`) is used as a presenter that would separate individual views. It is written based on an older version of Manim, which means that installing it break some of the dependencies used. Reinstalling current Manim dependencies does not influence presenting with `manedit`, as the changes in Manim were a refactor of internal structures not touched by what happens in mere presentations. In a pinch certain video-players can stand in as presentation-tool as well.

4.2 Workflow

There are two steps that need to be taken before the program can run as intended. The first step is to prepare the IPE-presentation, the second is to encode the required animations within a script.

Preparing objects for selection in IPE The first step is to create and prepare the presentation. This part of the workflow of creating a standard IPE-presentation. After this there is only the step of marking the content to edit. The only thing that differs from the standard workflow in IPE is that *handles* need to be added to the objects that are animated. This is done by editing the `custom` property of the object. The scope

considered here is the entire page, rather than the active layers of the view. If multiple objects in a page are given all objects are returned to use in the individual method. Saved changes in the IPE-presentation will be reflected by the next run of the program.

Adding Animations The next step is creating a corresponding `.py` file to the IPE-compatible `.pdf` or `.ipe` file in the directory the IPE-file. The Python script must share the filename. The script should consist of a list of decorated methods as described in Section 4.3. The signature of these methods are to follow the following specification:

method_name(**page**, **view**, ipe_handles, additional_arguments)

Arguments: The order of the arguments given above is strictly defined. However not all of these arguments are equally as important to the framework. There are certain bugs that can happen with errors in these arguments. They are explained in Section 4.5. The arguments correspond to values in the decorators described in Section 4.3.

- **page and view:**
  These arguments are an integer value ≥ 0 each. They describe the IPE-internal numbering of the corresponding pages and views. These two arguments are strictly necessary to make the framework function. Even for the default animation, these arguments must be given, if only to serve as empty filler. The decorators in which these values must be declared are the `@in_context` and `@default` decorators.
- ipe_handles:
  This describes a list of strings corresponding to the amount of different IPE-handles that should be considered in the animation. In theory, these arguments can be left out, this can for example be used for an animation that simply adds a circle somewhere. They correspond to the `@objects` decorator.
- additional_arguments:
  This category of argument is the most optional of all of these, as it is entirely up to the user if they want to hard-code the value into the method or have it defined in a decorator outside of the scope. This is a list of arbitrary arguments that are used within the method. The types and usage is entirely defined by the user. The corresponding decorator is `@settings`.

The main issues that can happen with the arguments is that for some reason the amount of arguments is mismatched. Fixing issues such as this is described in Section 4.5.
Additionally to these arguments the `@default` and `@in_context` decorators take an optional argument defining the method's behavior. This argument defines if the standard animation should be overridden or combined with the new method.

- **PRE** adds the defined method as a sort of prefix to the existing default method, playing it before the default method.

- **REPLACE** is the default value of the optional argument and simply overrides the existing method either in its entirety or the specified page and view pair.
- **POST** acts as a type of suffix to the existing method and adds it after the default method.

Method outline: The method is to follow the pattern given in Listing 4.1 The method name must be unique within the file.

Listagem 4.1: truncated example of a valid function within ipe_animations

```
@in_context(page,view \
#position=Position.REPLACE
)
#alternatively to in_context
#@default( \
##position = Position.DEFAULT
#)
@objects(handle_name)
@settings(additional_arguments)
def method_name(page, view, \
ipe_handles, additional_arguments):
    # insert method content
     ...
```

Running ipe_animations After all of these precursory steps are done, the program can be run by calling

```
Python <path_to_ipe_animations>/renderer.py \
<path_to_ipe_file>/filename
```

The output is generated in the `media` directory.

4.3 Decorators

In order to utilize the core features of `ipe_animations` a set of decorators was created to allow for easier access to the IPE-generated context. The following ease-of-access methods are defined:

Controllers These decorators are responsible for ensuring the transitions are properly situated. They are mutually exclusive and share an optional argument *position* that is a relative reference to the current default transition. They are strictly necessary to insert an animation.

@in_context: places an animation method to replace the default animation. This decorator takes page and view arguments.

@default: This decorator is used to override or extend the current default animation. The default transition/animation is called whenever there is no specifically overridden transition. Baring the optional position argument, it takes no arguments.

The Python script is run from start to end, and the individual methods are copied in each decorated method. This is to prevent issues caused by self-references. Therefore the order in which they are declared will define how the program is run.

Argument-Shortcuts This class of decorators is intended to be a shorthand for arguments within the method.

@objects The semi-optional `@objects` decorator fetches the objects with the defined handle from the page. It is not intended to fetch objects from a different page. This would need to be done by calling the corresponding

```
find_on_page(page, *objs,separate_by_names)
```

method in `Renderer.py`. The amount of arguments must match the number of objects returned. The optional argument `separate_by_names` is a boolean that groups all elements that share the predefined custom properties into a list of objects that can then be considered a group.

@settings This optional decorator is intended to fill in any other positional arguments in the method definition. The order of given arguments must match the method signature.

The order in which the decorators are to be called is the following

- default *or* in_context
- (objects)
- (settings)

4.4 Context-management

The term context describes what is currently contained within the animation. It roughly correlates to the contents of the scene. The control-unit of the context is the `renderer` object. When writing animations that and the objects passed are what will need to be changed.

objects The objects returned by `@objects` are of the `ipe_animations`-internal type `Displayable` or rather its subclasses. This is done so that the methods of these classes can be called. If only simple Manim-operations are intended it suffices to use the objects `render_object` property. Group-objects implement all list-methods and can be used like iterators.

Basic object management: A few common use cases would be realized like this:

Adding an Object:

```
context.add(object.render_object)
```

Removing an Object:

```
context.remove(object.render_object)
```

Animating an Object:

```
context.play(Animation(object.render_object, \
additional_arguments))
```

Resetting the canvas The canvas for each view is set in the method `init_view`, an optional argument defines if a new section is to be begun. This `init_view` method prepares all objects to be rendered as the view properties define it. To simply reset the canvas, call

```
init_view(page,view)
```

within the generating function.

