3D-Visualization of Theory Graphs

Master's Thesis in Computer Science

submitted by

Richard Marcus

born on November 29, 1994 in Forchheim

written at

Professorship for Knowledge Representation and Processing
Friedrich-Alexander-Universität Erlangen-Nürnberg

Supervisor: Prof. Dr. Michael Kohlhase

Start of Thesis: October 24, 2018

End of Thesis: April 24, 2019

Abstract

With the ever-growing amount of information, knowledge management systems have immensely gained in importance. Mathematical libraries are especially challenging; the contained knowledge admittedly is complex but, at the same time, also very structured.

OMDoc/MMT **theory graphs** are capable of representing this knowledge, but the special structure causes problems to traditional graph systems. Two spatial dimensions may not suffice to organize theory graphs in way that they are usable in practice.

Consequently, this thesis explores how 3D techniques can improve this. Virtual reality devices in particular offer efficient ways of visualizing large amounts of data. But it is not possible to simply predict the effectiveness of these measures. so an experimental system, **TGView3D**, was developed. TGView3D uses the Unity game engine for rapid prototyping and aims at implementing a set of features to support 3D theory graph workflows. Therefore, an overview of related systems and algorithms is given. Then, an introduction to the knowledge management systems that surround theory graphs follows. These include powerful tools that can be useful to users of very different systems. Starting from this, requirements are derived to design the system architecture of TGView3D and identify interactions with theory graphs that can actually help users.

Accordingly, this work describes the integration of 3D and VR methods into the ecosystem of theory graphs. The actual implementation of TGView3D starts with finding a way to visualize theory graphs in 3D in the first place. Furthermore, it is necessary to investigates ways of exploiting the 3D space to compute a graph layout so that users can better understand the represented structure. As even the best algorithms fail when graphs exceed certain sizes, approaches to explore the graph and reduce the information are vital. Here, the VR devices — which are optimized for interacting in 3D worlds — are ideal to replace traditional 3D input devices.

Based on the lessons learned, this thesis evaluates the benefits of visualizing theory graphs in 3D and gives an outlook on future research topics regarding 3D data visualization in general.

Contents

Chapter 1

Introduction

Sharing and storing knowledge has always been an important factor in driving technology forward and the amount of newly generated information is not slowing down — instead, we even speak of **information explosion**. Since humans cannot keep track of this by themselves, we need sophisticated systems that can manage this knowledge.

Especially, mathematical knowledge requires careful integration because of its inherent complexity. Libraries of both informal and formal mathematics have reached enormous sizes: at least half-a-dozen of them exceed 10^5 statements. Thus, it is getting more and more difficult to organize this knowledge in a way that humans can understand and therefore access it. This can lead to the duplication of formalizations because users are unaware of them or cannot find them. As countermeasure, systems often employ modularization features to introduce a high-level structure that helps to navigate through the libraries.

Many of these features can be captured in the language of theories and theory morphisms as used in, e.g., the **OMDoc/MMT** format [Koh06; RK13], which has been used as a uniform representation standard for libraries related to theorem proving, computation, and mathematical documents [KR16]. Based on this standard, we can derive **theory graphs** [Koh14], which can graphically present the formalized knowledge. To help users understand this high-level structure better, there are first experiments that have explored visualizing these graphs with the help of the interactive 2D graph viewer **TGView** [RKM17]. But the sheer size of theory graphs — they can easily contain thousands of nodes and an order of magnitude more edges — complicates the process of computing structured graph layouts.

Therefore, it may be necessary to go beyond. 3D data visualization techniques offer solutions to explore graphs more efficiently. The third, spatial dimension naturally extends the space that is available for organizing the graph. This becomes especially relevant with the recent advances of hardware. On the one hand, consumer **graphics cards (GPUs)** and **processors (CPUs)** can easily render huge amounts of data or at least handle data in sizes that make sense to present to the user.

On the other hand, **virtual reality (VR)** devices are gaining traction. While this had already happened in the past, the quality was only sufficient for niche gaming purposes or required significant effort to use them in the form of **CAVEs** (Cave Automatic Virtual Environment) [CNSD93]. Virtual reality today is usually realized in the form of **HMDs (Head-**

mounted displays). These offer a wide field of view and an appropriate display resolution to visualize scientific data. Furthermore, these devices offer intuitive interactions with virtual worlds and 3D data. They achieve this by introducing sensors that can fully track controller and head movements. Accordingly, the fusion of knowledge management systems and 3D data visualization is a very promising endeavor.

1.1 Research Question

This leads us to the research question of this thesis: **what is the potential of 3D visualization for theory graphs?** Does this help us to understand them better, explore them more efficiently or interact with them more intuitively? To answer these question, however, we need to consider multiple aspects. First of all, there is the visualization itself as there is no predetermined way of representing theory graphs in 3D. Elements such as text labels or different forms of edges cannot simply adopted from 2D graph viewers. Then, there is the layout computation. At first glance, an additional dimension seems to be an advantage for organizing the graph, but monitors can only show 2D projections anyway.

Closely connected, there is the navigation within the space. While a computer mouse is ideal to move objects within a plane, different concepts to go forward and backward may become necessary. An efficient 3D visualization in combination with appropriate exploration tools can then again be very promising. Another crucial part of this work is the utilization of virtual reality devices. Controllers optimized to interact with 3D worlds and a huge field of view can be solutions to many problems by default, but how can we integrate these into the workflows of scientists? Especially the communication between an immersive virtual reality application and external systems poses a challenge. To find solutions, it is necessary to experiment with the implementation of these features in practice.

1.2 Contribution

Consequently, I have developed a virtual reality graph viewer, **TGView3D**. The system uses the **Unity** game engine and is licensed under GPLv3. The code is is available at `https://github.com/UniFormal/TGView3D` and a web port at `https://tgview3d.mathhub.info`. A demo video can be found at `https://www.youtube.com/channel/UCnEdCn7X4m7dqcNxq7okzDA`.

Looking at existing systems, the combination of theory graphs and VR graph interaction is not feasible with them. The reason for this is that theory graphs have particular structure and its users have special information needs. A subset of certain edge types induces a hierarchical inheritance structure that is a key component of theory graphs and thus needs to be reflected in the layout. Beyond, theory graphs contain nodes and edges that are rich in information, even seen individually. Other systems often only support very basic access to the information contained within the graph and focus on highlighting the structures. At the same time, there is a huge variety of work that is in some way related to this thesis; I will introduce these specifically in Chapter 2.

With TGView3D as an experiment for exploring the space of all 3D interactions for theory

graphs, I first of all propose a way of representing theory graphs in 3D. This also includes presenting the contained information directly within the graph as well as in a user interface. To visualize the graphs, I compare different ways of integrating the hierarchy of theory graphs into the layout computation and arrive at a new way of computing the layout, which we can apply to graphs with hierarchic structure in general. This variant defines additional hierarchic forces and constraints that guarantee that the directed edges point upwards consistently.

Furthermore, I contribute concepts of interacting with 3D graphs with the help of virtual reality devices on the one side and special filtering methods on the other side. In particular, these allow the user to focus on certain paths of the graph to explore it locally — down to the information contained within a node — as well as globally — by navigating through structures that are highlighted by the layout algorithm.

To make the system itself interesting for the future, I have identified and experimentally integrated workflows of actual users. In this sense, TGView3D is not only an experiment but also a system that users can already work with.

1.3 Structure

First, we will take a closer look at the related work in Chapter 2. After that, we will establish the foundations of this thesis and also TGView3D in Chapter 3. This includes the concepts that introduce the need for graphs to structure mathematical knowledge as well as systems and algorithms I have used as a starting point. In Chapter 4, we will learn about different use cases and arrive at a system architecture that can support these. For the general implementation of TGView3D, we distinguish between three parts.

1. **Rendering** The process of representing a theory graph in 3D, starting from mathematical archives (Chapter 5).

2. **Layout Computation**The development of different variants to incorporate the hierarchy of theory graphs into a classical forces-driven layout algorithm(Chapter 6).

3. **Interaction** The design of intuitive ways of exploring a theory graph with the help of VR interactions (Chapter 7).

After that, we look at the system as a whole in Chapter 8 and evaluate how well it achieves the goals defined before. Chapter 9 concludes this thesis and gives an outlook on possible future research topics that have emerged during this work.

1.4 Acknowledgments

As project that is influenced by different fields such as knowledge processing, mathematics and computer graphs, many experts have been involved. First of all, I appreciate input regarding potential use cases from Max Rapp, Florian Rabe, Claudio Sacerdoti Coen, Theresa Pollinger, Navid Roux and Thomas Koprucki. This was crucial for exploring how users want to interact with 3D theory graphs. Furthermore, I acknowledge hardware support and very

helpful discussions about layout algorithms from Jonas Müller, Roberto Grosso and Marc Stamminger. Without them, this thesis would have not been possible in this form. I also must not forget Dennis Müller and Tom Wiesing, who both helped me with the technical aspects of MMT. Of course, I also thank the remaining members of the KWARC group for always being glad to give feedback. This especially includes Michael Kohlhase, who was very open to experimenting with new methods and technologies, and Marcel Rupprecht, with whom I constantly found myself debating about various design choices.

Chapter 2

Related Work

Developing a tool such as TGView3D borders on different areas. First, there are graph algorithms such as layout and cluster computation as a theoretical foundation of visualizing the data. Then, we discuss systems and frameworks that implement these algorithms in respect to specific environments. In addition to that, we need to look at research that generally explores interacting with data in VR. Note that I address concepts that are integrated into TGView3D or part of its ecosystem in the Preliminaries.

2.1 Graph Layout Algorithms

When computing the layout of an arbitrary graph there is — as is often the case in computer science — a trade-off between quality and speed. Consequently, there is research towards finding different methods to this as well as efficient implementation of existing algorithms.

2.1.1 Quality Optimization

Without fixed geometrical positions, we have to position the nodes of a graph in a way that the whole structure is visually pleasing and organized. The general way of doing this is to define certain qualities, derive heuristics and use these to optimize the layout iteratively [DB+94]. While generally more iterations result in a better layout, the quality of a layout algorithm foremost depends on the used heuristics and their implementation.

Force-directed Layout Algorithms A very intuitive way of doing this are **force-directed** algorithms. These are based on an early approach of Tutte [Tut63], which calculates the layout for specific graphs by solving a system of linear equation. The concept of physics inspired forces was introduced by Eades [Ead84] in the form of **spring embedders**. Based on springs, nodes only attract other nodes that are connected to them, but repel all of them. By performing multiple iterations a force balance is attained, generating a layout that forms groups and reduces the average edge-length. Since this will be a core part of the layout algorithm for theory graphs, I will describe a variant proposed by Fruchterman & Reingold [FR91] in Section 6.1. There is also an alternative way of modeling springs proposed by Kamada &

Kawai[KK+89], which uses the graph theoretic distance[1] to optimize the layout.

Representing Specific Structures While spring embedders do generate aesthetically pleasing graphs, the common criteria do not apply to all graph applications. Theory graphs, for example, must represent the theory inheritance, which is a prime factor for understanding.

One way of doing this is to base the layout on tree structures. For example, Walker's Algorithm [QWI90] tries to optimize tree layouts by minimizing their width. If we look at a tree, we will roughly recognize parts of the tree as triangles. Two triangles next to each other then have a lot of free space in the upper areas where they are very narrow. The general idea here is to organize the upper tree levels in a way that they fill[2] the space efficiently. There are two problems with this algorithm: first, the optimization of the tree with in 3D becomes a significantly more complex problems and, second, trees are problematic as basis for the hierarchy of theory graphs as we will learn in 3.4. Nonetheless, a similar approach will be the foundation of the layout algorithm for theory graphs in Section 3.5.

2.1.2 Performance Optimization

As a medium of representing information, graphs are utilized in very different areas such as road networks, social media, molecular interaction or even the World Wide Web itself. Some of these can be extremely vast — orders of magnitude bigger than the mathematical libraries we have mentioned — and easily include millions of edges and nodes. To name an example, the Texas road network dataset [TRN] already contains nearly two million edges. This complicates any kind of interactive computations and requires sophisticated algorithms and systems as countermeasure. Such approaches range from developing compromises regarding layout algorithms and interaction methods to finding optimal ways of exploiting the available hardware, e.g. many-core architectures. When designing systems that are supposed to handle interactions with extremely big networks, researchers often combine these approaches. For example, Mi et al. [Mi+16] first reduce the computational load by approximating the force calculation and then also implement their algorithm on the GPU.

For TGView3D, we can largely avoid these topics because the existing theory graphs currently only include at most thousands of nodes edges and the motivation of this thesis was rather the exploration of how interaction with 3D theory graphs can work in general. Yet, we will need to consider the performance on occasion in order to guarantee a pleasant user experience. In these cases we will mostly focus on removing the biggest performance bottlenecks and not attempt be as efficient as possible.

2.2 Clustering

An additional step to improve forming of groups and organizing the graph is **clustering**. Clustering means that based on some kind of closeness measure, certain sets of nodes are put

[1] The graph theoretic distance is the length of the shortest path between two nodes
[2] Imagine playing Tetris

into one group, the so called cluster. The cluster information can then be incorporated into the layout. There are different ways of applying clustering to theory graphs.

Semantic Clustering For this variant, additional information acts as the basis for clusters. This could range from the corresponding mathematical topics to meta–information such as the year it was added. We can find a similar approach in the form of the OpenMathMap project [DKL13], which uses other visualization methods than graphs. This is the most effective method to form meaningful groups within theory graphs because these groups are already defined externally. At the same time it can be very difficult to use this kind of clustering. If there is no additional information available or if it is not accessible for TGView3D there is no way to implement this variant. Unfortunately, this is currently the case for theory graphs.

Graph Clustering Not relying on any external sources of information, graph clustering algorithms build clusters by computing special measures derived from the nodes and edges. Again, the quality and performance considerations are equivalent as described for the graph layout algorithms in Section 2.1 Methods like this can of course only guess where to draw the line between to clusters, but can in theory be applied to every existing graph. Schaeffer gives an overview of different approaches [Sch07].

These often operate on specific measures or even combine these [ZCY09]. Yet, many of them do not explicitly address directed graphs or hierarchic structures, which occur in theory graphs. TGView implements an algorithm based on the **betweeness centrality** [Bra01] of nodes, which is defined by the number of shortest paths that pass to through each node. The result of this approach is interesting in its own right as it emphasizes the relations within the theory graph, but it it loses the hierarchical structure.

There also is the concept of **hierarchical clustering**, which can be understood as recursive clustering. While this is essentially what we would like to have, there is a decisive difference: this variant does not operate on known hierarchical information, but instead tries to discover hierarchy within arbitrary data. I will touch on possible ways of exploiting the information we already have in Section 8.2.1. Beyond that, the author decided to leave the exploration of graph clustering methods for future work, especially as there is a useful alternative in the form of spatial clustering.

Spatial Clustering Spatial Clustering forms groups by only considering the positions of nodes. Such techniques are often used to separate data sets in machine learning contexts. TGView3D uses an adjusted DBSCAN [Est+96] algorithm, which I describe in Section 7.6.

2.3 Graph Interaction and Visualization

Data visualization in general is a very diverse topic. Just as theory graphs have very specific attributes, other graphs also have their own characteristics. Beyond that, graphs are not the only kind of data that is suited to visualize data. Point clouds, heat maps or even regular charts offer very valuable insights. Despite their differences, they often share certain properties. All

of them need to be rendered somehow and often benefit from interactivity. User interaction and layout algorithms also do not only apply to one specific data set, but instead to a whole category.

2.3.1 Visualization Frameworks

To support as many use cases as possible, frameworks only offer optional modules and features that can be used to build custom applications. Still, there always will be some kind of specialization to guarantee that the framework offers high quality implementations.

The frameworks that are designed to handle different kinds of data often only implement a basic set of features and require the programmer to fully implement use case specific interactions.

vis.js Vis.js [VJS] is a JavaScript based browser based library that follows this concept. Regardless, it introduces some specific features by grouping possible applications into modules such as charts, timelines or networks. For the purpose of graph visualization, the network module is relevant. It includes individual node and edge interaction, e.g., manual clustering, dragging or event triggers, as well as one force-directed layout and a hierarchical one for tree structures.

D3 Also based on JavaScript, the D3 [D3] library supports an even wider array of applications. It follows a very interesting concept of binding the visualized data to document elements to allow direct manipulation [BOH11]. While there are less predefined interactions, D3 is very flexible and efficient in regard to visualization. Modules offer a large variety of features to represent different types of data. For graphs, there is a force-directed and hierarchical layouts similar to vis.js.

Although it would be possible to build a 3D system based on these frameworks, they would not offer much help. We can neither use the supplied layout algorithms nor the interactions in combination with 3D visualization. We should also note that these frameworks are limited by their respective rendering systems. Even for 2D applications, WebGL [WGL] offers a big advantage in performance we will learn in 3.6.

2.3.2 Graph Systems

To support specific workflows there are full visualization and interaction systems. While they often allow modifications, they come with a defined set of interactions and algorithms to allow users to directly work within these environments.

Graphviz Graphviz is not only a single system but rather a collection of different tools [Ell+01]. For example, directed graphs can be visualized with **dot** [Gra]. It processes input in form of the DOT language. As Graphviz can generate many formats such as SVG and PDF it is an ideal choice to integrate the graphs into documents. However, it is difficult to integrate interactions into these static formats, which also has been the motivation to build TGView.

Gephi Gephi [BHJ09] is a platform to specifically visualize and explore graphs and runs on common operating systems [GEP]. It is based on the OpenGL 3D engine and thus can visualize graphs in 2D as well as 3D. Furthermore, Gephi implements graph algorithms such as community detection, filtering methods also different layouts. Combined with its exploration tools, Gephi offers many features that are also desirable for theory graphs. Unfortunately, it would be hard to integrate VR devices into Gephi and there would be no option to use the application in a web browser. Although this is not part of this work, it may be interesting to load theory graphs into Gephi to investigate which additional methods might be helpful for theory graphs.

Neo4j Another interesting system is Neo4j [Mil13]. Its main motivation is the implementation of a **graph database (GDB)** system. Based on the Neo4J platform, there is a whole range of products and solutions. These range from GDB features such as data storage and querying to different graph algorithms [N4J]. This also includes several graph visualization tools [N4JV] including Bloom, which, for example, allow editing nodes or initiating queries based on the presented graph. In its essence, this is the same idea that is behind MMT and theory graphs. MMT, however, offers powerful features that allows the representation of mathematical archives. Additionally, Neo4J seems to be mainly aimed at business customers, although it does offer a community version.

2.3.3 Theory Graph Visualization: TGView

As already mentioned, TGView was the first real[3] approach of developing an interactive theory graph viewer. It is a web application that uses traditional 2D visualization. Since theory graphs cannot be represented very well by existing systems, it uses the vis.js library. This allows the necessary flexibility while still offering interaction components that can be used as basis. TGView already had identified force-directed layout algorithms as very valuable, but the hierarchic structure proved to be more important for use cases in general.

Figure 2.1 [RKM17]

Layout Computation TGView offers mainly three different layout: a force-directed one, a hierarchic one with different heuristics and a combination of both. This combination tries to find a compromise between the correct representation of the hierarchy and optimizing the layout by reducing the edge length and edge crossing etc. TGView prioritizes the latter and thus allows edges to proceed against the hierarchy direction. This also reduces the computational load, as there is no full determination of the correct node ordering.

Graph Interaction TGView offers a base set of filtering and navigation options. For example, the user can disable edge types individually and zoom into parts of the graph. The area where TGView really shines are the interactions on the node and edge level. It implements the full stack of possible editing functions, which even allows to create graphs from scratch.

[3]There was an experiment of introducing interactivity to GraphViz with the help of an overlay, but that did not work that well...

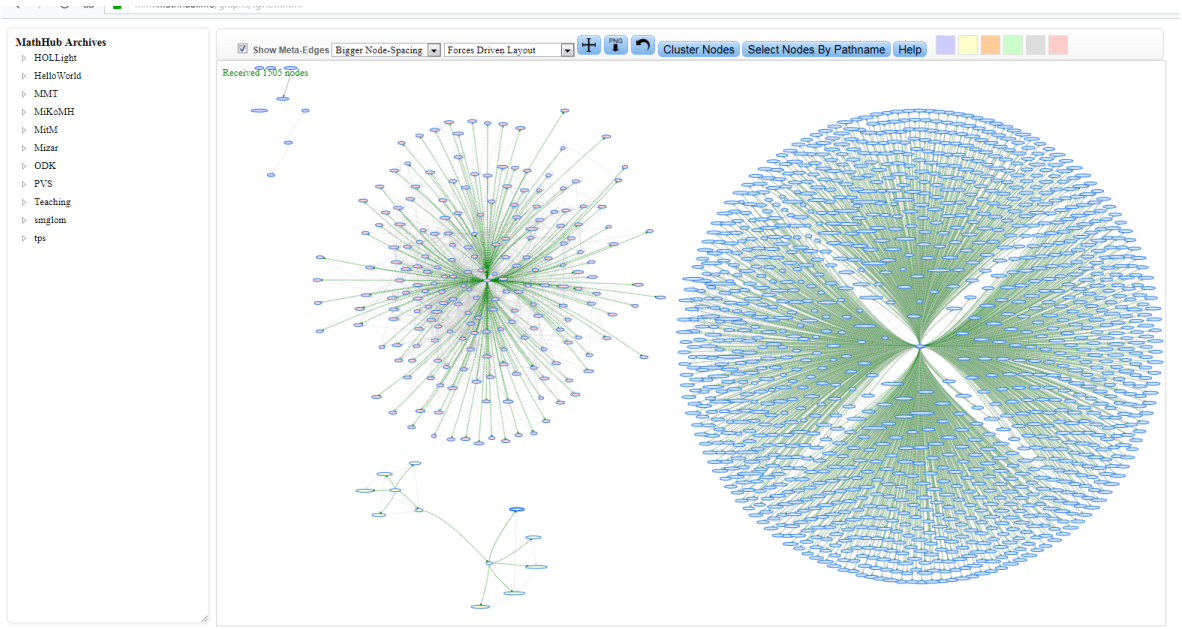


Figure 2.1: Screenshot of Theory Graph within TGView

First of all, it is possible to delete and create nodes as well as edges. The user can undo and redo his actions or even save the created graph. Furthermore, there is the option to select multiple nodes and accumulate them within a new cluster node. All nodes can be moved with the mouse and reveal options to access further information, e.g., by following the MMT URIs.

As an experiment for 3D interaction with theory graphs, TGView**3D** ports some of these interactions to 3D, leaves out others and implements a range of new features.

