Authoring conceptual diagrams by codifying visual representations

Wode “Nimo” Ni 倪沃德

CMU-S3D-24-XXX

September 13, 2024

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Kenneth Koedinger and Joshua Sunshine, Carnegie Mellon University, Co-chairs
Brad Myers, Carnegie Mellon University
Titus Barik, Apple
Shriram Krishnamurthi, Brown University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

To the pool gods.

Abstract

Visual representations like diagrams are powerful tools for thought. Diagrams are used extensively to understand abstract relationships, explain complex ideas, and solve difficult problems.

I conducted an interview study to understand how domain experts create diagrams and identified key limitations in current tools. To illustrate concepts effectively, experts find appropriate visual representations and translate concepts into concrete shapes. This translation step is not supported explicitly by existing diagramming tools. Our participants reported how they create, adapt, and reuse visual representations using both sketches and digital tools. However, they had trouble using digital tools to transition from sketches and reuse components from earlier diagrams. Based on these results, we suggest four opportunities of diagramming tools—exploration support, representation salience, live engagement, and vocabulary correspondence—that together enable a *natural* diagramming experience.

The findings from these studies informed the design of PENROSE, a language-based system that allows authors to codify domain-specific concepts and their visual representations. In PENROSE, the visual representation is user-defined in a constraint-based specification language; diagrams are then generated automatically via constrained numerical optimization. The system is designed to be user-extensible to many domains. In contrast to tools that specify diagrams via direct manipulation or low-level graphics programming, PENROSE enables rapid creation and exploration of diagrams that faithfully preserve the underlying visual representation. I demonstrate the effectiveness and generality of the system by showing how it can be used to illustrate a diverse set of concepts from various domains.

Atop PENROSE, I built EDGEWORTH, a tool designed to help educators easily create visual problems. EDGEWORTH works in two main ways: firstly, it takes a single diagram from the user and systematically alters it to produce many variations, which the educator can then choose from to create multiple problems. Secondly, it automates the layout of diagrams, ensuring consistent high quality without the need for manual adjustments. I collected a dataset of diagrammatic multiple-choice problems to show that EDGEWORTH can create problems in three domains: geometry, chemistry, and discrete math. EDGEWORTH generated usable answer options within the first 10 diagram variations in 87% of authored problems. I then performed a user study to measure authors' efficiency at creating translation problems using EDGEWORTH, compared with a conventional drawing tool. The results show that once authors make a correct diagram, they are about 3 times faster at making diagrammatic options for translation problems using EDGEWORTH compared to Google Drawings. Finally, in response to walkthrough demonstrations, expert educators gave positive feedback on EDGEWORTH's utility and the real-world applicability of its outputs.

PENROSE and EDGEWORTH demonstrate that codifying visual representations allow diagrams authors to reuse their design effort, produce new diagrams faster, and thus make diagrams at a larger scale.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Thesis Overview and Research Questions . . . 2
- 1.3 Thesis Outline . . . 3

2 Background and Related Work **5**
- 2.1 Diagrams . . . 5
- 2.2 Learning how to use diagrams . . . 6
  - 2.2.1 Representational fluency and contrasting cases . . . 7
  - 2.2.2 Multiplicity of examples . . . 7
- 2.3 Digital diagramming tools . . . 8
- 2.4 Tools for Problem Generation . . . 9

3 Understanding the Diagramming Process **11**
- 3.1 Introduction . . . 11
- 3.2 Method . . . 13
- 3.3 Results . . . 14
  - 3.3.1 Representation finding . . . 14
  - 3.3.2 Choosing the right tools . . . 16
  - 3.3.3 Reusing elements from earlier diagrams . . . 18
- 3.4 Implications: Natural Diagramming . . . 19
  - 3.4.1 Exploration Support . . . 20
  - 3.4.2 Representation Salience . . . 21
  - 3.4.3 Live Engagement . . . 22
  - 3.4.4 Vocabulary Correspondence . . . 23
- 3.5 Conclusion and Future Work . . . 24

4 PENROSE: From Notations to Beautiful Diagrams **26**
- 4.1 Introduction . . . 27
- 4.2 System Design . . . 27
  - 4.2.1 Language-Based Specification . . . 29
  - 4.2.2 Optimization-Based Synthesis . . . 32
  - 4.2.3 Plugins . . . 33
- 4.3 Language Framework . . . 34

4.3.1 The DOMAIN Schema . . . 34
4.3.2 The SUBSTANCE Language . . . 35
4.3.3 The STYLE language . . . 36
4.4 Layout engine . . . 38
4.4.1 Compiler . . . 39
4.4.2 Solver . . . 40
4.4.3 Plugins . . . 41
4.4.4 Rendering . . . 42
4.4.5 Development Environment . . . 42
4.4.6 Implementation . . . 43
4.5 Examples and Evaluation . . . 43
4.5.1 Sets . . . 43
4.5.2 Functions . . . 44
4.5.3 Geometry . . . 44
4.5.4 Linear Algebra . . . 49
4.5.5 Meshes . . . 50
4.5.6 Ray Tracing . . . 54
4.5.7 Large-Scale Diagram Generation . . . 55
4.5.8 Performance Evaluation . . . 55
4.6 Discussion and Future Work . . . 56

5 EDGEWORTH: Diagrammatic Problem Authoring at Scale 58
5.1 Introduction . . . 58
5.2 Formative interview . . . 60
5.3 System Design of EDGEWORTH . . . 61
5.3.1 Author Workflow . . . 61
5.3.2 Diagram Notation and Layout . . . 63
5.3.3 Program Mutation . . . 64
5.4 Limitations . . . 65
5.4.1 Domains of instruction . . . 65
5.4.2 Numerical and textual variations . . . 65
5.4.3 Usability of UI components . . . 66
5.5 Translation Problem Dataset . . . 66
5.5.1 Summary Statistics . . . 66
5.5.2 Euclidean Geometry . . . 66
5.5.3 General Chemistry: Lewis Structures . . . 70
5.5.4 Discrete Math: Graphs . . . 70
5.6 Summary . . . 71

6 Evaluating EDGEWORTH 72
6.1 Reliability Evaluation (RQ3.1) . . . 73
6.1.1 Methods . . . 73
6.1.2 Results . . . 73
6.2 Experimental Evaluation of Authoring Efficiency (RQ3.2) . . . 74

6.2.1 Study Design . . . 74
6.2.2 Results . . . 77
6.2.3 Discussion . . . 80
6.3 Expert Walkthrough Demonstration and Feedback (RQ3.3) . . . 80
6.3.1 Participants and Procedure . . . 81
6.3.2 Ecological Validity of Generated Problems . . . 82
6.3.3 Expert Feedback . . . 82
6.4 Limitations of the Studies . . . 84
6.4.1 Ecological Validity . . . 84
6.4.2 Tool . . . 84
6.4.3 Authoring Speed vs. Problem Quality . . . 84
6.5 Summary . . . 86

7 Conclusion and Future Work **87**
7.1 Summary of contributions . . . 87
7.2 Future work . . . 87
7.2.1 Composable visual representations . . . 88
7.2.2 Knowledge-infused problem variation . . . 88
7.2.3 Interactive diagrams . . . 91
7.3 Concluding remarks . . . 92

Bibliography **93**

List of Figures

2.1 Several examples of ancient diagrams, from left to right: (1) Phases of the Moon: Abu Rayhan Muhammad ibn Ahmad al-Biruni (Iranian, 973-1048), (2) Babylonian clay tablet diagramming an approximation of $\sqrt{2}$ (1900 -1700 BCE), and (3) Geometric proof of the Pythagorean theorem in Zhoubi Suanjing 周髀算经 (1st century BCE). 5

3.1 Diagrams explain concepts visually in many domains, e.g.,: (a) Diffie-Hellman key exchange with colors representing prime multiplication [175]. (b) Linking two views of the Klein 4-group [181]. (c) Unrolling a recurrent LSTM network [129]. (d) Natural numbers as 2D areas in a visual proof [71]. 11

3.2 A good visual representation (b) of forces on a truck (a) is easily understandable, whereas (c) loses essential information and (d) is non-standard and harder to understand. 15

3.3 P9 made sketches to explain projector and camera calibration. The complexity of these sketches increased and visual representations evolve over time. *Left*: an initial sketch represents connectivity as line segments. *Right*: a later sketch represent connectivity as arrows. 16

3.4 P13 manually tracks versions of a diagram in Illustrator using multiple canvases. 19

3.5 P3 uses a cheat sheet to track frequently used style attributes. 20

3.6 P7 organizes all previously made TikZ diagrams in a single document. 21

4.1 PENROSE is a framework for specifying how mathematical statements should be interpreted as visual diagrams. A clean separation between abstract mathematical objects and their visual representation provides new capabilities beyond existing code- or GUI-based tools. Here, for instance, the same set of statements (left) is given three different visual interpretations (right), via Euclidean, spherical, and hyperbolic geometry. 26

4.2 High-level pipeline: a compiler translates mathematical statements and a chosen visual representation into a constrained optimization problem. This problem is then solved numerically to produce one or more diagrams. 28

4.3 An optimization-based approach has myriad benefits. Here a logically inconsistent program fails gracefully, providing visual intuition for *why* the given statements cannot hold. 28

4.4 By specifying diagrams in terms of abstract relationships rather than explicit graphical directives, they are easily adapted to a wide variety of use cases. Here we use identical PENROSE code to generate ray tracing diagrams for several targets (Section 4.5.6). Though the arrangement and number of objects changes in each example, the meaning remains the same. 29
4.5 Most PENROSE users need only use the SUBSTANCE language, but can benefit from packages written by more expert DOMAIN and STYLE programmers. This is similar to the TEXecosystem, where most users only write documents, but benefit from expert-authored packages. 29
4.6 One benefit of a unified framework is that different domains are easily combined. Here, two existing packages (for meshes and set theory) were combined to illustrate that a simplicial complex *(left)* is closed with respect to taking subsets *(right)*. 30
4.7 Extensibility enables users to adopt conventions and notation *(center)* that reflect the way they naturally write mathematical prose *(left)*. Here, the resulting diagram *(right)* plays the role of the concluding statement. 31
4.8 An optimization-based approach makes it possible to jointly optimize visual attributes that are difficult to coordinate by hand. Here for instance we optimize color contrast according to the spatial proximity of adjacent disks *(left to right)*, ultimately discovering a two-color solution *(far right)*. The system can also be used to debug the optimization process itself—in this case by drawing the hue of each disk as a dot on a color wheel. 32
4.9 A language-based design makes it easy to build tools on top of PENROSE that provide additional power. Here we use standard techniques from program synthesis (Section 4.5.7) to automatically enumerate how the given relationships can be realized. Generating such examples helps to see important corner cases that might be missed when drawing diagrams by hand (where perhaps the top-left diagram most easily comes to mind). 33
4.10 A DOMAIN schema specifies the building blocks available in a given mathematical domain, as well as any associated syntactic sugar. This schema (abbreviated) enumerates some basic constructs from linear algebra. 35
4.11 When used with the STYLE defined in Figure 4.12, this SUBSTANCE code (with or without syntactic sugar) produces the diagram shown at right. 35
4.12 The STYLE program defining the visual style used in Figure 4.11, *right*. Note that this STYLE program can be reused for many different SUBSTANCE programs in the same domain. 37
4.13 Pipeline view of the layout engine. Rather than a single static image, compilation yields an optimization problem that can be solved and re-solved to produce many diagrams, or (in principle) used in an interactive tool. 39
4.14 Applying the mapping defined by STYLE code to a SUBSTANCE program yields a graph that describes how to draw the diagram—here, for part of Figure 4.11. Some values are known (in blue), whereas others (in orange) depend on unknowns that must be determined via optimization. 40

4.15 The computation graph is further expanded to produce graphs representing the objective and constraint space for our optimization problem. From there, we can easily use automatic differentiation to obtain derivatives. This figure depicts part of the optimization graph for Figure 4.11. 40
4.16 Our solver can lay out diagrams even if we do not initially know how to satisfy all the constraints. Here we show several steps of optimization. 41
4.17 Our system supports integration with web-based applications. Here a PENROSE IDE provides automatic syntax highlighting and autocomplete for any user-defined domain. 42
4.18 Here, some SUBSTANCE code is used to specify set relationships. Different STYLE programs not only tweak the visual style (e.g., flat vs. shaded disks), but allow one to use a completely different visual representation (e.g., a tree showing set inclusions). `Sets.sty` above describes the flat disk style. 45
4.19 Different visual representations provide different ways of thinking about an idea. Here, the notion of injections, bijections, and surjections is illustrated in both discrete *(left)* and continuous *(right)* styles. In the former, functions with the desired properties are randomly generated by an SMT solver, allowing the user to learn from many different examples. 46
4.20 Here, abstract function composition is realized as explicit composition of functions produced via an SMT solver, illustrating the fact that the composition of an injection and a bijection is an injection. 47
4.21 The cascading design of STYLE enables one to modify a base style with relatively little code. Here the two SUBSTANCE programs from Figure 5.2 and the listing above are visualized in three different styles, all of which build on the same basic constraints and objectives. 48
4.22 Once a complex diagram has been built, it can be easily broken into pieces or stages by, e.g., commenting out lines of SUBSTANCE code. Here we illustrate steps in Euclid's proof of the Pythagorean theorem, turning Byrne's static figure *(far right)* into a progressive "comic strip." 49
4.23 Composition of mathematical statements naturally translates into composition of graphical transformations with no explicit programmer effort. Here we compose linear maps, showing addition and scaling, to illustrate the two defining properties of linear maps. 51
4.24 A language-based specification makes it easy to visually inspect data structures or assemble progressive diagrams with only minor changes to program code. Here we draw the simplicial *link* by building it up from simpler constituent operations. 52
4.25 Domain-specific notation makes it easy to explore an idea by trying out many different examples. Here several subsets of a simplicial complex are specified *(top)* to explore the definition of the "link" *(bottom)*. An external plugin generates random example meshes, further enriching exploration. 53
4.26 When drawing ray tracing diagrams by hand, it can be difficult to construct geometry that permits the desired path types. Here we jointly optimize path and scene geometry to match multiple path types simultaneously. Shown are several diagrams generated for the same program. 54

4.27 We evaluated the performance of the PENROSE compiler by running it on a large collection of programs, showing that the execution time of the compiler grows slowly as the number of selector matches increases *(left)*. To stress-test the system and collect timing information, we generated and visualized random SUBSTANCE programs of different sizes, revealing that optimization dominates the execution time *(right)*. 56

5.1 **left**: a translation problem that helps students discern the structure of linear equations (adapted from [88]). **right**: an EDGEWORTH generated problem that trains student to recognize diagram configurations [93] for triangle congruence. . 58

5.2 EDGEWORTH is a diagrammatic problem authoring tool that automatically generates diagram variations from a single diagram: the author creates an example diagram (1), then EDGEWORTH generates a myriad of diagram variations (2), from which the author selects diagrams (3) to form a diagrammatic multiple choice problem (4). 59

5.3 **The user interface of EDGEWORTH.** The author first provides a textual prompt (a) as an input scenario in SUBSTANCE notation (b). Then, clicking "Generate Variations" (e) generates the specified number of diagram variations (d) at random based on a string seed and weights on Add, Delete, or Edit mutations (c). In the diagram panel, the top-left diagram (f) corresponds to the input scenario and the rest are diagram variations generated by EDGEWORTH. The author can visually select diagrams (g) to assemble a diagrammatic multiple-choice problem (h). If needed, the author can fine-tune the mutator using "Advanced options" (i j). 62

5.4 Diagram and SUBSTANCE notation for the Lewis structure of phosgene ($COCl_2$). 63

5.5 We used EDGEWORTH to recast real-world problems as diagrammatic translation problems. Left: Determine if triangles are congruent. Middle: Identify the correct Lewis structure for hydrogen cyanide. Right: Identify graphs with Euler circuits. 67

5.6 The first ten diagram variations generated by EDGEWORTH for the problem shown in Figure 5.5 (left). 68

6.1 Tasks used in the EDGEWORTH experimental evaluation. Each participant is given a textual prompt and a correct diagram to this prompt at the beginning of each task. They are asked to first re-produce the correct diagram using the designated tool in the correct segment, and then edit this diagram to produce up to 10 incorrect diagrams to the prompt in the incorrect segment. 76