Referencing objects on a different page In some cases one may want to reference an object from a different page. A call to

```
find_on_page(page, object_handle)
```

returns all objects with the handle in their outermost layer. These will all be put in a list in the same order as in the document.

Extracting still-frames It is possible to extract still-frames from the current Manim context. This is done with the method

```
extract_current_frame()
```

as Manim does not immediately provide the current frame. Calling

```
extract_image(page, view)
```

initializes the page and then returns an image of the page's initial state.

Working with invisible layers: For some use cases it might be necessary to work with invisible objects. As individual views can change the appearance of an object, the position of the objects is not immediately set at initialization, but just before it is rendered. To make sure that the object is correctly initialized the method

```
object.prepare_render(current_outside_transformation)
```

must be called.

4.5 Miscellaneous tips

Besides the basic functionality there are some other aspects that can cause problems. A few common error sources or other adaptations are discussed here.

Text: It is important to make sure no empty text objects are in the file, as the implementation cannot handle these.

Number of arguments error: To resolve this check how many objects share the property given in the method handle. Because there is an option to pass these as parameters to the method itself, an additional object with the property can cause problems. The scope here is page-internal.

Counting pages and views: The basis of counting in `ipe_animations` is consistent with IPE's internal library. IPE's GUI and library, however, are not. The GUI begins counting at 1, whereas the library uses 0-indexing. Causes for errors could be that the page or view arguments are shifted by one.

Cached results: Manim caches its animation output in partial animation files. These cached results are used to speed up the process of subsequent runs, by skipping the animation-process if the animation has previously been played. The partial files are hashed. The way this functions Running `ipe_animations` may not always reflect the most recent state of the files. In this case it is best to delete any generated video-files from the `media` directory. For more complicated cases the option of disabling caching can be set in

```
    Renderer.prepare_config
```

Changing the configuration: There are cases in which one might wish to adapt the output format of the program. To a degree, Manim provides options to do so. The intended method to do this. can be found in in the `renderer` class as the `prepare_config` method.

Format and Video players: Depending on which Video player the video is played with and what configuration is being used there may be issues in displaying the video. This is because there is a level of disparity in the different implementations of common video-codecs. The following issues are known:

- Choppy transitions in `h264`
- laggy video
- Wrong color image using VLC and `config.transparent = True`

Updating the libraries may resolve some issues. Otherwise a format conversion with `ffmpeg` can become necessary. A codec that seemed to have no issues, but is not guaranteed support is `h265` or `HEVC`. If all else fails, playing the video over the `ffmpeg` included video player `ffplay` will work.

CHAPTER 5

Implementation Details

The previous chapter, Chapter 4, gives an overview of what `ipe_animations` is. It explains the basic functionality and usage primarily from a user's perspective. Contrasting the user focused manual, this chapter is written from a development perspective. It draws strongly from Chapters 3 and 4. It is an application of the theory outlined in 3 and outlines the processes described in Chapter 4. Chapter 6 is also closely related, as it takes into consideration the faults and omissions in the current state of the code.
This chapter is structured in the following way:
As a first step, in Section 5.1, an overview on the architecture of `ipe_animations` is given. Initially the general course of the program and the main control flows are examined. This is then followed by an explanation of the technical details of what the translation of IPE looks like in Section 5.2. Closing the chapter Section 5.4 is a reflection of more general issues in the implementation.

5.1 Overview

`ipe_animations` needs to fulfill the task of generating an animated presentation from parameters given in an IPE-file and a Python-script. This can be split into a number of sub-tasks, that being

1. importing all relevant data from the IPE-file (See Subsection 5.1.1 and Section 5.2)
2. injecting the defined methods in the script file (See Subsection 5.1.1)
3. combining the information into a generated video (See Subsections 5.1.2 and 5.4.2)

This section treats the structure of the program in a very shallow way. Details will be elaborated on in later sections.

5.1.1 Handling Input

Previously – in Chapter 4 – the way external input from IPE and the Python script is processed was described. This was abstracted to a rough overview of the way that data is processed. Here, the technical details are explained in more detail. The focus of this section is put on the integration of IPE and the animation script. Both of these resources, that is the IPE-file and the corresponding Python-script, are loaded from within the `renderer` class. They share a name and path, only differing in their file extensions:

IPE Files contain a large portion of the data required to construct the final animation. For usability's sake the goal was to directly read in the data from the file. For this different approaches could have been taken. One of them would be to parse the local file. IPE stores data in an XML-format. That means parsing would be a relatively straightforward, if tedious, approach. Taking this approach has the drawback of potentially breaking the program if any part of the specification changes.
Another approach, that is taken in `ipe_animations`, is to load the data from a runtime object from within IPE. As IPE is not a Python-written library, some form of adapting the content is necessary. The tool `ipepython`[1] binds an abstraction of the IPE-library via Lua. This Lua interface wraps the c++ library: This can become somewhat more difficult to follow, as the inbuilt c++ data types are seemingly more analogous to Python datatypes. Debugging can in addition become more complex with that added layer. A linkable ipe library is required regardless of whether `ipepython` is used or not. With `cppyy` there is a good and well documented binding option for bridging Python and c++. It also allows for relatively straightforward adaptation of all relevant objects.

Animations are programmed in by replacing a default animation on a slide per slide basis. It uses the decoration feature Python provides to ensure the animation is played at the defined position. The way this is implemented is by processing and storing the generating methods in a dictionary with the position, consisting of a page and view pair, as search index and input variables. Other variables are assigned by either declaring them in the methods themselves or inserting them via the designated decorators.