2.3.4 Immersive 3D Data Interaction

Interaction with 3D data also creates many challenges and requires reassessment of solutions for 2D interactions, which can draw on a significantly longer tradition. Especially virtual and augmented reality devices offer new solutions. Integrating these into graph visualization systems such as the ones described in Section 2.3.2 is often difficult as there is no direct interface for this purpose. Game engines on the other side provide this, which I will describe in Section 3.7.

Looking at related experiments of visualizing graphs with the help of virtual reality devices the volatility of this field becomes apparent. There are early approaches that use Caves, which propose the use of multiple projectors to create a stereoscopic 360 degree virtual reality environment. While there is research, how this can benefit scientific visualization in general [Bry94], others have built applications to support certain workflows. Yang et al. [Yan+06], for example, visualized metabolic networks with this method and support interactions with an additional 2D GUI.

While we still can draw conclusion from these experiments, the technology today has drastically changed: **HMDs (Head Mounted Displays)** can be manufactured with high

resolution and movements can be tracked in real time. As these devices are entering the mainstream, virtual reality data interaction concepts are revisited, e.g., focusing on general virtual reality graph exploration [CEG18] or interfaces [EMP18].

Other work rather focus on user studies that compare a proposed virtual reality system with traditional 2D displays regarding certain interaction tasks such as finding paths within the graph or recalling node locations [Kwo+16]. There are also studies that investigate the influence of virtual reality environments on visual memory techniques[KPV18], which can allow insights on how systems should be designed and how we benefit from visualizing data this way.

With theory graphs as a very special type and the different devices that have released recently it is difficult to adapt one specific interaction concept as general solution. Instead, I conducted multiple demos with potential future users and gathered useful feedback to develop features that can assist their workflows. The conduction and evaluation of specific user experiments to compare the effectiveness of TGView3D against other systems, however, is not part of this thesis.

Chapter 3

Preliminaries

This thesis is situated in the intersection of different research areas such as knowledge management, graph visualization and virtual reality. Accordingly, we will begin with the origins of theory graphs; they have emerged in the context of not only representing knowledge but also bringing different systems together. We then will proceed with the algorithmic and the technical foundations of building TGView3D. More precisely, we will first revisit layout algorithm concepts and then discuss the choice of framework and hardware.

3.1 Graphs in the Context of TGView3D

As very first step, we need to look at graph concepts in general. This will be important for the aspect of representing knowledge as well as for the visualization itself.

3.1.1 The General Structure of Graphs

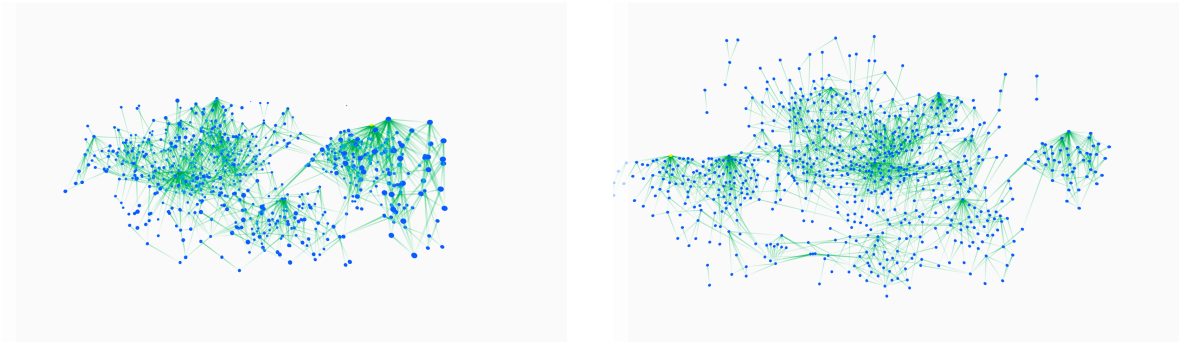
Graphs have two components: **nodes** that represent arbitrary elements and **edges** between nodes that represent relations. According to the type of relation, an edge can have either have a direction or not. If a node is reachable from itself — moving along edges in respect to their directions — we refer to this as **cycle**. While undirected graphs contain cycles by definition, there can be directed graphs without them. In this case, we call the latter **directed acyclic graphs (DAG)**.

With these attributes, graphs may show certain patterns, e.g., there can be parts that are strongly connected and others that are connected weakly. Representing knowledge as graphs not only allows machines to process it, but also offers humans an intuitive way to explore and better understand the contents. Imagine you are given the task to present a topic in the form of a poster. Often, such posters will be very similar to graphs. You will find closely related topics in proximity and maybe also arrows that indicate some relations between them [PK15]. Even this thesis can be understood as a graph. Chapters, for example, have hierarchic relations to other entities such as sections and subsections. References are then directed edges between these entities and can introduce cycles.

Yet, there is no predetermined way of visualizing arbitrary graphs. As a result, there is

a whole area of computer science that addresses this topic, called **Graph Drawing**. This brought forth a wide variety of different **Layout Algorithms**. The main idea behind these is to bring out the selient structures in the respective graphs so that it becomes easier to recognize them.

For example, we should position elements that have many relations between them close to each other. This way we minimize the length of the edges and form groups in the layout. Traditionally, these algorithms use two spatial dimensions to span the graph.



(a) The 3D Layout shows cluttered graph

(b) The 2D Layout has better separation

Figure 3.1: NASA PVS Library Theory Graph with 739 nodes and 2851 edges

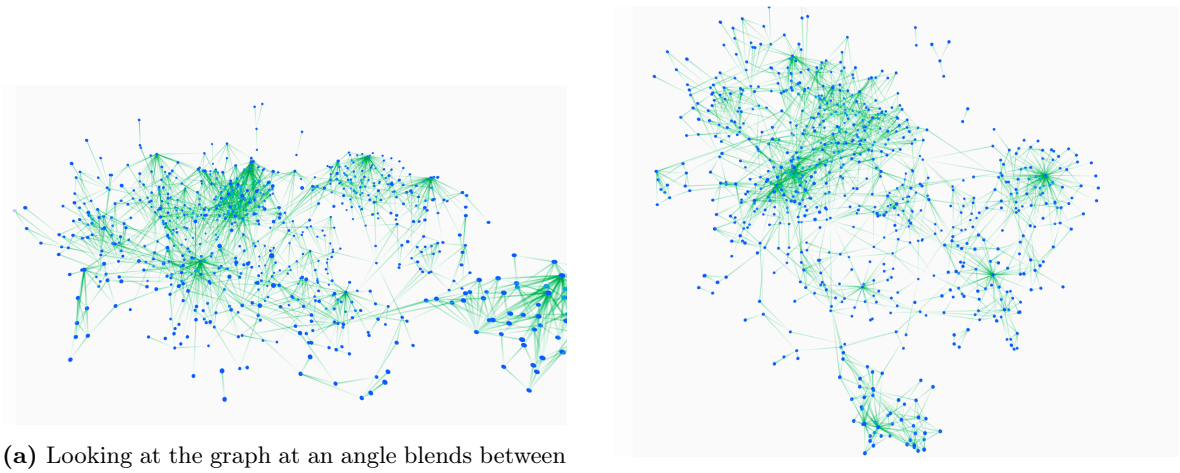
3.1.2 Graph Visualization in TGView3D

Although using three dimensions offers more space to portray complex graph structures, there are significant practical problems. First of all, any static image of a 3D graph is a projection to the 2D space and thus a 2D graph.

An image of a 3D layout can then easily degrade compared to using only two dimensions to begin with, as Figure 3.1 shows. Therefore, I present visualizations of theory graphs in this thesis as screenshots of 2D layouts. Aside from that, I have externally modified the screenshots to make them more pleasing to view on paper. Within the application, there is a dark color theme for the background and vibrant colors for the graph. To avoid the possibly distracting contrast to the white of the paper, I have inverted black and white. An exception are screenshots of VR interactions; these naturally require 3D graphs and benefit from the original color scheme.

The actual advantages of 3D graphs arise from using **interactive** graph viewers. These allow to explore the 3D structures of the graph and filter information by inherently highlighting the structure of the graph that emerges from the current viewing angle. While impossible to show with pictures — we can try to get a brief glimpse of the advantages of a 3D graph layout with the help of the screenshots in Figure 3.2. They are taken from different angles to showcase how a single 3D layout contains multiple different 2D layouts. Actually inside the virtual world, users have an unlimited number of such projections and perceive data in novel ways.

However, navigating a 3D graph with keyboard and mouse is not very intuitive.



(a) Looking at the graph at an angle blends between the groups and the structure from bottom to top.

(b) View from above reveals multiple groups.

Figure 3.2: Screenshots from Multiple Angles: NASA PVS Library Theory Graph

3.2 Interaction in Virtual Reality

This is where virtual reality devices come into play. These offer solutions to all interaction issues and are ideal medium to visualize huge amounts of data. This is especially true for HMDs, which I further characterize in Section 3.8. The general concept is that the users put these devices onto their heads and are immersed within a virtual world. In particular, users benefit as follows:

- A wide field of view makes it possible to show more information at once.
- Tracking the head rotation effectively increases this to an infinite field of view. Wherever you look, there is content.
- The screen no longer shows a simple 2D projection. HMDs are by design stereoscopic displays. Therefore, the user can perceive depth within the 3D world just like in the real world.
- Controller and hand tracking allow intuitive interactions that may be more convenient than using classic mouse and keyboard inputs.

Furthermore, these advantages generally apply to different kinds of similar solutions. CAVEs, for example, do not include special controllers, but can utilize the physical walls as giant touch screens.

Another technology is **Augmented Reality**. In contrast to VR the devices are in comparatively early stages and thus only offer a small field of view. Instead of fully diving into a virtual world, virtual objects are projected into the physical world. The most common AR applications for consumers today are only based on smartphones that use their inbuilt camera to do this. Recently, a spin-off of the Pokemon series [POGO] has reached many users but the possible features do not offer much for interacting with data. Yet, the advantage of interacting

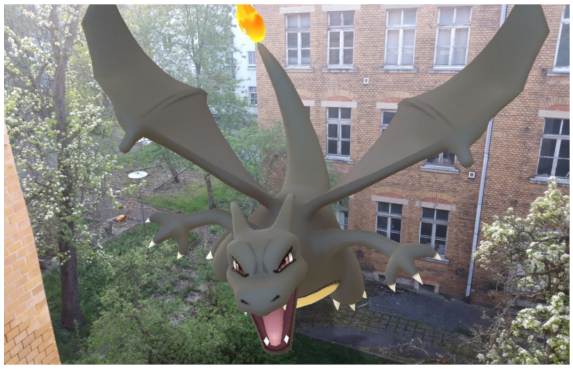


Figure 3.3: Screenshot from Pokemon Go. A creature is projected into the real world.

with the outside world eases the process of using other systems simultaneously. In this thesis, however, we will only focus on VR devices.

3.3 MMT and The Math-in-the-Middle Approach

We have already talked about the growth of mathematical knowledge systems in Chapter 1. This could be, for example, theorem proof assistants or libraries that simply focus on certain areas of mathematics. Consequently, there are often different priorities regarding the way knowledge is represented. Even the exact same concepts may then be expressed very differently between two different libraries. Generally, users of these systems are not be affected by this at first — but as soon their work starts to span multiple fields of mathematics, they would love to have access to external tools — and that is the crux of the matter. There is no automatic way to allow interoperation between two arbitrary systems. Nevertheless, they all share the common language of mathematics and, in theory, it should be possible for all of these tools to communicate with each other.

The requirement for this is a common format that we can use to represent various mathematical archives the same way. Accordingly, the **Math-in-the-Middle (MitM) Approach** [Deh+16] proposes the use the OMDoc/MMT language [Rab13]. In this role, MMT stands for **meta-meta-theory**, which emphasizes the aspect of representing knowledge in a generic way.

OMDoc/MMT This format is designed represent mathematical knowledge in a very generic and modular way. MMT itself follows in the footsteps of the **OMDoc (Open Mathematical Documents)** format [Omd], which uses an XML format to represent mathematical knowledge so that it can be processed by machines.

While the implementation of principles that are necessary to allow utilization as meta format are not relevant for this thesis, the different structural elements are crucial for visualizing theory graphs correctly. There are four levels of knowledge:

1. **Documents** We can regard these as files that contain groupings of the lower levels.

2 . **Modules** A module is the core structural component. Knowledge and relations are represented as **theories** and **theory morphisms**.

3. **Symbols** These are contained within the theories. While they are not relevant for the base structure, we will revisit these in section 4.1.1.

4. **Objects** An object is a mathematical expression that uses Symbols among other components.

Especially for this thesis, level 2 is very important. Theories consist of groups of declarations that describe the respective theory. Then, we can understand theory morphisms as truth-preserving mappings between theories. We will look into these in detail in Section 3.4. Note that every knowledge level except Objects have direct MMT URIs for identification and further processing.

OMDoc/MMT Theory Graphs Now that the knowledge is modularized, we could build a graph by combining all of these components. Similar to the way we have characterized the relation between knowledge and graphs in Section 3.1. For example, a document node would contain modules and the module node would, in turn, contain symbols and objects — combined with different relations between them. But this is not beneficial for understanding the global structure of the knowledge. Therefore, a theory graph only includes the theories, which are represented as nodes, and the theory morphisms, which are represented as edges — yet, we keep in mind that each of these can contain further knowledge. We see, that knowledge formalized with OMDoc/MMT languange can directly be understood as theory graph. This also means that TGView3D can visualize any kind of data that uses this format.

The MMT System Notably, OMDoc/MMT also comes with a system [MMT] that helps to take care of the actual interoperation between different tools and libraries. This system is also called MMT, but the abbreviation has a different meaning: **meta-meta-tool**. As such, there are plugins to support specific workflows and an API at the core to access MMT functionalities. A very important factor for TGView3D is the use of MMT as an interface to obtain the theory graphs and access the information contained therein. MMT can generate JSON files from the OMDoc/MMT sources and supply these to external applications. These are the main data sets for TGView3D.

Mathematical Archives In particular, we are mainly talking about the archives that are maintained on MathHub [MH] We picture the theory graphs that represent the MitM Ontology [MitMO], the MMT Libraries [MMTL], the ODK archive [ODKA] or the PVS NASA Library [Mmt] on multiple occasions within this thesis. These have been formalized in the OMDoc/MMT format and thus, we can directly visualize them within TGView3D.

```
55  view NatAsMonoid : ?Monoid → ?Nat =
56      u = Nat |
57      comp = plus |
58      unit = zero |
59      assoc = allI ([x] allI ([y] allI (plus_assoc x y))) |
60      unit_axiom = allI ([x] plus_defl x) |
61  |
```

(a) View

```
31  theory Abelian : ?NatDed =
32      include ?Semigroup |
```

(b) Inclusion

Figure 3.4: View and Inclusion.

3.4 The Structure of Theory Graphs

We have already learned about theories and theory morphisms in general. But the latter can have very different characteristics, i.e., there are different kinds of morphisms. So, let us now break these down.

- **Inclusions** Often, related theories share many representations. To avoid the need to redefine these for every theory we use the concept of inheritance. An inclusion morphism (compare Figure 3.4b) then hands down the declarations to the target theory. Based on these, theory graphs usually exhibit an inheritance hierarchy that starts with general theories that get increasingly specialized.
- **Views** Views serve a slightly different purpose compared to inclusions. Where the latter simply express hierarchy relations, views can possibly connect two arbitrary theories (compare Fig. 3.4a) . Again, this relation maps declarations from the source to the target theory, but precisely because there is no simple inheritance, we need to carefully define the mapping.
- **Structures** These can be seen as a special form of inclusions as they also implement an inheritance relation. The difference is that we can name the relations in order to define theories based on multiple hierarchically related other theories. We can find a code snippet with structures in Figure 3.5, which includes explanations if you are interested but these are not necessary for this thesis.
- **Meta Relations** From a technical standpoint, meta theory statements behave like inclusions. However, they rather serve the purpose of giving related theories a common origin.
- **Alignments** Alignments [Mül+17] complete our set of morphisms. They serve the purpose of creating mappings between different representations of the same mathematical concepts or expressions. Instead of directly connections between libraries, there are abstract interface theories that act as intermediary.

The core component for the structure of theory graphs are the inclusions. These form the hierarchy that I have already mentioned occasionally. Concretely, the subgraph of a theory graph that only considers the inclusions is a DAG, i.e, there are no cycles. From this, we can derive the most important aspect of computing the layout of theory graphs: **we can arrange the nodes in a way that the edge directions of inclusions are consistent**.

```
63  theory Ring : ?NatDed =
64      /T We will add two structures, one for addition and one for multiplication. |
65      /T Addition forms an abelian group, hence we will use that theory for our structure. |
66      /T In the structure, we first give the universe u a new ALIAS R. This way, we don't
67          have to refer to the universe of our Ring with add/u all the time. |
68      /T Afterwards, we provide new notations for the other symbols. Note, that none of the symbols
69              we "modify" are declared directly in the theory we're including (i.e. AbelianGroup).
70              Structures can modify along includes without problems. |
71      structure add : ?AbelianGroup =
72          u @ R |
73          comp # 1 + 2 prec 40 |
74          unit # 0 |
75          inverse # - 1 |
76      |
77      /T We can now add a DEFINITION to Multiplication/u, making it equal to R=Addition/u.
78          That way, both + and · are defined on the SAME
79          universe (i.e. R), while being two separate operations nonetheless (as opposed to when
80          using plain includes): |
81      structure mult : ?Monoid =
82          u = R |
83          comp # 1 · 2 prec 45 |
84          unit # I |
85      |
86      /T Alternatively we could have defined mult/u directly as add/u
87
88      /T Finally, we can use the symbols in our structures jointly in distributivity: |
89
90      distributive : ⊢ ∀[a] ∀[b] ∀[c] a · (b + c) = a · b + a · c |
91  |
92
```

Figure 3.5: Structures

We can also integrate structure and meta edges the same way while keeping the DAG intact. This is also how TGView3D currently handles these. An alternative would be to consider them as undirected edges. While this is only an option here, we need to do so for views and alignments. Both can potentially introduce cycles and are not part of the inheritance hierarchy. Accordingly, the layout algorithm interprets these as undirected edges. This is always only for the computation of the node positions, we visualize the edge as directed edge.

Furthermore, TGView3D defaults to not including meta edges into the layout, but offers the option to activate them (see Section 7.5). The reason for this is that generally there is one meta theory that connects to a whole range of different theories, which does often not only introduce irrelevant edges, but also dominates the layout.

Finally, as representation of mostly mathematical sources, we need a way to formulate mathematical expressions within the rendered theory graphs and, before this, in the JSON file. We achieve this with MathML (Mathematical Markup Language) [Matb], which uses an XML–inspired structure.

3.5 Layout Algorithms for Theory Graphs

We have seen that the main challenge of visualizing theory graphs is to display the DAG structure within. Standard spring embedders cannot achieve this because they do not consider the direction of edges. Therefore, we have to look at alternatives.

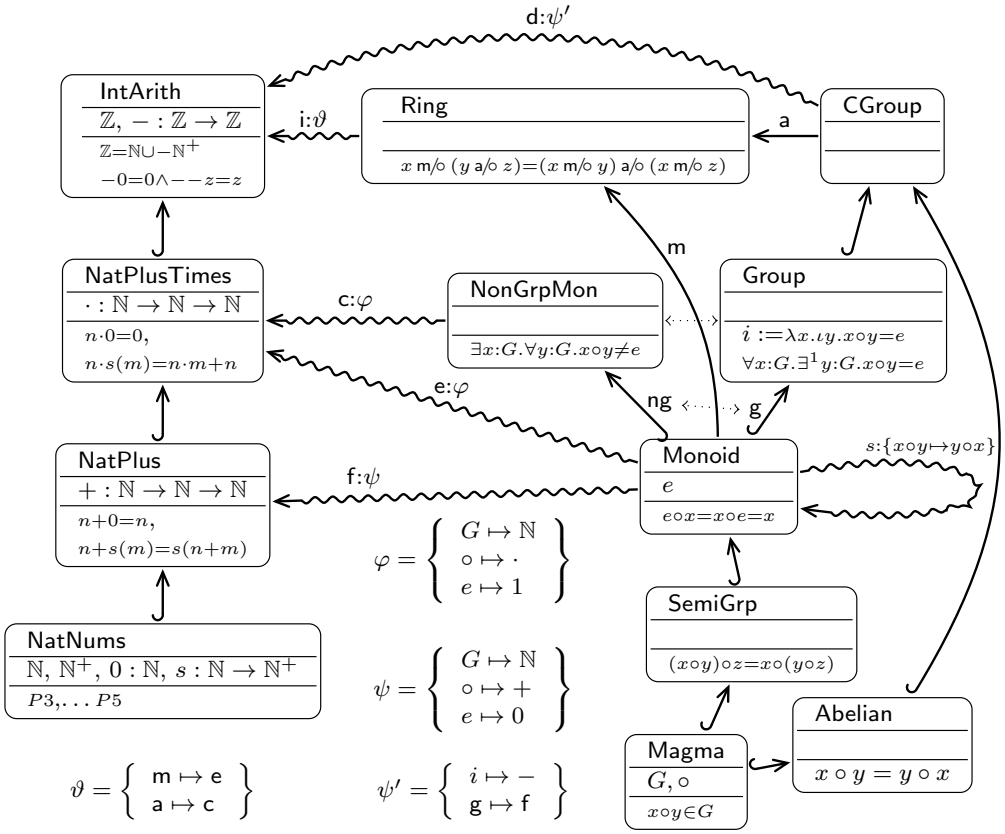


Figure 3.6: A Theory Graph of Elementary Algebra and Arithmetics [KRMT]. It presents contents that are slightly different from what we have seen in the MMT code, but this is not important here. We can see inclusions as edges with hooks, structures as straight ones and views as wavy lines.

3.5.1 Layered Graph Drawing

Hierarchy can not only be introduced by trees but also by graphs. However, this is only true for DAGs. The standard way of organizing such graphs is known as **Layered** or **Sugiyama Graph Drawing** [STT81], in fact, this is also used in GraphViz, for example. This method even works for arbitrary graphs, but removes cycles to reduce the graphs to DAGs in the first step. For theory graphs, we can ignore this part of the algorithm as we already know which part of the graph induces the DAG substructure. Starting from a DAG, three steps remain:

- **Layering** The most crucial work happens here. By computing hierarchy layers, we determine an ordering from bottom to top.