6.2 Participants were provided both Google Drawings and SUBSTANCE examples throughout the study. The SUBSTANCE code (left) was given in the EDGEWORTH tasks and a Google Drawings file that visually resembles the PENROSE output (right) was given for the Google Drawings tasks. 77

6.3 Violin plots showing the distribution of time-on-task for both correct (**Left**) and incorrect (**Right**) segments of tasks. The shape of the violins represents a smoothed approximation of the data distribution, with wider sections representing higher density. The embedded box plots within the violins show the median (white line) and inter-quartile range (thick black bar), with the whiskers (thin black lines) extending to the data range. 79
6.4 Screenshots of Google Drawings *(top)* and EDGEWORTH selections *(bottom)* of diagrams by P4 of the user study (Section 6.2). They are instances of "shortcuts" participants took when using both tools, avoiding large layout edits *(top)* in Google Drawings and selecting counterexamples seemingly at random in EDGEWORTH *(bottom)*. 85

7.1 A screenshot of the EDGEWORTH interface, after generating examples for a translation problem focusing on improper subsets. The first pool of mutants aren't suitable for this problem. 90

List of Tables

3.1 Interview participants' primary domains. 14

6.1 Distribution of diagram variation classes. 74
6.2 Participants were divided into 4 groups by the tools they used and diagramming domains of the tasks. Each row corresponds to the task sequence of one of the groups. Participants used both EDGEWORTH and Google Drawings to author problems for two prompts in chemistry or geometry (Figure 6.2). 75
6.3 Summary of Average Time, Diagram Count, and Time Per Diagram by Domain for both chemistry and geometry domains, and two segments of each task (Section 6.2.1). 78
6.4 Survey responses for chemistry and geometry tasks using EDGEWORTH and Google Drawings. Higher numbers (visualized in green hue) indicates positive responses and lower numbers (yellow and red hue) negative responses. 80
6.5 Demographics of walkthrough demonstration participants. 81

Chapter 1

Introduction

1.1 Motivation

In *Doing with images makes symbols*, Alan Kay describes the widespread use of visual representations among experts in math and physics[1]:

> Jacques Hadamard, the famous French mathematician, in the late stages of his life, decided to poll his 99 buddies, who made up together the 100 great mathematicians and physicists on the earth, and he asked them, "How do you do your thing?" They were all personal friends of his, so they wrote back depositions. Only a few, out of the hundred, claimed to use mathematical symbology at all. Quite a surprise. All of them said they did it mostly in imagery or figurative terms.[85, 52:43]

Cognitive science research corroborates Kay's insight that visual representations are powerful tools of thought. In a more controlled setting, Koedinger and Anderson [93] showed evidence of experts using an internal diagrammatic representation to skip steps when solving geometry proof problems. In "Why a Diagram is (Sometimes) Worth Ten Thousand Words", Larkin and Simon [100] theorized that effective diagrammatic representations are computationally more efficient and thus help problem-solving.

One caveat, as suggested in the "(sometimes)" in the title of [100], is that the efficacy of diagrams depends on both the diagram design and the problem-solver's ability to use diagrams. In the same talk, Kay lamented that while experts seemed to have this ability, students were not getting enough training to do the same. Kay continues:

> The sad part... is that every child in the United States is taught math and physics through this [symbolic] channel, the channel that almost no adult creative mathematician or physicist uses to do it... They use this channel to communicate, but not to do their thing. [85, 54:36]

Learning science researchers have advocated for explicitly using diagrams in the learning process to enable more robust learning [112]. They further suggested that when students work with symbols and diagrams together, they build better conceptual understanding and more flexible mental models that go beyond memorized procedures [140, 88, 174].

[1] Kay is summarizing Jacques Hadamard's book entitled *The Mathematician's Mind: The Psychology of Invention in the Mathematical Field* (1945)[69].

Diagramming complex concepts involves transforming abstract ideas into tangible illustrations. While experts have powerful visual intuitions in their heads, it's difficult to communicate them to each other and teach them to students. As William Thurston noted in "On proof and progress in mathematics":

> People have very powerful facilities for taking in information visually or kinesthetically, and thinking with their spatial sense. On the other hand, they do not have a very good built-in facility for inverse vision, that is, turning an internal spatial understanding back into a two-dimensional image. Consequently, mathematicians usually have fewer and poorer figures in their papers and books than in their heads. [162, p. 164]

Indeed, this "inverse vision" process demands both a deep understanding of the subject matter and, at present, expertise in graphical design and tools—skills that are not commonly found together. As a result, despite the demand for diagrams, the ability to create effective diagrams is limited to a small group of specialists [41]. Further, the diagrams made by these experts cannot be easily reproduced. Consequently, much of the research literature and learning materials are sparsely or poorly illustrated.

This dissertation aims to bring diagrams and visual intuition to more people by identifying the tooling challenges in diagramming and building a series of artifacts that help people author and reuse diagrams.

1.2 Thesis Overview and Research Questions

I investigated how experts create diagrams via semi-structured interviews (Chapter 3), focusing on the following research question[2]:

> **RQ1:** How do diagrammers utilize the strengths and cope with the limitations of their diagramming tools?

The interviews revealed that existing diagramming tools often require hours of low-level tweaking of geometric primitives and do not capture the core task of diagramming: representing ideas visually. Consequently, the diagrams created by existing tools don't have semantics, as they are merely a collection of pixels and geometric primitives. Others who want to build upon existing diagrams often cannot reproduce the work, because diagrams are currently delivered in low-level formats such as rasterized images and Scalable Vector Graphics (SVG). The results from the interviews led to a series of design goals for next-generation diagramming tools, one of which is *representation salience*: tools should allow authors to define visual representations for domain-specific concepts in a manageable, scalable, and reusable way.

To support representational salience, I designed a tool called PENROSE (Chapter 4). Diagrams made in PENROSE contain the *source information* of diagram design: using PENROSE, diagram authors encode domain-specific concepts and how to visually represent them in plain-text languages. PENROSE generates diagrams from this encoding through automatic layout. I demonstrate

[2] All the work presented in this proposal was carried out in collaboration with others, and to recognize this, I use "we" rather than the singular first person in the subsequent chapters.

the effectiveness and generality of the system by showing how it can be used to encode visual representations across a wide range of domains:

> **RQ2:** How effectively can PENROSE's language-based specification express a wide range of diagramming domains without requiring significant modification to the system's core design?

PENROSE has several potential audiences of users and use cases. I chose to validate its usefulness for making diagrammatic problems because:

- the learning sciences literature provides ample evidence for the use of them [140, 112, 19].
- by making diagrammatic problems using PENROSE, problem authors can inform us about the ecological validity of PENROSE-generated diagrams.
- problem authors are a concrete user group that have high demand for more diagrams and use them for social good.

Atop PENROSE, I built EDGEWORTH, a tool designed to help educators easily create visual problems (Chapter 5). EDGEWORTH works in two main ways: firstly, it takes a single diagram from the user and systematically alters it to produce many variations, which the educator can then choose from to create multiple problems. Secondly, it automates the layout of diagrams using PENROSE, ensuring consistent visual quality without the need for manual adjustments. To assess EDGEWORTH, I aim to answer the following research questions about various aspects of the system:

> **RQ3.1 Reliability:** Can EDGEWORTH reliably generate translation problems with relatively few variations required?
> **RQ3.2 Efficiency:** Comparing with a conventional drawing tool, are authors more efficient at making translation problems using EDGEWORTH?
> **RQ3.3 Ecological validity:** Do real-world instructors consider EDGEWORTH-generated translation problems to be useful?

To answer these questions, I carried out: a technical evaluation to evaluate reliability, a user study to evaluate efficiency, and expert walkthrough demonstrations to evaluate ecological validity (Chapter 6).

1.3 Thesis Outline

In Chapter 2, I first provide some historical context and discuss related work on diagram use and diagramming tools. In the rest of this dissertation, I present a body of work that is a mix of *descriptive* empirical research and *artifacts* of novel interactive tools [178]. Chapter 3 is an empirical study on existing diagramming processes and limitations of existing tools. The findings of this study drive the design and implementation of PENROSE, presented in Chapter 4. PENROSE's design responds directly to the limitations identified in current practices, aiming

to bridge the gap between abstract conceptualization and tangible representation. Therefore, the research contribution is the artifact [178] of a novel diagramming system. The subsequent chapters detail how EDGEWORTH, another artifact contribution built atop PENROSE, address the identified needs and support the process of diagrammatic problem authoring in educational settings. In Chapter 5, I present the system design of EDGEWORTH and show its expressiveness by collecting a dataset of diagrammatic translation problems in multiple domains. Chapter 6 describes a series of evaluative studies of EDGEWORTH. In Chapter 7, I assess the contributions and insights developed in this dissertation and outline potential directions for future research of diagramming.

Chapter 2

Background and Related Work

Figure 2.1: Several examples of ancient diagrams, from left to right: (1) Phases of the Moon: Abu Rayhan Muhammad ibn Ahmad al-Biruni (Iranian, 973-1048), (2) Babylonian clay tablet diagramming an approximation of $\sqrt{2}$ (1900 -1700 BCE), and (3) Geometric proof of the Pythagorean theorem in Zhoubi Suanjing 周髀算经 (1st century BCE).

This chapter provides background on diagrams in general, existing diagramming tools, and research on using diagrams for learning.

2.1 Diagrams

Tversky [166, 167] defines diagrams as "*an arrangement of marks on a virtual page (stone, paper, or screen) that represents a set of ideas and their relations*". This definition is broad enough to include many ancient and modern graphical representations of ideas, including graphs, charts, infographics, and many more. Under this definition, diagrams are perhaps one of the oldest form of human communication and expression. For instance, Figure 2.1 shows a few examples of diagrams in ancient times around the world. Although there are many more definitions of diagrams (for example, [14, 31, 72, 6, 165]), this chapter does not aim to provide a comprehensive discussion of how diagrams are defined. Rather, we examine two aspects of a diagram to motivate the types of graphics this dissertation focuses on: content and utility.

First, this dissertation contribute tools (Chapters 4 and 5) that produce diagrams that depict logical, non-quantitative *concepts*, rather than quantitative *data*. As we will discuss in Section 2.3 and Chapter 3, this focus on conceptual diagrams is largely driven by the relative dearth of tools

for making conceptual diagrams. In contrast, statistical graphics [176] largely depict quantitative data, and are well supported by authoring tools [22, 150].

The second aspect is the utility of diagrams, which set them apart from decorative paintings, photos, floor plans, and more. Ervin [52] distinguishes between *pictorial* and *propositional* graphics: instead of directly visualizing data or depicting naturalistic scenes, diagrams (propositional graphics in Ervin's terms) "constitute knowledge and embody media-independent abstractions for inference-making." The specific utility of diagrams for inference-making is significant enough to prompt psychologists and cognitive scientists to study their role in problem-solving and learning. Diagrams have been shown to have cognitive benefits to reasoning and problem-solving [100, 92, 112]. Compared to textual representations, diagrams facilitate fast recognition and direct inference by making the most relevant information explicit and easily findable [100]. As an external representation of abstract structures of tasks, diagrams can work together with one's mental representation and are an indispensable part for accomplishing distributed cognitive tasks [185]. Akin to Ervin [53], Hegarty and Kozhevnikov [73] distinguish between *pictorial* and *schematic* visual representations and show that schematic representations of relative spatial relationships significantly outperform pictorial ones that encode visual appearances. In addition to their values as an external, static representation of knowledge, diagrams are also beneficial when people learn *with*, instead of *from* them [163]. In educational contexts, explicit training of drawing, including the creation of new visual representations and adoption of new ones, significantly improve students' ability to work with multiple representations and improve learning, reasoning, and communication skills [1]. Moreover, creating diagrams as visual explanations also improves learning, since they can act as a check for completeness and a medium for inference [20].

Beyond diagrams' utility for more efficient cognitive inference and learning, logicians, mathematicians, and philosophers have argued for diagrams' fundamental role in reasoning. Historically, diagrams were often dismissed as mere illustrations—supplementary tools rather than central components of logical arguments. This traditional view has been predominantly influenced by the dominance of symbolic languages in the history of logic, where precision and formal rigor were prioritized over visual representation [154]. Peirce's existential graphs [136] are a form of diagrammatic logic that he argued could represent logical relations in a manner both clearer and more intuitive than traditional symbolic logic. Peirce believed that existential graphs could express logical relationships with a degree of generality and precision that rivals, if not surpasses, that of symbolic logic, particularly in their ability to represent the continuity of logical processes. Jon Barwise and John Etchemendy's work on the importance of diagrams in logical reasoning [11] aligns closely with the earlier insights of Peirce. Peirce's existential graphs, which visually represent logical propositions and their relationships, serve as a precursor to Barwise's concept of "heterogeneous reasoning," [10] where visual and symbolic methods are integrated to solve logical problems more effectively.

2.2 Learning how to use diagrams

Whether for more efficient problem-solving or logical reasoning, one must learn how to use them properly. At the end of "Why a Diagram is (Sometimes) Worth Ten Thousand Words", Larkin and Simon [100] noted that:

> [D]iagrams are useful only to those who know the appropriate computational processes for taking advantage of them. Furthermore, a problem solver often also needs the knowledge of how to construct a "good" diagram that lets him take advantage of the virtues [of diagrams] we discussed. [100, p. 99]

In this section, we provide background on why students need to practice for better fluency in visual representations and how diagram variations may help students practice using them more effectively.

2.2.1 Representational fluency and contrasting cases

Representational fluency refers to the ability to quickly understand a visual representation and to use it to solve domain-specific tasks [140]. To become representationally fluent, an important first step is to identify meaningful aspects of a particular representation. Kellman et al. [88] show that mapping between symbolic and visual representations leads to intuitions about the way equivalent structures relate to each other. The learning that results from constructing connections between symbols and diagrams can be more flexible. Students are better at transferring their learning from the problems they have explicitly practiced to more open-ended problems and their conceptual understanding is better [70].

In addition to mapping between representations, Marton [109] also showed that contrasting cases help students discern crucial parts of a particular representation. Early on, students benefit from discerning instances and noninstances that differ in only one dimension of variation. As students become more fluent, a *fusion* of multiple varying dimensions in problems may be necessary [36]. Arnheim [6] characterized this need for many diagrams (or animation of diagrams) in *Visual Thinking*:

> The usual illustrations in textbooks and on the blackboard help to make a problem visible, but they also freeze it at one phase of the range to which the proposition refers. Therefore, they tempt the student to mistake accidental circumstances for essential ones. The solution is not to leave out illustrations but either to produce mobile models… or, at least, to use immobile illustrations in such a way that the student realizes which of their dimensions are variables. [6, p. 182]

2.2.2 Multiplicity of examples

Indeed, in addition to training representational fluency, multiple examples and repeated, varied practice are well-documented strategies for broader learning goals in the learning science literature. Many studies have demonstrated substantial STEM learning benefits for multiple worked examples per topic [134]. Equally important is research indicating the importance of active learning [35, 43] and repeated practice [51, 152] that occurs within varied contexts [133, 147] and involves direct explanatory feedback [88].

Rau [141] reports that, unfortunately, providing computational support for representational fluency is time-consuming with current tools. Our formative study (Section 5.2) confirmed this claim and revealed barriers resulting from the limitations of diagram authoring tools. To address

these limitations, EDGEWORTH (Chapters 5 and 6) aims to simplify the workflow for creating diagram variations for repeated practice.

2.3 Digital diagramming tools

As this dissertation investigates diagram authoring empirically (Chapter 3) and contributes a new diagramming tool (PENROSE, Chapter 4), we survey existing digital tools for making diagrams. Although many diagramming tools support both text-based and graphical interfaces, we categorize current diagramming tools by their dominant mode of interaction: programming-language based (PL) tools and direct manipulation (DM) tools.