5.1.2 Coordination

There are certain limitations to what the call-order of can be as the mechanics for storing animations hinges on the IPE-objects being processed beforehand. This is because all decorated methods are compiled with their decorators already applied. That means that the dynmaically referenced objects need to be known beforehand. The `renderer.py` file within the main module as well as `animations/animate.py` in the animations sub-module are the main coordinating files, ensuring the process runs as intended. The main controller of the tool lies within `renderer.py`. This includes setting up the

[1] https://github.com/otfried/ipe-tools/tree/master/ipepython
last accessed: 08.03.2025

Manim-environment. Described in a simplified manner the succession of calls is the following:

```
prepare_config()
prepare_objects(obj)
_load_animations()

def prepare_objects(obj):
    obj._load_objects()
    obj._count_views()
    obj._get_layer_matrices()
```

This means that the ipe file's data is at first extracted to create runtime objects that can then be used when loading and playing the animations. At first, all animations are read from the file. This process uses concepts of code-introspection as is further described in Subsection 5.4.2. The way playing animations works is by then loading in all animation-generators from the script and then running all methods. A call to `init_view` initializes the content of the view and makes sure all relevant objects are shown on the page. This is summarized in the following code-snippet that is minimally changed from the source code for legibility's sake.

```
#read all annotated methods from .py file
...
play_all()


def play_all():
    for page in pages:
        for view in views:
            #exclude view and page if necessary
            ...
            f = default
            # look up if page and view has a stored animation
            if(lookup(page, view)):
                f = lookup(page, view)
            #play the animation
            init_view(page, view)
            f(page,view)
```

5.1.3 Overarching changes between iterations

`Ipe_animations` is currently in its second major iteration. The first iteration was more experimental. It focused on figuring out the mechanics of combining IPE and Manim.

Most of the changes were focused on cleaning up the code, simplifying and generalizing parts of the code as well as possible. There were however more structural changes, such as restructuring the animation process to be structured with layered decorators as well as pre-processing the generating functions to only take the position as a final argument. In the beginning there was a higher focus on trying to create a library that would provide a few standard animations. As Manim turns out to be quite versatile and well-documented in and of itself, however this idea was later disregarded. Within the `deprecated` directory there is some older code that could be used as a starting point to implement some disregarded features, particularly those described in Subsection 6.1.4.

5.2 Importing IPE data

This section tackles the implementation and architecture of the translation from IPE. It mirrors Section 3.1.1, as the implementation of the structures described there are detailed. First, the details on properties from a wider scope are detailed. That means those properties that relate to *Pages* rather than individual *Objects*. Then further detail on the individual *Objects*' implementation is revealed.

5.2.1 Document and Sectioning

To begin with the broader structure of IPE files is summarized again. The uppermost level of an IPE-document is the *Document* itself. This object is loaded into the Python context. Disregarding the *Pages*' contents, and the *Cascade* the *Document* contains only a few properties that are relevant in the implementation. One example of this is a boolean on whether *Pages* are numbered. IPE is sectioned into *Pages* and individual *Views* for each of those. *Objects* are assigned a *Layer* each. These *Layers* can have a transformation-matrix (an additional affine projection applied to them) per *View*. Additionally, the View-Attribute-mappings are to be found there.
All of these properties are decoupled from the objects and are handled within the `Renderer` class. Because the data structures here can either remain as `c++` type or they are simple ones like a square matrix the implementation of this is just simply reading what is in the file.

5.2.2 Integrating the objects

The basic features of determining the coordinates and overall shape of the rendered objects is handled by the `Displayable` class and its respective sub-classes. As the *Object* class within IPE is practically abstract, so is the `Displayable` class here. To resolve this the factory-method `ipe_to_displayable` separates the objects to their subtypes accordingly. The factory method looks like this:

```
    @staticmethod
    def ipe_to_displayable(ipe_object: ipe.Object, \
```

```
    layer: int) -> Self:

        ipe_object = Displayable.return_as_class_pointer(\
        ipe_object)
        match ipe_object.type():
            case ipe.Object.Type.EGroup:
                return Group(ipe_object, layer)
            case ipe.Object.Type.EReference:
                return Reference(ipe_object, layer)
            case ipe.Object.Type.EText:
                return Text_Object(ipe_object, layer)
            case ipe.Object.Type.EPath:
                return Vector_Object(ipe_object, layer)
            case ipe.Object.Type.EImage:
                return Image(ipe_object, layer)
            case _:
                raise ValueError()
```

There is a second part to initializing objects that handled by the `Style.py` class because certain qualities are dependent on IPE's *Stylesheets*. It acts as an analogue to IPE's *Cascade* `Style.py` file. All qualities beyond an *Object*'s shape and transformations are potentially symbolic *Attributes*, where their values need to be either only extracted or looked up from the *Cascade*. *Text* is determined by its styling even further as the LaTeX-source determines said *Object*'s shape and size.

Certain qualities are directly encoded into the *Object*, others need to be looked up. As the *Object* class in IPE is practically an abstract superclass, there is a distributing factory method within the `Displayable` class. Each `Displayable` object is separated into a rendered object and into its base shape, that is their untransformed shape placed onto the coordinate origin. Any previous translations are encoded into their transformation matrices, as Manim does not preserve their position as the object is transformed. This base object is used to reset the object before each render. After briefly summarizing each object's function some implementation details are given.

Paths are defined by their shape. One of the main complexities of this implementation is properly extracting and distributing the shape-generating functions. IPE has a two-layered encapsulation with minimally varying implementations. This encapsulation is dissolved. Additionally this object type has styling properties that must be applied as a final step in `finalize_object`. This method handles creating object fills Another complication is how objects are filled. This is an issue discussed in Section 6.1.

Groups are composited of several objects and as such its implementation is largely a recursive one. This class includes iterators and implements Python's built-ins used for lists.

Images consist of a bitmap and the canvas they are drawn on. The bitmap is extracted by structuring the buffer and ensuring the channels are correctly sorted. Currently they are rendered with an intermediary step of transforming the canvas and then rendering it in its correct projection via OpenGL[Inc]. This result is then considered a new image where its edges have no opacity. Clipping images is done by applying a bit-mask onto them.

Text objects are mainly created in the `Stylesheet` as their realization is heavily dependent on the LaTeX-source code that is strongly dependent on their styling.

References are practically pointers to an entry in a Stylesheet. In this the pointer is de-referenced and a new Manim-object generated from it.