- **Crossing Reduction** By now, we only know how to position nodes vertically but not horizontally. Therefore, the algorithm introduces the aim of minimizing edge crossings. These are only computed by taking the layer of the nodes into account and also uses an horizontal ordering instead of geometric positions. A variety of heuristics help to optimize the layout.

- **Coordinate assignment** Another way of improving the visual structure of a graph is to have as many straight edges as possible without intersecting any nodes. If this cannot be avoided, we have to bend the edges. Consequently, we employ another optimization step that tries to minimize the number of bends. This step now actually assigns position in the plane to the nodes but may only do so within the boundaries of the computing vertical and horizontal ordering, i. e., the ordering itself stays constant.

3.5.2 Combination of Hierarchy and Forces

It is not trivial to port the last two steps to 3D. Edge crossings always relate to the perspective that we use to look at the graph and the one-dimensional ordering would also somehow need to be adapted. However, we do want to build upon the concept of a vertical hierarchy generated by the layering step. We will revisit this in Section 6.2 and describe an implementation based on the height of nodes within the graph in detail. Now we have to think of a different way to distribute the nodes within the remaining two dimensions — luckily, we already have such an algorithm: force-directed graph drawing can be applied independently from the number of dimensions. Additionally, we can still fall back to 2D layouts, which is often very helpful as we have seen in Section 3.1.2. We will discuss the combination of these two concepts in detail in Chapter 6.

A more elegant approach is to include hierarchy based forces directly into the force mix of an force directed layout algorithm. This approach already exists in the form of an Magnetic-Spring Algorithm [SM94] and the introduction of **hiearchy energy** [CHK04], which I had not discovered during my initial literature research. Independently from those, I have arrived at a comparable solution, see Section 6.3, where I also discuss the relation to the earlier methods.

3.6 WebGL and WebVR

An important choice to make when trying to develop software systems is the kind of framework. There are reasons for working with low level APIs, e.g., when aiming at computational efficiency, but we are rather interested in rapid prototyping. Also, we also cannot resort to existing graph frameworks because they do not sufficiently support interactions with 3D graphs as described in Section 2.3. Nonetheless, the option to build browser-based web applications is very desirable for TGView3D as we wish to support a wide range of users with different platform requirements and almost everyone has access to a web browser. To render interactive 3D–content browsers utilize WebGL. WebGL is a JavaScript API based on OpenGL and maintained by the Khronos Group [KHR]. It allows direct access to the GPU, although this is limited by the software overhead of communicating with the browser. Additionally, multi–threading and other hardware features have to be supported by the respective browser as well as WebGL itself. While there these aspects are improving currently, it is still problematic for performance–reliant applications such as TGView3D.

As mentioned before, WebGL does support virtual reality devices and can address them with the help of the of the WebVR [WVR] standard. This has the advantage that all devices

use the same API calls. On the other hand, device-specific functions may not be supported or custom ports of already existing API-specific algorithms may become necessary.

3.7 Unity as the Framework of Choice

We have already expressed that we want our framework to be as high-level as possible and we also do not want to be restricted to the web as only platform. At the same time, however, extending existing systems is no option because there would be many compromises and difficulties to integrate features specific to theory graphs and virtual reality. Fortunately, there is yet another alternative: game engines. Since interaction with virtual worlds is exactly what TGView3D is supposed to make possible, these are the ideal choice. Just as video game developers, we want to quickly iterate on ideas and build an application that should be available to a wide range of users.

In particular, TGView3D uses the Unity game engine, which is popular for its flexibility regarding virtual reality integration and multi-platform support, including the web. To give a better context to how Unity specifically can help to build applications of all kinds, we will discuss different areas in the following.

Platform Support Unity can deploy builds to many popular platforms, altogether more than 20 different ones [UMP]. It uses different methods to translate C# code and compile platform specific executables.

Device Integration While there are efforts to build a single API to address different virtual reality devices the same way [OXR; VRTK], currently an application often has to be manually adapted for each device. More specifically, there are special Unity software integration packages for the different devices. These take care of processing the tracking data of the HMD and the controllers and button presses. This data is then available within Unity in the form of an interface that gives access to this data. Furthermore, those software packages usually come with implementations of basic functionalities, e.g., the Oculus Unity Integration includes a example for grabbing objects with the touch controllers, which are represented as hands in the virtual world.

Rendering Unity fully takes care of rendering. The programmer only has to define **game objects** with transformation matrices, which are responsible for position, scale and rotation. Usually, such an object also contains a mesh describing the geometry and a material for the shading. A material, in turn, supplies parameters such as color or texture to a shader that computes the final look of the object. Different settings are available, that affect performance and visual fidelity.

Interaction Unity supports the implementation of custom interactions. We can categorize them roughly into three areas:

- **API Functions** There is a variety of different functions that support the programmer. This ranges from common computer graphics operations such as vector math to external interfaces such as web requests or JSON parsing.
- **Interaction Components** To act on collisions or user input, Unity offers components that we can attach to game objects. We can then refer to these events in the program code and define behaviors.
- **User Interface Elements** This includes common button types and input fields.

3.8 The Oculus Rift

To develop virtual reality interactions, a head-mounted display (HMD) and some form of controllers are necessary. Popular manufactures usually bundle those together. For TGView3D, we use the **Oculus Rift** [OCR] The two main reasons for this decision are:

- **Portability** The tracking system uses up to three external sensors. These are equipped with a stand and do not require any kind of permanent mounting mechanisms. This is very helpful when demonstrating the system at various places, as it allows to quickly pack up the hardware and set it up again.
- **Intuitive Interaction** The so called **Oculus Touch Controllers**, shown in Figure 3.7, are designed to emulate natural hand gestures. This is important to achieve intuitive interactions and thus easy mastering of the application.

  Tracking of the HMD and the controllers allows six degrees of freedom (6DoF). Consequently, not only rotation is tracked but also any kind of lateral device movement, which is the prerequisite to achieve immersion. For example, moving the head closer to a certain node and moving closer by walking on the real world floor could not be supported otherwise.

The remaining specifications of the Oculus Rift do not differ much from current competition. We assess them in the following way:

- **Horizontal FOV: 110 degrees** While the field of view (FoV) does not reach the 210 degrees of the FOV of the human eyes, it works very well to give the user a wide overview of the graph. The human eye does not recognize details in the outer perimeters anyway. To inspect those, the user can simply turn his head toward other parts of the graph.
- **Resolution: 2160x1200 pixels** Resolution is very important for TGView3D as it relies on presenting information in the form of text. The Oculus Rift definitely works for this task, but higher resolutions would simply allow even smaller text to be read easily and improve the user experience.
- **Refresh Rate: 90 Hz** To avoid any inherent discomfort of using the device, it is important to not exceed a motion-to-photon (MTP) latency threshold of 20ms [Elb+18]. MTP latency describes the delay between the physical motion of the head and the frame

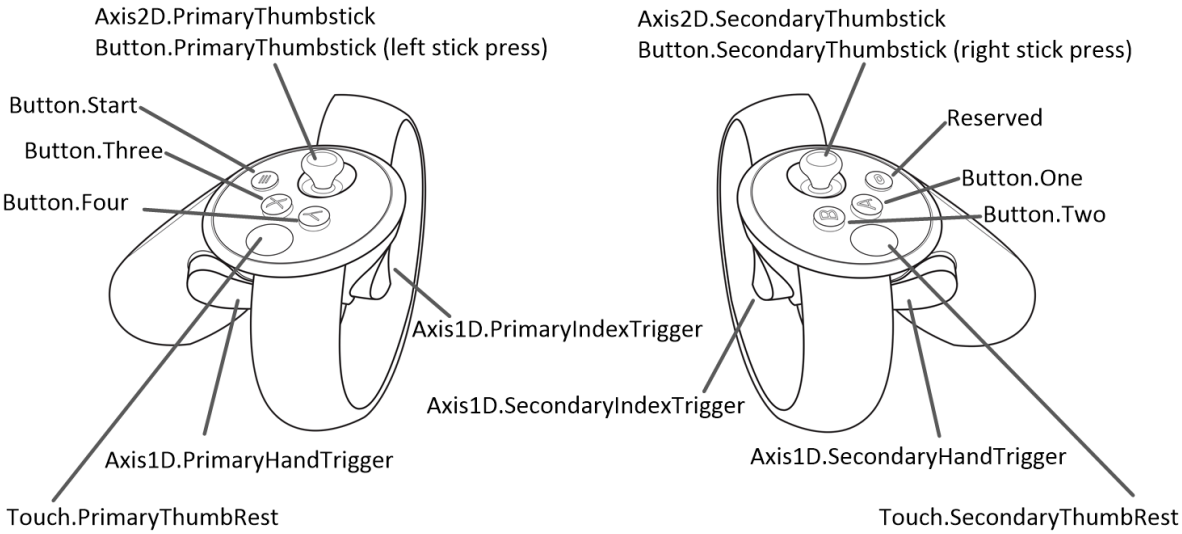


Figure 3.7: Oculus Touch Controllers [OCTC]

refresh of the screen. With a refresh rate of 90 Hz, we arrive at a latency of 11ms. This just leaves enough time for other computations and signal processing. Therefore, it is extremely important to also maintain a frame rate of at least 90 frames per second within the application. If this cannot be achieved symptoms such as nausea or loss of orientation are possible. They arise from the discrepancy between the current view on the screen and the view the brain expects from head movements. This is known as **Virtual Reality Sickness**.

Last, we need to mention the system requirements of the Oculus Rift. Unfortunately, these are very strict [REQ] and demanding, requiring Windows 10 as operating system and a modern dedicated GPU.

Chapter 4

Requirements and Architecture

There is still a very important factor for the system design — namely the way users would like to work with TGView3D. There are very different mathematical libraries and there might even be multiple use cases and workflows for one particular library alone. By demonstrating TGView3D to potential users and discussing their needs, I have identified a set of use cases and requirements. I will describe these in the following and then derive global requirements as well as an architecture that is capable of implementing them.

4.1 Use Cases

There are multiple use cases for TGView3D. Even though some have only emerged during this work but the basic desire of exploring mathematical archives has been there from the beginning as it was part of the motivation for this thesis.

4.1.1 Mathematical Libraries

In particular, we are talking about the archives that are formalized in OMDoc/MMT, which are exactly the graphs we refer to as theory graphs. Workflows with these archives focus on the following interactions:

L1 **Graph Exploration** The archives contain so many theories and morphisms that it is not trivial to obtain the knowledge from traditional file browsing interactions. Visualizing these as graph can support this workflow. Users can localize structures in the 3D space and directly see the relations in the form of edges. To offer significant advantages, we need an organized layout and methods to filter the information overload. Furthermore, the resulting structures in the layout can offer new insights about the represented data.

L2 **Accessing Node/Edge Information** Beside the visual structures, there is also the local information that is hidden inside nodes and edges. This means that after finding where this information is located within the graph, users often want to directly access this information. Thus, it must be possible to do so within the application.

Recall that the information contained within a theory can also represented as a graph that uses the dependencies between symbols. While it still does not help much to immediately show all of these, it may be helpful to have the option of **expanding** nodes so that the contents of selected expanded nodes are integrated into the global graph layout.

4.1.2 Model Pathway Diagrams

Mathematical modeling and simulation is an integral part of modern scientific research. **Model Pathway Diagrams** (MPDs) [Koh+17] can represent the structure of these mathematical models. They use circles to present physical quantities and rectangles for physical laws, which connect the quantities. Again, we can directly formulate this as theory graph. We simply need two theory types for this purpose: **quantity theories** and **law theories**.

Furthermore, here is a very interesting aspect regarding the structure of MPDs, in particular the loops within. These directly relate to the represented system and equations, respectively [Kop+18]. If we look at the van Roosbroeck system in Figure 4.1, we can see connections between different substructures of the MPD. Beyond this, it is neither necessary for this thesis to further describe details of the physics visualized in MPDs nor do we need to discuss the importance of mathematical modeling.

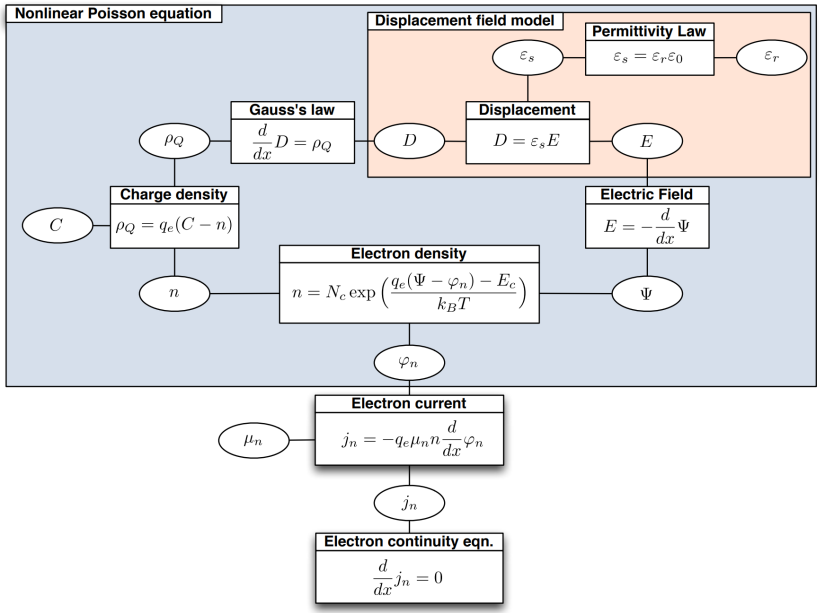


Figure 4.1: MPD of the unipolar van Roosbroeck system [Koh+17].

This leads to the following requirements:

M1 **Creating and Editing Graphs** While MPDs usually only contain a few dozen nodes and edges at most, they represent complex structures. Layout algorithms cannot sufficiently organize these because this requires expert knowledge of the physical models.

Therefore, the knowledge engineer either wants to use an initial layout to build upon or directly create the graph within the application.

M2 **Rendering MathML** For the mathematical formulas, we use MathML. However, there is currently no standard way to display these contents in 3D, so this requires a custom solution.

M3 **Access Media to Help Understand Phyiscal Models** It is also possible to visualize the simulations of the modeled physical process. Often, we can directly see relations between the parameters of the model graph and the simulation. Watching the simulations and the graph side by side promises to be beneficial for understanding the model.

4.1.3 Coq

Coq [Coq] is an interactive theorem prover. There are many mathematical as well as computer science related libraries but they are not maintained centrally and instead often hosted on GitHub [Con+]. Similar to the other mathematical archives managed by MMT, Coq libraries could be represented in the OMDoc/MMT format. At the time of experimenting with Coq libraries, however, this was not done yet. Still, a Coq plugin [Coe19] was ready in time to export the libraries to XML and extract a graph in the form of CSV files. These generalize the contents to directories (modules, theories, etc.) and objects (definitions, lemmas etc), connected by edges that represent the dependency. While the dependencies between objects induce a directed acyclic subgraph, the relationship between different directories as well as between directories and objects forms a tree. Again, we only need to know about this structure to be able to visualize the graph.

Compared to the usual OMDoc/MMT theory graphs, this results in slightly different requirements:

C1 **Library Exploration** Coq libraries are often too big – by orders of magnitude — to load into graph viewers at once. Therefore, users only want to look at parts that are small enough to be explored effectively. Furthermore, the exported dependencies between objects are not transitively reduced, which makes it hard for users to make sense of the graph.

C2 **Clustering Graphs with External Meta Information** The XML exports also include meta information about the author or the GitHub repository. To get a better overview and possibly new insights, users would like to see clusterings based on these or maybe even based on mathematical keywords.

4.1.4 Attack Graphs

Graphs also are ideal for representing the way humans argue. In particular, there is the notion of an **argumentation framework** [Dun95], which consists of **arguments** and **attack** relations between them. We refer to the graphs that represent this structure as **attack graphs**. There are different ways to prune the resulting attack graph resulting in subgraphs containing

only the arguments that should be accepted. These procedures are called **semantics** and the resulting subgraphs are called the **extensions** of an argumentation framework under a given semantic.

The extensions yield a straight-forward labeling for arguments. We call members of all of the extensions **accepted**, members of at least one extension **undecided** and members of none **rejected**. Of course, there is much more to formal argumentation [BGG18], but for our purpose, there is no reason to go further into detail at this point.

Interacting with attack graphs requires features to interact with the argumentation structure:

A1 **Initiating Computations** Depending on whether an argument can be accepted or has to be rejected, we can use different colors for the nodes. This changes when the user, for example, modifies the graph source or uses different argument graph specific settings. Therefore, users need the possibility of initiating an update of this coloring as well as performing these changes in the first place.

A2 **Creating and Editing Graphs** While argument graph users can also make use of this feature the way that MPD users do (M1), the motivation here is different: altering the the graph, for example, removing nodes or changing the targets of edges directly influences the meaning of the argument graph. Re-initiating computations (A1) afterwards can then change how arguments are supported.

4.2 Global Requirements and Architecture

Since the use case specific requirements are often comparable, we can combine these into more generic requirements. Together with issues that apply to all types of graphs, we can formulate four global requirements:

GR1 **Using TGView3D** Although this may seem trivial at first, this is actually very important. Users often work with certain operating systems and hardware and, while we are exploring the possibilities of 3D and VR, we cannot ignore that not everyone has access to these. Also, once a user is ready to use TGView3D, he still needs to be able to import his desired graph. Even if there is the option to directly create a graph within the application, the user may want to load this graph again later.

GR2 **Visualizing Graphs** After importing the graph source file, there must be a defined 3D representation to render the graphs, including elements such as the MathML expressions (M2). Additionally, we need an appropriate layout algorithm to organize the graph; we can handle MPDs and argument graphs, in particular, with standard force-directed algorithms.

GR3 **Data Interaction** For the interaction with the graphs, we have two extremes: exploration of huge graphs (L1, C1) on one hand and working with small graphs (A2, M1) and would like to have the full range of editing features that are common in text editors, for example. Nonetheless, there are intersections: bigger graphs may have smaller

substructures (L2) that users may want to interact with and even small graphs can have enough complexity so that filtering methods are helpful.

GR4 **System Interaction** There is external information or certain computations that are very use case specific (M3, A1) and thus cannot be included into TGView3D itself. Instead, we request this information from external systems. This could either be a server request that returns raw information to be processed in TGView3D or ready to use content such as pictures or even web URLs.

Of course, these requirements also influence each other, e.g., interacting with the graph affects the result of following system requests. Based on the defined requirements, we can build a model of the system architecture (see Figure 4.2) that describes TGView3D and reflects the dependencies of the different system parts.

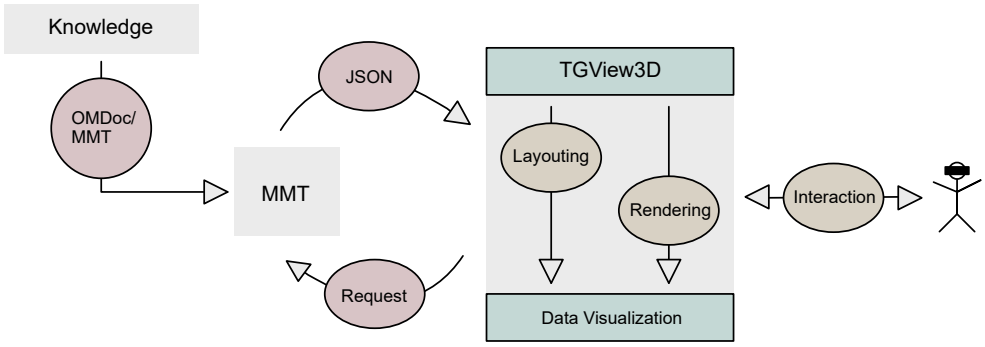


Figure 4.2: System Architecture.

Chapter 5

From MMT to VR

To arrive at a graph that can be experienced in VR, several steps are necessary. We start with an arbitrary source of structured information, such as one of those described in Chapter 4 and want to represent this structure as a theory graph. To achieve this, we use the formalization mechanisms of MMT.

5.1 JSON Creation and Data Structures

With MMT, knowledge can be transferred into OMDoc/MMT format. Once this is done, MMT can generate the JSON files that represent the resulting theory graphs. This file consists of a JSON array of nodes and a JSON array of edges. Nodes and edges then have further attributes such as text labels, type specification and a unique ID. Although the latter does not need to be considered for visualization purposes, it is a very crucial element for the purpose of running the system. The node ID consists of the full MMT path and is used to formulate origin and target of edges. Similarly, the edge ID is the combination of two node IDs. We see a shortened JSON file below.

```
{
"nodes":[
    {
        "id":"http://cds.omdoc.org/examples?Monoid",

        "style":"theory",

        "label":"Monoid",

        "url":"/?http://cds.omdoc.org/examples?Monoid"
    },
        ...

],
```

```
"edges":[

    {
        "id":"http://cds.omdoc.org/examples/tutorial?Group?
        [http://cds.omdoc.org/examples/tutorial?Monoid]",

        "style":"include",

        "from":"http://cds.omdoc.org/examples/tutorial?Monoid",

        "to":"http://cds.omdoc.org/examples/tutorial?Group",

        "url":"/?http://cds.omdoc.org/examples/tutorial?Group?
        [http://cds.omdoc.org/examples/tutorial?Monoid]"
    },
    ...
]
}
```

We can then load the JSON file into TGView3D and build the respective data structures. For convenience, we use an object-oriented approach and reuse the setup of the JSON file. This is very helpful for prototyping because we can easily add more attributes and access them across different files by accessing an node or edge.

An alternative for this would be data-oriented design techniques or even an entity-component-system, which fully decouples the entities from the data structures. For example, we we would then use separate arrays for each attribute. The advantage would be faster layout computation because of improved memory access patterns. Despite this, we leave the evaluation of this approach for future work.