We use PL tools to refer to text-based diagramming tools, including imperative or declarative programming languages, libraries, frameworks, and embedded domain-specific languages. General-purpose tools such as Processing [143], Asymtote [23], PGF/TikZ, and Paper.js [1] provide program constructs that model graphical primitives and operations akin to those in Scalable Vector Graphics (SVG) [177]. Many of their shared disadvantages are well summarized in TikZ's manual [161]: "steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really "show" how things will look like." Domain-specific tools allow diagram specifications that are higher-level and specialized to the problem domain to smoothen the learning curve. They are developed either from scratch (e.g., GraphViz and the DOT language for graph visualization [48]) or on top of general-purpose tools (e.g., TikZ's extensions, `tkz-euclide` for Euclidean geometry). However, many of them still inherit the other disadvantages from above.

DM tools represent interactive diagramming tools that support WYSIWYG interfaces and direct interaction with shapes. Akin to PL tools, general-purpose DM tools such as Adobe Illustrator, Inkscape, and Figma also have similar sets of primitives, but often provide a large number of widgets or drawing tools (e.g., Illustrator CC has nearly 100 built-in tools [2]). To overcome the disadvantage of their highly manual interaction model, both Illustrator and Inkscape provide language bindings or command-line tools for automation, but they still suffer from the above problems of PL tools. Popular domain-specific diagramming tools such as draw.io and Gliffy are template editors that provide predefined, mostly box-and-arrow style shapes, limiting users to a narrow set of diagrams. Research prototypes such as Sketchpad [158] and ThingLab [21] automate diagram layout using constraint solving, but many edit actions like selection and shape construction remain manual. Other prototypes like Apparatus [3] and Bret Victor's dynamic visualization tool [26] incorporate some limited programmatic operations (e.g., macro recording, variable declaration, and computed properties) via direct interactions.

As discussed by Satyanarayan et al. [149], data visualization tools have transformed over the past decade. The major advances are characterized by three "waves": (1) improvement of individual charts' quality, (2) theories and tools that enable mass-production of visualizations, and (3) the convergence of tools [45]. Whereas the benefits of conceptual diagrams are clear and theoretical foundations exist, most of the diagramming tools are still not easily scalable and there are large gaps in existing technologies, notably between PL and DM tools. In other words, the 2nd

[1] http://paperjs.org/
[2] https://helpx.adobe.com/illustrator/user-guide.html
[3] http://aprt.us/

wave of conceptual diagramming is still not here. In the interview study presented in Chapter 3, we aim to gain a deep understanding of people's diagramming process to drive the design of tools that fill these gaps.

2.4 Tools for Problem Generation

In addition to making standalone diagrams, this dissertation also covers diagrammatic problem authoring with EDGEWORTH in Chapters 5 and 6. In this section, we cover related digital systems for practice problem generation in general, and their support for diagram authoring. Kurdi et al. [97] conduct a systematic review of automatic problem generation tools and show that the majority of tools address language learning. In this section, we focus on problem generation tools in STEM learning and discuss how they relate to diagrammatic problem generation and EDGEWORTH.

Intelligent Tutoring Systems (ITS) are automated curricula that include practice problems with personalized feedback (*inner loop*) and customize problem selection to improve students' performance (*outer loop*) [169]. Problem banks are an important component of ITS tools, so many systems have built-in authoring support to generate a large number of problems via templating. For instance, Cognitive Tutor Authoring Tools (CTAT) is an ITS authoring platform [2]. CTAT has a "Mass Production" feature that lets the user create a problem template and insert problem-specific values via a spreadsheet [3]. Similarly, the ASSISTment builder allows authors to "variabilize" numerical values in problem templates for automatic generation [142].

In the context of testing, researchers proposed systems that generate test problems (*items*) automatically for adaptive testing and cost-effectiveness [59]. Due to the need for numerous test items, automatic item generation systems also rely on templating (*item models*) to generate items [60, 76, 135]. For instance, IGOR [59, Chapter 13] has a similar approach to templating as CTAT and ASSISTment. While the templating approach is suitable for symbolic problems, they do not automate diagram generation. Authors still need to provide individual diagrams in templates in CTAT, ASSISTment, or IGOR.

In Chapter 5, we present EDGEWORTH, which complements these tools by enabling authors to automate diagram variation production. Diagrammatic problems generated by EDGEWORTH can be integrated into problem banks and managed by the outer loop of ITS for an adaptive learning experience. EDGEWORTH does not currently support template variables in the textual prompt or diagram labels. However, it is possible to parameterize the example diagram as a problem template and use existing template-based systems to generate problem variations.

Other problem generation systems employ different methods from templating. A number of systems use *program synthesis* to synthesize a program that produces many problem instances [65]. Singh et al. [156] generate algebraic equality proof problems from example problems. Weitekamp et al. [173] speed up ITS authoring in CTAT by synthesizing ITS problems from user demonstration of problem solutions. Andersen et al. [5] model procedures to solve algebra problems as imperative programs and use execution traces of these programs to generate a series of problems. Notably, Gulwani et al. [67] generate solutions to geometry drawing problems by synthesizing programs of ruler-and-compass geometry constructions from a program specification. Though not strictly a problem generation tool, the generated solutions can be illustrated diagrammatically. However, the approach in [67] is specific to the domain of geometry, whereas EDGEWORTH's approach is

domain-agnostic. Synthesis-based systems often have an advantage of a simpler user experience, since the author can provide examples and the tool automates problem generation itself. The approach of EDGEWORTH takes inspiration from these tools in that EDGEWORTH only requires the author to provide one example diagram. However, EDGEWORTH does not need to generate programs from a specification. It merely performs mutations on an example diagram.

Commonly used in human intelligence tests and as computer vision benchmarks, Figural Analogy Problems (FAPs) give a series of diagrams and ask the respondent to infer or select the next diagram given some patterns in the given diagrams [180]. Early automatic FAP generators were based on human-crafted shape composition rules [77] and cognitive models [49]. Newer systems [172, 9] encode variation rules [32] as first-order logic constraints. While FAPs are by definition highly diagrammatic, FAPs focus on pure visual reasoning, while in STEM problems often focus on mapping symbolic notations to visuals. Moreover, diagrams in STEM are much more diverse due to the multitude of disciplines, and are not limited by a few variation rules. That said, EDGEWORTH takes inspiration from FAP generators' rule-based approach. However, EDGEWORTH's mutations are domain-agnostic and operate on logical objects, not fragments of the diagram itself.

Chapter 3

Understanding the Diagramming Process[1]

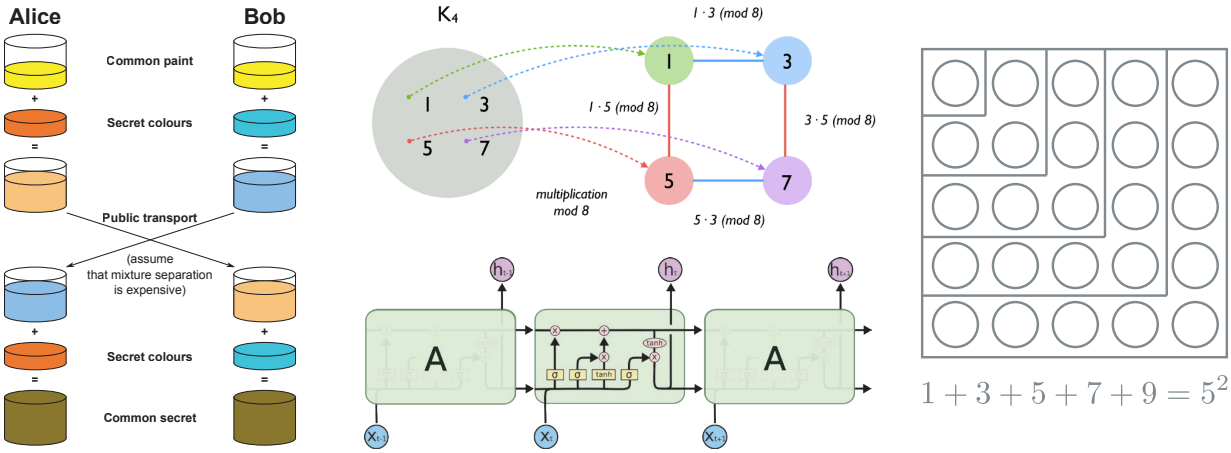


Figure 3.1: Diagrams explain concepts visually in many domains, e.g.,: (a) Diffie-Hellman key exchange with colors representing prime multiplication [175]. (b) Linking two views of the Klein 4-group [181]. (c) Unrolling a recurrent LSTM network [129]. (d) Natural numbers as 2D areas in a visual proof [71].

Chapters 1 and 2 laid the foundation by discussing the importance of diagrams in learning and problem-solving and reviewing existing tools for diagramming. This chapter aims to investigate how domain experts create diagrams, examining their methods, the tools they use, and the difficulties they encounter. This chapter presents an empirical study to highlight the gaps in current tools and suggest opportunities for developing more intuitive and effective diagramming solutions that align better with the diagramming processes of experts.

3.1 Introduction

Visual representations of knowledge allow us to understand and disseminate information more effectively than text alone [111]. This chapter focuses on conceptual diagrams, which communicate conceptual, procedural, and metacognitive knowledge [96] in visual form (Section 2.1). By

[1]This chapter is adapted from "How Domain Experts Create Conceptual Diagrams and Implications for Tool Design" [108].

giving abstract concepts visual representations, these diagrams help explain concepts to oneself and communicate them with others. Explaining concepts using visuals is profoundly important for dissemination of scientific knowledge and for learning.

While conceptual diagramming is clearly an important form of knowledge work, unfortunately, tools for creating conceptual diagrams are still limited. Current tools for diagramming stand in tension between: a) General-purpose drawing tools such as Illustrator and Figma that offer simple pen-and-canvas or box-and-arrow metaphors, but are *viscous* [63]—users must constantly commit to exact positions, sizes, and styling of shapes. b) Dedicated diagramming tools such as Lucidchart and Gliffy that allow rapid changes, but rely heavily on templates, limiting diagrammers to a fixed set of visual representations.

We argue that this relatively limited support for diagramming in tools is in part because the process of diagramming is poorly understood. For instance, often diagrammers start with informal media such as paper or whiteboards, and edit diagrams digitally before they are presented, but how do diagrammers manage the evolution of diagrams? How do diagrammers utilize the strengths and cope with the limitations of their tools? Which tools are chosen for what purposes? Such a detailed understanding of the process can help design interactive tools to support diagramming.

This chapter contributes a description of the process of creating conceptual diagrams, the difficulties people face while diagramming, and opportunities for tool design. In Section 3.2, we describe how we conducted interviews with 18 domain experts from a wide variety of disciplines such as math, computer science, architecture, and education. The findings presented in Section 3.3 reveal that diagrammers have diverse interactions with visual representations in both physical sketches and digital tools, including finding, creating, storing, and reusing representations. When diagrammers transition from sketches to digital tools, their tool selections are influenced by their *sense of control* over object placement and diagram layout. Participants were concerned with two kinds of control: local object placement, and global diagram layout. Current tools, both those that use programming languages (PL) and those that use direct manipulation (DM) as their interactive metaphor trade-off one kind of control to support the other more effectively. Consequently, we found that diagrammers invented their own set of *ad hoc* and personal reuse patterns to iterate, simplify, and automate the diagramming workflow.

One implication of our results is the opportunity to design tools informed by the processes of diagramming, and practices that domain experts already use, making digital diagramming more intuitive and efficient. In Section 3.4, we identify four key opportunities for **natural** [122] diagramming tools that allow diagrammers to express their ideas visually the same way they think about them:

- *Exploration support*: supporting exploratory behaviors such as undo and backtracking during both abstract-level, breath-first exploration of the design space and low-level refinements of visual details.
- *Representation salience*: allowing explicit creation and management of visual representations, i.e., the *mappings* from domain constructs to shapes instead of geometric primitives themselves.
- *Live engagement*: providing diagrammers with the sense of agency by designing for liveness and directness of the diagramming experience.
- *Vocabulary correspondence*: enabling diagrammers to interact with their diagrams using

vocabularies that is conventional in their domain.

For each of the opportunities, we survey existing techniques from relevant areas to provide tool designers with technical insights on how it might be implemented.

3.2 Method

Participants and Recruitment

We conducted interviews with 18 participants (13 male, 5 female). Participants were recruited through posts on social media, and our research group website. Of these, four participants were university faculty, 10 were PhD students or postdocs, one was a professional masters student, one was a K-12 math instructor, one is an independent software developer, and one is an enterprise software engineer. Prospective participants filled out a survey which allowed us to screen participants for our interviews. We selected the interviewees based on the following criteria: the interviewee (1) creates conceptual diagrams on a frequent basis and (2) uses digital diagramming tools to create these diagrams.

Because we had more potential interviewees than we originally envisioned (64 in all), we used a saturation method [15] to determine the number of participants. We conducted several batches of interviews (with 2-3 interviews per batch consisting of diverse participants), and did a preliminary analysis of the transcripts from each batch. When the analysis stopped revealing new insights, we stopped interviewing more participants.

Semi-structured Interviews

Interviews lasted between 30 and 80 minutes and were semi-structured. Interviews were conducted either in person (7 participants) or online using Skype (11 participants.) We encouraged participants to bring any digital and hand drawn diagrams that they had previously created that they could share with us. We also encouraged them to have a pen and paper (or a whiteboard) available to draw during the interview.

Four of the authors are involved in the development of PENROSE [128, 181], a new diagramming tool. Our initial interview questions were developed to inform PENROSE's design. The focus of the interview eventually broadened to participants' past experience diagramming (using the critical incident technique [57]), tool preferences, and reuse practices. The full interview protocol is included as supplementary material. Example questions from our script are: (e.g., "*What is the last diagram you made?*" and "*What is the diagram you are most proud of?*"), accompanied by appropriate follow-up questions and requests for participants to share diagrams under discussion. Table 3.1 includes all the participants categorized by the primary focus of their work.

Analysis

Interviews were video recorded and transcribed using either human or machine transcription. The first two authors then manually validated and corrected any transcription errors.

Domain	Participant
Abstract algebra	P1, P17
Category theory	P4
Discrete mathematics	P14
Computer graphics	P16, P6, P10, P3
Algorithms	P12
Topology	P7
Human-computer interaction	P13, P18
Programming language theory	P11
Software engineering	P9, P2
User interface design	P5
Architecture	P8
K-12 education	P15

Table 3.1: Interview participants' primary domains.

We employed thematic analysis methods [25] to analyze interview transcripts. The first two authors began by conducting an open coding session and discussed initial insights for every batch of interviews. Then, following all interviews, the authors discussed the codes and created a coding guide with operationalized definitions of codes. Using the agreed coding guide, one of the authors did a second phase of coding. While conducting the second coding phase, the author also summarized the transcripts using sticky notes containing highlights of the interview sessions. All authors then reviewed both the codebook with the sticky notes to further refine the set of codes.

Finally, the authors analyzed the codes by clustering lower-level codes during multiple interactive discussion sessions. Through the higher-level clusters, a few themes with high numbers of codes emerged, such as *Reuse* and *Representation*. We present these themes and the resulting insights next.

3.3 Results

In this section, we present the results from our analysis of the interview data. The section is organized in terms of the high-level themes that emerged from our analysis.

3.3.1 Representation finding

When illustrating a concept visually, a crucial step is to decide how every abstract object will be represented graphically. For example, Larkin and Simon, in their classic paper "Why a Diagram is (Sometimes) Worth Ten Thousand Words," chose to represent forces with arrows in the diagram shown in Figure 3.2b [100]. This step, which we call *representation finding*, is crucial to diagram effectiveness. If Larkin and Simon had represented forces using concentric circles with different radii instead of arrows (Figure 3.2c), the directionality of the forces would be lost. If they had represented forces with chocolate bars with different lengths (Figure 3.2d), the diagram would

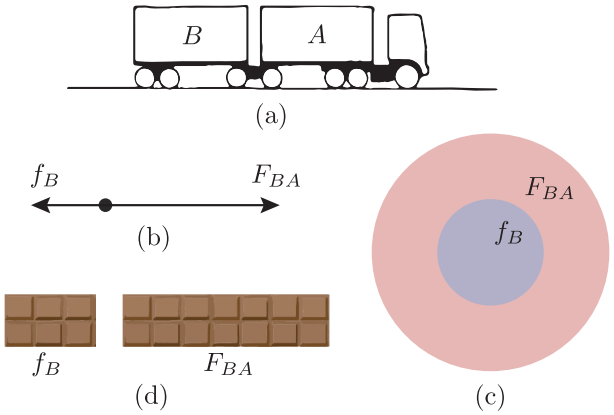


Figure 3.2: A good visual representation (b) of forces on a truck (a) is easily understandable, whereas (c) loses essential information and (d) is non-standard and harder to understand.

have been inconsistent with other physics diagrams and the extraneous detail would have distracted from the core purpose.