Each object has the properties `custom`, `render_object`, and `shape`. This `render_object` is what is rendered on the screen - all view-specific properties are applied on it. The `shape` is the object read in - transformations still need to be applied. It is used as a stored state to reset objects to. The `custom` property is used to extract marked objects for a predefined animation.

5.2.3 Style-Cascade

The style cascade is a relatively straightforward data-structure. It is a lookup-table for any properties. In the code-base the functionalities of most IPE-attributes are handled in the class the lookup-table is implemented in. It is responsible for extracting and translating the attributes of IPE into a Manim-legible format as well as storing document-wide styling properties, such as the default cap-styles, line-joins or fill-rules. Its implementation is in the class `Style.py`. This class ensures that the LaTeX-strings are correctly compiled with their styling environments.

5.3 Differences in Representations

Most of the object-translations are straightforward as many elements are commonly used. However, IPE and Manim are designed with different purposes in mind and implemented independently from one another. This results in a certain number of differences that require consideration in converting one format to the other.

5.3.1 Coordinate Systems and Aspect Ratio

A relatively minor difference in the systems is the way coordinates are handled. As Manim initializes the image first thing a temporary `config` needs to be created to override the global values during rendering. Additionally the coordinate origin can be reset in IPE, not so in Manim. This can be solved by simply converting the coordinates through a view matrix.

5.3.2 Initializing Text in IPE

One of the longer standing error-source was incompletely initialized IPE *Objects*. Some *Object* properties are not automatically initialized to their final properties. This applies to *Text* in particular. This is because their width is reset whenever the object is compiled. For this, `ipe_animations` runs a LaTeX-compilation to make sure the size is correctly set.
Another initialization method had to be called to make sure that a page title's styling attributes were correctly set. This was a somewhat hidden error source as looking up a wrongly initialized *Attribute* would return an *Attribute* containing a variably of an arbitrary type. This issue remained undiscovered as in many cases the variable would resolve to something usable. It also seemed that delaying the code execution with prints would resolve this issue. This issue was finally solved by calling the method that applies the attributes to the object.

5.3.3 Unexpected Conversions

Aside from the most common issues with coding there were a few more specific issues that came from differences between ipe and Manim. Not all attributes of an object are standardized. Implementations can differ. These differences in implementation can lead to certain problems in importing objects from one format to another. For instance IPE, it is enough to consider an object in a static position, whereas Manim's objects are dynamic.

Dashed objects: When creating dashed objects in Manim, the object is split into a series of partial outlines. This means that with dash-patterns the connectivity of the objects gets lost. This happens because internally Manim considers only the outline of the object and then splits it up into a number of partial lines. This means that an additional object must be stored to preserve the overall shape, as well as any other properties such as its fill-color. The object's outline is discarded and the dashed outline is added as an additional object. Additionally only simple on-off patterns are implemented in Manim. This was resolved by dissolving the dash-pattern into multiple patterns that combine into the original pattern.

LaTeX: Another issue was based on the way that Manim renders LaTeX-objects. Generating a text-object in Manim is based on first compiling a LaTeX-string and then converting it to an svg-object. The svg-object is then truncated to its closest borders. This svg-object does not necessarily share the same dimensions as the initially compiled LaTeX-string. For simple Text-objects whose actual size and compiled center or their alignment matches, this is not a problem. Minipages with fixed sizes and alignments are, however, a problem. They are formatted with transparent spaces around them. These transparent spaces do not transfer to the svg object. In the end the converted object disregards alignment. This issue is solved by creating a reference object with added helper frames as wide as the Text-object really is, so that it retains its intended size.

Images: While it is not necessarily a bug in Manim, but a difference in specification instead, there is a bit of a discrepancy in how IPE and Manim handle the opacity of images. It is usually not noticeable, because it only becomes an issue when one wants to change the opacity of a partially transparent image. This happens because all α-values, describing a pixel's opacity-values, are changed to the given value. That means an image with a partially transparent background is then given a black border instead. This is in part due to the differences in object representations - particularly that of images. Manim considers only the object that is shown, and changes its state when the user tells it to do so. IPE, on the other hand works with a bitmap that is projected onto the final shape of the object. Resolving this would not be difficult. Adding an additional bitmap object and mapping the opacity to a new scale would be one solution to this. Considering that a number of image-related bugs, mentioned in Subsection 3.2.5, were already worked around with OpenGL, adding transparency into the workaround was a simple solution.

5.3.4 Boolean Operators

The way boolean operators are implemented in Manim depends on Skia-paths. This is not ideal, as is acknowledged by the maintainers[2]. As with images, this is not a fault within Manim, rather than a difference in how the concept of boolean operators is approached. For IPE they serve as a mask on the object, whereas Manim considers them entirely new objects. The discrepancy this causes with IPE is the following: In Manim a comparison of shapes requires both shapes to be filled objects . Otherwise an empty object is generated. Consequently unfilled paths receive an edge they did not previously have. An example for this would be an open curve that is intended simply as a line. While it would be theoretically possible to truncate these edges, there is an issue of uncertainty in both finding intersections as well as determining what intersections are relevant. This is an issue that also comes up with the filling of objects, more concretely considered in Subsection 6.1.3, where solutions to this are discussed. Another, more easily resolved issue based on this same concept comes with the masking object itself. That is, the edge of that object becomes the edge of all other objects. This discrepancy was reduced by adding a white border to the masked object.

5.4 Other Challenges

There were other challenges involved that were more related to overarching issues in the overall architecture and implementation rather than concrete issues related to IPE.

5.4.1 Oddities in the connection of Render and Scene

When working with Manim, one would intuitively assume that for all intents and purposes objects are directly added to the scene. This is for most cases an adequate understanding.

[2] https://github.com/ManimCommunity/Manim/issues/3529,
last accessed: 08.03.2025

In the context of this project, however, a more detailed understanding is required. That is because changing the contents of a scene does not mean that the current frame has already been updated. Resolving some of the issues encountered in implementing `ipe_animations` – both solved (Chapter 5) and disregarded (Chapter 6) – requires extracting a momentary capture of what the Scene contains. A concrete example for this would be the way in which masking images is implemented. Subsection 3.2.5 goes into further detail on why this is necessary. While not implemented yet, there are solutions to yet untackled issues that can make further use of said captures. This concerns almost any potential implementation of the PDF-transition effects IPE provides. This approach is described in Subsection 6.1.3. The need for an extracted still-frame and the lack of an immediate frame update requires manually triggering an update.