5.2 Representing Theory Graphs in 3D

Now that we have the required data structures in place, we still need to discuss how we incorporate the numerous elements of a theory graph into the 3D rendering. TGView3D handles this as follows (also compare Figure 5.1):

- node: sphere
- node label: text mesh
- edges: quadratic tube
- edge type: color
- edge direction: gradient from light to dark
- edge label: accessible in interface

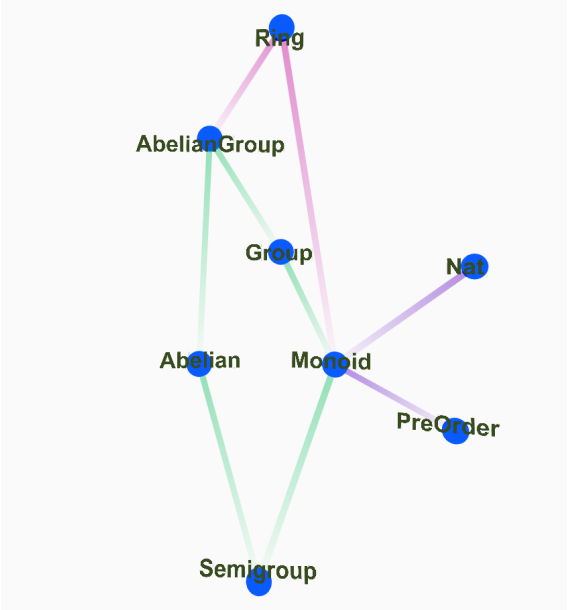


Figure 5.1: Subgraph within MMT Archive. This time, it is the exact graph that is created from the MMT code in Chapter 3. Inclusions are green, views purple and structures magenta. Alignments would be yellow and Meta edges red

.

5.2.1 Efficiency Considerations

Partly, these decisions result from efficiency considerations. For pure data visualization tasks, no expensive lighting effects or similar are necessary, so the the GPU performance is only limited by the number of polygons in the scene. More precisely, the number that is captured in the current camera view frustum[1] is relevant, as objects that do not fulfill this criterion are culled, i.e., there is no draw call for non-visible objects. Note that in the worst case scenario the whole graph is visible and thus, no culling takes place.

As this can be very relevant, especially for bigger graphs, we use simplistic edges. Still, they should have a certain thickness and allow for a color gradient, so each edge is represented by a straight, quadratic tube. Such an edge can have openings at the beginning and at the end that are not visible because those are occluded by the nodes. Consequently, the mesh of an edge consists of eight vertices. Because of this choice, they have to be specially constructed and maintained. When the positions of nodes change, we also have to update the edges. While we can simply assign a new position coordinate to a node, edges require a more refined treatment. Basically we have to update the position of every vertex according to the new origin and target position of an edge.

A bigger issue is the sheer number of edges and nodes. Generally, each object means an additional draw call for the CPU. Although the culling described above also applies in this case, the CPU runs into a bottleneck very quickly. To cope with this, we have to reduce the effective number of draw calls drastically.

[1] region of space visible by the camera

(a) Holes at both ends of the edge.

(b) Each side of the mesh consists of two triangles.

Figure 5.2: Screenshot of Edge Mesh

As we do not interact with edges by touching individual edges, there is no need for multiple edge objects – in contrast to the nodes. Hence, all edges can be combined into a single object and mesh, respectively, which is the most efficient solution to minimize draw calls. While the above would be problematic for the nodes, other methods can be applied. Unity specifically offers three variants:

- **Static Batching** This is similar to combining the mesh directly, but not as efficient. In exchange, the game objects can remain separated, but are not affected by object transformations anymore. Therefore, this is no option for the node spheres.

- **Dynamic Batching** This does not have this restriction, but only considers objects that do not exceed a certain size threshold. Although it is less efficient than its static counterpart, it could be used for the nodes.

- **GPU Instancing** This deploys a different mechanism, which only works for objects with identical geometry. Instead of combining several meshes into a single one, the draw call for the object is committed once, but multiple copies are drawn simultaneously. This even allows different colors and sizes, which makes it the ideal choice for rendering the nodes.

Also, we must not forget the node labels. With the help of Unity, their texts can be easily transformed into text meshes and thus, they are the perfect target for dynamic batching. Beyond that, it is also very useful to only render them up to a certain viewing distance. This does not only increase performance, but also improves the visual clarity since distant text is too small to be read anyway. We implement this in Unity by defining a layer for the text objects and adjusting the far plane for this layer specifically.

5.2.2 Usability Considerations

There are further usability considerations, especially concerning the edges. Mainly, we we want to express their type and direction.

Edges Using different styles such as dotted or waved lines do not translate very well into three dimensions and would also require additional vertices. Hence, different colors proved to be the natural solution.

Similarly, the usual way of using arrows is problematic as the user cannot discern the direction if the beginning and end of the edge is not clearly visible. We solve this by using a gradient from light to dark.

Yet, this does not cover the whole range of edge occurrences. It is possible to have multiple edges between two nodes as well as a self–loop. This is problematic to solve with only straight edges, especially for the latter. For multiple edges, we arrange the edge origins and targets, respectively, in a circle orthogonal to the edge direction (see Fig. 5.3). Using a color gradient would distract from the color-type-coding and limit the amount of available colors. While we cannot indicate self-loops like this, we do not have to treat them like the other edges in the first place. Since they is only depend on a single node, we just have to specially highlight such a node. To stay as close as possible to common representations of graphs, we can attach a torus to the node (comp. Fig. 5.4). Since this may impede other edges, one could either take special care when choosing where to place the torus or use a different indication such as an outline. Last, edges also may have labels. These are very difficult to show within the graph because edges can be relatively long and cross each other. Furthermore, 3D edges can also proceed parallel to the view direction. A label in the middle of the edge might then be too far away. Hence, we include the labels into the UI and allow the user to iterate through edges of a certain node, which will be described in Section 7.4.

Figure 5.3: Two edges going to the same node.

Nodes By default nodes have a fixed color. If there are multiple not connected subgraphs within one graph file, we will assign different colors to these. For the support of grabbing interactions, the nodes at least require a **Collider** component. A **Rigid Body** component allows physically-based reactions, e.g., nodes pushing each other away when colliding or throwing nodes. However, we do not use the latter because it results in less predictable behavior and the benefit of moving nodes like this is not relevant[2] for current applications of TGView3D.

Figure 5.4: Node has an edge loop.

The placement of node labels is also important. In 2D graphs, a node is displayed by a text box, e.g, oval or square ones, containing the node label. This way, it is obvious where the label belongs to and the node stands out from the background very well.

Nevertheless, we opt for equally–sized spheres (we can also see this in Fig. 5.4) that do not directly contain the text. These offer several advantages:

- Larger texts do not require larger objects to contain them.

[2] ... even though users enjoy throwing nodes around and playing pool with them a lot

- The size of the node object is available to be given a special meaning, independently from the size of the text. This attribute is revisited in Section 8.2.1.
- The node label can have a certain offset towards the node label. Then, holding the node with the hand does not interfere with the text itself. By rotating the node, the user can also move the text closer or further.

Nodes can also contain additional labels. For example, MPD–nodes also have corresponding formulas that are represented by MathML expressions. While there are no tools that can generate text meshes accordingly within Unity, there is a tool that can create arbitrary meshes from SVG files. There can be major differences between SVG representations of letters: they either can be encoded as SVG text elements or generic path elements. For Unity, we require the latter, which, again, is not supported by default. We solve this problem by relying on MathJax [Mata], which implements several features related to displaying mathematical content in the web. It offers a server-side conversion of MathML and supports our required format. It is possible to send an appropriate web request within Unity and receive the result for further processing. We describe the setup in the Appendix A.

5.2.3 Platform Considerations

When trying the 3D graph in virtual reality, one also has to think about the differences between platforms. For our purposes, there is no efficiency optimization that has negative effects for virtual reality. Still, the requirements are different. While even a frame rate of 30 frames per second can be considered smooth for regular displays, virtual reality application should achieve 90 consistently. The reason is the issue of VR sickness as described in Section 3.8.

Therefore, it is very important to optimize performance. Different platform– and hardware–dependent methods are available. They exploit the special traits of HMDs, e.g., the two very similar images that are rendered simultaneously.

Another issue is the combination of platform and hardware. For example, the Oculus Rift integration only works with Windows 10. Utilizing the special controllers further complicates the process of supporting multiple devices and platforms. The interaction concepts then have to specifically adapted, often even requiring different API calls.

A similar problem arises when trying to port the application to WebVR. Even though the Oculus Rift itself is supported, the Oculus integration is not designed for this and thus the controller interaction would have to be rewritten. Additionally, the WebVR and WebGL standards are often one step behind in terms of optimization techniques, with the web browser itself also causing performance overheads.

Using TGView3D without virtual reality devices also requires a different set of interactions, possibly even design choices, e.g., a different visual representation of nodes and node labels. We describe a partial port of the interactions in Section 8.1.

Chapter 6

Automatic Graph Layouting

Even though we can now visualize the components of theory graphs within TGView3D, they will not be of much use to the user as we have not defined a layout algorithm for theory graphs yet. In particular, we need to consider their specific structure that we have learned about in Section 3.4. The prime user request is to project the hierarchy of theory graphs to one specific dimension. Especially for 3D, this still leaves two dimensions to organize the graph freely. Edges with types that do not contribute to the hierarchy can also be placed arbitrarily. Nevertheless, the algorithm can also be applied in 2D — of course, with the drawback that there is only one dimension left.

To form the hierarchy, we have to look at the include DAG that is induced by the theory graph. The most central property for theory graphs is the **local hierarchic consistency**, which means that each predecessor needs to be positioned "below" its successor, for example. This is equivalent to postulating a consistent edge direction within the DAG — in this case, from bottom to top. Our goal is to make sure that the produced graph layout preserves this property.

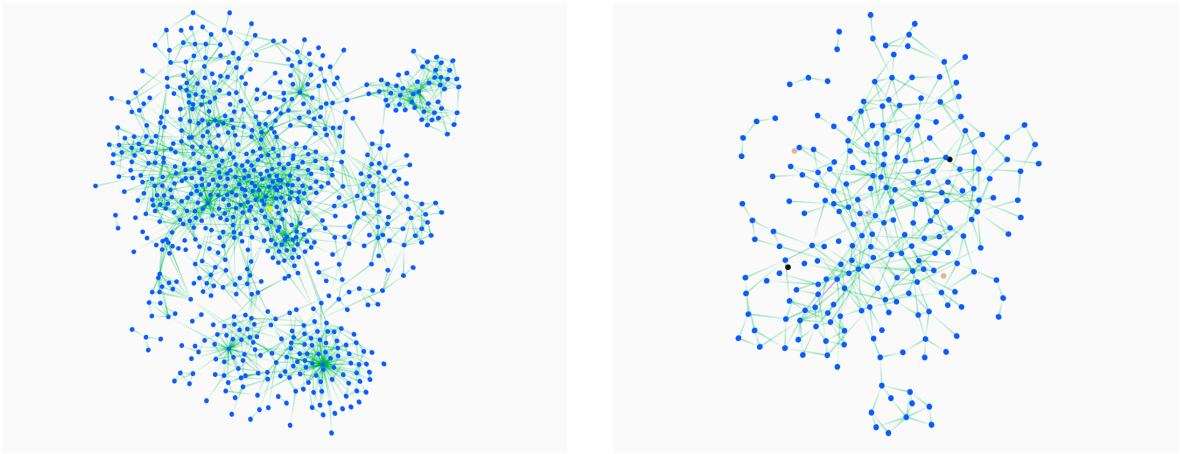
While we could in theory also arrange the hierarchy in a horizontal way instead, the vertical orientation is more intuitive. The reason for this is that we generally keep our body upright when moving in the real as well as the virtual world. To illustrate this, we can look at real world trees. When walking through a forest we can always easily find the root of a tree.

We call nodes without successors leaf nodes and nodes without predecessors root nodes. While it is not inherently defined whether to put the roots or the leafs to the top of the graph, the consensus for theory graphs is to place the roots at the bottom, which also aligns with our metaphor of real world trees in contrast to the computer science practice of placing the root at the top. Since this strategy only specifies how to determine the vertical order of nodes, we need an algorithm to at least take care of the other two dimensions.

6.1 Force-directed graph drawing

Force-directed graph drawing techniques emerged as the ideal choice for this purpose. Looking at the results of this algorithm works very well to organize theory graphs as we show in Figure 6.1. Of course, groups and other relation structures have to exist in the particular

graph in the first place and this algorithm completely ignores the inheritance structure of theory graphs because it considers all edges as undirected.



(a) NASA PVS Library **(b)** MitM Archive

Figure 6.1: Force-directed layouts of theory graphs

Algorithm The original motivation of this class of algorithms are mechanical springs that replace the edges between nodes. The more two nodes are apart from each other, the more the spring tries to contract. But as there are many springs, pulling two nodes closer together might then extend a second spring in the system. Consequently, the system will reach a force balance that minimizes the energy. To avoid the cluttering of nodes, there is traditionally a second, repulsive force between all nodes.

The implementation in TGView3D is based on Fruchterman–Reingold algorithm. It defines the optimal distance between vertices so that they spread evenly, i.e., each node is supposed to conquer the same amount of space. This is expressed as

$$k = C\sqrt{\frac{area}{number\ of\ nodes}}, \tag{6.1}$$

where C is a constant to control the spacing[1] of the nodes. Depending on the distance d between two nodes, we can then define the attractive force f_a and the repulsive force f_r as follows:

$$f_a(d) = d^2/k \tag{6.2}$$

$$f_r(d) = \text{-}k^2/d \tag{6.3}$$

If two nodes have the optimal distance to each other, the sum of these forces is zero. If not, the result will be strength of the force that moves the node towards its desired position. In practice, there are of course often more than two nodes, but the principle remains the same.

[1]This is not really important for TGView3D because we can interactively increase and decrease the spacing.

Note that for the displacement, we need to multiply the normalized direction vector of two nodes with the computed force.

The algorithm itself begins with distributing the nodes randomly within the assigned space. Then, it performs multiple iterations to minimize the energy in the system. To better find local minima, the algorithm uses the optimization concept of **simulated annealing** [KGV83]. It is inspired from the annealing process in metallurgy, which is applied when cooling down metal to control the resulting properties. Depending on how the energy minimization progresses, the layout algorithm will adapt the temperature to make smaller or bigger update steps. A single iteration consists of mainly three steps:

1. For each node, compute the sum of attractive forces f_a and repulsive forces f_r.
2. Limit the displacement of the node to the current temperature.
3. Reduce the temperature as the layout converges.

The algorithm usually also restricts the displacement to the specified space, but this is not necessary for TGView3D because users can manually set the scale of the graph after layout computation.

For a 3D adaption, we simply need to use 3D instead of 2D coordinates and slightly modify the definition of the optimal distance:

$$k = C\sqrt[3]{\frac{volume}{number\ of\ nodes}} \tag{6.4}$$

6.2 Strictly Hierarchic Layout

For this hierarchic variant, we first need to define the following terms:

- The **height** of a node is the length of the longest path between the node and a leave node.
- The **depth** of a node is the length of the longest path between the node and the root node.

Consequently, this means that the height of a leaf node as well as the depth of a root node is zero. Thereby, we follow the terminology of trees and will continue like this for the description of this algorithm but simply turn the graph upside down during visualization so that the root node is at the bottom.

For graphs, the determination of height and depth is not as trivial as for trees because nodes can be reached from more than one path and there are also multiple root nodes according to our definition. In particular, we require the maximum path length. Since the computation of the heights is analogue to the computation of the depths, we will only describe the latter. We can either calculate the depths in a breadth-first or a depth-first manner, starting from each root node. A recursive step of an arbitrary node n during this proceeds as follows:

1. We compute a temporary depth $d_{tmp} = d_n + 1$ for all successors s_i.

2. If d_{tmp} is greater than the current depth of any s_i, we update the depths and continue the recursion with these nodes.

3. Otherwise, there is no need to continue as we are interested in the maximum depth of each node.

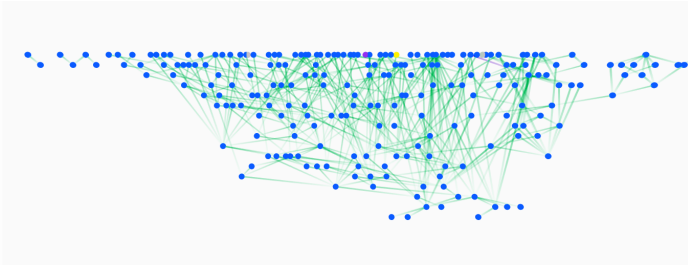
Computing either the height or the depth of each node implicitly results in a strictly hierarchic layout, which is also inherently locally consistent. We can deploy this layout by placing nodes of the same height or depth, respectively, into the same layer. Then, each layer has a certain thickness, so that the nodes keep a reasonable vertical distance from each other. We then distribute the nodes randomly, but set their vertical coordinate so that they are placed in the center of their layer.

After calculation of the strict hierarchy, we use the force-directed layout algorithm to organize the nodes within their height layer. This means we can freely adjust the position within a horizontal plane. To guarantee the consistency of the hierarchy we cannot do the same with the vertical positions and need certain constraints: We can

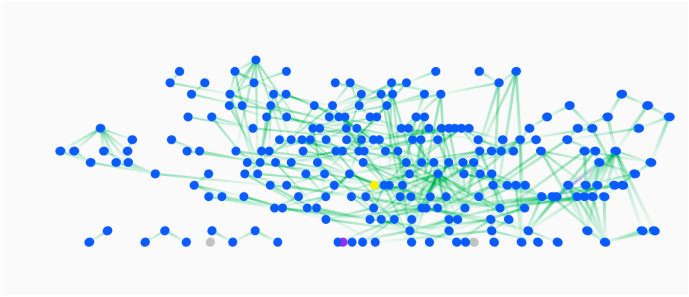
H1 keep them constant.

H2 change them only within the height layer of the respective node.

H3 change it, as long the local hierarchy is not violated.



(a) Result of computing the height: leaf nodes of height 0 all gather on the top level.



(b) Result of computing the depth: root nodes of depth 0 all gather on the bot level.

Figure 6.2: MitM Archive: Strictly Hiearchic Layouts

H1 and H2 are straightforward and do not require special attention. The integrity of the global hierarchy remains untouched as well. The problem here is the static organization of such a hierarchy. Some kind of graphs may benefit from it because it may be helpful to keep two comparable nodes on the same hierarchy level. Users can then retrieve information directly from the placement of a node, e.g, from the fact that each node within the lowest layer is a root node. We show results of this variant in Figure 6.2. Note that we encounter the problems of two dimensional graphs here. Using one dimension for the hierarchy makes it difficult to show the inherent organization that we can observe when using a standard force-directed algorithm. Exploring this layout in 3D, however, reveals this structure when we look at the graph from a top down perspective. In 2D, the layout tends to stretch out horizontally, because the nodes only move left and right within their layer. There is simply not enough place within the height layer to organize the graph efficiently.

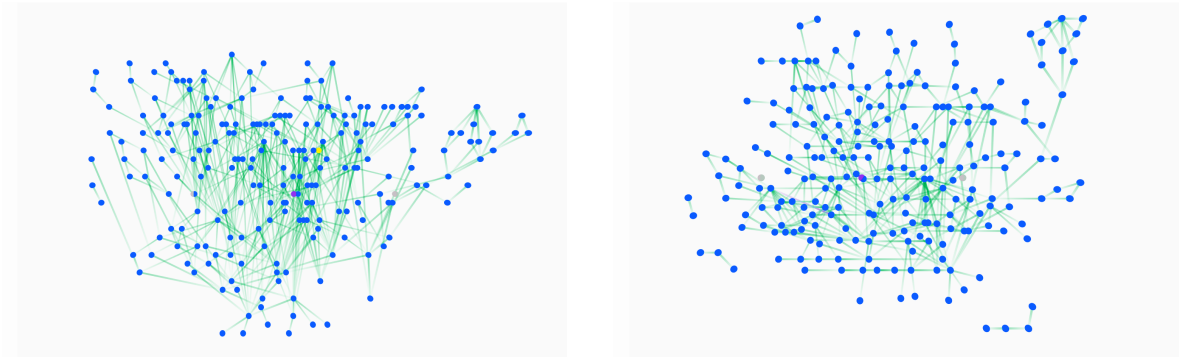
A possible way of improving this is to take the average of height and depth, see Figure 6.3a. This will generate additional layers **in between**, e.g., a node of height zero and depth three will be placed into a new layer between layer one and two. Yet, this creates computational overhead and voids attributes such as, e.g., root nodes always lying in the lowest layer. While the latter is even welcome for theory graphs, we not only want a better overall distribution but also want the algorithm to use the vertical dimension to be used to put closely related nodes close to each other — limited by the hierarchy constraint.

H2 does also not offer much room for movement as nodes can at most move up or down by the width of the layer. Although it results in a less static layout and groups may be vaguely perceptible, it does not help to achieve our goal of efficient use of space. Therefore, H3 seems very attractive, especially with the global hierarchy being possibly harmful for theory graphs. Consequently, we see this variant as a separate algorithm, which specifically achieves locally consistency.

6.3 Hierarchy-based Force-directed Layout

To validate this local hierarchic consistency, we have to compare the node position with positions of pre– and succeeding nodes. We can then determine the distance to the nearest predecessor and successor. Now, we know how far we can move a node up or down without violating the hierarchy. Still, as we update the positions for all nodes simultaneously, two nodes could travel towards each other that are already have hierarchically correct positions. If this forces are sufficiently strong, they could even overtake each other in the hierarchy and reach incorrect positions. To prevent this, we have to limit the movement to half the distance to the nearest nodes. Then, two nodes can at most touch each other and the hierarchy stays consistent. Finally, we achieve exactly want we want: a layout that optimizes the use of space, forms groups and retains a consistent local hierarchy. The global hierarchy computation is then the initialization for a second force-directed step with the described limit acting as an optimization constraint (compare Figure 6.3).

Despite this, this algorithm is still problematic as the optimal force-directed layout would often want to move nodes opposite to our desired position. This can lead to slow convergence



(a) Averaging height and depth: more layers are generated and nodes are distributed more evenly.

(b) Allowing vertical node movement with local consistency constraints.

Figure 6.3: MitM Archive: improving the hierarchic layout

and subpar results. Nevertheless, we do know that the nodes should be positioned between their direct predecessors and successors. This means we actually can formulate an additional, hierarchic force to be included into the force-directed layout.

6.3.1 Hierarchy–based Forces

The force that tries to move the node to the correct position in the hierarchy is similar to the usual repelling force. However, it only considers connected nodes and repels the node as it approaches direct predecessors or successors. Such a force can even result in a correct layout without any specific initialization such as a strictly–hierarchic one. First, we need to define the sign s of the hierarchic force that affects a node as 1 or -1, depending on whether the specific neighbor is a successor or predecessor. We can then formulate the hierarchic force f_h, with d_v as the vertical distance between two nodes and as

$$f_h(d_v, s) = sk^2/d_v. \tag{6.5}$$

This means the closer a node travels towards a wrong local hierarchy level, the stronger is the repulsive force. Note that if it has already entered a wrong hierarchy level, we set d_v to an epsilon close to zero, which already creates the maximum repulsion. Just like usual repulsive force, we then add up the individual hierarchic forces that affect the current node.