This process of *representation finding* usually preceded the creation of any formal or informal diagrams. Participants engaged in two representation finding activities: (1) seeking and finding existing representations from prior work and (2) creating novel representations.

Diagrammers seek existing representations from prior work

In many domains, there are well-established visual representations for abstract concepts and objects. Therefore, diagrammers tended to look at existing diagrams for representations when starting to create their representations:

> "*Sometimes I look for inspiration in other papers just to know what kinds of standard people are using. Sometimes there are some conventions that people actually use in my field like how to represent a camera for instance. So you kind of have to stick with these conventions.*" (P3)

Diagrammers generate new representations to tell new stories

Other domains lack standardized representations and diagrammers creatively *generate* their own representations:

> "*The whole purpose of those diagrams [in my book] is to make something that has never been seen before visually obvious... Why didn't anybody draw that picture before? I have been taking something that almost seems completely confusing or unimportant and having a picture that makes you know what's going on... is truly satisfying.*" (P1)

When creating diagrams for explanatory purposes, diagrammers also carefully craft visual representations to ensure that the diagram is intuitive and clear for their target audience. For instance, P8 developed new representations to reduce visual complexity:

> "*When a diagram has too many working elements, it becomes too hard for your brain to process it. If you can boil it down to two main things interacting, that will make the diagram much more intuitive to someone. It's very much about choosing the right colors, lines... putting the emphasis in the right place.*" (P8)

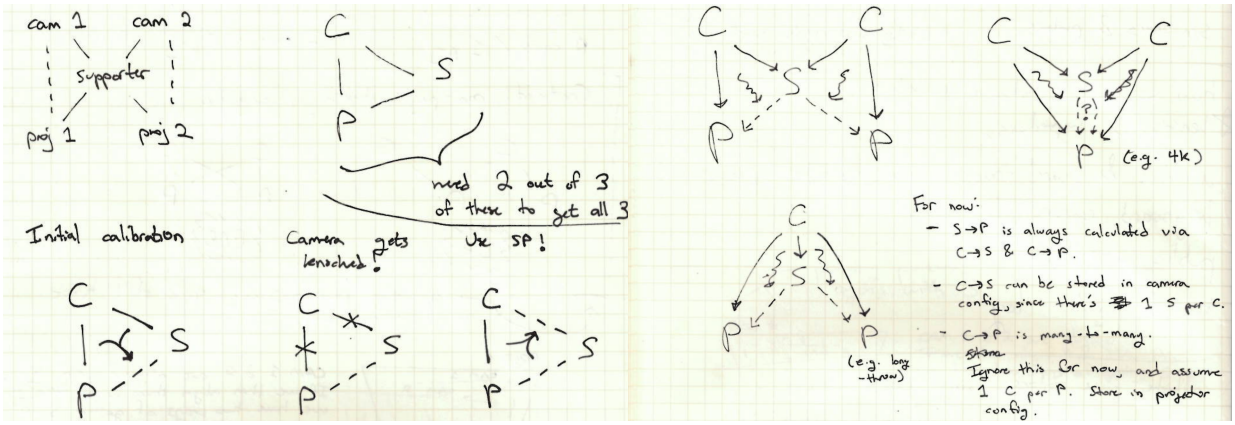


Figure 3.3: P9 made sketches to explain projector and camera calibration. The complexity of these sketches increased and visual representations evolve over time. *Left*: an initial sketch represents connectivity as line segments. *Right*: a later sketch represent connectivity as arrows.

Diagrammers use sketches to discover appropriate representations

Sketching plays an important role in generating new visual representations or choosing among existing ones. For instance, P8, P12, P5, P9, and P13 reported iterative processes of refining their visual representations as they sketch. For instance, P9 described the evolution of diagram sketches and the changes of visual representations of the design of a complex camera-supporter-projector system, also shown in Figure 3.3:

> "*At this stage, I don't even know how these machines would be connected, so there's lines, but at this later stage I was actually thinking about 'Oh, how are we going to represent these things and compute with them in practice?' So I did arrows. There's also certainly increased complexity in the beginning thing, I'm just looking at the situation of a single camera projectors supporters system. Then here on the next page I'm starting to look at different configurations of multiple cameras and projectors.*" (P9)

3.3.2 Choosing the right tools

When participants eventually chose to move to a digital medium, their choice of tool was systematic, if not conscious. Specifically, we found participants' preferred either programming-language based (PL) tools or direct-manipulation based (DM) ones. Below, we analyze the reasons for their preferences.

People choose DM tools for faster feedback and global control

DM tools were often described as "easier" and thus have lower barriers to entry when compared with PL tools. One particularly common reason for choosing DM tools was the need to place shapes in relations with other shapes, which is difficult to do without immediate visual feedback.

> "*So I like [a DM tool] because it gives me this very fine control over how things are aligned and when they're straight up and down.*" (P2)

Because of the synchronized visual preview, DM tools provide better support for *global* control over diagram layout, i.e. the relationships among graphical primitives. Diagrammers used DM tools similar to how they used pen and paper, to offload their working memory [100], but with the additional benefits provided by interactions supported by the tools:

> "*I'm trying to draw things down on the papers [or DM tools] because my head is getting crowded and I need to be able to keep track of everything on [digital] paper and be able to interact with it the same way I would in my head.*" (P7)

People choose programming languages for better abstraction and local control

Comparing to the easy *global* control provided by DM tools, PL tools make *local* control easier: they let users control *local* placements of shapes by specifying exact pixel coordinates:

> "*I want this [a shape] to be exactly a hundred pixels... because there's definitely times where I want to get this right to this point and it's hard to do that with the mouse.*" (P15)

Global layout of diagrams can be specified more precisely using PL tools, but requires more advanced programming skills and, as discussed, more time commitment:

> "*If it's something where the relationships among the things you want to specify in a precise way, then it's a lot easier, if you know how to program, to introduce a programming language where you can specify exactly the relationships you want, how you want them to change, and so forth.*" (P1)

Programming languages provide affordances to create abstractions and automate the diagramming process:

> "*Once you have made a visualization [using PL tools], if you want to tweak things about it, you can. Just put what you do into a script, add some parameters, and you could repeatedly get the same visualization with variations... It will generate the thing automatically, you don't have to create a whole picture by hand again.*" (P1)

The complexity of languages, however, incurs a higher upfront cost and steep learning curve, making more diagrammers without programming background reluctant to commit to them. Another downside of PL tools is that they often require compile-and-run cycles and hence delayed feedback:

> "*There's a long learning curve on [a PL tool] and then it's slow. It's a lot of typing and a lot of iterative–'I type something and I see what it looks like'. So there's a lot of delays in modifying [the diagram].*" (P15)

> "*I'm willing to put in the effort, but it's like 20% of the time is that, and like 80% of the time is fighting with LaTeX.*" (P11)

Advanced diagrammers use PL tools to automate their diagramming workflows

In some cases, diagrammers find the need to create families of similar diagrams for the use of, for instance, writing a textbook or a problem set. A small number of our interviewees automated their diagramming workflow extensively by leveraging the abstraction affordances of PL tools. One automation pattern is to parameterize complex diagrams and generate multiple instances with variations to explore design alternatives and populate diagram collections:

"*If I invest the time upfront to just write it, parameterized by, and then I do the diagram in terms of n and k. And then later I realize that, 'Aw, $n = 15$ and $k = 4$ is just a mess!' Okay, I go to the top of the file and I set $n = 12$ and $k = 3$ and I re-render it and it looks like this, and I go, 'That's what I want.' If I'm not sure, okay, let's try $n = 10$, try that. You can just make a new diagram in 15 seconds instead of four hours, but they also demand more time and skills up front. If I'm just going to do one diagram, it's not worth it.*" (P1)

Another pattern is creating *ad hoc*, embedded domain-specific languages that allow specification of diagrams at a higher level:

"*I have learned a style that is highly idiomatic and not something that I could teach someone else... you look at the sort of syntactic objects that you're going to work with in a certain proof theory, and you define the macros at the top level.*" (P11)

As commented by P1 and P11 above, the automation requires a high upfront cost and advanced skill sets, and only advanced diagrammers invested in the skills and time commitment to do so.

3.3.3 Reusing elements from earlier diagrams

Diagrammers backtrack frequently and informally track prior versions

Diagrammers were iterative even in the formal diagramming stage, which involves frequent backtracking behaviors. Therefore, keeping track of version history becomes an essential task for diagrammers.

In the case of DM tools, however, versioning can be challenging in many existing tools, due to the lack of textual storage formats. As a result, *ad hoc* solutions are again created to compensate for this limitation such as keeping multiple versions of the diagram on the same canvas, as illustrated in Figure 3.4 and described by P5:

"*I ... duplicate each [art board], change something about that, and take it out again. That's really helpful not only to present the overall trajectory of the process, but then you can go back and reference a previous state without having to look through the undo history and destroy all of your redos. I [like to] branch out fractally with different areas that are relevant to me.*" (P5)

In theory, standard version control systems such as `git` [164] make it easy to track versions with PL tools. However, even with their textual file formats, versioning can still be challenging for PL tools because textual representations of diagrams can be too low-level to be human readable. As a result, P6 tracks prior versions without using standard version control systems:

"*I rely either on Dropbox to store different images or I have my own custom-made back-up system that keeps hard copies of things... I have a separate script that every day pulls all of my folders and keeps copies of them if there is any change.*" (P6)

Diagrammers organize reuse libraries by representation

The most common form of library we saw diagrammers maintain was a "cheat sheet". Cheat sheets are configuration files that contain low-level parameters such as line-weight settings and hexadecimal strings of colors (P3, P6, P13). Diagrammers used cheat sheets to reduce stylistic inconsistencies and to simplify the repetitive, manual tasks in the diagramming process with cheat sheets. For instance, one participant said:

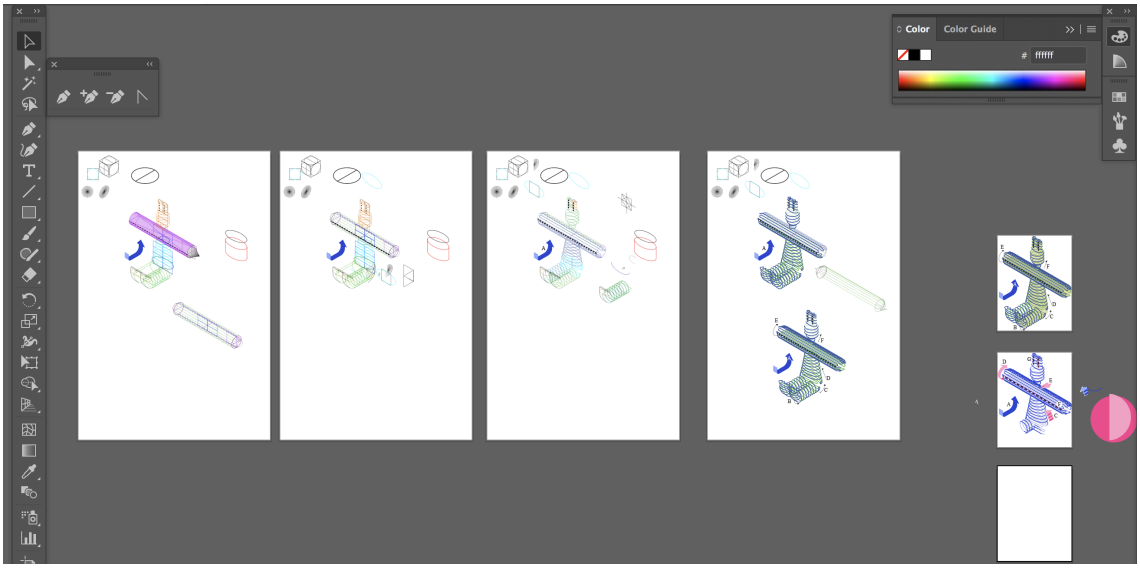


Figure 3.4: P13 manually tracks versions of a diagram in Illustrator using multiple canvases.

"*Usually I have this little* `txt` *file where I basically remember the color so I have the color codes for primary and secondary [objects]... I saved the [line] width as well for primary secondary [objects], and that's kind of like my cheat sheet that I reuse.*" (P3)

This is one area where tool support was mostly lacking. For instance, participants often took notes manually (sometimes these notes were handwritten.)

More advanced diagrammers maintained collections of existing diagrams or diagram components, organized by representation. For instance, one participant keeps a document of previous visualizations, as shown in Figure 3.6:

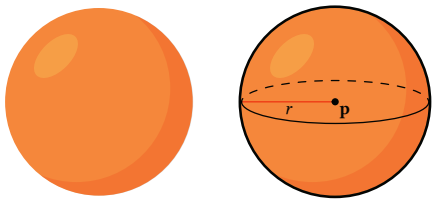
"*I keep a document, that is almost all the TikZ diagrams I've ever had to because I find that they helped me think about how to represent diagrams for new situations.*" (P7)

Another participant collects commonly used diagram components in a personal library:

"*So over the time I've settled on specific representations for the camera and the light source, I keep copies of them. I have my own small library of things.*" (P6)

3.4 Implications: Natural Diagramming

The results discussed above suggest unique strengths and weaknesses of existing PL and DM diagramming tools. This section offers some possibilities to combine the strengths of PL and DM tools and create more intuitive and efficient next-generation tools. Just as previous work advocated for creating programming tools "for people to express their ideas in the same way they think about them" [122] as an opportunity for *natural programming*, we advocate for creating diagramming tools for **natural diagramming**. Natural diagramming presents four distinct opportunities to leverage the strengths while alleviating the weaknesses of existing tools: exploration support, representation salience, live engagement, and vocabulary correspondence. We end the description of each of these natural diagramming opportunities by highlighting existing techniques that may be further developed to achieve it.




```
=== COLORS ===
* Orange
Base: #f7883c
Dark: #000 @ 15% (overlay)
Darker: #000 @ 30% (overlay)
Light: #fff @ 20% (overlay)
Lighter: #fff @ 40% (overlay)
=== STROKES ===
Main (silhouette only): #000 2pt thickness (solid)
Secondary: #000 1pt thickness (solid)
Tertiary: #000 @ 50% (overlay) 1pt thickness
Behind: #000 @ 1pt thickness (dashed: 6pt dash, 7pt gap,
      round cap, align to corners)
=== FONT ===
Base: Linux Libertine (add with TikZ directly in TeX)
Size: 11pt
```

Figure 3.5: P3 uses a cheat sheet to track frequently used style attributes.

3.4.1 Exploration Support

Exploration pervades the diagramming process for making conceptual diagrams, from choosing representation to determining stylistic details. We characterize two types of exploratory activities by adopting terms from Goel [62]: (a) *Lateral transformations* involve ideation and exploration at the high-level to broaden the design space, e.g., finding the appropriate visual representation for forces in Figure 3.2. (b) *Vertical transformations* are more detailed refinements to a pre-determined visual representation, such as deciding the arrowhead style in Figure 3.2b.

Sketching is an important, if not necessary, part of the design process [29]. Our participants produced physical sketches to explore design alternatives before transitioning to digital tools, which are perceived as a medium of higher commitment. They performed this type of exploration by looking at prior work and multiple alternatives *laterally*. Early informal sketches naturally affords lateral transformations [62].

Diagrammers also attempt to leverage precision, automation, and abstraction afforded by digital tools. After the visual representation stabilizes, they move to digital tools and refine this determinate representation, i.e. *vertically* refine their design. Unfortunately, existing tools do not provide sufficient flexibility to support even small vertical changes. Whereas DM tools require significant manual efforts to propagate local changes, PL tools require a high upfront cost to create abstractions that reduce future repetition. As a result, our participants described various *ad hoc* workarounds to perform common activities for exploration such as backtracking, versioning, and reuse in both PL and DM tools.