A ruse to update the frame: Rendering the scene's contents is very strongly tied to playing an animation. The stored image representing the current frame, will not update if no animation-event is triggered. This applies to still-frames as well. They are represented as a sort of null-animation. A call to `Scene.wait(...)` wraps a call to playing this empty animation. It is, in theory possible to call this animation with no duration. That approach compiles. It works right until the rendered result is combined into the final output of Manim. The issue lies in the animation being indexed, but there being no file that could be attached to the output. Obviously, this breaks the entire program. How to update the frame then?
The way to do it is to trick the scene into playing said animation, but without triggering the mechanism that would save the animation. Taking a closer look at the implementation of the animating mechanism this is possible. `Ipe_animations`' implementation of this truncated mechanism can be found in the `extract_current_image` method within the `Renderer` class.

Using that image: There are a number of potential uses for this image. Some of which are already implemented. This includes the masking of images in particular. Masking images is done by creating a temporary image that only contains the masking object that is then compared to the image as described in Subsection 5.3.4 Future use cases include image-based view transitions, described in Subsection 6.1.4.

5.4.2 Injecting Animations

The way in which animations are injected into the context is by reading all decorated methods from the corresponding `.py` script. This came with a number of hurdles. The script needs to be compiled at runtime, which Python provides features for. That file needs to be extracted for all relevant methods. This is an issue as methods need to be invoked from somewhere. As the file contents are arbitrary, this invocation needs to be based on meta-information about the methods within the script. The final step of storing and later playing the method had its own issues mainly about correctly implementing the corresponding decorators.

Processing the Python script: The solution needed to be able to insert arguments defined outside of the scope of the method. Using special decorators this is doable within Python. It is easily done for with a decorator passing a fixed amount of arguments. As soon as an arbitrary amount of arguments is involved, however, more consideration is required. Because of this arbitrary list of arguments combined with a specified amount of them, the nesting order needed to be defined. Additionally there was the problem of the `@objects` decorator requiring the context given in the decorator it is contained in. Nesting two different decorators, namely `@object` and `@settings`, both defined with an arbitrary amount of arguments was the next challenge. Another issue related to this of encapsulating the function call was in how this method would be stored. Python considers functions to be objects in the same way any other object. That means it can be stored just as an arbitrary object can be stored. Allowing for different options of either replacing or expanding a previously existing standard operation required a way of changing the method. This was an error source for a while. The object status of the method caused recursion related issues, as the name serves as a pointer and changing the function requires a deep copy of the object. A final improvement on this methodology reduced the number of variables at actual runtime to the page-view tuple.

Automated method calls: Another part of the mechanism that was more challenging to implement than anticipated was loading the animation specification into the context. One issue with this was that decorators are automatically when the methods are compiled. This means that any referenced object needs to be initialized beforehand. This is a non-issue for statically defined or initialized objects. The premise of `ipe_animations` is that an arbitrary document can be read. That means that each object is dynamically loaded. Consequently, the animation-script can only be compiled after all relevant objects are initiated. This is something that can be done in Python with relative ease. More difficult was figuring out how to automatically run all defined functions automatically. Doing that came with the need of invoking methods from the script by their name. One issue there is that importing any other dependency adds that method to the scope of the script. That means that the available methods needed to be filtered. Additionally, decorated methods are a separate method to their undecorated base. Calling the incorrect method meant that necessary objects were unknown.

Store and render: Currently what is stored within a dictionary is the animation-generating function. During the development there was a time where the generated sections were stored instead. This lead to a problem where sections would need to be sorted in a post-processing step, which is relatively convoluted within ipe. Finding the object in which the rendered objects are stored required looking through the source code. After finally resolving all issues with the relevant decorators, the final rendering step checks if there is a stored method for the specified view-transition. If none is found the program falls back to a default-method instead.

CHAPTER 6

Further work

In Chapter 2 the potential for creating a better presentation tool has been evaluated. The basics of building this tool have been considered in Chapter 3. The final result of the project has been introduced in both a broad overview as is described in Chapter 4. A more detailed implementation record of the tool including a reflection of issues in the development is given in Chapter 5. This chapter gives a rundown of what future improvements to `ipe_animations` could look like. First, in Section 6.1 a list of omitted features is given. This list is followed by suggestions to include said features. In Section 6.2 further consideration is given to possible expansions in new animation modules. Section 6.3 considers options for further integration with IPE.

6.1 Omissions

Not every feature offered by IPE is as well-implemented as it possibly could be. This is partially because bridging the gap between the implementation of IPE and Manim proved to be more challenging than anticipated. Another reason for choosing to leave them out is infrequent usage. Table 6.1 provides insight into the features that were not included. In further notice, implementation suggestions for these features will be given.

6.1.1 Quick fixes

Some of these features were not included simply because they are infrequently used. They could relatively simply be included. Their implementation could with already existing code be included or fixed relatively quickly.

References: Outlines of references are incorrectly drawn. Presumably this is an issue caused in either the class' `style` method or in the way incoming values are handled in other classes' `style` methods.

Omitted and incomplete Features	
Feature	**Status**
Quick fixes (6.1.1)	
References	~
Arrows	-*
Decorations	-
Attribute-Mappings	-
Sections/Subsections	-
Page Cropping	-*
Constrained Transformations	~
Current shortcomings in Manim (6.1.2)	
Tex-engine	~
Boolean operators	~
Images	~
Overhauling the implementation with cairo (6.1.3)	
Gradients	-
Fill-Rules	-*
Tiling	~
Transitions (6.1.4)	
Effects	-

Table 6.1: Overview of omitted and improvable features
Legend:
-: barely included, -*: attempt at implementation in code-base, ~: included but certain issues remain

Decorations: Decorations and Arrows broadly fill similar roles within IPE. Their implementation is also very similar. In principle this also resembles the existing implementation of References. An implementation would need to look up a *Decoration* Attribute, scale it to the size of the overall group and then render it before any other group elements.