Instead of the sum, we could also only take the worst violation into account. Because of the nature of this force, wrongly positioned nodes would still dominate the overall force balance for this node but this yielded worse results so the author did not further pursue this. Still, this external component has to be carefully integrated into the existing force-directed algorithm. In the original algorithm, the forces will converge in a reasonable balance between repulsive and attractive forces. Now, the repulsive portion is increased by the additional hierarchic repulsion, which we have to compensate somehow.

6.3.2 Combination with Traditional Forces

The main idea is that we simply combine the vertical component of the usual repulsive force with the new hierarchic force. In particular we calculate the full hierarchical displacement y_h of each node as

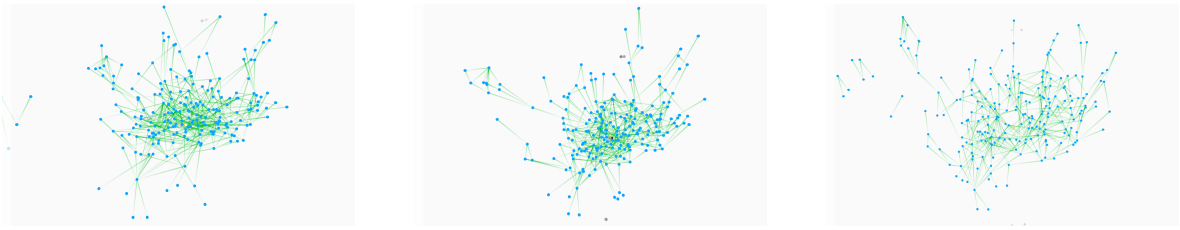
$$y_h = \sum_{i=0}^{N} f_h(n_i), \tag{6.6}$$

where n_i are the direct neighbors. If we define the Fruchterman-Reingold displacement as $disp_{fr}$ and its vertical component as y_{fr}, we can integrate the hierarchical displacement as follows:

$$y_{fr} = \begin{cases} ay_{fr} + (1-a)y_h & \text{if } sgn(y_{fr}) \neq sgn(y_h) \\ y_{fr} & \text{otherwise} \end{cases}, \tag{6.7}$$

with a as experimentally determined constant. We leave y_{fr} intact if the sign of both displacements is the same, so we only override the repulsion from unconnected nodes when they push the nodes in the wrong hierarchic directs and use $a = 0.1$.

Note that the specific way we combine the forces is has emerged from experiments and other values or ways of integration may yield better results. Nonetheless, the result achieves our goals: finally, groups also form vertically and the nodes scatter evenly, all while keeping the local hierarchy consistent, but we can change the way we initialize the layout.



(a) After 10 iterations, the graph is still cluttered in the middle

(b) After 20 iterations, the graph spreads out, but is still cluttered

(c) After 50 iteration, we reach a quite even distribution.

Figure 6.4: MitM Archive: layout algorithm optimization progress

6.3.3 Impact of Initialization

We have learned that we can guarantee a correct hierarchy by using a strictly-hierarchical layout as initialization, using according constraints. However, the deterministic computation of the strict hierarchy can quickly become very expensive for bigger graphs. We consider the following alternative initialization methods:

- **Random Initialization** Here, we place the nodes randomly within the volume that is allocated to the layout algorithm. Since there is no initial hierarchy, we cannot use the constraints in this case.
- **Root–Leaf Initialization** This method exploits the concepts of root and leaf nodes. Consequently, we define three layers: a root layer at the bottom, a leaf layer at the top

and a middle layer exactly in between. Interestingly, his is already a locally consistent hierarchy as we allow equal heights. Hence, it works very well in combination with the height movement constraints.

Generally, we observe that with a sufficiently high number of iterations, all initialization methods converge into similar layouts. The biggest impact for such a final layout is whether we use the constraints or not. While not using them often helps the algorithm to find a better force convergence, a hierarchically consistent layout is only achieved with many iterations. Even then, we cannot guarantee consistency. There might be other local optima or there might be edge cases where the other forces outweigh the hierarchic ones.

Taking into account that we only want to spend a certain time on computing the algorithm in the context of an interactive application, the initialization type becomes more relevant. Computing the strict hierarchy reduces the amount of iterations required to achieve a visually pleasing layout, but the additional overhead negates this advantage. As we have already ruled out the random initialization without appropriate number of iterations, the Root–Leaf Initialization seems like the best choice, especially in the constrained variant. Despite its slower convergence it often produces hierarchically consistent layouts that are sufficiently close to the equilibrium results faster than the other variants — they will either be incorrect or will have spent too much time during initialization. Note that this is very subjective to the particular graph and possibly also the implementation of the algorithms. Still, the strict hierarchy computation will always be very exhaustive as it needs to consider each possible path through the graph in the worst case.

To summarize, our algorithm of choice is the constrained Hierarchy–based Force-Directed Layout with Root–Leaf Initialization (compare Figure 6.5). It produces a well structured graph layout that is also guaranteed to have a locally consistent hierarchy. We can see the progress of the algorithm with different iteration counts in Figure 6.4

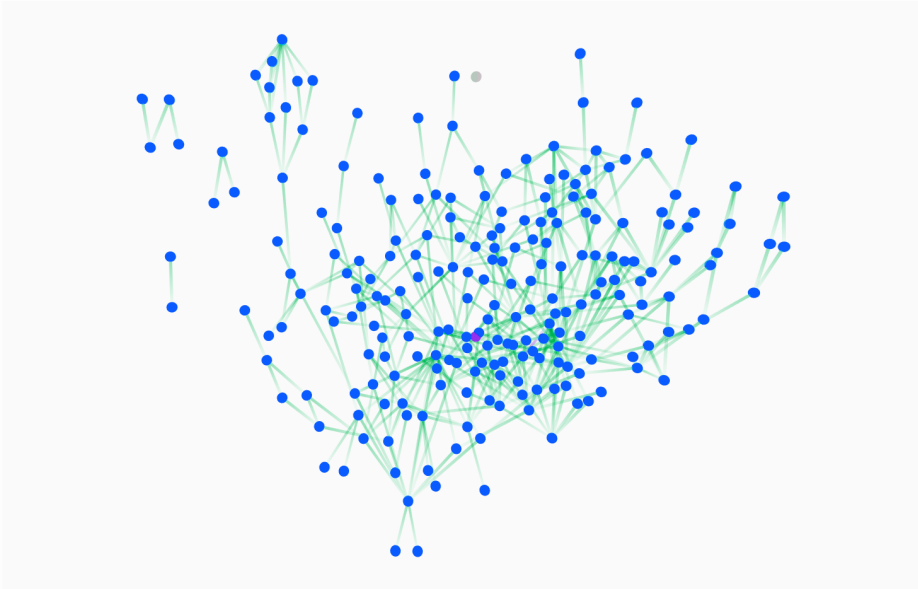


Figure 6.5: MitM Archive: Hierarchy–based Force-Directed Layout with Root–Leaf Initialization

6.3.4 Alternative Approaches

After talking about the experiments of integrating the hierarchy into 3D layout algorithms, we can now compare this to the earlier methods that combine hierarchy with force directed algorithms. Note that we have to do this based on a theoretical basis, as there was no time to implement these appropriately to directly compare the results. This will be an important task for future work.

Magnetic Spring Force Instead of introducing the hierarchic forces relative to node positions, the magnetic spring force [SM94] considers the edges. We have already defined that the local hierarchic consistency is equivalent to consistent edge directions. If we include an appropriate magnetic field, the magnetism tries to align the edges correctly. This means that we can imagine an edge as the needle of a compass, which rotates accordingly. Based on the length of the edge and the angle compared to the orientation of the magnetic field, the rotational magnetic force f_m is then

$$f_h(d, \theta) = d\theta, \tag{6.8}$$

when leaving out the additional parameters that are not necessary for the algorithm and the variable for specifying different magnetic fields.

The main difference here is the missing constraint to guarantee the local consistency of the hierarchy. If we compare the forces, the hierarchic force might lead to better results because the magnetic force reaches its balance when the edge is parallel to the magnetic field (in our case, for example pointing **straight** upward) — something that is not necessary for our criterion.

Hierarchy Energy In contrast to a hierarchic force, the hierarchy energy [CHK04] directly describes the term we want to minimize. Dwyer et al. [DK05] have extend this by introducing a constrained optimization that does separate between the two axes and defined additional constraints [DKM06] to counteract edge crossing.

As this does not make a difference for the hierarchy energy, we can stay at the original definition:

$$E_H(y) = \sum_{i,j=1}^{n} (y_i - y_j - \delta_{ij})^2, \tag{6.9}$$

where y is a vector of coordinates and δ_{ij} is the ideal vertical distance between two nodes. Similar to the algorithm of Kamada & Kawai [KK+89] this function is used to minimize the stress in the system.

Although the technique is different from what we do in TGView3D, it has similar implications: the stress in the system is minimized when each node has reached the correct position in the local hierarchy. However, this does not guarantee that the consistency is reached or maintained before the system reaches it equilibrium, so we still would need the additional constraints for theory graphs. I assume that other than that, the relation between the hierarchic force and hierarchy energy is comparable to the relation between the algorithms of Kamada & Kawai and Fruchterman & Reingold.

Iterations	10	20	50	100	200
NASA Library, Time [ms]	314	445	1150	2190	4273
MitM Archive, Time [ms]	106	107	211	319	542

Table 6.1: Layout Algorithm Time Measurements without Meta Edges (NASA Library with 771 nodes and 2851 edges, MitM Archive with 239 nodes and 673 edges), CPU used: AMD Ryzen 2600X.

6.4 Optimizations

While we now have an algorithm to handle the inheritance structure within theory graphs, there is still room for a few optimizations.

6.4.1 Performance

Since the layout computation is the most critical part of the graph loading process, we parallelize the force-directed part to take advantage of multi–core systems. We can implement this in Unity by using the Unity Job System. It allows the programmer to implement safe multi-threading by introducing several restrictions to avoid, for example, race conditions. For the actual parallelization, we split a single iteration into two jobs:

1. **Node Update** We can perform this step for each node in parallel. Nodes only need to read data from other nodes and compute their new position for the next iteration. Additionally, we have to store an energy value for each node for the next step to avoid synchronizing the sum update.

2. **Reduce the Temperature** This is a sequential step. We declare it as depended from the node update in Unity and thus use it as a way of synchronizing the threads. Then we build the sum of the individual energy values to determine the temperature reduction.

We then issue one pair of those jobs for each iteration, with each pair declared as depending from the previous one. Otherwise, there is no change to the algorithm. To give a rough idea of the performance we can expect from the overall layout computation in respect to the number of iterations, nodes and edges, I have included some measurements in Table 6.1. Note that the algorithm is in no way optimized aside from the parallelization and the time also includes generation of the 3D objects. While smaller graphs load almost instantly, we have to wait a few seconds for the bigger graphs that are available on MathHub.

6.4.2 Interaction Radius

Another possibility to increase the performance is to only compute forces for a certain defined neighborhood. This is usually implemented with the help of a grid. Then, only the nodes within the same grid are considered. This can produce worse results — especially without special initialization or other preprocessing — and the current performance is already acceptable for our purposes.

Nonetheless, we can use a similar technique to solve another problem: given enough space, very weakly connected subgraphs can drift of to the corners of the volume assigned to the layout algorithm. The reason for this is that we can only guess this volume heuristically based on the number of nodes and edges. To limit the repulsion causing this problem, we introduce an interaction radius to each node. This radius is as big as the distance to the farthest connected node. Then, we only apply the repulsive force if the interaction radii intersect. We can control this optimization by scaling these radii with an arbitrary factor.

6.4.3 Node Radius

Another option to influence the layout indirectly is to actually treat the nodes as spheres instead of points in the 3D space. When calculating the distance between two nodes, the radii then have to be subtracted from the distance of the node positions. Using the true radius of the spheres seems like the natural choice but the radius is small enough to only have very little effect on the layout. Generally, using a bigger radius increases the node spacing, but makes it harder to recognize groups and other structures in the layout.

This method becomes notably more important when using different sphere sizes, e.g., when bundling nodes as described in section 8.2.1. Then the layout will actually take care to not assign more space to bigger nodes.

Chapter 7

Immersive Graph Interaction

We have determined that the benefit of 3D graphs mainly arises from the combination of interactive graph exploration and VR interactions. This is especially true in cases where graphs are so big that even the best layout algorithm cannot represent the contained knowledge efficiently. To solve this problem, we need to reduce the information load. Accordingly, this chapter describes methods to explore the graph globally and locally, up to interacting with individual nodes.

7.1 User Interface

We start with discussing the way we can access basic options within TGView3D. There is a limit to how many interactions we can map to button combinations or gestures. We should should not focus on the actual number of available buttons too much, but rather keep the user experience in mind. If there are too many things to remember to use the application efficiently or the learning curve is too steep, users might be discouraged quickly. Hence, we need to have a virtual user interface (UI) with meaningful button labels instead of fully relying on the physical controller buttons. Furthermore, not every interaction should even be implemented by buttons with only two states, i.e., on and off. Imagine, multiple modes, from which the user can select one; then, a user interface with corresponding selections is also often the better alternative.

Interface Types There is no single UI that is optimal for all applications. Generally, there are three categories [VRUI]:

- **Non–diegetic** They refer to screen–overlay–type user interfaces. However, this type can not be utilized for virtual reality because the human eyes cannot focus elements that are too close to the game camera.

- **Diegetic** This term is used for interfaces that are connected to objects in the virtual world. A prime example would be a clock on the wall, but simply attaching text fields to objects also fall in this category.

- **Spatial** This type applies to interfaces that are affixed to a certain world coordinate. A special case of this variant is an interface that is placed relative to the camera.

Hand Interface While we could place a spatial UI in front of the camera, it would constantly occlude some part of the vision. Instead we use a diagetic UI attached to one hand of the user. Affixing an interface to the hand introduces several advantages regarding positioning, since the user can move the menu independently from the camera. This makes it possible to move it out of the way or adjust the distance freely. In addition, an offset of the object pivot further affects the distance to the menu when rotating hand.

This is also exploited by the structure of the menu itself. We split it into three parts that are placed next to each other. Rotating the hand can then change the positions of these relative to each other. The user can then, for example, let parts line up and put the left one to the front position. The same script that we have described for use with the node labels 5.2.2 also guarantees a constant orientation of the hand interface towards the user. Each consists of the same transparent rectangle. We place buttons for different interactions. This suffices for the current state of TGView3D, but in the future a more sophisticated interface might be appropriate. We could split the interface and allocate parts to both hands. We would also like to have a simple gesture to hide and show the interface, but this is difficult to implement with the Oculus Touch Controllers in a way that it is not triggered accidentally and always gets recognized. Instead, we the user can turn the interface on and off by pressing down the control stick of the hand the menu is attached to.

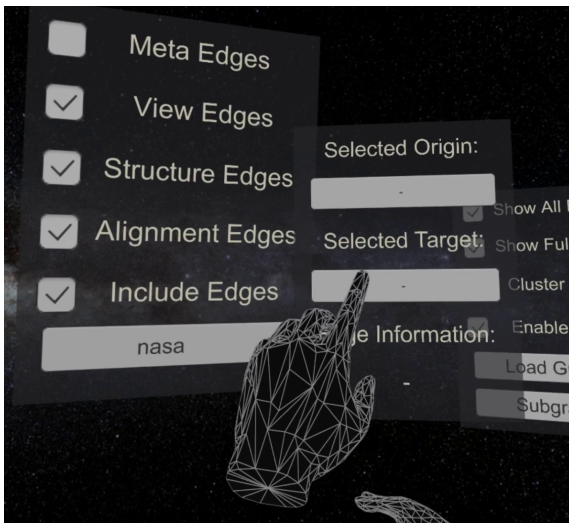
For the purpose of interacting with the interface, it serves as a virtual touchscreen. The user can trigger actions by touching the buttons with the index finger of the opposite hand. This is especially intuitive with the Oculus touch controllers, which can recognize whether this finger is pointing forward or not. Still, without any haptic feedback such a touchscreen cannot reach the user experience of a physical one. Luckily, this is no big issue when performing single press interactions, e.g., touching a tick box. More complex gestures such as swiping are not implemented for the touchscreen. Instead the controller buttons and movements are used.

7.2 Loading Graphs

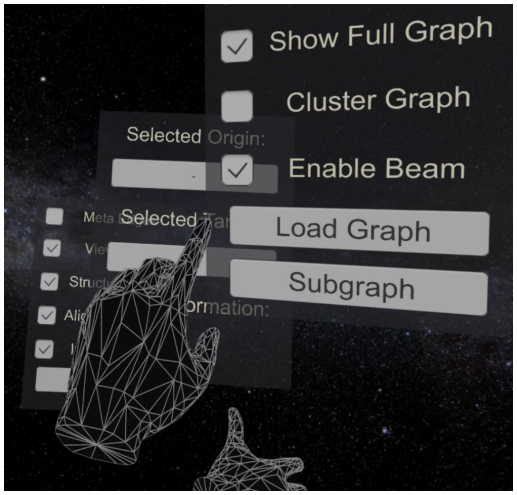
For the purpose of interacting with the graph, we have to load it first. Generally, there are two different scenarios of access:

- **Loading the application with one predefined graph** When using the graph visualization as supportive tool, users often only need to inspect one graph at a time, in particular the one containing the data that the user currently works with. Ideally, the user can access the graph by feeding a URL into the viewer. Refer to Section 8.1 for implementation details.

- **Choosing the graph from within the application** This is important when working with TGView3D as primary application, e.g., directly editing graphs. In the same sense a text editor can quickly load and store local files, doing this with graphs must be



(a) Users can trigger events by touch the menu with the left hand.



(b) Rotating the right hand switches the ordering of the menu parts (the fingers of the right hand are now oriented to the right side).

Figure 7.1: Hand Interface

possible. Alternatively, we could request the data that is necessary to browse through available graphs from a server, which, however, is not implemented yet.

Additionally, it might become necessary to recompute the layout after the graph has been modified. Since nodes already have computed positions, they could be used as initialization for the algorithm. The advantage of this would be the improved global orientation for the user: it is more likely that parts of the original graph will remain located at similar places and the user can watch how the algorithm moves the nodes. However, the final result could suffer as such a initialization can be biased towards the original layout, depending on how the algorithm is set up.

Loading Progress Beyond that, the user experience during the loading duration has to be considered. Without going too much into detail here, users do not like to look at frozen screens, not knowing whether progress is being made or not. Implementing a loading screen would for example improve this. A more elegant solution is asynchronous[1] computation. In combination with multi-threading we can even compute the layout without interrupting the rendering. This also allows the user to follow the individual iterations of the layout algorithm.

7.3 Graph Manipulations

Of course, users do not only want to load graphs but also want to interact with them. The possible manipulations can be categorized into two types: manipulating the whole graph, i.e.,

[1]In Unity this is implemented in the form of **Coroutines** which are decoupled from the rendering. This means that the computation can happen during multiple frames and the rendering can continue independently

translating, scaling and rotating and modifying individual nodes. Interacting with the graph as a whole rather serves the purpose of interactively exploring the graph. Changing the size of the graph allows the user to quickly switch between a global and a local view. Additionally, this allows to manually adapt to the varying inherent sizes of the individual graphs. Rotation and translation can be seen as a convenient alternative to moving around within the world. It does not make any semantic difference if the user pulls the graph closer or flies towards it. Most importantly, this way the problem of virtual reality sickness is largely avoided as the user itself does not move.

Furthermore, all of these operations are triggered by gestures that try to mimic the movement that is expected from real world habits. However, the user still has initiate them with the help of buttons, as the finger tracking of the touch controllers is not accurate enough in order to use it for gestures.

Rotation Pressing the right index finger button and moving the controller horizontally rotates the graph, akin to steering a wheel in a car. Note that this is intentionally limited to rotations around the axis pointing upwards, because this guarantees to preserve a consistent hierarchy, i. e. edges pointing downwards will keep this property.

Translation The graph follows the hand while the right index finger button is pushed down, respectively. Whether the amplitude of this movement is directly translated into the change of position or has some factor attached to it is a matter of preference. One might argue that the latter is less intuitive, but for the purpose of efficient navigation in a relatively vast space even a factor of 30 is reasonable.

Scaling Pressing the respective button on both controllers simultaneously commences the start of the scaling operation. Bringing the controllers closer together scales down the graph, while the opposite happens, when the distance is increased. Depending on the angle that is spanned between the vector from one hand to the other and the ground three different modes are possible: scaling along all dimensions, only considering the height or everything except the height. It is not necessary to change the width and the depth of the graph separately from each other, since the layout algorithm does not differentiate between those two dimensions. For the height, however, an additional component is introduced. Hence, it may help to visually split different hierarchy layers further by increasing the height of the graph or vice versa.

Note that resizing an object in computer graphics usually results in a uniform movement of each vertex of the geometry. In contrast to this, the user only wants to affect the node spacing when scaling graphs. Therefore, only the length of edges should change, while their diameter as well as the size of nodes must stay constant. We achieve this by scaling the position of the nodes and redrawing the edges accordingly.

Node Modification Individual node modifications are mostly use case specific and are beyond the scope of this thesis except for proof of concepts regarding the use cases as described in Section 8.2. A general attribute would be the position of nodes. For graphs of enormous

size, it is reasonable to compute the layout once and then store the positions for further uses. Smaller graph types like MPDs require externally supplied node positions because of another reason: their final layout is not generated by a an algorithm but by a domain expert. Possibly, the graph structure could even be built within TGView3D. However, this requires many different editing features such as undo, redo, delete, create etc. and thus is very hard to integrate into VR.

Note that this also affects the layout itself.

7.4 Node Interaction

(a) Puling a node closer with a tractor beam.

(b) Selected node and target appear in the UI.

Figure 7.2: Selecting nodes.

Interacting with single nodes requires a selection mechanism. The most intuitive variant for a 3D–world is grabbing. The Oculus touch controllers achieve this by placing a trigger so that it is pressed down when closing the hands in the real world. In the virtual world, the virtual hand then also performs a grabbing motion. When the hand comes into contact with a node while doing so, it will actually grab the node. We then mark it as selected within TGView3D.