In essence, a gap exists between *drafting* and *crafting*, a design dilemma that has plagued other sketching tools (such as digital pens) [146]. Solving this dilemma requires tools that continuously support diagrammers to explore lateral and vertical changes, allowing more fluidity to make, track, and revert changes.

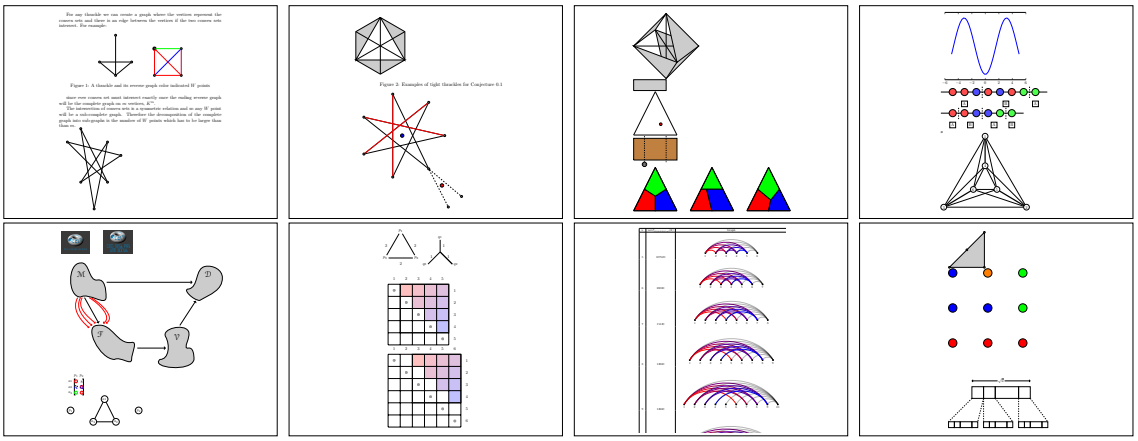


Figure 3.6: P7 organizes all previously made TikZ diagrams in a single document.

One solution to the problem of exploration and change management can be seen in tools for exploratory programming. *Exploratory Programming* (EP) characterizes a practice of experimenting and prototyping adopted by programmers across a wide range of domains [153, 90]. Data scientists reuse and iterate on small snippets of scripts to analyze data exploratively. VARIOLITE [89] support local versioning with "variant boxes" around regions of code. Software engineers often backtrack by manually deleting or re-typing code when developing software [183]. *Selective undo* techniques allow complex backtracking in code editing [184], which was also shown to be effective for painting applications [123].

Some systems also show opportunities for automatic vertical refinement. For instance, to support the transition from freehand sketches to final UI implementation, SILK recognizes hand-drawn shapes and translates them to real UI components [99]. DREAMSKETCH generates 3D models from sketches [86]. Sketches for conceptual diagrams, however, can often be intentionally ambiguous and unstructured, making recognition, beautification, and generation of reusable diagram components challenging [159].

Finally, in addition to better support for history management, another approach may be to reduce unnecessary changes through improved abstractions. *Program synthesis* can be used to generate reusable functions from examples of lower-level interactions with a potentially non-programmatic interface [68]. SKETCH-N-SKETCH is a vector drawing tool with synchronized code and graphical views of the same drawing [74]. It synthesizes reusable functions from direct manipulation of objects in the graphical view and thereby enables users to avoid repetition. Ellis et al. [47] synthesize imperative programs from hand-drawn sketches. Currently, synthesizing high-level abstractions and surfacing them in a non-obstructive, meaningful manner are still open problems for future conceptual diagramming tools.

3.4.2 Representation Salience

Constructing and interpreting representations are crucial skills for learning new concepts and developing domain expertise [1]. Many of our participants track prior representations, curate reuse libraries, search for ones created by others, and inventively create new ones in their

representation finding phase. This suggests an opportunity for natural diagramming tools that support **representation salience**, treating visual representations as first-class entities and providing operations to easily interact with them.

Bret Victor uses the analogy of climbing "up" (*abstraction*) and "down" (*concretization*) the *ladder of abstraction* to characterize the process of understanding complex systems using visual representations [170]. Abstraction and parametrization of diagrams allows fast generation of families of similar diagrams, which can be used in multiple places or serve as a set of design alternatives. In existing tools without easy access to abstraction constructs, generating design alternatives can be a time-consuming manual process. Our participants use existing abstraction constructs such as macros and functions to encode representations, but they tend to be highly personalized and brittle solutions. As a result, these custom abstractions are rarely scalable or composable, as one frequent TikZ user said *"macros are terrible, [I make macros that are] 20 or 30 braces deep... it's just hard to write and edit."* (P11) In addition, these poor abstractions still require significant time investment and force diagrammers to concretize concepts manually instead. Consequently, diagrammers' representational encodings are scattered in manually curated personal libraries, online examples, and cheatsheets of lower-level elements.

For visual representation to be *salient*, both the underlying structures and mappings to visual elements need to be encoded in the diagramming system explicitly. Further, these encodings must be specified with *manageable, scalable, and composable abstraction constructs* that allow diagrammers to move "up" and "down" the ladder of abstraction easily.

The management of visual representations at different levels of abstraction can be seen in many fields. For instance, Gross [64] tackles the problem of the fixed low-level representations of computer-aided design (CAD) tool by supporting gradual transition from sketches to more structured diagrams and suggesting concrete representations given early conceptual sketches. Data visualization tools such as Dashiki [115] and Draco [119] can manage of multiple representations of the underlying data.

The lack of representation salience often manifest in highly *viscous* diagramming tools that operate on low-level primitives and lose deeper *semantics* of graphical components. To solve this problem in data visualization, the *grammar of graphics* [176] formalizes a rich set of operations to transform data into visual components. For mathematical diagrams, PENROSE [128] includes two domain-specific languages that decouple visual representations from declaration of abstract objects and encode visual representations by pattern-matching on the objects and declaring visual elements. Apart from these domain-specific solutions above, however, conceptual diagramming tools still lack a general and accessible approach to specify problem domains and the visual representations thereof.

3.4.3 Live Engagement

Hutchins et al. [81] introduce *direct engagement*, "the *qualitative* feeling that one is directly engaged with *control* of the objects," as an important criterion for effective interfaces. Our results show that direct engagement for diagramming can be perceived differently depending on the kinds of interfaces and the sense of *control* they afford, i.e., the sense of agency over essential operations in conceptual diagramming [105]. DM tool design affords continuous representations of objects and immediate visibility of incremental changes [155]. As a result, they naturally afford a sense

of *global* control over the global rearrangement of diagram layout. Yet, *local* control over precise specification of visual properties and creation of high-level abstractions is still challenging in direct-manipulation tools. On the other hand, while these same operations are directly supported in PL tools, our participants reported frustration with their high latency and long compile-and-run cycles. Our results suggest immediate visual feedback (or *liveness* [160]) is also essential for abstract operations. In other words, the sense of control and direct engagement is diffused among DM and PL tools.

Traditional direct manipulation interfaces can be augmented by novel interaction and programming techniques such as programmatic brushes [82], *programming by example* [66] and *programming by manipulation* [78]. *Bidirectional programming* [37] proposes to combine direct manipulation and textual programming by surfacing both direct manipulation and text-based interfaces and synchronizes and synchronizing changes in both directions: (i) from the program text to the output (liveness) and (ii) from the output to the program text (direct engagement). These techniques, although currently limited to narrower domains, provide promising directions towards bridging the gap between PL and DM tools.

There have been also significant advances in programming languages to support liveness. *Live programming* techniques can provide responsive and continuous feedback on program changes. The methodology has been increasingly adopted in computer science education, web development, and traditional programming environments [113, 79, 144]. Tanimoto [160] proposes four levels of liveness with the top level featuring "stream-driven updates" and "informative, significant, responsive and live" visual representations of programs' dynamic behaviors. Techniques such as incremental compilation and type inference [114] and typed holes [130] allow fast compilation and facilitate liveness of programming environments. However, many live programming systems only offer one-way updates from the program to its output visual representation, inhibiting the opportunities for direct interactions with the visuals.

3.4.4 Vocabulary Correspondence

Conceptual diagrams are made of "abstract and topological" [53] shapes that are mapped from domain objects in the *content model* [145]. In Figure 3.2b, the concept of two counteracting forces (domain objects) is mapped to two arrows, but the exact styling and lengths of the arrows do not change the meaning of the diagram. As a result, users' vocabularies for conceptual diagramming are often abstract, topological, and domain-specific. Therefore, there is an opportunity for natural diagramming tools that support **vocabulary correspondence** by having a grammar or an interface that directly maps to users' vocabulary for diagramming.

As shown in our results, diagrammers define new abstractions to both automate and *naturalize* their diagramming process. More advanced diagrammers create abstractions to quickly generate new diagram instances and fit their mental models, but, even for advanced diagrammers, abstractions are brittle and often not share-able, due to the lack of tool support. In addition, many participants described diagrams in terms of relative, high-level relationships such as "smaller"/"bigger" and "overlaping"/"non-overlapping". But the tools they use tend to operate on absolute units and do not provide support for specifying such relationships. One participant shared a vision of an ideal tool:

"*The best tool... would have fairly high level primitives. I might say 'Okay, I want it to be*

symmetric in this way. I want this thing always to be attached to that.' I want to be able to define my own higher level primitives." (P2)

In other words, large semantic and articulatory distances [81] still exist between interaction metaphors and diagrammers' vocabulary, creating an opportunity improve the closeness of mapping [63] while maintaining users' control and the expressiveness of diagramming tools.

One opportunity to do so is allow users to introduce their own vocabulary to the tool, for instance, through domain-specific languages (DSL). DSLs provide focused expressive power within specific problem domains at the cost of generality [168]. For instance, domain-specific diagramming systems such as GraphViz allow succinct, high-level specification of diagrams and leverage domain knowledge to solve for diagram layouts algorithmically [48]. So far, DSLs have been most useful areas such as graph visualization, but they may prove to be useful elsewhere too. To allow end-users to introduce their own DSLs, *language workbenches* may be a viable implementation route. These workbenches allow efficient definition, reuse, and composition of DSLs [50], but much work remains, as existing language workbenches such as MPS [171] are still complex to learn for end-users like many of our participants.

Another way to model abstract and topological relationships is using high-level constraints, an idea that has existed since the invention of SketchPad [158]. Constraint-based systems are extensively used in Computer-Aided Design (CAD) tools [157]. CAD users commonly use constraints in *parametric drawing*, exploring different configurations of complex shapes. Automatic formatting of documents, digital drawings, and web pages are often modeled as constraints and can be optimized by solvers [80, 131]. To further simplify the process of constraint specification, some systems allow visual interactions with constraints [78, 61] while others intelligently infer constraints by examples [98]. By offloading the burden of low-level specification to constraint solvers, diagrammers often lose *control* of diagram elements, which poses usability challenges to future diagramming tools.

3.5 Conclusion and Future Work

Conceptual diagrams are essential for understanding concepts, communicating ideas, and improving instructions effectively in many fields. This chapter provides the first empirical study of how domain experts create conceptual diagrams.

Our results demonstrate *representation finding* as a vital step in the diagramming process and the role that sketches play in this step. However, due to limitations of current tools, notably the trade-offs between direct manipulation tools and programming languages, reusing representations is still challenging. As a result, diagrammers creatively circumvent these limitations by employing a set of *ad hoc* techniques to reuse diagram components and to scale up diagram production.

Based on our results, we introduce the concept of **natural diagramming** and four opportunities for natural diagramming support: exploration support, representation salience, live engagement, and vocabulary correspondence. For each of them, we discussed how recent advances from various research communities can help improve existing tools and design future tools.

Future work can leverage the substantial amount of conceptual diagrams that exist in the wild and perform large-scale analysis to gain a more quantitative understanding of existing diagrams. Similarly, it is possible to leverage the large number of current diagramming tools, some of

which support a subset of natural diagramming. In particular, it may be possible to isolate and quantify the benefits of each opportunity of natural diagramming introduced in this chapter by analysing existing tools. A deeper, more systematic analysis of the relationships and trade-offs among the four natural diagramming opportunities may further inform tool designers to make design decisions more critically. Further, this study does not focus on collaboration support for conceptual diagramming. Future work may explore how to support, for instance, *vocabulary correspondence* when multiple diagrammers from diverse backgrounds collaborate on a single conceptual diagram.

Natural diagramming embodies our vision for future diagramming tools—tools that seamlessly and naturally translate diagrammers' high-level ideas to beautiful and illustrative diagrams. This chapter articulates a concrete vision for systems designers to create more effective diagramming tools. In subsequent chapters, we present PENROSE (Chapter 4) and EDGEWORTH (Chapters 5 and 6), two tools designed with this vision in mind.

Chapter 4

PENROSE: From Notations to Beautiful Diagrams[1]

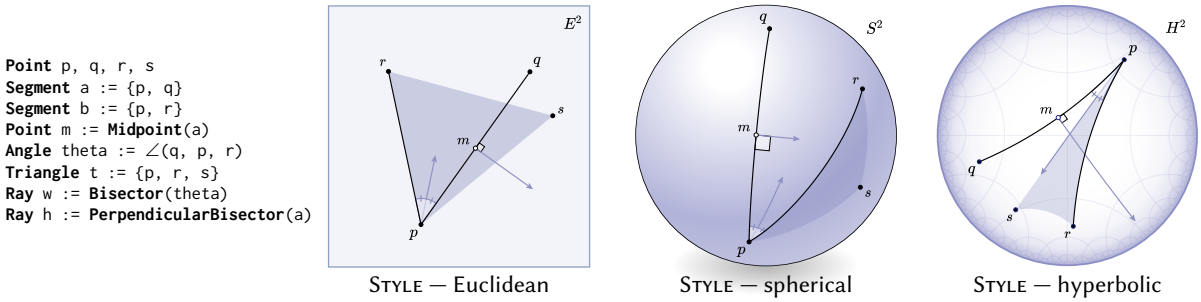


Figure 4.1: PENROSE is a framework for specifying how mathematical statements should be interpreted as visual diagrams. A clean separation between abstract mathematical objects and their visual representation provides new capabilities beyond existing code- or GUI-based tools. Here, for instance, the same set of statements (left) is given three different visual interpretations (right), via Euclidean, spherical, and hyperbolic geometry.

Informed by the results from the interview study (Chapter 3), we developed PENROSE, a language-based diagramming platform [182]. The core PENROSE system addresses **representation salience**: it has first-class support for creating and reusing visual representations. To accomplish this, PENROSE decomposes the concerns of diagramming into two domain-specific languages (DSLs) with distinct purposes: SUBSTANCE contains the mathematical content of the diagram. STYLE explicitly specifies mappings from mathematical objects to visual icons. In contrast to tools that specify diagrams via direct manipulation or low-level graphics programming, PENROSE enables creation and exploration of diagrams that faithfully preserve the underlying mathematical meaning. Sections 4.1 and 4.2 explain the overall design goals, Sections 4.3 and 4.4 detail PENROSE's language design and automatic layout engine that powers its runtime. Section 4.5 demonstrates the effectiveness and generality of the system by showing how it can be used to illustrate a diverse set of concepts from mathematics and computer graphics.

[1]This chapter is adapted from "Penrose: From Mathematical Notation to Beautiful Diagrams" [182].

4.1 Introduction

A central goal of PENROSE is to lower the barrier to turning mathematical ideas into effective, high-quality visual diagrams. In the same way that TEX and LATEX have democratized mathematical writing by algorithmically codifying best practices of professional typesetters [12], PENROSE aims to codify best practices of mathematical illustrators into a format that is reusable and broadly accessible.

Our approach is rooted in the natural separation in mathematics between abstract definitions and concrete representations. In particular, the PENROSE system is centered around the specification of a *mapping* from mathematical objects to visual icons (Section 4.2). Such mappings are expressed via domain-specific languages (DSLs) that reflect familiar mathematical notation and can be applied to obtain new capabilities that are difficult to achieve using existing tools (Section 2.3). A key distinction is that PENROSE programs encode a *family* of possible visualizations, rather than one specific diagram. Hence, effort put into diagramming can easily be reused, modified, and generalized. This approach has several broad-reaching benefits:

- **Accessibility.** Novice users can generate diagrams by simply typing mathematical statements in familiar notation, leveraging the efforts of more expert package developers.
- **Separation of content and presentation.** The ability to easily swap out different visual representations helps deepen understanding by illustrating the same mathematical concepts from many different visual perspectives.
- **Evolution of collections.** Existing collections of diagrams can easily be improved and modified to meet the needs of a target platform, e.g., desktop vs. mobile, different printing processes, or different language localizations.
- **Large-scale generation.** It becomes easy to generate large collections of illustrations to explore an idea, or to accompany, say, randomly-generated homework exercises.