Arrows: Arrows are in principle very similar to Decorations. They have fixed sizes stored in their own attributes. Their position and angle is defined by the Path-object they are attached to. Because these objects are based on to cubic bézier curves as described in Subsection 3.2.2, these can easily be read out from the object's shape. The first and second handles of the object are per definition the position and direction of the shape's starting point. Analogously, the ending direction and position are defined in the penultimate and ultimate handles. Arrows would need to be read from the `Stylesheet`, scaled to their size, rotated by the angle, to finally be positioned to the beginning and ends of the shape. At some point the implementation of arrows was semi-functional. In a subsequent refactor, however their implementation broke. Fixing their implementation would need to be done in the constructor for path-based objects, as well as said class'

`finalize_object` method. In theory all necessary steps have been taken, the methods would only need to be adequately debugged.

Attribute Mappings: They are a sort of dictionary that overrides the standard values of Attributes within a view's scope. They could be introduced in a relatively simple way. Analogously to how layer matrices are stored on a view-per-view basis a set of Python dictionaries could be passed to the methods. This would in theory be relatively simple, but requires a subsequent refactor of almost all methods involved in styling in the objects. This touches all subclasses of the `Displayable` class as well as large parts of the `Stylesheet` class.

(Sub-)Sections: Implementing sub-sections would require changing the naming of the new section in the method that initializes new sections for each view. The decision to exclude this feature was initially made to maintain a chronological order when sorting already rendered sections. This was at a point where – as described in Subsection 5.4.2 – animation results were stored instead of generating functions.

Page Cropping: IPE allows to ensure there is no content outside of a page's frame. This is a setting found in the IPE-file's *Layout* structure, meaning it applies to the entire document. While no significant effort was put into implementing this feature, existing code could be reused to implement this feature. Concretely, this applies to Boolean operators as described in Subsection 5.3.4. Steps necessary would be to pass the Page frame to each rendering pass as a clipping object if the option to crop images is enabled.

Constrained Transformations: Transformations are correctly restrained, however, there is an issue where the positions of objects are not correctly rendered. Going from a previous issue in Manim[1] it is possible that an outer-lying matrix was not considered in the conversion from ipe to Manim. It is also possible that this has something to do with the fact that Manim always re-centers objects when they are transformed. The underlying theory of this issue is discussed in Subsection 3.2.1. This issue would need to be fixed in the `_permitted_transform` method of the `Displayable` class.

6.1.2 Current shortcomings of Manim

Certain features are limited simply due to implementation details in Manim.

LaTeX-compilers: Manim provides only two of the three LaTeX-compilers IPE gives as an option. For the non-included *Xetex*-compiler a warning is given and `ipe_animations` defaults to *Pdf-Latex*.

[1] https://github.com/otfried/ipe/issues/573
last accessed: 08.03.2025

Boolean Operators: As referenced in issues #3529[2] and #3456 [3], the current implementation of boolean operators is not necessarily final. Currently `ipe_animations` implements object-masking in a way where issues in the Manim-internal implementation are being superficially fixed. This should be reconsidered and reworked if a new implementation is given in Manim. This could potentially resolve issues mentioned in Section 5.3.4, where objects receive additional outlines.

Images: For now the implementation of images is partially outsourced to OpenGL. Details on this are explained in Subsection 3.2.5. This implementation choice was made in an earlier stage of the project, and works to resolve some of the bugs Manim has with images. This outsourcing should likely be removed when issues within Manim are resolved.

6.1.3 Overhauling the implementation with Cairo

There is a number of effects featured in IPE that for technical reasons are not implemented within `ipe_animations`. These features cause higher levels of disparity than those mentioned previously. Manim is not intended to implement these features, but they could visually be mocked. IPE and Manim share a dependency in Cairo. Cairo is a c-based library that renders 2d-vector based graphics. Both tools allow for rendering in a Cairo-based environment. And yet, not all attributes of an object are directly transferable. Some aspects of the objects are very different. The shared dependency does, however, suggest that features could be supplemented by utilizing parts of Cairo. The simplest approach would be to simply let Cairo handle rendering the object. This would resemble the way images are currently transformed as described in Subsection 3.2.5. However, the current implementation of `ipe_animations` is not intended use Images within non-image-objects. Adapting an approach based on pre-rendering images would require a refactor of the current data structure. Additionally the same issues as with rendering images would apply.

Separating Static Objects: Another thought to consider is separating static and dynamic objects when objects are rendered. One could render all static objects with `pycairo` directly. This would increase the visual similarity of the objects, as the issues coming from various workarounds would not apply. One drawback of this would be an added complexity to the rendering process. Additionally one would need to consider how dynamic objects in between static objects are separated.

Gradients and Tiling: Two conceptually similar attributes featured in IPE are gradients and tiling patterns. They provide alternative means of filling a path-based

[2] https://github.com/ManimCommunity/Manim/issues/3529,
last accessed: 08.03.2025

[3] https://github.com/ManimCommunity/Manim/issues/3456,
last accessed: 08.03.2025

object. Tiling patterns are currently implemented by reducing the pattern to line-like rectangles and then masking that line with the initial shape. Combining this workaround with other effects like dashed objects, or a purely visual approach in implementing fill-rules, could lead to inaccuracies. This would make it more beneficial to create this fill with Cairo directly instead. This same approach can be used in implementing gradients. In Manim the realization of gradients is not implemented with the same fidelity as it is in IPE. Radial gradients cannot be drawn as the fill of an object. The easiest approach for both of these issues would be to utilize Cairo and generate a texture to serve as a fill. This texture would be stored in an image that is rendered below the object's outline.

Fill-Rules: Fill rules are a set of approaches in which a shape can be filled. There is two common rules that can be applied; (see Subsection 3.2.4). Manim considers only one of those rules. This can for certain types of objects be an issue. A question of principle that needs consideration is whether a purely visual render suffices. In this case it would be enough to replace the fill with a Cairo-rendered fill-object. However if the partition of the object in empty and filled is relevant in a broader sense, for example, if the object is used as a clipping mask, then the object needs to be transferred from the even-odd-rule to the winding rule. Details of such a conversion are described below.