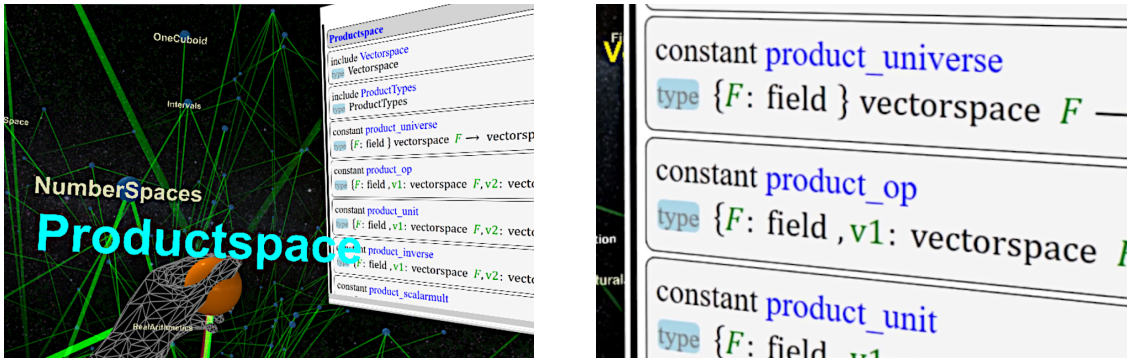
But how to handle nodes that are not in range of the hands? Constantly moving is stressful, so a different method was implemented: performing the grab interaction launches a tractor beam that drags the first node hit towards the respective hand. The tractor beam stays active until the node has reached the hand, even if no hit was registered. The user can then use the now visible cone as a reference to guarantee that the next try will work. Note that in this case, we do not use the beam as a ray cast but instead, check whether any node sphere collides the beam cone.

The selection of a node itself has direct consequences:

- The edges of this node are redrawn every frame.
- The node label and sphere will change their colors.
- The node label appears within the interface.

Furthermore, additional interactions are possible. The user can use the right stick to iterate through the connected nodes. We show the target label and the edge label for the current selection. This is effectively an edge selection. We also automatically orient the camera towards the target node, which is highlighted with a specific color.

Both — the origin node label and the target node label — now serve as buttons that create a window containing information regarding this node (compare Figures 7.2 and 7.3). Pressing the button a second time will close the window again. We access this information by following the URI that is stored for each node. Unity first loads the respective URL with the default web browser. Then, we stream the window from the desktop into VR, based on an external Unity VR Desktop Mirror Asset [VRDTM].



(a) Grabbing and selecting a node.

(b) Virtual screen with node details.

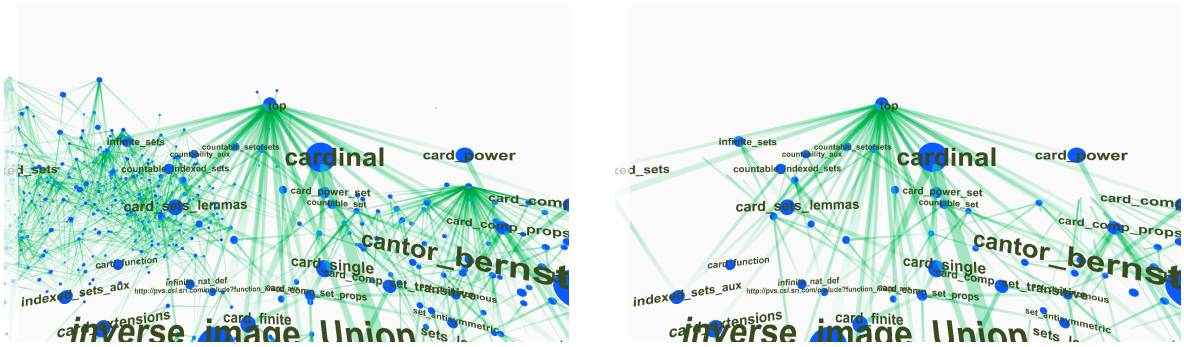
Figure 7.3: Interacting with nodes.

The window can be seen as spatial interface. We spawn it in front of the user at a comfortable viewing distance and then stays at this position. Similar to the hand interface, it allows interaction in the form of a touchscreen. We emulate mouse clicks by touching the window with the index finger and stream the clicks back to the browser.

We could implement various interactions with other systems similarly. Alternative interfaces might then also be reasonable such as a diagetic one attached to the selected node, which are capable of directly show a limited amount of extra information.

7.5 Filtering of Information

Although we have now introduced tools to explore the graph, complex graphs often contain so much information that it is almost impossible to recognize structure. An abundance of edges is very problematic, as it is already difficult to follow single edges with the eye in such a case (compare Fig. 7.4), but even worse, the strong connectivity often prevents layout algorithms from forming communities within the graph. On the other side, graphs that are sparse, this means they have a high number of nodes compared to the number of edges, are organized quite well by those algorithms, even if they are big. Exploiting the third dimension already helps by creating more space for the graph, but the parts in the distance still interfere with the part in focus. However, there is a simple solution to this. We can configure the camera to



(a) Too many edges make it hard to focus.

(b) Cutting away parts in the distance

Figure 7.4: Basic 3D Information Filtering (NASA PVS Library theory graph)

cut away any geometry that lies beyond a certain distance by setting the far plane accordingly. Depending on whether the user currently wishes to explore the graph locally or globally, we move the plane closer within the application.

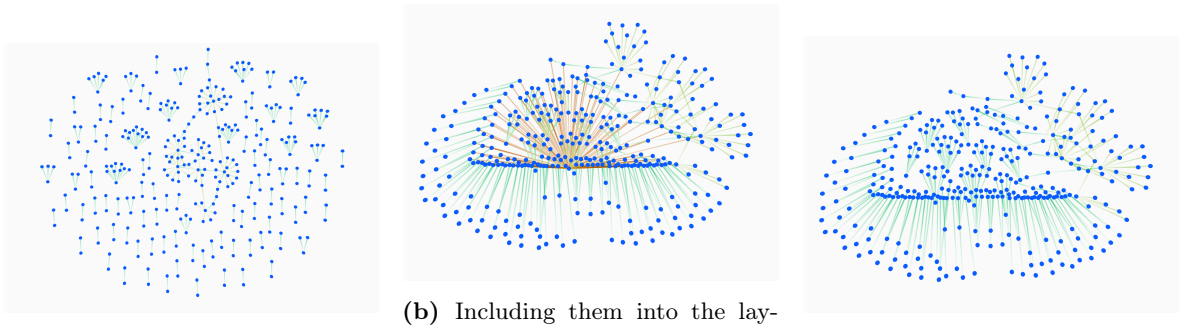
7.5.1 Filter Methods

We can achieve similar effects by only showing a subset of edges, which also can be applied to the 2D graphs. The UI offers different ways of choosing them. The user can:

- disable certain types of edges.
- only show the direct neighborhood of a node.
- calculate a specific reachable subgraph of a node.

Selecting Edge Types As each edge has a certain type, we allow the user to choose which type of edges should be active and which not. Furthermore, the layout can be recalculated with only considering the active edges. The effectiveness of these measures depends on the specific graph and how important it is to see different types of relations at the same time. Still, this is a very effective way of reducing the number of nodes, as it also grants new insights how certain combinations of edge type influence the relations between nodes. For example, imagine a graph with two different kinds of hierarchic relations, e.g belonging to and depending on a directory. Such a combination can have big impacts on the layout, possibly even creating multiple not connected subgraphs. In this sense, especially meta edges are a double-edged sword. While they can create additional structure, there often only are a few nodes that have meta edges to almost every other node (see Figure 7.5).

Direct Node Neighborhood This is very useful when following the connections of a single node. Having up to two selected nodes at once proved to be very useful, because this allows to also check all edges of nodes connected to the first selection. We implement this by having separate selections for each hand. Without this option, it might be better to also include the second set of edges right away.



(a) Without meta edges, the graph fragments.

(b) Including them into the layout reveals more structure, but the meta edges occlude it partly.

(c) Deactivating them again improves this.

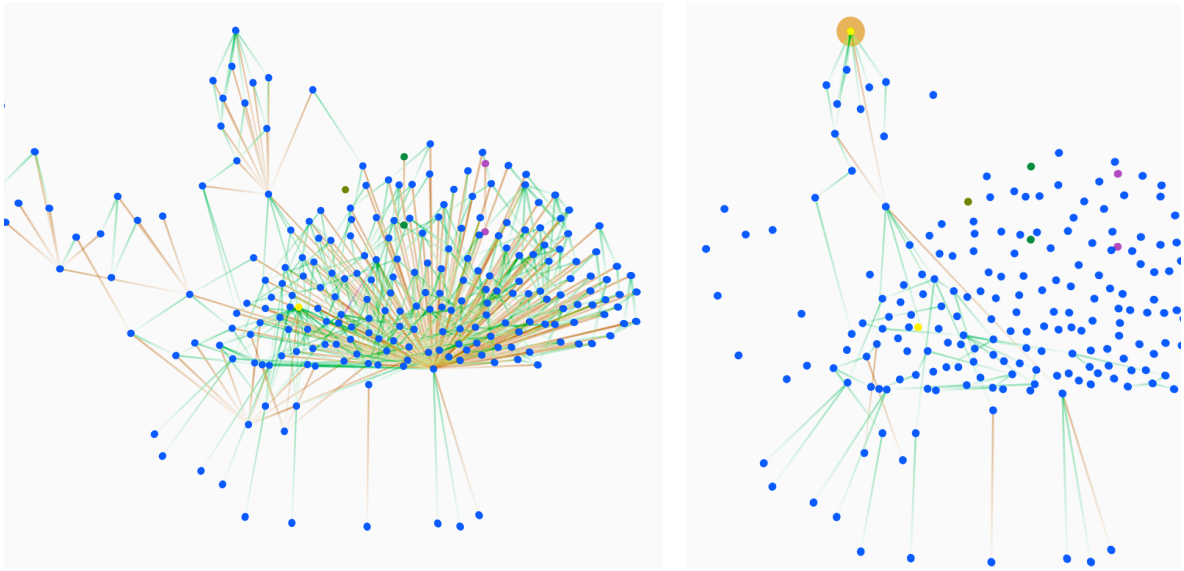
Figure 7.5: The effect of meta edges in the HOL Light theory graph.

Calculating Special Subgraphs For this variant, we only take the directed edges into account. As the graphs we use in TGView3D are generally hierarchic, edges can either point downward of upward. In particular, we calculate two reachable subgraphs based on one selected node. The first one is the usual one, progressing along edge directions, starting from the selected node. The second one is exactly the opposite subgraph, that is reached by following the edges opposite to their directions.

In comparison to the variant with direct neighborhood, the number of nodes within such a subgraph is far greater. Note that we only keep the edges that are part of the combined subgraph and not all edges of all nodes that are part of the subgraph. As a consequence, this intuitively shows what part of the original graph is related to the selected node. Following the edges along one direction reveals all roots, from which this node can be reached; by following the edges along the other, the user arrives at the leaf nodes. Looking at the whole subgraph, the occupied volume relates to the strength of the connectivity within the whole graph. A weakly connected subgraph would then appear closely packed into one part of the space, while a strongly connected one would spread out.

7.5.2 Reloading the Layout

Additionally, recalculating the layout with only considering the remaining active edges provides very interesting insights as well. The information how the subgraph fits into the global structure is lost, but this can be very helpful, since the layout algorithm can now focus on a smaller graph. An originally very dense selection will become expanded, whereas a sparse one will become compressed. Yet, in both cases new patterns or groups may become discernible, because the respective parts of the graph have been too close or too far from each other before. Starting point for such a recalculation could be the results from edge selections in the form of the described subgraphs or edge type selections.



(a) Base layout, dense structure is problematic.

(b) Subgraph reveals structure.

Figure 7.6: Activating subgraphs (MitM Ontology theory graph with meta edges)

7.6 Theory Graph Clustering

- DBSCAN randomly picks an existing feature vector — in our case a node position — and initializes a cluster.
- It adds all nodes to the cluster that do not exceed a certain distance threshold. For our theory graph, we only put nodes into the same cluster that are reachable from the current node.

In contrast to the other mentioned clustering approaches, here the layout has to be computed first to give the clustering algorithm meaningful positions to work with. The advantage is that we can work with the result of the specifically developed hierarchical algorithms.

Despite this measure, the adapted DBSCAN does not work for all graphs. We define the distance threshold relative to the average edge length within the graph, more specifically as $\frac{1}{8}$ of the average edge length. But even then, substructures sometimes are too small to be recognized as different clusters, although other spatial clustering algorithms or even an advanced heuristics might improve this. However, it still is very helpful in some cases: it can

- reveal structures that are hidden by too many edges or very dense parts in the graph, which we demonstrate in Figure 7.8.
- emphasize groups visually that are formed by the layout algorithm, compare Figure 7.7.
- be used to change the layout or bundle nodes as described in section 8.2.1.

Although we have have only been able to touch clustering methods during this work, it already turns out to be a very effective tool to not only organize regular graphs, but also theory graphs — if we adapt the algorithms correctly.

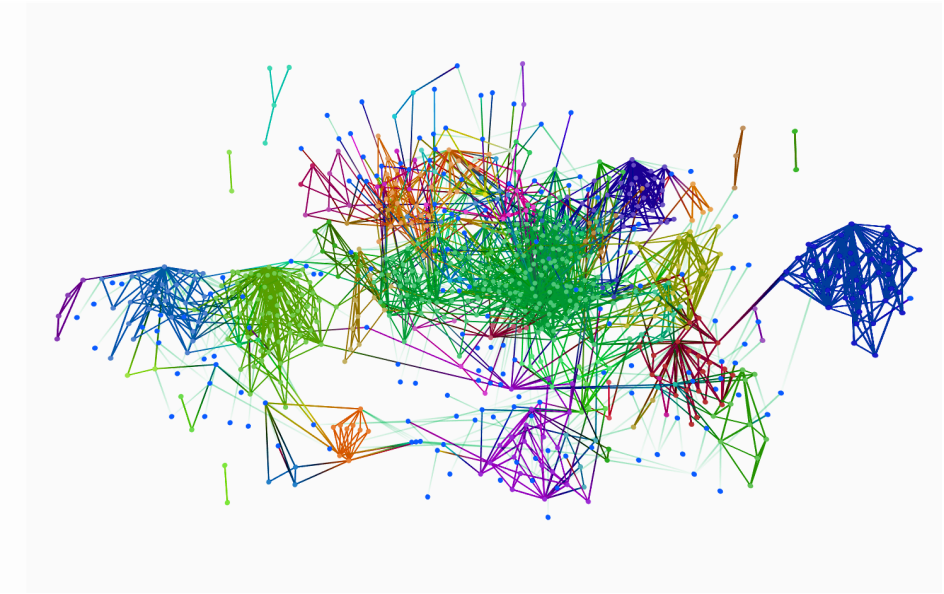
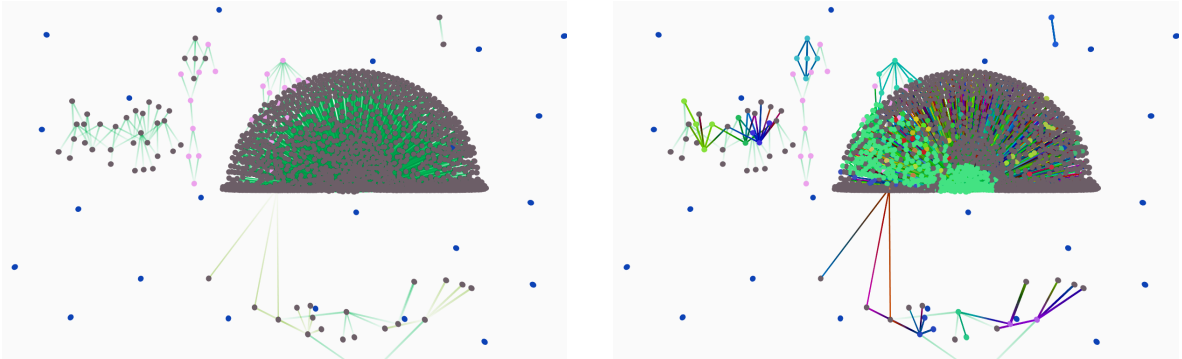


Figure 7.7: Spatial Clustering (NASA PVS Library)



(a) Although there is some structure outside, the part in the middle seems to just be a pile of nodes.

(b) Clustering reveals a structure that is hidden within the left part of the big ball.

Figure 7.8: Revealing hidden structures (ODK Archive)

Chapter 8

Implementation and Evaluation

We have now learned about the core parts of TGView3D and its ecosystem. This already covers big parts of the system implementation; to complete this, I will now describe the non-VR implementation and the integration of the different use cases and workflows. With the results of these experiments, we can then evaluate the methods of 3D and VR data visualization.

8.1 Non-VR Implementation

We have already noted that 3D graph visualizations have certain disadvantages compared to regular 2D layouts, especially without the help of VR devices. Furthermore, many potential users do not have access to these devices. To reach these users and provide comparable experience, we can emulate the described VR interactions with mouse and keyboard. We implement them as follows:

- **Navigation** For moving within the 3D space, we adopt controls that are known from video games and 3D modeling software. The mouse is responsible for changing the viewing direction and the keyboard buttons allow movement along all axes in relation to this direction.

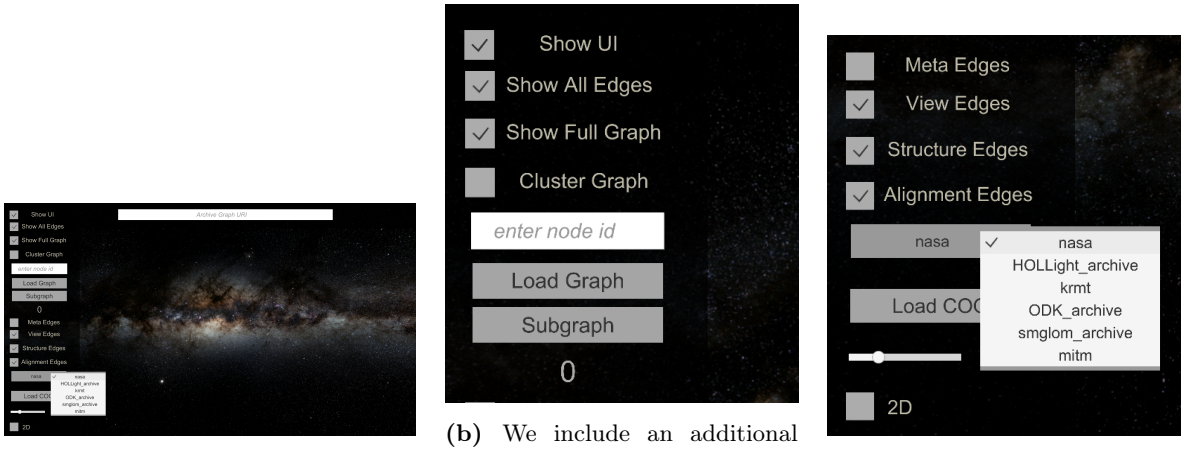
- **Node Interaction** Nodes can be dragged around with the mouse. To adopt this method for three dimensions, we compute the distance of the node to the screen and move the node, so that it will always keep this distance. This means that users can also simply pick up nodes with a mouse click and carry them along while navigating through the virtual world. A double click on a node opens the respective URL in the web browser but in contrast to the virtual reality variant, we will not stream the page into the application as users can simply work with multiple windows.

- **Visual Filtering** We use the same methods as before, but replace the hand interface by a screen overlay that brings together all the menu options on the left side of the screen as we show in Figure 8.1.

While these interactions are often less intuitive, there are certain problems that can be solved significantly easier with a physical keyboard. Most notably, inputting text, which allows two new interactions:

- **Node Search** As an alternative to clicking on nodes, the user can input a string into a text field. We then start a search that returns either the node with this exact label or — if there is no such node — the first node whose label includes this string. Although the labels are not unique sometimes and thus cannot identify one particular node, we do not use the MMT IDs, as inputting these long strings is not very user friendly. If there is a match, we trigger the subgraph computation with the found node as origin.

- **Graph URL Input** Instead of choosing from a set of predefined graphs, the user can enter the MMT URI of a graph and directly load it into TGView3D.



(a) Full view of 2D UI, Graph URL input field

(b) We include an additional node search field and a number for the layouting progress.

(c) More 2D specific features: 2D mode and spacing scaler

Figure 8.1: 2D User Interface with Space Background

Based on these features, we can build a separate application that does not require any VR hardware on the one hand, but also include the screen overlay into the VR application on the other hand. It is then only visible on the monitor and serves as a workaround for the lacking text input capabilities.

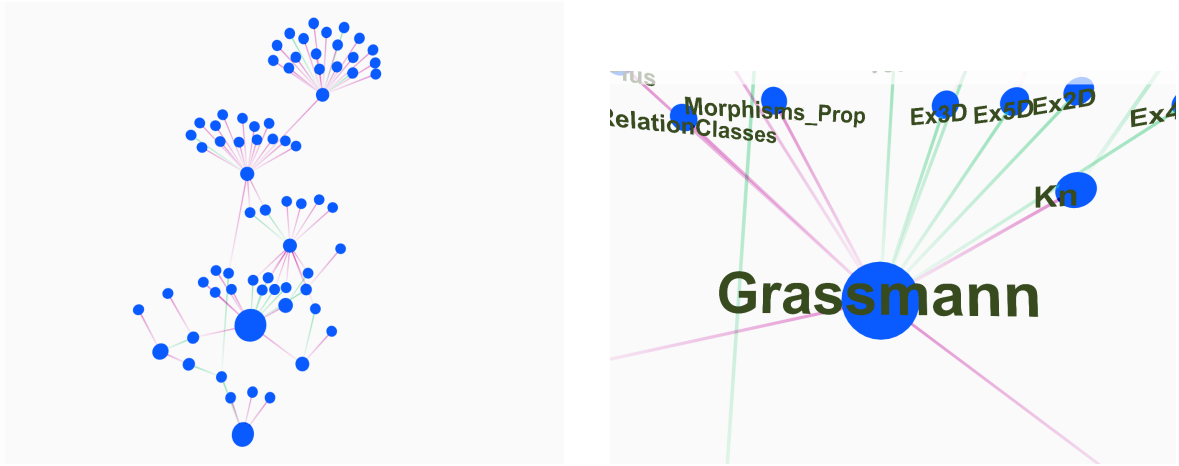
Furthermore, we can use this to deploy a web build based on WebGL. Although our current VR interactions are not compatible yet, users can at least activate a VR view and look around in addition to the regular mouse and keyboard controls. This is achieved with the help of WebVR and even allows other VR devices than the Oculus Rift to be used. Using a browser also offers another advantage. We can include an MMT graph URI into the link to the web instance of TGView3D and load the passed theory graph. However, we lose the benefits of multi-threaded calculations and other platform specific optimizations.