Beyond describing the implementation of PENROSE, the purpose of this chapter is to explore the general challenge of designing systems for diagram generation. We hence start with a discussion of goals and trade-offs that inform our system design (Section 4.2). Readers may also find it helpful to periodically refer to the more detailed but purely descriptive account of the system given in Section 4.3 and Section 4.4.

4.2 System Design

Our aim is to build a system for converting abstract mathematical ideas into visual diagrams. Choices about system design are guided by several specific goals, many of which are supported by interviews presented in Chapter 3:

1. Mathematical objects should be expressed in a familiar way.
2. The system should not be limited to a fixed set of domains.
3. It should be possible to apply many different visualizations to the same mathematical content.
4. There should be no inherent limit to visual sophistication.

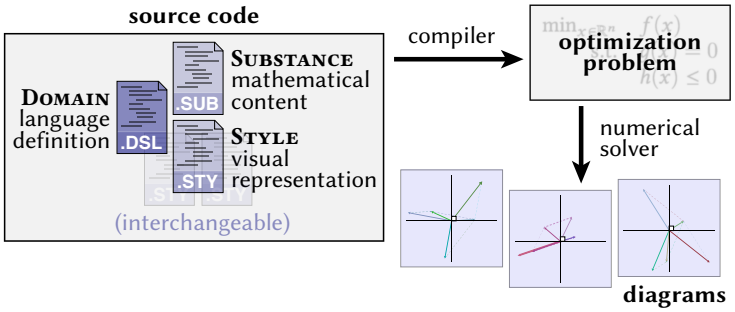


Figure 4.2: High-level pipeline: a compiler translates mathematical statements and a chosen visual representation into a constrained optimization problem. This problem is then solved numerically to produce one or more diagrams.

5. It should be fast enough to facilitate an iterative workflow.
6. Effort spent on diagramming should be generalizable and reusable.

To achieve these goals, we take inspiration from the way diagrams are often drawn by hand. In many domains of mathematics, each type of object is informally associated with a standard *visual icon*. For instance, points are small dots, vectors are little arrows, etc. To produce a diagram, symbols are systematically translated into icons; a diagrammer then works to arrange these icons on the page in a coherent way. We formalize this process so that diagrams can be generated computationally, rather than by hand. Specifically,

> The two organizing principles of PENROSE are:
> (i) to **specify** diagrams via a mapping from mathematical objects to visual icons, and
> (ii) to **synthesize** diagrams by solving an associated constrained optimization problem.

Just as the occupant of Searle's "Chinese room" does not actually understand a foreign language [39], a system designed this way need not perform deep reasoning about mathematics—it simply does a translation. We hence do not expect our system to solve all challenges of diagramming. Users are still responsible for, say,

- choosing meaningful notation for a mathematical domain,
- inventing a useful visual representation of that domain, and
- ensuring that diagrams correctly communicate meaning.

Likewise, a system cannot be expected to solve hard computational or mathematical problems (e.g., the halting problem or Fermat's last theorem) in order to construct diagrams. Yet despite this shallow level of reasoning, PENROSE is able to generate quite sophisticated diagrams. In fact, even in the absence of such reasoning, naïve visualization often provides useful observations (Figure 4.3).

```
Set A, B
Point p
A ⊂ B
p ∈ A
p ∉ B
```

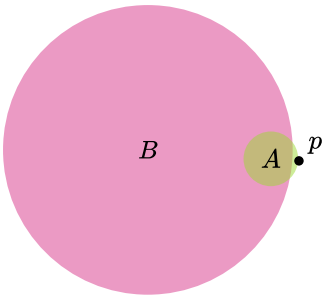


Figure 4.3: An optimization-based approach has myriad benefits. Here a logically inconsistent program fails gracefully, providing visual intuition for *why* the given statements cannot hold.

The resulting system effectively models diagram generation as a compilation process, where the compilation target is a constrained optimization problem rather than (say) a binary executable or a static image. Once compiled, this problem can be used and *reused* to generate visual diagrams; Figure 4.2 illustrates the high-level system flow. From a programming language point of view, a mapping expressed

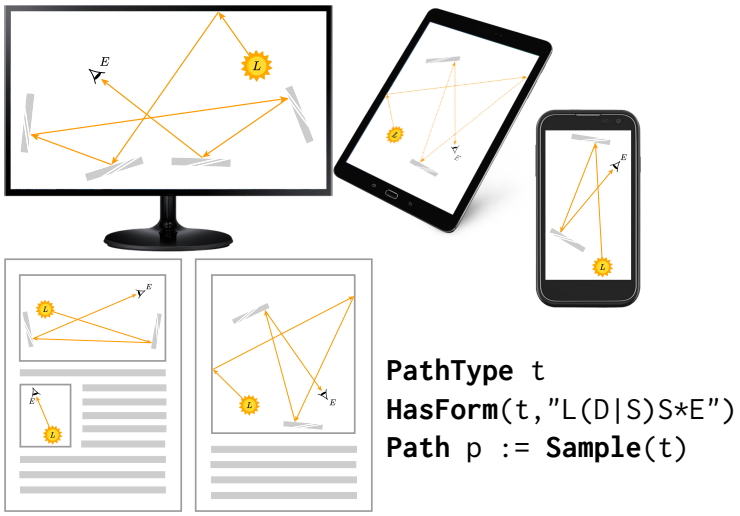


Figure 4.4: By specifying diagrams in terms of abstract relationships rather than explicit graphical directives, they are easily adapted to a wide variety of use cases. Here we use identical PENROSE code to generate ray tracing diagrams for several targets (Section 4.5.6). Though the arrangement and number of objects changes in each example, the meaning remains the same.

in this framework defines an *executable visual semantics*. That is, it gives a specific, visual, and computable interpretation to what were previously just abstract logical relationships.

4.2.1 Language-Based Specification

A major decision in PENROSE is to use programming languages to specify both mathematical objects (Section 4.2.1) and their visual representation (Section 4.2.1). Graphical (e.g., sketch-based) specification would demand that users already know how to visualize abstract ideas, and it ties mathematical content to one specific visual representation, which conflicts with Goal 3. A language-based specification provides the level of abstraction needed to separate content from visualization. This technique supports Goal 1, since language is the most common means by which mathematical ideas are expressed. From a system design point of view, a language-based encoding provides a unified representation for identifying and transforming mathematical objects throughout the pipeline. Moreover, a language-based interface makes it easy for PENROSE to provide a *platform* on which other diagramming tools can be built (as in Section 4.4.5 or Section 4.5.7). One trade-off is that a language-based approach requires users to express themselves in formal mathematical or computational language, making it more difficult for (say) visual artists and designers to contribute new representations.

A secondary decision is to split specification of mathematical content and visualization across two domain-specific languages: SUBSTANCE and STYLE. A good analogy is the relationship

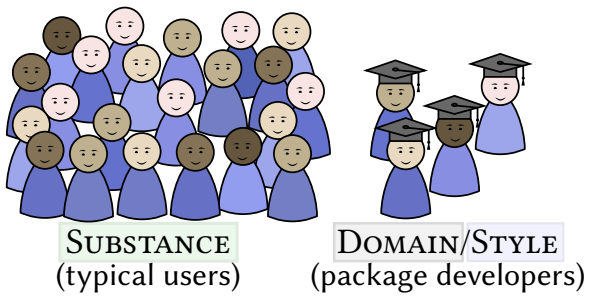


Figure 4.5: Most PENROSE users need only use the SUBSTANCE language, but can benefit from packages written by more expert DOMAIN and STYLE programmers. This is similar to the TEXecosystem, where most users only write documents, but benefit from expert-authored packages.

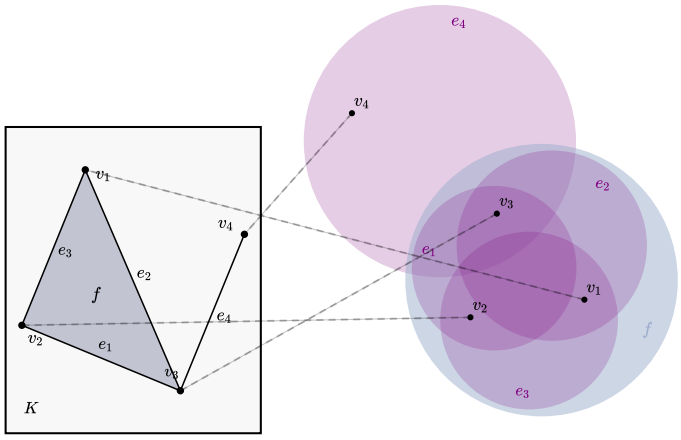


Figure 4.6: One benefit of a unified framework is that different domains are easily combined. Here, two existing packages (for meshes and set theory) were combined to illustrate that a simplicial complex *(left)* is closed with respect to taking subsets *(right)*.

between HTML [16], which specifies content, and CSS [104], which describes how it is rendered. A schema called DOMAIN (analogous to XML or JSON schemas) defines the mathematical domain of interest, supporting Goal 2. In line with Goal 3, this division allows the same styles to be reused for different content, and likewise, the same content to be displayed in many different styles. Our goal is for this division to support an ecosystem where novice users can benefit from packages written by more experienced developers (Figure 4.5). Finally, as in mathematics, the ability to adopt user-defined, domain-specific notation (Section 4.2.1) enables efficient expression of complex relationships in a way that is both concise and easy to understand [95]. Encoding ideas directly in the idiom of a problem domain often results in better program comprehension than (say) a sequence of library calls in a general-purpose language [168]. We discuss the scope and limitations of our languages in Section 4.6.

Mathematical Domain (DOMAIN)

One of our primary goals (Goal 2) is to build a unified system for diagramming, rather than to focus on specific domains (as in, say, *GraphViz* [48] or *GroupExplorer* [33]). A unified design enables objects from different domains to coexist in the same diagram, often by doing little more than concatenating source files (Figure 4.6). Moreover, effort put into (say) improving system performance or rendering quality is amortized across many different domains.

Users can work in any area of mathematics by writing so-called DOMAIN schemas (Section 4.3.1) that define DSLs tailored to a given domain. This design also empowers users to adopt their own notation and conventions for modeling the domain. Use of domain- and user-specific notation reflects common practice in mathematical writing, where the meaning of a symbol is frequently overloaded depending on context. Importantly, a DOMAIN schema is purely abstract: it does not define an internal representation for objects, nor does it give definitions to functions or predicates. This level of abstraction is crucial for Goal 3, since it allows multiple visual representations to later be applied to objects from the same domain.

Mathematical Content (SUBSTANCE)

To define the content of a diagram, one must be able to specify (i) the objects in the diagram, and (ii) relationships among these objects. In line with Goal 1, SUBSTANCE uses concise assertions

For any vector space X, let $u, v \in X$ be orthogonal vectors of equal length, and let $w = u + v$. Then u and w make a $45°$ angle.

```
VectorSpace X
Vector u, v ∈ X
Orthogonal(u, v)
EqualLength(u, v)
Vector w ∈ X
w := u + v
```

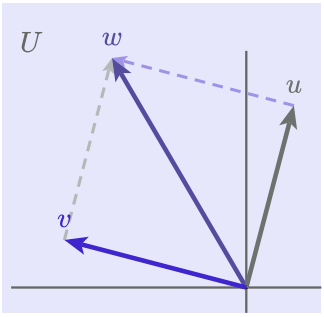


Figure 4.7: Extensibility enables users to adopt conventions and notation *(center)* that reflect the way they naturally write mathematical prose *(left)*. Here, the resulting diagram *(right)* plays the role of the concluding statement.

that resemble standard mathematical prose (see for example Figure 4.7). Formally, it can model any domain expressible in a compositional language of types, functions, and predicates, which are the basic constructs found in conventional mathematical notation [58]. Just as definitions are typically immutable in mathematics, SUBSTANCE draws inspiration from strongly typed functional languages (such as *ML* [117]) where objects are stateless. This choice also simplifies system implementation, since the compiler can assume fixed definitions. A conscious design decision, in line with Goal 3, is to exclude all graphical data (coordinates, sizes, colors, etc.) from SUBSTANCE—since its sole purpose is to specify *abstract relationships* rather than *quantitative data*. All such data is instead specified in STYLE or determined via optimization. This division relieves users from the burden of tedious and repetitive graphics programming, which can instead be factored out into reusable STYLE code.

Existing languages would be difficult to use in place of SUBSTANCE since they lack the semantics needed to encode complex logical relationships and do not provide language-level extensibility. For instance, TEX [12] and *MathML* [118] markup provide only enough information to translate plain text into mathematical glyphs; computer algebra systems like *Mathematica* and *Maple* have limited type systems or provide only a small set of fixed predicates (e.g., asserting that a number is real). Conversely, the much richer languages used by automated theorem provers and proof assistants (such as *Coq* [17] and *Lean* [120]) are overkill for simply specifying diagrams. A custom language provides simple, familiar syntax and clear error messages. We do however adopt some ideas from *Coq*, such as the ability to customize syntactic sugar (Section 4.3.1).

Mathematical Visualization (STYLE)

The meaning of a diagram is largely conveyed by relative relationships rather than absolute coordinates. Moreover, diagrams are often underconstrained: relationships needed to convey the intended meaning determine a *family* of possible solutions, rather than a single unique diagram. STYLE hence adopts a constraint-based approach to graphical specification in the spirit of *Sketchpad* [158]: diagrams are built up from hard constraints that must be satisfied and soft penalties that are minimized (Section 4.3.3), then unspecified quantities are solved for via numerical optimization (Section 4.4). Though procedural definitions can still be used, the programmer need not provide absolute coordinates (as in imperative languages like *PostScript* or *OpenGL*). Though an implicit specification can make output hard to predict, part of the allure

of PENROSE is the potential to find interesting or surprising examples. Moreover, the approach yields more concise code; for instance, STYLE programs are much shorter than the SVG files they produce.

An alternative design might be to use an application programming interface (API), though historically APIs have been eschewed for specification languages for good reason. Language provides far more concise expression of complex relationships—imagine styling an entire website via the DOM API's `getElementById()` and `setStyle()` methods, versus a few short lines of CSS. Visual programming languages (like *LabView* [46] or *Grasshopper* [116]) might suffice for basic specifications (e.g., vectors should be drawn as arrows), but don't scale to more complex concepts that are easily expressed via language [28].

A key design challenge is identifying objects that appear in a SUBSTANCE program. Objects in a given mathematical universe are distinguished not only by their type, but also by their relationships to other objects. A widely-used mechanism for specifying such relationships is through CSS-like *selectors*. STYLE adopts a similar mechanism that performs pattern matching on the types, functions, and predicates appearing in a DOMAIN schema (Section 4.3.3).

4.2.2 Optimization-Based Synthesis

The second major design decision in PENROSE is to use constrained optimization to synthesize diagrams satisfying a given specification (Section 4.4). This approach is again inspired by how people often draw diagrams by hand (e.g., using GUI-based tools): visual icons are placed on a canvas and iteratively adjusted until no further improvements can be made. In difficult scenarios, a diagrammer may try several global arrangements before refining the final design, though typically no more than a few. Automating this process makes it easy to perform layout tasks that would be tedious by hand (Figure 4.8).