Foundation of a conversion: What needs to be known for this conversion is information on how the object is partitioned. For this, a method exists within the `Vector_Object` class. `Even_odd_mask` is a method that defers the rendering of an object to Cairo, rendering a black and white image for further use. Working under the assumption that the direction of each line is (semi-)arbitrary, *even-odd* and *winding* can diverge if the object consists of multiple disjoint curves, where the directions were not adjusted. In a more complicated case these disjoint curves may intersect with one another. A conversion of an even-odd-filled object to a winding-rule-object needs to result in an object with the following qualities:

- all generated points of both the initial and generated bézier-curve match
- sub-curves that create a hole are wound in the opposite direction of the overall shape.

Implementing this may require partitioning the curve at select points. For this critical points would need to be determined and the object split up at these points.

A failed attempt: One simplified approach was based on first checking the object for all intersections. All sub-curves would be partitioned by these critical points. Finding self-intersecting paths is possible in Manim, however, it is very time-consuming. The time needed could likely be reduced, as there are optimizations that could be made based on geometric properties of cubic curves and the implementation of the used Manim-methods. This was abandoned to only consider disjoint curve parts. The object would be compared

to the next pixel in direction of the object's center. However the approach was attempted and later abandoned because it had issues in its logic. In concrete terms, the approach failed when given partially concave objects. It is not guaranteed that a comparison value in direction to the object's center would be on the inside of the shape, as there is no way to know on which part of the object the outline is situated. This attempt at an implementation can be found in the `calculate_new_object` method of the `Vector_Object` class found in `Displayable.py`.

Image processing: A more sophisticated solution would be based on analyzing the cairo-rendered image. There is a large number of techniques available for analyzing images. The most relevant techniques here would be edge or blob detection. A method to convert an object would either generate an entirely new object from the image or find critical points and split up the original object. An approach that could work well for this would be to separate the objects by their outmost edge, remove those and repeat this for as long as there are objects to be found. There are methods to approximate bézier curves by a number of points [DJEF24]. These approximations should be accurate enough for these use-cases. These groups would alternating between two winding directions. A possible implementation is given as a broadly abstracted pseudo-code in Listing 6.1. Seeing as this method works with close-to binary images, edge detection would be possible and likely cheapest using morphological operations. Methods for tracing a curve from an image would need to be substituted from an external source.
Listagem 6.1: A simplified algorithm for converting even-odd-filled objects to winding filled objects

```
def reorder_shape(shape):
    image = render_mask(shape)
    outside = []
    inside = []
    #add black border to make sure
    #that borders on the edge are
    #correctly rendered
    image = add_1px_black_border(image)
    is_inside = True
    while not all_pixels_black(image):
        image, outline = fill_white_from_corner(image)
        outline = edge_detection(outline)
        subcurves = []
        for blob in find_all_cohesive_shapes(outline):
        subcurves = subcurves + trace_curve()
        if is_inside:
            inside = inside + subcurves
        else:
            outside = outside + subcurves
        image = invert(image)
        is_inside = not is_inside
    return Difference(Union(inside),Union(outside))

#fills image using floodfill and
#checks where the image was changed
def fill_white_from_corner(image):
    outline = image.copy()
    for row in range(image.rows):
        for column in range(image.columns):
            outline[row, column] = False if \
            outline[row, column] == [image row, column] \
            else True
    return image, outline

#traces an image and returns the image as a bézier curve
def trace(image):
    ...

# finds all cohesive shapes as a series of images
def find_all_cohesive_shapes(image):
    # an approach like fill_white_from_corner could be used
    # approach would fill all white pixels iteratively
    ...
```

6.1.4 Transitions

IPE provides a certain set of PDF-transition-effects that can be shown in select PDF-files. These are focused on the slide as a whole and would best be compared to image-transitions. They offer semantic value only in very specific and tailored use-cases. Manim is not intended to provide such transitions. As discussed in Section 3.2.5 images are not necessarily the object-type most focused on in Manim. This feature was left-out as it is expected to bring little value to presentations, but requires high effort to implement. A pure Manim-implementation, depending on the used effect would also significantly slow down the rendering process.

Typification of Effects: Taking a look at what transition effects are provided in IPE they can be grouped into three groups, where each subsequent group requires more effort in implementation and computing capacity.

1. Simple translations: These are relatively easy to add into Manim, because they can be handled with previously implemented methods. Categories consisting of this are: A simple Fade, a None-Transition and images where the old slide is moved away and/or a new image is moved onto it.

2. Wipes: These Linear and/or shape-driven transitions are implementable in Manim, however due to the way images work, this is a relatively long render-time. it is likely more convenient to pass these rendering processes to a dedicated rendering library.

3. Tesselated surfaces: Tesselation refers to changing the geometry of an object within a rendering call. Effects like the PDF-dissolve effect superficially resemble this. These effects split up the surface into tinier partial surfaces and manipulate those.

Implementation: These would all be implementable by extracting the current and next frame and then, if necessary, partitioning them into various sub-objects. When implementing them, it may be a good decision to create these animations as video-frames in a library such as OpenGL that are inserted into the rendered video. Libraries such as OpenGL even provide methods to tesselate objects within a rendering process. The default method in `animations/animate.py` would need to be overridden. Likely the easiest method to do this would be to create a dictionary that links the enumeration of IPE-effects to a Manim-translated animation. This means that for each implemented Effect a method would need to be created as a method. There is an idea of how such image-transitions could look in the classes included in the `deprecated/Animations.py` file. They could be reused for the implementation of specific effects. Another approach could be to combine this with a bitmap based interpolation similar to the approach described in Subsections 5.3.4 and 5.4.1.