8.2 Use Case Implementation and Evaluation

With TGView3D as VR as well as normal 3D system, we have all the tools to implement different use cases. We can directly skip the mathematical archives. As core application of TGView3D, we already have discussed the necessary interactions in detail during the previous chapters. To get quick results, I have implemented each of the remaining workflows as proof of concept.

8.2.1 Coq Library Graphs

Let us start with requirement to cluster the libraries (C2). The meta information available does only help to compute clusters between different Coq libraries and not within a single one. Although very interesting — such global relations are only secondary for the users I have interviewed. Accordingly, I have decided to not further pursue this within the limited time of this project and focus on the library exploration (C1) instead.

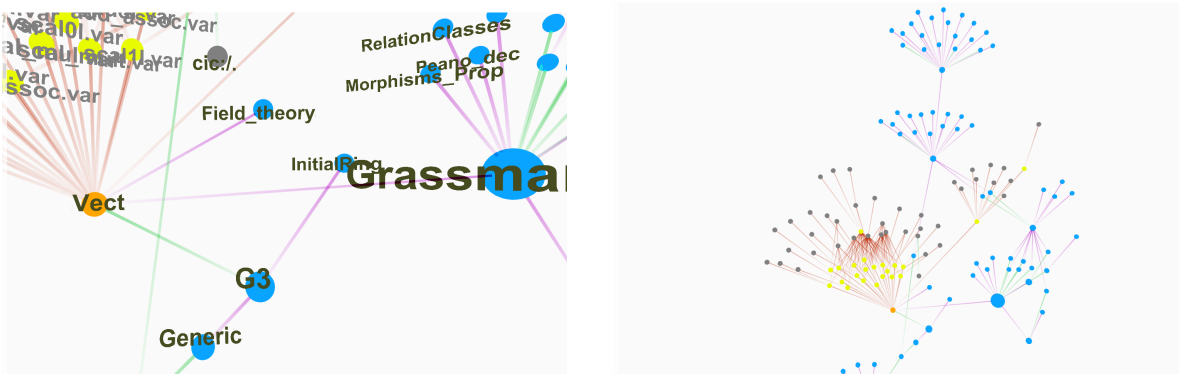


(a) Reduced Coq Graph **(b)** Zoom to Node

Figure 8.2: Coq: Geometric Algebra

The first step is to import the CSV files and perform a transitive reduction. As this is not part of the TGView3D feature set, we must process this data externally to arrive at a JSON file similar to the one we use for theory graphs. Even after reduction, the remaining amount of edges makes it hard for humans to maintain an overview, so we need to reduce the information load. We can achieve this by exploiting the structure of directories and objects. This structure is ideal for lazy loading, i.e., we do not load the contents of directories at first and give the user the option to expand directories selectively. In the case of the the formalization of Grassmann-Cayley algebra [FT11], we can see the initial result in Figure 8.2. The user can then either select a single node or use the search function. Based on the selection, more nodes and edges are added to the graph structure after reloading the graph via the usual graph load button, which I demonstrate in Figure 8.3.

At the time of finishing the writing of this thesis, a formalization of the coq libraries as



(a) The objects that have been hidden within Vect are now visible.

(b) The relations between these forms as substructure within the graph.

Figure 8.3: Expanding Nodes (Coq: Geometric Algebra)

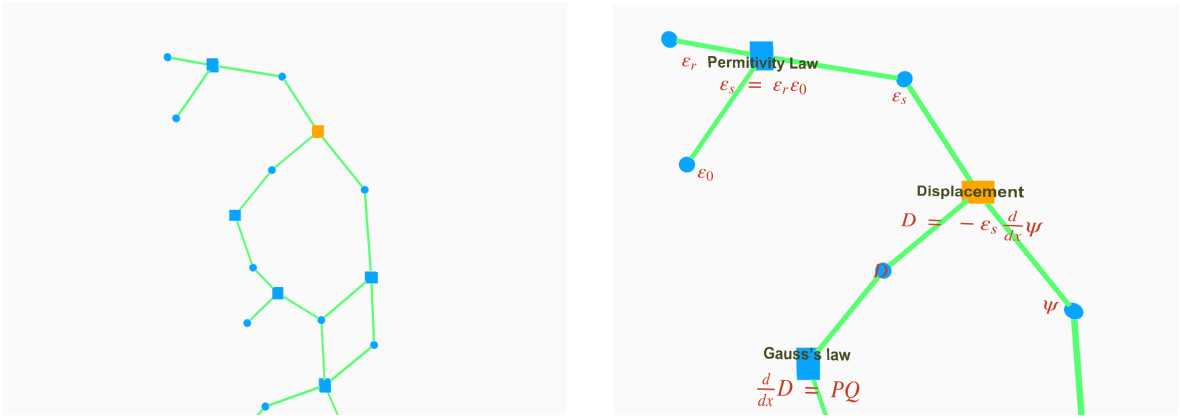
theory graphs [MRS19] had begun. Consequently, there is not much benefit of supporting coq libraries separately; instead we can profit from the lessons learned and integrate the concepts into the general theory graph workflow. In particular, we could extend the concept of node encapsulating to theory graphs. There are two areas where we can exploit their format structure:

- There is a tree relation between MMT graph files, e.g., we can access the MitM library as a whole or a part like MitM/Foundation specifically.
- There are also the contents of theories. Despite their role as requirement of mathematical archives, we have skipped them so far as we have the option to access the information in the form of an web page.

In general, we could also implement this based on an appropriate form of clustering as described in section 2.2 or based on the graph hierarchy. For the latter there two ways of encapsulating nodes. The first option is to cut the graph at a fixed depth. This is trivial to implement but might create very unbalanced groups. Alternatively, we can combine the nodes as soon as paths leading down within the graph have collected a certain number of nodes. Both of these are basically forms of clustering and we potentially could introduce additional criteria and define a method of clustering arbitrary DAGs, but exploring this further is out of scope for this thesis.

8.2.2 MPDs

For the MPDs, we need to modify the regular appearance of 3D theory graphs. To represent the circles of physical quantities and rectangles of physical laws we utilize the 3D pendants: sphere and cuboid. We can see the 3D visualization of the van Roosbroeck system in Figure 8.4. To convert the MathML expressions to SVGs and then to meshes (M2), we have only used the server locally so far. To use TGView3D offline, we use an external file to store the SVGs. Accessing external media (M3) is possible via the node URLs, which do not have to be MMT



(a) Full MPD Graph

(b) Zoom to Node– and MathML–labels

Figure 8.4: MPD of the unipolar van Roosbroeck system as theory graph

URIs. Instead, they could link to other web pages, which we can stream into the virtual world, or even directly load pictures into TGView3D. While there are plans for this, there currently is no such information available.

(a) We can recognize the cube easily.

(b) There are no edge crossings, but we can only see a cube at second glance.

Figure 8.5: Planar Projections of a Cube.

Among the graph editing and creation features (M1) we only have the interactions described in Section 7.3. While these do not suffice to build graphs, the MPD prototype allows users to export and import the node positions. Doing so, they can at least build custom layouts. To evaluate this feature in respect to other systems, we need to compare how experts can emphasize the structure of the physical model between 2D and 3D. To do this, we can also look at the general behavior of geometry in 3D in Figure 8.5. Traditional algorithms that try to reduce edge crossings may lose attributes that are valuable for understanding, but trying to preserve them is also problematic. For example, we use the 2D results of the layout algorithm to produce static pictures for this thesis. Going back to the example of the cube, only an interactive 3D application can really portray this structure as we can move the camera

and view the cube from many different perspectives. In the same sense, we can understand the advantages of 3D representation of the MPD structures. As to how this actually helps in practice, we cannot really draw a final conclusion based on the experiments of this thesis alone.

8.2.3 Attack Graphs

(a) Attack graph: Roe v. Wade, 410 U.S. 113 (1973) **(b)** Simple Reasoning with Tweety the penguin.

Figure 8.6: Attack graphs with attacks and inclusions

The attack graphs used here are already formalized with the OMDoc/MMT language. Beside the attack edges, they also may have include edges to express logical reasoning (compare Fig 8.6).

```
theory background_knowledge =
  include found:?Logic |
  include found:?InformalProofs |
  human: type |
  black: human |
  fetus: human |
  is_person: human → prop |
  has_rights: human → prop |
  is_born: human → prop |
  is_conceived: human → prop |
  is_black: human → prop |
  is_caucasian: human → prop |
  def_fetus: ⊢ is_conceived fetus ∧ ¬ is_born fetus |
  def_black: ⊢ is_conceived black ∧ ¬ is_caucasian black |
```

(a) Grounded

```
theory fourteenth_amendment =
  include ?background_knowledge|
  person: ⊢ is_person black |
```

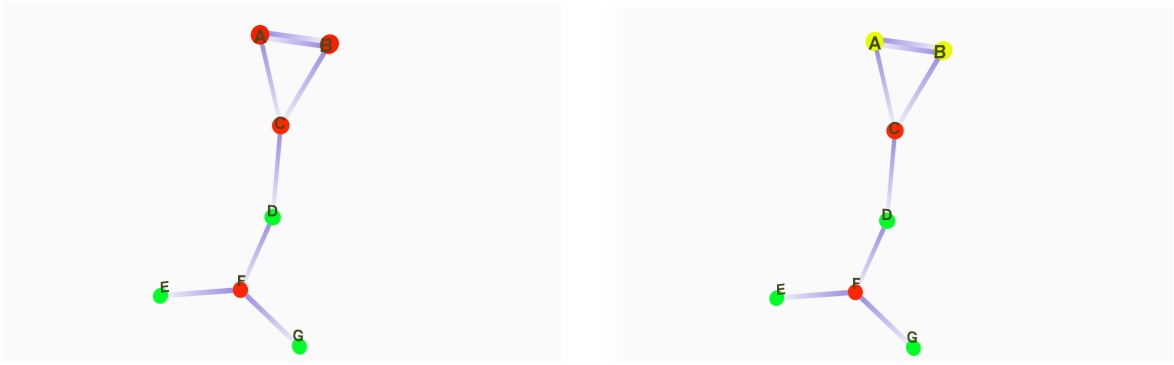
```
theory constitution_pro_slavery =
  include ?background_knowledge |
  person: ⊢ ∀[x:human] (is_person x ⇒ is_caucasian x) |
```

(b) Stable

Figure 8.7: MMT source code of the Roe v. Wade attack graph.

The reason for this is the limited implementation of graph editing features. This means that we also cannot modify the nodes and edges to have an impact on the acceptance within TGView3D (A2). Instead, we rely on the user directly editing the MMT source file (we can see code snippets of a attack graph in Figure 8.7); updating this file would then affect the graph in TGView3D upon reloading.

Within TGView3D, we introduce further options to the URI for the requirement of initiating computations (A1). Users can specify a **semantic** [BCG11] (currently **grounded** or



(a) Grounded: A and B are rejected

(b) Stable: A and B are undecided

Figure 8.8: Results of different semantics, C attacks A and B, which attack each other

stable) and a solver (currently only one default solver [CVG17]) to compute the acceptance of arguments. The grounded semantic selects the minimal subgraph that does not contain conflicting arguments and defends in a certain sense all of the arguments it contains. The stable semantic selects any subgraph that is conflict-free and attacks any argument it does not contain. As there may be more than one of these subgraphs, the stable semantic can yield undecided arguments (compare Fig. 8.8). To visualize this in the graph, we color accepted arguments green, undecided arguments yellow and rejected ones red.

For the evaluation of 3D to represent attack graphs we can refer to the MPDs in Section 8.2.2. The difference is that in attack graphs, we can see loops and similar dependencies between argument groups that affect each other, whereas in MPDs, the physical dependencies result in loops. Note that in both cases, users could create not only complex, but also very big graphs. However, that is counterproductive for human understanding, so researchers often limit their models on purpose. With help of interactive visualization tools, we can maybe motivate researchers to create bigger models that also highlight global dependencies between different parts that possibly have been excluded before.

8.3 General Evaluation

After analyzing the use cases, we have completed the full description of the TGView3D functionalities. We can now go back to the four global requirements we have set in Section 4.2: **Using TGView3D**, **Visualizing Graphs**, **Data Interaction** and **System Interaction**.

8.3.1 GR1: Using TGView3D

Although our first requirement not only applies to 3D application but all kinds of software in general, supporting 3D and VR workflows has special implications.

Platform With the help of Unity, we can deliver builds to all necessary platforms and also do so. However, VR introduces practical problems. First, possible users often do not already own an HMD like the Oculus Rift and would have to buy one specifically for TGView3D.

Second, the APIs to build interactions for different devices are still in their early phases, which makes it sometimes necessary to rewrite code to support devices of other manufactures. This also applies to maintaining a VR and non VR version at the same time. While it is very helpful to have a version with an easy entry barrier, there is the danger of users getting a wrong impression of the full capabilities of TGView3D.

Graph Access While there are have been some detours to prototype different workflows, the current path to access theory graphs are MMT request, which can return a wide range of different theory graphs accessible by URI. This was no problem for the TGView3D experiment, but poses a limitation compared to other graph systems, which often can import different types of files, including common graph formats such as GraphML or CSV. In the same sense, it was never necessary to load local graph files via file explorer, which should be included in the future.

8.3.2 GR2: Visualizing Graphs

Visualizing consists of two parts: rendering the graph and computing the layout. Nonetheless, both of these heavily influence each other.

3D Graph Visualization The experiments with TGView3D has shown that there are cases where the 3D visualization is a significant advantage when users want to better understand global structures — this can either be the inheritance aspects of the OMDoc/MMT format or new unknown structures that emerge from the computed layout. This is achieved by the improved separation of data in combination with the option to look at the graph from different perspectives. Exploring the 3D graphs in virtual reality multiplies this advantage.

Despite this, 2D graphs are superior when it comes to static images. The advantage of multiple layers in 3D turns into a disadvantage as content within the front layers can occlude what lies behind. Furthermore, 3D visualization engines sacrifice resolution to be able to handle a huge number of polygons. Even when increasing the polygon count, they cannot reach the clarity of SVG images, which is a possible output format in traditional graph systems such as GraphViz.

Theory Graph Layout Computation Based on a regular force-directed approach we arrive at an algorithm that incorporates new hierarchic forces. This force is exerted exclusively vertically and allows the remaining forces to function as usual. To guarantee that the edge directions consistently point upwards, we add constraints to the algorithm that restrict the movement of nodes so that the layout cannot become invalid.

For this algorithm, three dimensions are essential. If we are talking about traditional layout algorithms for undirected graphs such as force-directed approaches, moving from a 3D layout to a 2D one is very acceptable, especially for non interactive graph representations. For the hierarchical algorithm, however, it is extremely problematic. In 2D, there is only one dimension to fully exert the non hierarchically forces. The vertical dimension is heavily constrained and only allows little influence of the repelling and attractive forces.

This is also the reason why existing systems usually use other algorithms to draw DAGs or trees. These optimize the layout with carefully designed heuristics, which, for example, specifically minimize the number of edge crossings. Because of this, we cannot convert them directly to 3D. Although the hierarchical algorithm of TGView3D is by no means fully optimized and there are alternative algorithms that would work in 3D, the results perfectly achieve our goals. Even the 2D variant of it can hold up against the sugiyama method; we can see the GraphViz result of the MitM theory graph in Figure 8.9. While the edges are looking cleaner, the graph does not feature the small group that always shows in TGView3D and the layout needs to stretch horizontally because of the fixed layers.

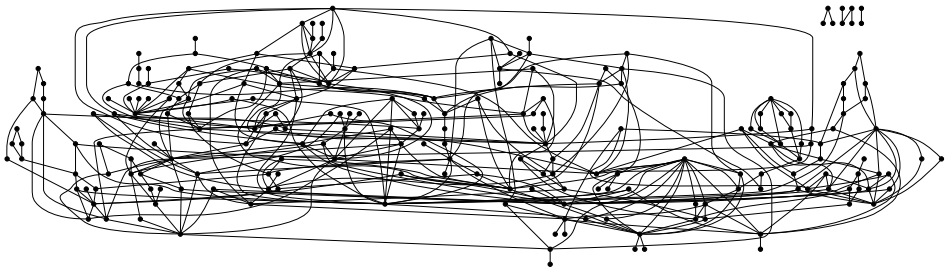


Figure 8.9: GraphViz: MitM Archive.

8.3.3 GR3: Graph Interaction

Interacting with graphs includes exploring them on a global level up to accessing information of individual nodes. In particular, we need think about these workflows in the context of existing systems.

Data Interaction Interacting with 2D data is well explored — we use these everywhere whether they are touchscreens or regular monitors with a computer mouse. Hence, there are efficient interaction solutions, which of course are utilized in state-of-the-art visualization systems. For example, moving nodes of a graph within a plane with the help of the mouse is extremely intuitive.

These, however do not translate very well into 3D visualizations. Existing 3D systems often avoid interaction with direct individual elements and focus on offering different settings and exploration. VR interactions offer solutions. Grabbing nodes with virtual hands and moving them is just as natural as moving a node with a mouse in 2D.

However, there are technical problems. First, the hand is currently emulated with special controllers. While these serve their purpose, they do not fully reach the level of, e.g., grabbing something with our real hands. The user first has to learn about the necessary buttons and how the device works. Second, VR interactions are generally supposed to happen within an immersive environment. Interacting with the outside world, i.e., a book or desktop applications is difficult. TGView3D uses the possibility of streaming desktop content into virtual reality, which can only solve this problem to some degree.

This means that currently, VR interactions cannot fully replace 2D workflows, but are

often superior to alternative 3D methods. In particular, exploration of big archive theory graphs works really well in VR, but the feature set necessary for editing graphs introduces many problems — to the point that I consider simply forcing these VR interactions into TGView3D as harmful for the overall user experience. While new devices and interaction concepts promise to offer solutions, the combination of 2D and 3D workflows also seems very potent. There is already a basic implementation of this is TGView3D: for example, we can use the 2D interface to search a node and activate certain filters. Then, we can put on the Oculus Rift and explore the graph. This way it is also feasible to separate 2D graph creation and 3D graph exploration. In this sense, TGView, for example, can act as the initial tool to build a theory graph that we then can import into TGView3D.

Information Filtering Although 3D graphs already filter the information during the exploration of the 3D structures inherently, specific methods to reduce the information density work very well for both 3D and 2D graphs. Especially for DAGs, the hourglass subgraphs proved to be a valuable tool to quickly focus on the currently relevant parts. But there is also more room for experimentation. Extending the concept of encapsulation and clustering that I have started to implement for the Coq libraries promises to be very helpful. Other systems offer features such as manual clustering or other ways of highlighting parts in the graph that we can also build upon in the future.

8.3.4 GR4: System Interaction

The most generic variant is to simply open web pages that relate to the graph content. We can either open them next to TGView3D or stream the content into the virtual world for use in VR. The main way of interacting with actual systems, in turn, are MMT requests. This is also necessary to access the graph files in the first place. The most interactive workflow is the computation of a different a coloring for argument graphs, as the results of user interaction with the graph and changes to the source files directly affect each other. Based on such requests to HTTP or HTTPS web servers in general, i.e., not only MMT, we can offer an interface for communication with external systems. Of course, that is not to say that existing graph systems cannot achieve this the same way but rather that nothing is stopping us to use this in a 3D or VR setting.

A related question that was part of the motivation for this thesis, is how we can support workflows of scientists with 3D visualization systems. And while we can implement them like this, there must be the need for interacting with graphs like there is for argument graph users or at least data that we can represent as graphs like MPDs. In the most basic form, TGView3D could just be a tool to debug dependency graphs that are very cumbersome to go through in the form of list, maybe compiler error messages to name a simple example. To truly understand how much scientists actually benefit from such tools, we need to find even more workflows and most importantly, formally evaluate the impact of 3D — which we now can accomplish with the help TGView3D.

Chapter 9

Conclusion and Future Work

As an initial experiment, it was naturally not even planned to exhaustively answer all of our initial questions, but we have explored all of the relevant aspects to some degree. Accordingly, I will first conclude this thesis and then point out possible research directions for the future.

9.1 Conclusion

To answer the main research question on how 3D visualization helps user to understand theory graphs, we need to consider two aspects. First, the inheritance structure that is formed by the inclusions induces an directed acyclic subgraph, which needs to be emphasized in the layout and this largely occupies the vertical dimension. I have introduced an algorithm that is capable of handling this by integrating a hierarchical force into traditional force-directed approaches. Having 3D dimensions is then a clear advantage as it basically allows to combine a 2D force-directed and a 2D layered graph layout into one single 3D layout.

Second, 3D visualizations tend to produce structures that are closer to what humans are used to in the real world. However, we cannot answer if this actually helps users to understand the structures better. Furthermore, any 3D benefits only apply in combination with an interactive graph exploration system. To come close to the effectiveness of traditional workflows, we use VR devices, but while they achieve this goal in some cases such as selecting simple settings and exploring the graph, more complex tasks are still problematic. In the same way, the use cases rather served to show which features TGView3D should support than to show where 3D really shines.

Overall, it is clear that many parts can be implemented differently; finding optimal ways of doing this has never been the purpose of this thesis. Instead, I have demonstrated methods to improve theory graph workflows by employing 3D graphics and virtual reality. Still, some of the lessons we have learned may very well be useful for similar applications that deal with data visualization and interaction in general — in the same manner that TGView3D already has and will profit from them in the future.

9.2 Future Work

TGView3D revealed many different paths that promise to be interesting topics for future research. This includes possible steps for the system itself and ideas for 3D data interaction in general.

Interactive Information Filtering The current filtering methods of TGView3D are static in the sense that the user has to activate them from the menu. An alternative would be an interactive and automatic filtering mode that makes use of the concept of encapsulating nodes. This also requires an extension of the clustering mechanisms; we can imagine the whole interaction, for example, as follows:

1. Recursively build clusters from the theory graph hierarchy.
2. Encapsulate the clusters within the parent node.
3. Automatically expand the encapsulations when the user enters its proximity.

Of course, this process includes significant efforts to guarantee seamless adaption of the layout. But I consider this the next level of reducing the information load and creating a smooth gradient from global to local knowledge.

Edge Bundling Many of the graphs we have seen contain huge numbers of edges. When looking at the graphs at a whole, individual edges become less relevant as we can no longer make out their paths within the graph. In such cases, **edge bundling** [HW09] can significantly tidy up the layout. The different edge types of theory graphs introduce an interesting component to this technique, i.e., we can bundle the edges based on their type. Note that this makes it even harder to make out paths of individual edges as we can see in Figure 9.1. Counteracting this with specific interactions is another interesting aspect. In this context, it will also be important to further investigate different clustering methods.