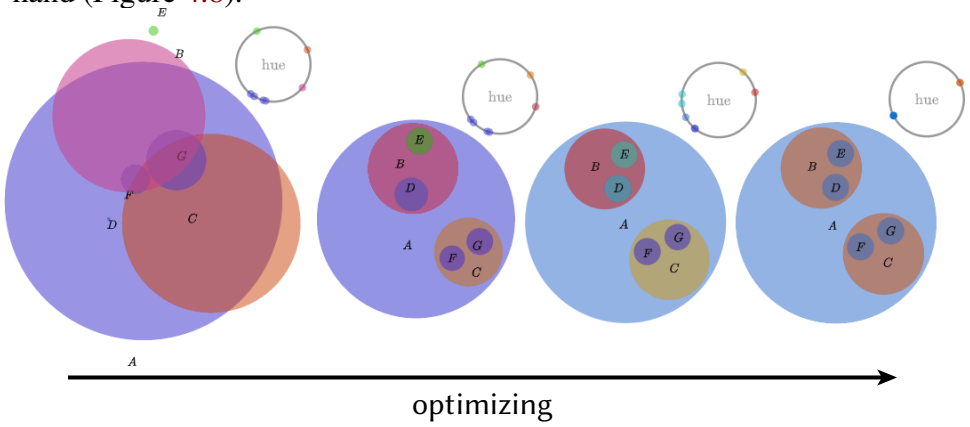


Figure 4.8: An optimization-based approach makes it possible to jointly optimize visual attributes that are difficult to coordinate by hand. Here for instance we optimize color contrast according to the spatial proximity of adjacent disks *(left to right)*, ultimately discovering a two-color solution *(far right)*. The system can also be used to debug the optimization process itself—in this case by drawing the hue of each disk as a dot on a color wheel.

There are good reasons to believe that an optimization-based approach can scale to very complex diagrams. First, attractive diagrams need not be optimal in a *global* sense—they should simply not permit obvious *local* improvements, such as text that could easily be moved closer to the item it labels. In fact, disparate local minima can provide useful examples that help build

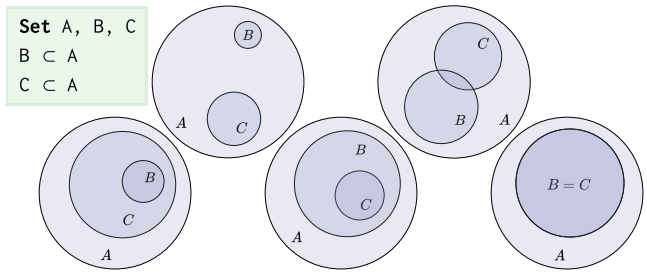


Figure 4.9: A language-based design makes it easy to build tools on top of PENROSE that provide additional power. Here we use standard techniques from program synthesis (Section 4.5.7) to automatically enumerate how the given relationships can be realized. Generating such examples helps to see important corner cases that might be missed when drawing diagrams by hand (where perhaps the top-left diagram most easily comes to mind).

intuition (Figure 4.9). Second, even sophisticated diagrams have surprisingly few degrees of freedom in comparison to many modern optimization problems (e.g., tens or hundreds, versus thousands or millions). Finally, strategies employed by expert diagrammers can be applied to manage complexity, such as independently optimizing small components of a diagram (akin to nonlinear Gauss-Seidel), rather than optimizing all degrees of freedom simultaneously.

In line with Goal 2 and Goal 3, an optimization-based approach can be applied generically and automatically for any user-defined domain and visual representation, without requiring programmers to think about the details of the layout process. In our system, the optimization problem is defined using common-sense keywords (Section 4.3.3) in STYLE and chaining together basic operations (e.g., arithmetic). Since the diagram specification is divorced from the details of the solver, optimization strategies can be changed and improved in future versions of the system while preserving compatibility with existing code. The main cost of an optimization-based approach is that it puts demands on system design "upstream": all expressions used to define a visual style must be differentiable. As discussed in Section 4.4.2, these requirements are largely satisfied via standard techniques (e.g., by using *automatic differentiation*).

In general, diagram optimization is a challenging problem in its own right, which we of course do not aim to solve conclusively in this chapter. Currently, we just use a generic constrained descent solver (Section 4.4.2). However, we have been pleased to find that this simple approach handles a wide variety of examples from different domains without requiring domain-specific strategies.

4.2.3 Plugins

The final design decision in PENROSE is to provide system-level extensibility via a *plugin* interface for calling external code in SUBSTANCE and STYLE. Providing a plugin system is essential to enable users to integrate external code that is specialized to solve particular logical or graphical challenges. In fact, such interfaces for integrating external code are already provided by many systems (e.g., TEX, *Adobe Illustrator*, and TikZ's plugin system for graph layout algorithms [48]). The interface for PENROSE plugins is designed to define a clear and simple boundary between the PENROSE system and the plugin while enabling each component to focus on its strengths. A plugin can analyze and augment the set of abstract objects defined in SUBSTANCE, as well as analyze and augment the numerical information in STYLE. This simple interface allows plugins to be written in any language (or repurposed from other systems) and operate independently

from the implementation details of PENROSE. However, a plugin cannot change an existing SUBSTANCE or STYLE program or directly generate static graphical content, since such plugins would abandon the benefits that PENROSE provides, such as the ability to re-style content and avoid use of absolute coordinates. Figure 4.4 illustrates how a simple plugin can make use of SUBSTANCE and STYLE information to create "responsive" diagrams.

4.3 Language Framework

The PENROSE language framework comprises three languages that play different roles:

- A DOMAIN schema declares the objects, relationships, and notation available in a mathematical domain.
- A SUBSTANCE program makes specific mathematical assertions within some domain.
- A STYLE program defines a generic mapping from mathematical statements in some domain to a visual representation.

A *package* consisting of a DOMAIN, and one or more compatible STYLE programs, can be used to illustrate SUBSTANCE programs from a given domain (Figure 4.2). Though some starter packages are provided for the examples discussed in Section 4.5, the real power of STYLE and DOMAIN is that they enable PENROSE to be easily extended. In this section we illustrate these languages via the running example of a linear algebra package (Figures 4.10 through 4.12).

4.3.1 The DOMAIN Schema

A DOMAIN *schema* describes a domain of mathematics by defining the objects and notation that can be used by associated SUBSTANCE and STYLE programs. A partial example for linear algebra is shown in Figure 4.10. The `type` lines define the available object types, `function` lines define the domain and codomain for the set of available functions (where `*` denotes a Cartesian product), and `predicate` lines define the possible relationships among objects, including unary predicates. Importantly, a DOMAIN schema is purely abstract: it does not define a specific representation for objects, nor does it define bodies for functions or predicates. For instance, we do not say here that a vector is encoded by a list of coordinates, nor do we write an addition operation on such coordinates. A concrete visual interpretation of these definitions is given by a STYLE program (Section 4.3.3). Types can be given fields via *constructors*. For instance, the line

```
constructor MakeInterval: Real min * Real max -> Interval
```

assigns fields `min` and `max` to an `Interval`, which can be accessed from a SUBSTANCE or STYLE program (e.g., to assert a relationship between endpoints). Subtyping via the syntax `Subtype <: Type` facilitates generic programming. Finally, `notation` lines define optional syntactic sugar that can simplify code (e.g., in Figure 4.11).

```
type Scalar, VectorSpace, Vector      -- LinearAlgebra.dsl
function add: Vector * Vector -> Vector
function norm: Vector -> Scalar
function scale: Scalar * Vector -> Vector
...
predicate In: Vector * VectorSpace
predicate Unit: Vector
predicate Orthogonal: Vector * Vector
...
notation "v1 + v2" ~ "add(v1,v2)"
notation "|y1|" ~ "norm(y1)"
notation "s * v1" ~ "scale(s,v1)"
notation "Vector v ∈ V" ~ "Vector a; In(a,U)"
notation "v1 ⊥ v2" ~ "Orthogonal(v1,v2)"
...
```

Figure 4.10: A DOMAIN schema specifies the building blocks available in a given mathematical domain, as well as any associated syntactic sugar. This schema (abbreviated) enumerates some basic constructs from linear algebra.

4.3.2 The SUBSTANCE Language

Each statement in the SUBSTANCE language either (i) declares an object, (ii) specifies properties of an object, or (iii) specifies relationships among objects within some DOMAIN schema. As in mathematics, not all attributes need be fully specified. For instance, one can talk about a point without giving it explicit coordinates. Together, these statements describe a *context* that encloses all the mathematical objects and relationships that have been defined.

Figure 4.11 shows an example in which SUBSTANCE code specifies properties and relationships for a pair of vectors. Importantly, these statements do not induce any kind of numerical evaluation. For instance, no coordinates are assigned to `x1` in order to make it unit—in fact, the vector space `X` does not even have an explicit dimension. Instead, statements specify persistent

```
VectorSpace X
Vector x1, x2
In(x1, X)
In(x2, X)
Unit(x1)
Orthogonal(x1, x2)
label x1 $x_1$
label x2 $x_2$
```

(unsugared)

```
VectorSpace X
Vector x1, x2 ∈ X
Unit(x1)
x1 ⊥ x2
label x1 $x_1$
label x2 $x_2$
```

(sugared)

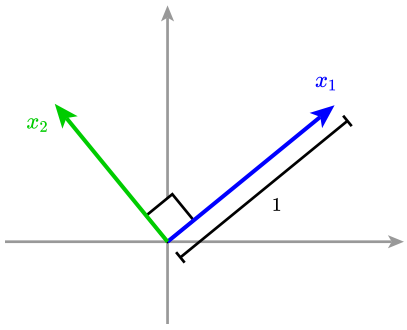
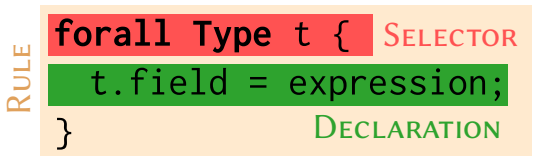


Figure 4.11: When used with the STYLE defined in Figure 4.12, this SUBSTANCE code (with or without syntactic sugar) produces the diagram shown at right.

and purely symbolic relationships that provide cues for visualization; specific coordinates and attributes are later determined by the layout engine (Section 4.4). The final lines specify label strings to be used by the STYLE program, here in TEX notation. Figure 4.11, *center* shows a "sugared" version of this program using notation defined in the DOMAIN schema (Figure 4.10). Users can write programs either way, depending on the capabilities of their editor (e.g., support for Unicode input).

4.3.3 The STYLE language



STYLE specifies how expressions in a SUBSTANCE program are translated into graphical objects and relationships. It is a declarative specification language that shares many features with CSS. The core principle is to sketch out basic *rules* (e.g., visual icons for basic types) and then refine these rules via *cascading* (Section 4.3.3). Each rule uses a *selector* to pattern match on objects and relationships appearing in SUBSTANCE code (Section 4.3.3). A sequence of *declarations* then specifies a corresponding visualization, e.g., by emitting graphical primitives or enforcing constraints. Each declaration either assigns a *value* to a *field* (Section 4.3.3) or specifies a *constraint* or *objective* (Section 4.3.3). An example is shown in Figure 4.12, which defines part of the style used for the SUBSTANCE program in Figure 4.11. We will use this example to highlight the basic features of the language.

Selectors

A selector uses pattern matching to specify which objects will be styled by a rule. Unlike regular expressions, selectors do not match literal strings of SUBSTANCE code, but rather objects and relationships in the context defined by this code. A simple example is a selector that matches every instance of a type, indicated by the `forall` keyword. For instance, Section 4.3.3 matches all vector spaces. In subsequent declarations, `U` refers to the vector space `X` from the SUBSTANCE program. The `where` keyword restricts matches to objects that satisfy one or more relationships; e.g., Section 4.3.3 matches all pairs of orthogonal vectors. One can also match by name using backticks; e.g., Section 4.3.3 matches only the vector `x2`. Selectors could be enriched in the future to allow other statements from first-order logic (such as $\exists$, disjunctions, and conjunctions).

Cascading

A cascading mechanism allows rules to be refined for more specialized objects or relationships. For example, the selector in Section 4.3.3 matches a specific vector, refining an earlier rule that applies to all vectors. Rule precedence is determined by order in the STYLE file, and later rules can refer to any previously defined *field* (Section 4.3.3). The `override` keyword (Section 4.3.3) hints that a rule will modify an existing field, otherwise the compiler issues a warning.

```
forall VectorSpace U {                          -- LinearAlgebra.sty
   U.originX = ? -- to be determined via optimization
   U.originY = ? -- to be determined via optimization
   U.origin = (U.originX, U.originY)
   U.xAxis = Arrow { -- draw an arrow along the x-axis
          startX : U.originX - 1
          startY : U.originY
          endX   : U.originX + 1
          endY   : U.originY
      thickness  : 1.5
          style  : "solid"
          color  : Colors.lightGray
   } -- (similar declarations omitted for the y-axis)
}
forall Vector u, VectorSpace U where In(u, U) {
   u.arrow = Arrow {
      startX : U.originX
      startY : U.originY
        endX : ?
        endY : ?
       color : Colors.mediumBlue
   }
   u.text = Text {
      string : u.label -- label from Substance code
       color : u.arrow.color  -- use arrow's color
   }
   u.start = (u.arrow.startX, u.arrow.startY)
   u.end = (u.arrow.endX, u.arrow.endY)
   u.vector = minus(u.arrow.end, u.arrow.start)
   encourage near(u.text, u.end)
   ensure contained(u.end, U.shape)
}
forall Vector u, Vector v
where Orthogonal(u, v) {
   local.perpMark = Curve {
        pathData : orientedSquare(u.shape, v.shape, U.origin, const.perpLen)
        strokeWidth : 2.0
        color : Colors.black
        fill : Colors.white
   }
   ensure equals(dot(u.vector, v.vector), 0.0)
}
... -- (similar rule omitted for Unit)
Vector `x2`  { override `x2`.shape.color = Colors.green;  }
```

Figure 4.12: The STYLE program defining the visual style used in Figure 4.11, *right*. Note that this STYLE program can be reused for many different SUBSTANCE programs in the same domain.

Fields

The visual representation of an object is specified by creating *fields* that are assigned *values* (Section 4.3.3). For instance, in Lines 16–22 a field called `u.arrow` is created and assigned an expression describing an arrow. Fields are created on assignment and can have any name not conflicting with a reserved word. Fields not naturally associated with a single object can also be assigned locally to a rule. For instance, Section 4.3.3 is used to draw a right angle mark between any pair of orthogonal vectors. Every object automatically has fields `name` and `label` storing its SUBSTANCE name and label string (*resp.*), as well as any field created via a constructor (Section 4.3.1).

Properties

STYLE provides built-in graphical primitives (circle, arrow, etc.) with a fixed set of *properties*. Like fields, properties can be assigned values (as in Lines 35–40). If a value is not assigned, it will be assigned a default value, possibly a *pending value* (Section 4.3.3). For example, an arrow might be black by default, whereas the width of a box might be optimized (akin to *flexible space* in TEX).

Values and Expressions

Atomic *values* can be combined to form *expressions*. For instance, Lines 10–12 assign values, whereas Lines 6–9 assign composite expressions involving inline computation. Section 4.3.3 specifies value via a *path*, i.e.,, a sequence of expressions separated by `.` characters; such assignments are made by reference. Expressions can also access values from plugins (Section 4.4.3). A very important construct is *pending values*, denoted by a `?` as in Section 4.3.3. This line specifies that the location of the arrow endpoint is not fixed and will be automatically determined by the solver (Section 4.4).

Constraints and Objectives

Constraints and objectives describe how pending values should behave. In particular, the **`ensure`** keyword defines a hard constraint that the diagram must satisfy. For instance, Section 4.3.3 specifies that two orthogonal vectors must be drawn at right angles. The **`encourage`** keyword specifies a relationship that should be satisfied as much as possible. For instance, Section 4.3.3 asks that the label for a vector be placed close to its endpoint. These expressions are translated into constraints and energy functions that make up a numerical optimization problem (Section 4.4.2).