6.2 Additional Animation Modules

`Ipe_animations` is designed as a framework that enables users to create their own animations as well. The user can just put together new animations from existing animations. Alternatively an entirely new subclass of the Manim-class `Animation` can be created. There is an overlap in topics covered in presentations. It may be useful to have additional modules handling different sorts of animations. Add-ons could be shortcuts to certain animations and can in certain presentation types become very useful. The submodule `./ipe_animations/animations` is intended to house a group of modules to handle different types of animations. As Manim allows for the creation of individual animation-type derivates, this could even become quite complex. A few examples for how some of these modules can look is taken from Figure 2.1.

Graph-based utilities: A potential expansion could be made to create animations more tailored to specific graphs and graph types. This would require defining a standard representation of graphs, regarding the data passed from IPE as well as creating data structures. If clearly defined this could even include reading a graph from IPE directly. Expansions of this type could be partially based on the graph classes Manim already provides.[4]

Text-tools: Some utilities relating to different texts could be useful, for example parsing the text for all its visible parts. Manim's implementation of LaTeX is not constructed to very easily change visible text. Separating parts of the code can lead to compilation errors where the object is not imported as it should be. The process of importing Text has other issues , as discussed in Subsection 5.3.3. This means that would likely be easiest by changing the LaTeX-code. Another option would be to figure out how individual characters are indexed in different contexts. Parsing texts in its visible characters could pave the way to new types of animations. Itemized lists could, for example be revealed in parts. This would not only mean less work in separating IPE-texts, but also create smoother text units.

Formulas: One thing that could be particularly useful is building a module that can modify and simplify formulaic expressions. This could be an extension of the Text-based animations. There is a precedent for similar concepts in Manim [5], however this hinges on the TeX-strings being properly separable. Depending on the intended complexity this could be highly complex. It could necessitate the deconstruction of the entire term for its constituents. As LaTeX expressions are formalized, this does not come with very high levels of ambiguity. One issue, that might occur is that different variables

[4] https://docs.manim.community/en/stable/reference/manim.mobject.graph.Graph.html
last accessed: 08.03.2025

[5] https://docs.Manim.community/en/stable/reference/
manim.animation.transform_matching_parts.TransformMatchingTex.html
last accessed: 08.03.2025

might be tokenized oddly, particularly in multiplication. A solution to this could be naming all variables and using those names as a dictionary. Another difficulty could be that structures such as exponents would require more attention. An add-on relating to formulas requires decisions on a format to be made. Outsourcing operations to another tool could simplify this.

6.3 Integration

For now `ipe_animations` is comparable to a backend library that renders the intended animations, but requires code as an input. Future iterations could benefit from further integrating this Python tool. Some actions to increase this integration are described here.

Exception handling: There are a number of exceptions that can break the code. They output the concrete call-stack for now, This could be improved on. One issue is that empty Text objects will cause the program to fail.

Adapting the configuration: There are certain parts of the Manim-configuration that could be changed to work better with the user. This would for example be changing the output directories of the relevant files to be bound to the path of the input files.

Consistency within IPE and ipelib: As mentioned in Section 4.5, the is consistent with the IPE-library rather than IPE's User Interface. This could be changed by changing the `in_context` decorator to automatically decrease the number by one, and having `init_view` output the files according to the 1-indexed user interface.

Combining saved individual methods: Another aspect could have improved consistency is how default and view-specific animations are handled. For now it is only possible to combine new methods with an existing default method. What could instead be done is that if a method for this scope is already defined the new method is added on.

GUI: It would be possible to create a front-end that can directly be addressed from within IPE. One approach that could be taken here is that said front-end writes the Python code responsible. An ipelet could be written for this purpose. The question is how exactly the interface for this would look. Keeping the written Python code could be used to further adjust specific methods in more precise and versatile ways.

CHAPTER 7

Conclusion

This thesis lays the groundwork for a tool that can animate IPE. It is for now a rather basic tool, that requires the user to directly encode the added animations. At first the possible usage and technical requirements were considered. To get a picture of what could be used a series of conference presentations were analyzed. Later the features of a selection of presentation tools was considered. The combination of tools was also considered in the. This led to the conclusion that an implementation based on the programs IPE and Manim could be the most useful approach to create such a tool. In a subsequent analysis of the fundament given by the choice of these tools some of the underlying theory of Computer Graphics were considered. This tool was implemented. The implementation was documented in its setbacks and the challenges that came with bringing together both ipe and Manim. The main challenges here were finding workarounds for features Manim was never designed to provide. The other part of this project was finding a mechanism that would effectively inject animations. This was a challenge insofar as advanced Python-features had to be taken into account. Current shortcomings were also analyzed in their cause. It could be argued that other tools, like RevealJS, could have provided for easier object translation, as svg-support in Manim could be better.[1]. On the other hand, Manim is specialized for animating, and allows for the easy creation of new animation-types. The already implemented animation types could also be reused for new Animations, using different objects. Using inheritance it should be comparatively easy to create new animations based on mathematical concepts. While Manim has its shortcomings, these can be worked around. The developed tool `ipe_animations` is intended as a backend-library. Integration with ipe could in future iterations be improved. Additionally lacking IPE-features could be implemented, as well as new animation-types created.

[1] https://github.com/ManimCommunity/Manim/issues/3709
last accessed: 08.03.2025

Bibliography

[DJEF24] Maria Dziuba, Ivan Jarsky, Valeria Efimova, and Andrey Filchenkov. Image vectorization: a review. *Journal of Mathematical Sciences*, pages 1–14, 2024.

[HA14] John F. Hughes and Kurt Akeley. *Computer graphics : principles and practice.* Addison-Wesley, Upper Saddle River, N.J., 3rd ed. edition, 2014.

[Inc] The Khronos Group Inc. Opengl.

[Ink] Inkscape Project. Inkscape.

[MS18] Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics.* A K Peters/CRC Press, Boca Raton, FL, 4th ed. edition, 2018.

[Otf24] Otfried Cheong. Ipe, April 2024.

[Ri6] Aleksas Riškus. Approximation of a cubic bezier curve by circular arcs and vice versa. *Information Technology and Control*, 35, 01 2006.

[The24] The Manim Community Developers. Manim – Mathematical Animation Framework, April 2024.