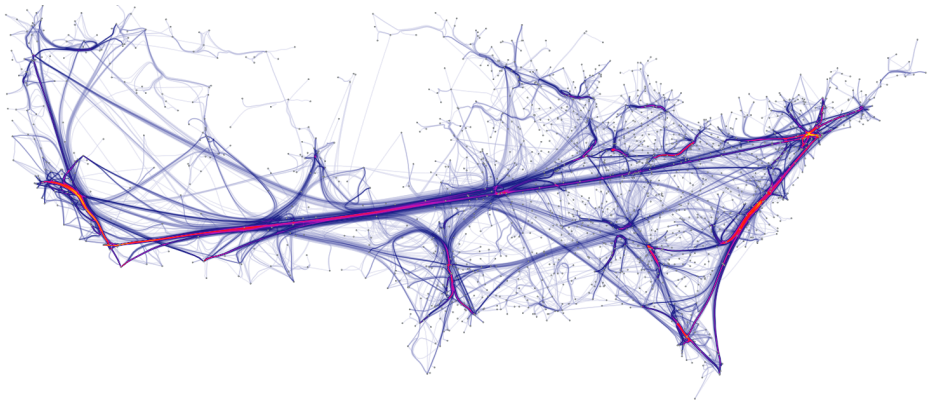


Figure 9.1: Holten et al. 2009: Graph with edge bundling.

Augmented Reality Recently, the second revision of the Microsoft Hololens [MHL2] has released. While this technology is not mature enough yet, it gives a glimpse of what is possible. In contrast to VR devices, they do not isolate the user from the environment, which allows more natural ways of complementing existing systems. Furthermore, the Hololens employs AI based gesture recognition to directly register interactions with the physical hands. There is also research that tries to implement this for VR controllers [Yan+18] and, at the same time, manufactures investigate new possibilities of user input methods. Experimenting with these will provide interesting insights.

Evaluation So far, I have only informally evaluated TGView3D and gathered feedback from demos. Now that the necessary features to perform various user tasks are implemented, it is possible to design formal evaluations that compare TGView3D against existing systems, especially in respect to how VR interactions hold up against traditional methods or even Augmented Reality. This will be a very important step to plausibly prove advantages of VR systems as well as for finding areas where the implementation or the device itself is still lacking.

It might also be interesting to collaborate with researchers that generally investigate the cognitive effects of virtual worlds. They often have to build systems only for the purpose of evaluation, whereas a system based on TGView3D could already serve as a foundation for such studies.

Furthermore, it would be helpful to perform a detailed comparison of the layout algorithm of TGView3D against the current state-of-the-art. This includes traditional hierarchical 2D systems and force-based variants that can achieve this. The latter, in particular, have not sufficiently been explored for use in 3D systems yet and thus would be the primary competitor.

Extending TGView3D Many of the features and algorithms in TGView3D are not optimized for use in production or offer room for optimization. Changing this will at least be important to improve the experience for potential users. Also, since not all of the described use cases are supported sufficiently yet, it will definitely be a task to improve this. Now that the general state of TGView3D is rather refined, even more workflows are emerging. But there are also applications beyond mathematics and computer science such as biological or medical networks. Some may have hierarchic structures similar to theory graphs or require interactions that TGView3D already supports. Others may simply benefit from the various exploration methods within TGView3D.

General Framework for VR Data Visualization In an environment with varying and changing workflows, it is not only important to have an common data representation such as theory graphs, but also give users the option to easily integrate their workflow specific interactions. We already have the option to include an URI to allow interoperation with external systems, but this may not always be sufficient, especially, for users and developers that are interested in 3D data visualization apart from theory graphs.

The first step to support this is the careful modularization of the system in order to ease the process of modifying the software. This mainly would allow to extend current functionalities

or add new features.

However, it be very helpful to also have a general framework for 3D and VR data visualization that offers greater flexibility than existing 3D systems like Gephi and notably faster prototyping compared to low-level APIs or even game engines such as Unity.

Hybrid System TGView already implements this concept to some degree in the form of builds that do rely on VR hardware, but we can extend this considerably. By giving users the option to switch to VR interactions on the fly, they can use the application according to their preferences and use the advantages of different devices to complement each other. Of course, this goes hand in hand with also switching between 2D and 3D. On the framework level, we can then also think about the way we render the graphs. For example, for static images of graphs, SVG representations are the preferred format. In this sense, it would be helpful to present the graph in different ways, based on the same data structures.

Outlook Even though VR devices are getting more and more accepted by consumers and researchers, we still have to consider them as niche products. Yet, technology is progressing rapidly. Device specifications and graphics hardware are improving every year, while, at the same time, they are becoming more affordable.

However, we have the dilemma of chicken or the egg. As long as there is no quality software, users are not inclined to buy these systems. Developers, in turn, expend no efforts to create such software because there are not enough users that already own VR devices. Especially for production or research environments, where people are used to traditional workflows and have worked with 2D displays for many years, we need to deliver convincing arguments to draw their attention.

In this sense, our first task is to find use cases that live from the advantages of VR devices and proof this extensively. Furthermore, we need to build hybrid systems that give users the best of both worlds as well as the choice of how they want to operate these systems.

I can imagine a future, where VR and especially AR systems have replaced 2D displays and are as common as smartphones are today. Then, we no would no longer argument how workflows benefit from VR or AR systems — but instead, find reasons why they do not.

Bibliography

[BCG11] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. "An introduction to argumentation semantics". In: **The knowledge engineering review** 26.4 (2011), pp. 365–410.

[BGG18] P. Baroni, D. Gabbay, and M. Giacomin. **Handbook of Formal Argumentation**. College Publications, 2018. URL: https://books.google.de/books?id=_OnTswEACAAJ.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. "Gephi: an open source software for exploring and manipulating networks". In: **Third international AAAI conference on weblogs and social media**. 2009.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3 data-driven documents". In: **IEEE transactions on visualization and computer graphics** 17.12 (2011), pp. 2301–2309.

[Bra01] Ulrik Brandes. "A faster algorithm for betweenness centrality". In: **Journal of mathematical sociology** 25.2 (2001), pp. 163–177.

[Bry94] Steve Bryson. "Virtual reality in scientific visualization". In: (1994).

[CEG18] Nicola Capece, Ugo Erra, and Jari Grippa. "GraphVR: A Virtual Reality Tool for the Exploration of Graphs with HTC Vive System". In: July 2018, pp. 448–453. DOI: 10.1109/iV.2018.00084.

[CHK04] L. Carmel, D. Harel, and Y. Koren. "Combining hierarchy and energy for drawing directed graphs". In: **IEEE Transactions on Visualization and Computer Graphics** 10.1 (2004), pp. 46–57. DOI: 10.1109/TVCG.2004.1260757.

[CNSD93] Carolina Cruz-Neira, Daniel J Sandin, and Thomas A DeFanti. "Surround-screen projection-based virtual reality: the design and implementation of the CAVE". In: **Proceedings of the 20th annual conference on Computer graphics and interactive techniques**. Citeseer. 1993, pp. 135–142.

[Coe19] Claudio Sacerdoti Coen. "A Coq plugin to export its libraries to XML". In: **Submitted to CICM** (2019).

[Con+] Andrea Condoluci et al. "Relational Data Across Mathematical Libraries". submitted to CICM 2019. URL: https://kwarc.info/kohlhase/submit/cicm19-ulo.pdf.

[Coq] **The Coq Proof Assistant**. URL: http://coq.inria.fr/.

[CVG17] Federico Cerutti, Mauro Vallati, and Massimiliano Giacomin. "An Efficient Java-Based Solver for Abstract Argumentation Frameworks: jArgSemSAT". In: **International Journal on Artificial Intelligence Tools** 26.02 (2017), p. 1750002. DOI: 10.1142/S0218213017500026. eprint: https://doi.org/10.1142/S0218213017500026.

[D3] **D3: Data-Driven Documents**. URL: https://d3js.org/.

[DB+94] Giuseppe Di Battista et al. "Algorithms for drawing graphs: an annotated bibliography". In: **Computational Geometry** 4.5 (1994), pp. 235–282.

[Deh+16] Paul-Olivier Dehaye et al. "Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach". In: **Intelligent Computer Mathematics 2016**. Ed. by Michael Kohlhase et al. LNAI 9791. Springer, 2016. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/CICM2016/published.pdf.

[DK05] T. Dwyer and Y. Koren. "Dig-CoLa: directed graph layout through constrained energy minimization". In: **IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.** 2005, pp. 65–72. DOI: 10.1109/INFVIS.2005.1532130.

[DKL13] Jan Wilken Dörrie, Michael Kohlhase, and Lars Linsen. "OpenMathMap: Accessing Math via Interactive Maps". In: **Contemporary Issues in Mathematical Publishing, JMM San Diego Special Session**. Ed. by Klaus Kaiser, Steven Krantz, and Bernd Wegner. EMIS, 2013, pp. 81–98. URL: http://kwarc.info/kohlhase/papers/cimp13.pdf.

[DKM06] Tim Dwyer, Yehuda Koren, and Kim Marriott. "IPSep-CoLa: An Incremental Procedure for Separation Constraint Layout of Graphs". In: **IEEE Transactions on Visualization and Computer Graphics** 12.5 (Sept. 2006), pp. 821–828. DOI: 10.1109/TVCG.2006.156.

[Dun95] P.M. Dung. "On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games". In: **Artificial Intelligence** 77.2 (1995), pp. 321–358.

[Ead84] P. Eades. "A heuristic for graph drawing". In: **Congressus Numerantium** 42 (1984), pp. 149–160.

[Elb+18] Mohammed S Elbamby et al. "Toward low-latency and ultra-reliable virtual reality". In: **IEEE Network** 32.2 (2018), pp. 78–84.

[Ell+01] John Ellson et al. "Graphviz—open source graph drawing tools". In: **International Symposium on Graph Drawing**. Springer. 2001, pp. 483–484.

[EMP18] Ugo Erra, Delfina Malandrino, and Luca Pepe. "Virtual Reality Interfaces for Interacting with Three-Dimensional Graphs". In: **International Journal of Human-Computer Interaction** (Jan. 2018), pp. 1–14. DOI: 10.1080/10447318.2018.1429061.

[Est+96] Martin Ester et al. "A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: **Proceedings of the Second International Conference on Knowledge Discovery and Data Mining**. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231. URL: http://dl.acm.org/citation.cfm?id=3001460.3001507.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: **Software: Practice and Experience** 21.11 (1991), pp. 1129–1164. DOI: 10.1002/spe.4380211102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211102.

[FT11] Laurent Fuchs and Laurent Théry. "A Formalization of Grassmann-cayley Algebra in COQ and Its Application to Theorem Proving in Projective Geometry". In: **Proceedings of the 8th International Conference on Automated Deduction in Geometry**. ADG'10. Munich, Germany: Springer-Verlag, 2011, pp. 51–67. DOI: 10.1007/978-3-642-25070-5_3.

[GEP] **Gephi, The Open Graph Viz Platform**. URL: https://gephi.org/.

[Gra] **Graphviz – Graph Visualization Software**. URL: http://www.graphviz.org/.

[HW09] Danny Holten and Jarke J. van Wijk. "Force-directed Edge Bundling for Graph Visualization". In: **Proceedings of the 11th Eurographics / IEEE - VGTC Conference on Visualization**. EuroVis'09. Berlin, Germany: The Eurographs Association & John Wiley & Sons, Ltd., 2009, pp. 983–998. DOI: 10.1111/j.1467-8659.2009.01450.x.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. "Optimization by simulated annealing". In: **science** 220.4598 (1983), pp. 671–680.

[KHR] **Khronos Main Page**. URL: https://www.khronos.org (visited on 03/07/2019).

[KK+89] Tomihisa Kamada, Satoru Kawai, et al. "An algorithm for drawing general undirected graphs". In: **Information processing letters** 31.1 (1989), pp. 7–15.

[Koh+17] Michael Kohlhase et al. "Mathematical models as research data via flexiformal theory graphs". In: **Intelligent Computer Mathematics (CICM) 2017**. Ed. by Herman Geuvers et al. LNAI 10383. Springer, 2017. DOI: 10.1007/978-3-319-62075-6.

[Koh06] Michael Kohlhase. **OMDoc – An open markup format for mathematical documents [Version 1.2]**. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[Koh14] Michael Kohlhase. "Mathematical Knowledge Management: Transcending the One-Brain-Barrier with Theory Graphs". In: **EMS Newsletter** (June 2014), pp. 22–27. URL: https://kwarc.info/people/mkohlhase/papers/ems13.pdf.

[Kop+18] Thomas Koprucki et al. "Model pathway diagrams for the representation of mathematical models". In: **Journal of Optical and Quantum Electronics** 50.2 (2018), p. 70. DOI: 10.1007/s11082-018-1321-7.

[KPV18] Eric Krokos, Catherine Plaisant, and Amitabh Varshney. "Virtual memory palaces: immersion aids recall". In: **Virtual Reality** (May 2018). DOI: 10.1007/s10055-018-0346-3.

[KR16] Michael Kohlhase and Florian Rabe. "QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge". In: **Journal of Formalized Reasoning** 9.1 (2016), pp. 201–234. URL: http://jfr.unibo.it/article/download/4570/5733.

[KRMT] **KRMT Lecture sources**. URL: https://gl.mathhub.info/Teaching/KRMT/tree/master/source/tutorial.

[Mata] **MathJax: Beautiful Math in all Browsers**. URL: http://mathjax.com (visited on 09/27/2010).

[Matb] **W3C Math Home**. URL: http://www.w3.org/Math/ (visited on 01/2009).

[MH] **MathHub.info: Active Mathematics**. URL: http://mathhub.info (visited on 01/28/2014).

[MHL2] **The Microsoft Hololens 2**. URL: https://www.microsoft.com/en-us/hololens/hardware (visited on 04/01/2019).

[Mi+16] Peng Mi et al. "Interactive graph layout of a million nodes". In: **Informatics**. Vol. 3. 4. Multidisciplinary Digital Publishing Institute. 2016, p. 23.

[Mil13] Justin J Miller. "Graph database applications and concepts with Neo4j". In: **Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA**. Vol. 2324. S 36. 2013.

[MitMO] **MitM Ontology Documentation**. URL: https://mathhub.info/library/group?id=MitM (visited on 04/02/2019).

[Mmt] **MathHub PVS/NASA Git Repository**. URL: http://gl.mathhub.info/PVS/NASA (visited on 05/15/2015).

[MMT] **MMT – Language and System for the Uniform Representation of Knowledge**. project web site. URL: https://uniformal.github.io/ (visited on 01/15/2019).

[MMTL] **MMT Libraries Documentation**. URL: https://mathhub.info/library/group?id=MMT (visited on 04/02/2019).

[MRS19] Dennis Müller, Florian Rabe, and Claudio Sacerdoti Coen. "The Coq Library as a Theory Graph". 2019.

[Mül+17] Dennis Müller et al. "Alignment-based Translations Across Formal Systems Using Interface Theories". In: **Fifth Workshop on Proof eXchange for Theorem Proving - PxTP 2017**. 2017. URL: http://jazzpirate.com/Math/AlignmentTranslation.pdf.

[N4J] **Neo4J, The Internet-Scale Graph Platform**. URL: https://neo4j.com/.

[N4JV] **Neo4J Graph Visualization Tools**. URL: https://neo4j.com/graph-visualization-neo4j/.

[OCR] **The Oculus Rift**. URL: https://www.oculus.com/rift (visited on 03/07/2019).

[OCTC] **Oculus Input API**. URL: https://developer.oculus.com/documentation/unity/latest/concepts/unity-ovrinput/ (visited on 02/02/2019).

[ODKA] **The archives of the OpenDreamKit EU Project**. URL: https://gl.mathhub.info/ODK (visited on 04/02/2019).

[Omd] **The OMDoc Project**. Project Homepage. URL: http://omdoc.org.

[OXR] **Khronos OpenXR Page**. URL: https://www.khronos.org/openxr (visited on 03/07/2019).

[PK15] Naomi Pentrel and Michael Kohlhase. "Relational Presentations Using Semantic Closeness Spatial Narrative for Mathematical Content". In: **Mathematical User Interfaces Workshop**. Ed. by Andrea Kohlhase and Paul Libbrecht. July 2015. URL: http://www.cermat.org/events/MathUI/15/proceedings/Pentrel-Kohlhase_Semantic_Closness-MathUI_15.pdf.

[POGO] **Pokemon Go Website**. URL: https://www.pokemongo.com.

[QWI90] John Q. Walker II. "A Node-positioning Algorithm for General Trees." In: **Softw., Pract. Exper.** 20 (July 1990), pp. 685–705. DOI: 10.1002/spe.4380200705.

[Rab13] Florian Rabe. "The MMT API: A Generic MKM System". In: **Intelligent Computer Mathematics**. Ed. by Jacques Carette et al. Lecture Notes in Computer Science 7961. Springer, 2013, pp. 339–343. DOI: 10.1007/978-3-642-39320-4.

[REQ] **Oculus Rift System Requirements**. URL: https://support.oculus.com/248749509016567 (visited on 03/07/2019).

[RK13] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: **Information & Computation** 0.230 (2013), pp. 1–54. URL: http://kwarc.info/frabe/Research/mmt.pdf.

[RKM17] Marcel Rupprecht, Michael Kohlhase, and Dennis Müller. "A Flexible, Interactive Theory-Graph Viewer". In: **MathUI 2017: The 12th Workshop on Mathematical User Interfaces**. Ed. by Andrea Kohlhase and Marco Pollanen. 2017. URL: http://kwarc.info/kohlhase/papers/mathui17-tgview.pdf.

[Sch07] Satu Elisa Schaeffer. "Graph clustering". In: **Computer science review** 1.1 (2007), pp. 27–64.

[SM94] Kozo Sugiyama and Kazuo Misue. "A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm". In: **Graph Drawing**. 1994.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. "Methods for Visual Understanding of Hierarchical System Structures". In: **IEEE Transactions on Systems, Man, and Cybernetics** 11.2 (1981), pp. 109–125. DOI: 10.1109/TSMC.1981.4308636.

[TRN] **Texas road network dataset**. URL: https://snap.stanford.edu/data/roadNet-TX.html (visited on 02/02/2019).

[Tut63] W. T. Tutte. “How to Draw a Graph”. In: **Proceedings of the London Mathematical Society** s3-13.1 (Jan. 1963), pp. 743–767. DOI: 10.1112/plms/s3-13.1.743. eprint: http://oup.prod.sis.lan/plms/article-pdf/s3-13/1/743/4385170/s3-13-1-743.pdf.

[UMP] **Information about platforms supported by Unity**. URL: https://unity3d.com/de/unity/features/multiplatform (visited on 03/07/2019).

[VJS] **vis.js - A dynamic, browser based visualization library**. URL: http://visjs.org (visited on 06/04/2017).

[VRDTM] **VR Desktop Mirror Github Page**. URL: https://github.com/Clodo76/vr-desktop-mirror (visited on 03/22/2019).

[VRTK] **VRTK Github**. URL: https://github.com/ExtendRealityLtd/VRTK (visited on 03/31/2019).

[VRUI] **Unity VR Interfaces Tutorial**. URL: https://unity3d.com/de/learn/tutorials/topics/virtual-reality/user-interfaces-vr (visited on 03/17/2019).

[WGL] **Khronos WebGL Page**. URL: https://www.khronos.org/webgl (visited on 03/07/2019).

[WVR] **WebVR specification**. URL: https://webvr.info/developers (visited on 03/07/2019).

[Yan+06] Yuting Yang et al. “Hierarchical Visualization of Metabolic Networks Using Virtual Reality”. In: **Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications**. VRCIA ’06. New York, NY, USA: ACM, 2006, pp. 377–381. DOI: 10.1145/1128923.1128992.

[Yan+18] Jiachen Yang et al. “Interaction with Three-Dimensional Gesture and Character Input in Virtual Reality: Recognizing Gestures in Different Directions and Improving User Input”. In: **IEEE Consumer Electronics Magazine** 7 (Mar. 2018), pp. 64–72. DOI: 10.1109/MCE.2017.2776500.

[ZCY09] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. “Graph clustering based on structural/attribute similarities”. In: **Proceedings of the VLDB Endowment** 2.1 (2009), pp. 718–729.

[Kwo+16] O. Kwon et al. “A Study of Layout, Rendering, and Interaction Methods for Immersive Graph Visualization”. In: **IEEE Transactions on Visualization and Computer Graphics** 22.7 (2016), pp. 1802–1815. DOI: 10.1109/TVCG.2016.2520921.

Appendix A

MathML and MathJax Setup

For the communication with MathJax we use the server from https://www.npmjs.com/package/mathjax-server without any adaptions. We can request SVG files that are created from the supplied MathML expressions. In the following, we find the Unity configuration to access this server locally and parts of the process that converts the SVGs to meshes.

```
...

private void CreateMathObject(int i, string svg)
{
    //instantiate default MathML Object for node i
    GameObject mathObject =
       (GameObject)Instantiate(Resources.Load("mathObject"));

    //convert svg to Unity mesh
    ImportSVG.ImportAsMesh(svg, ref mathObject);

    //correctly set parent and position the text mesh
    ...
}

class PData
{
    public string format = "MathML";
    public string math = "";
    public bool svg = true;
    public bool mml = false;
    public bool png = false;
    public bool speakText = true;
    public string speakRuleset = "mathspeak";
    public string speakStyle = "default";
```

```
25      public int ex = 6;
26      public int width = 1000000;
27      public bool linebreaks = false;
28  };
29
30  //requests conversion of MathML to SVG from local MathJax Server
31  private IEnumerator SVGRequest(PData pdata, int i)
32  {
33      string formData = JsonUtility.ToJson(pdata);
34      var data = System.Text.Encoding.UTF8.GetBytes(formData);
35
36      var www = new UnityWebRequest("http://localhost:8003");
37      www.method = "POST";
38      www.uploadHandler = new UploadHandlerRaw(data);
39      www.downloadHandler = new DownloadHandlerBuffer();
40
41      yield return www.SendWebRequest();
42
43      if (www.isNetworkError || www.isHttpError)
44      {
45          Debug.Log(www.error);
46      }
47      else
48      {
49          //replace certain symbols that are not supported by the
               Unity SVG Importer
50          string svg = www.downloadHandler.text.
51          Replace("ex\"", "px\"").
52          Replace("Infinity", "0").
53          Replace("currentColor", "white");
54
55          CreateMathObject(i, svg);
56      }
57  }
58
59  ...
```

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Erlangen, den 24. April 2019

(Richard Marcus)