4.4 Layout engine

The layout engine translates PENROSE code into images (Figure 4.13). There are two main stages: a compiler (Section 4.4.1) translates code into an optimization problem that describes possible diagrams, then a solver (Section 4.4.2) produces solutions to this problem. These values are used

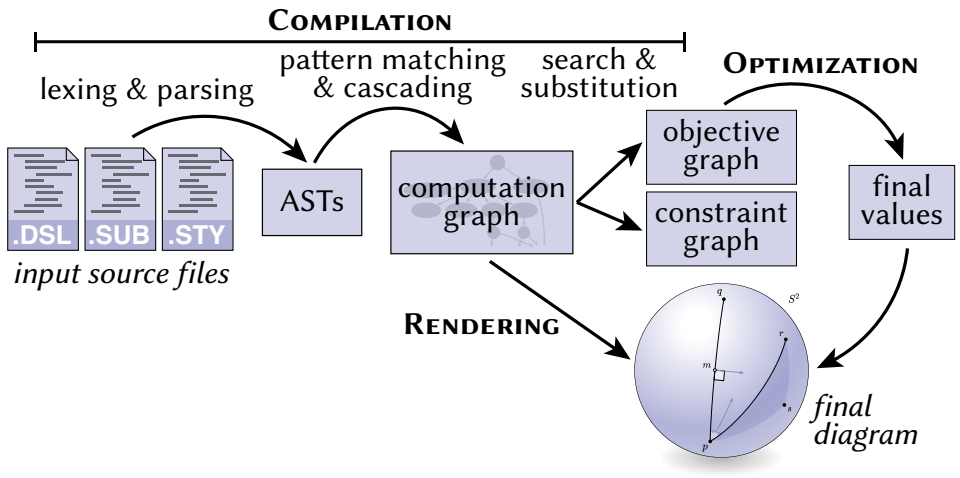


Figure 4.13: Pipeline view of the layout engine. Rather than a single static image, compilation yields an optimization problem that can be solved and re-solved to produce many diagrams, or (in principle) used in an interactive tool.

to render the final diagram (Section 4.4.4). For simplicity, the goal is to automatically produce one static diagram, but the same pipeline could be extended to support capabilities like interaction.

4.4.1 Compiler

The input to the compiler is a triple of files: a DOMAIN schema with SUBSTANCE and STYLE programs. The output is a constrained optimization problem, expressed as a computational graph.

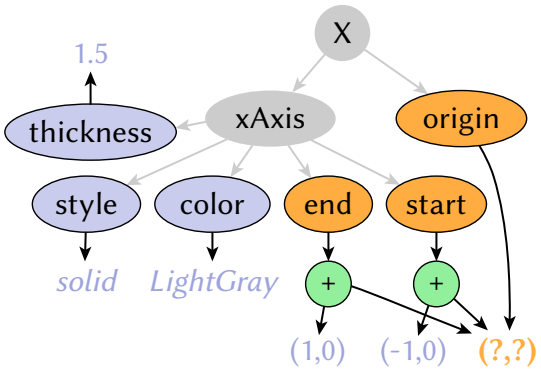
Parsing and Type Checking

We parse each of the input files into abstract syntax trees (ASTs), applying static typechecking to ensure that types are well-formed and variables are well-typed. We first typecheck the DOMAIN program since it defines the valid types for the SUBSTANCE and STYLE programs, then use these types to check the SUBSTANCE program and the selectors in the STYLE code.

Computational Graph

The ASTs are combined to define a *computational graph* that encodes operations that define the final diagram (Figure 4.14). To build this graph, we apply a standard pattern matching and cascading procedure: iterate over rules in the STYLE program, find tuples of SUBSTANCE variables that match the selector pattern, then modify the graph according to the declarations within the matched rule. For example, when the first selector `VectorSpace U` from Figure 4.12 matches the variable `X` from Figure 4.11, we add nodes to the graph that encode the axes of this vector space. In general, declarations could also remove nodes from the graph or connect previously added nodes. Once this transformation is complete, we have replaced all abstract mathematical descriptions with concrete graphical representatives. All that remains is to determine pending values (i.e., those marked by a `?`) and those values that depend on them, which will be done by the solver.

Figure 4.14: Applying the mapping defined by STYLE code to a SUBSTANCE program yields a graph that describes how to draw the diagram—here, for part of Figure 4.11. Some values are known (in blue), whereas others (in orange) depend on unknowns that must be determined via optimization.



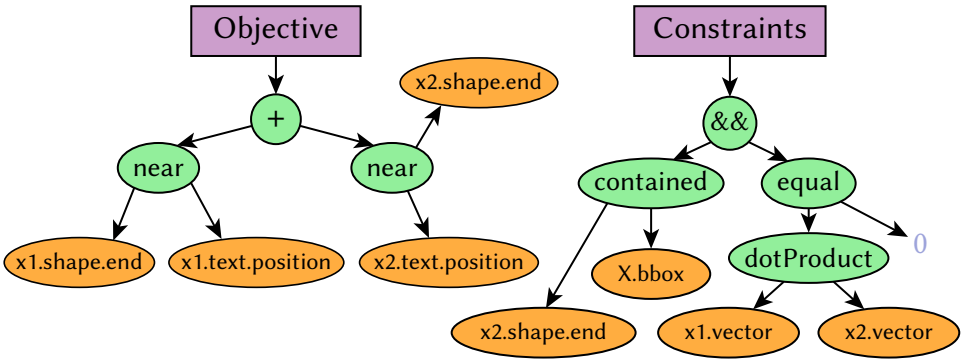


Figure 4.15: The computation graph is further expanded to produce graphs representing the objective and constraint space for our optimization problem. From there, we can easily use automatic differentiation to obtain derivatives. This figure depicts part of the optimization graph for Figure 4.11.

Optimization Graphs

To encode the optimization problem, we collect terms from the computational graph into an objective and constraint graph (Figure 4.15). Each `ensure` and `encourage` statement is then replaced by the corresponding mathematical expression. For instance, `ensure equal(x,y)` is translated into the constraint $x - y = 0$, which the solver seeks to enforce exactly, whereas `encourage equal(x,y)` becomes the objective $(x - y)^2$, which the solver seeks to minimize as much as possible. The overall constraint set is the intersection of all constraints, and the overall objective is a sum of objective terms. Currently PENROSE provides a fixed set of constraints and objectives, though it would be straightforward to extend STYLE to allow user-defined inline expressions.

4.4.2 Solver

The optimization graphs produced by the compiler describe an optimization problem in standard form, i.e.,, minimization of an objective function subject to equality and inequality constraints [24, Section 4.1]. Such problems may be solved with many standard methods. We currently use an *exterior point method* [75] that starts with an infeasible point and pushes it toward a feasible configuration via progressively stiffer penalty functions—mirroring a process often used by hand (Section 4.2.2). Moreover, the exterior point mehod is an appropriate choice since (i) a

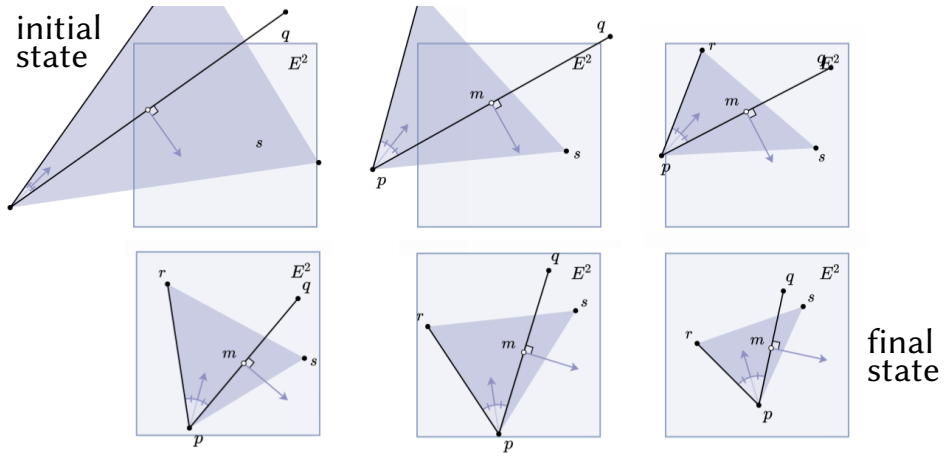


Figure 4.16: Our solver can lay out diagrams even if we do not initially know how to satisfy all the constraints. Here we show several steps of optimization.

feasible starting point is typically not known (Figure 4.16), and (ii) by converting constraints into progressively stiffer penalty functions, we can use descent algorithms that do not directly support constrained optimization. In particular, we use L-BFGS with a line search strategy suitable for nonsmooth objectives [103]. Given the rich structure of our optimization graphs, which can be linked back to program semantics, there are plenty of opportunities to improve this generic strategy, such as decomposing the problem into smaller pieces that can be independently optimized, or employing an SMT solver to find a feasible initial state.

Initialization

Just as a human diagrammer might consider several initial arrangements, we randomly sample several configurations and optimize only the most promising ones, i.e.,, the ones with the least overall energy in the exterior point problem. Initial values are sampled uniformly at random from a range related to their types; for example, RGB color values are sampled from $[0, 1]$.

Failures and warnings

Since our language framework is quite general, a programmer might define difficult or impossible optimization problems. Hence, we can't guarantee that PENROSE produces a valid diagram. However, the system can provide feedback by simply printing an error message if any of the constraint values are nonzero. The resulting invalid diagram might even provide useful visual intuition for why the STYLE program failed (Figure 4.3).

4.4.3 Plugins

A plugin is a piece of external code, written in any language, that is given information from a specific pair of SUBSTANCE and STYLE files and can produce more SUBSTANCE and STYLE information in specific files for PENROSE to use. A plugin is run when making diagrams with a particular STYLE. A STYLE may declare the plugins to be called at the top of the file with

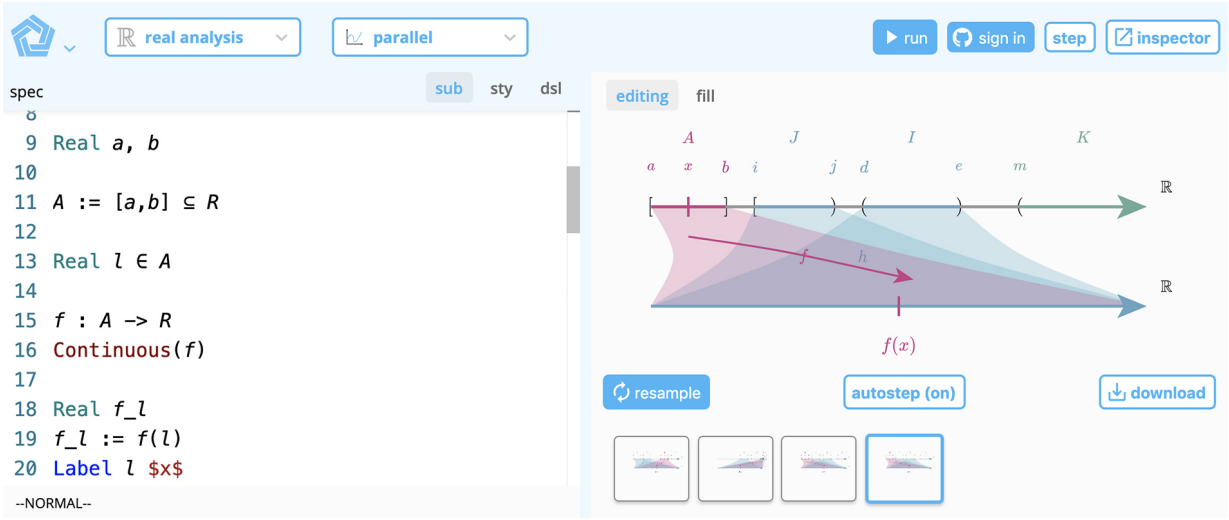


Figure 4.17: Our system supports integration with web-based applications. Here a PENROSE IDE provides automatic syntax highlighting and autocomplete for any user-defined domain.

the syntax `plugin "myPlugin" (args)`, which states that the plugin `myPlugin` should be run with the given argument list. When a diagram is generated, the plugin is given the SUBSTANCE program as a JSON file, as well as the parameters given in STYLE as command-line arguments. The plugin can output new SUBSTANCE code as a text file and/or a set of values for the fields of any SUBSTANCE variable, encoded as a JSON file. The SUBSTANCE code generated by a plugin is appended to the existing SUBSTANCE program, and the values generated by the plugin can be accessed in STYLE via the syntax `myPlugin[variable][field]`. Note that a plugin is run exactly once, prior to execution of all PENROSE code. Therefore, the values generated by a plugin are not optimized by the layout engine, so plugin code does not have to be differentiable. For examples of plugin use, see Section 4.5.2 and Section 4.5.5.

4.4.4 Rendering

In this chapter we focused on generating 2D vector graphics, but in principle nothing about our system design limits us to this particular target. For instance, the constraint-based approach is just as suitable for, say, generating arrangements of 3D objects that can be rendered via photorealistic ray tracing [137], or even constrained interactive diagrams that could be used in virtual reality. In our current implementation, graphical primitives are translated to SVG-native primitives via *React.js* [54] and labels are postprocessed from raw TEX to SVG paths using MathJax [34]. Since PENROSE code is typically quite concise, we embed it as metadata into the SVG, easing reproducibility. We also embed SUBSTANCE names as tooltips to improve accessibility.

4.4.5 Development Environment

To facilitate development, we built a web-based IDE (Figure 4.17) that highlights the potential for high-level diagramming tools built on PENROSE. For instance, since the DOMAIN grammar has a standard structure, the IDE can provide features like autocomplete and syntax highlighting for

any domain. We are optimistic that the design choices made in Section 4.2 will support the use of PENROSE as a platform for building diagramming tools beyond the use cases in this chapter.

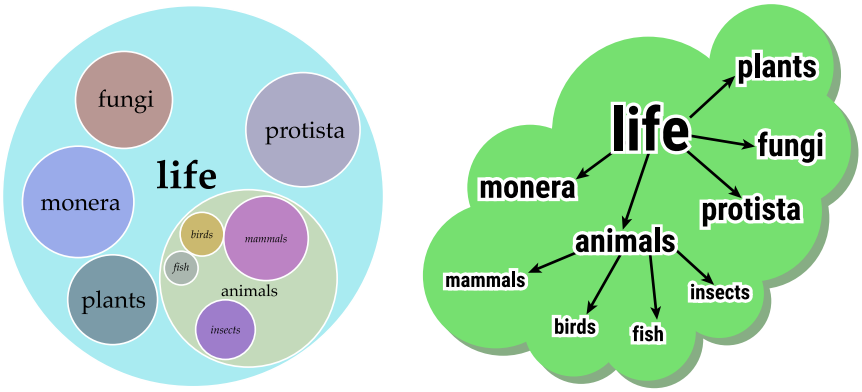
4.4.6 Implementation

The PENROSE system is written in Haskell and the rendering frontend is written in Typescript. We wrote our own solver using the Haskell library *ad* [91] to perform automatic differentiation. We provide one output target and renderer (SVG), together with a fixed set of graphical primitives that are loosely based on SVG (e.g., circles and paths), plus features that SVG users commonly add by hand, like arrowheads. We also provide a fixed set of objectives and constraints for specifying spatial layout, such as shape containment and adjacency relationships, and other functions for performing spatial queries, such as computing bounding boxes and pairwise distances. Sustained use by a community of users might point the way to a standard library. The system has been open-sourced here: `github.com/penrose/penrose`

4.5 Examples and Evaluation

Our main goal for PENROSE was to create a system that can automatically generate diagrams from many different domains using familiar syntax. Here we examine our design by exploring examples from a variety of common domains in mathematics and computer graphics; we also do some basic performance analysis (Section 4.5.8).

4.5.1 Sets



A simple example that illustrates many principles of our system design is the domain of basic set theory—Figure 4.18 shows a complete listing for one of three possible styles. Notice here the complete absence of explicit coordinates in both the SUBSTANCE and STYLE code. The other two STYLE programs (not shown) either improve the visual styling, or shift to a different representation where subset inclusion is indicated by a tree-like drawing rather that overlapping disks. Different representations are especially helpful for different types of examples—for instance, disks must shrink exponentially for deeply nested subsets, whereas a tree diagram remains easy to read (see inset, *left*). One limitation highlighted by this example is that the constraints and objectives appearing in STYLE are not yet extensible within the language itself—for instance, the statement

`ensure contains(y.shape,x.shape)` translates to a fixed function on the centers and radii of the two disks.

This example also demonstrates the benefit of a more explicit type system, rather than, say, interpreting raw mathematical strings as in TEX. In particular, Figure 4.9 shows how a DOMAIN schema can be used with program synthesis techniques (Section 4.5.7) to automatically enumerate different *logical* instantiations of the given SUBSTANCE code. To make this example, there was no need to model sets as an explicit datatype (e.g., a list of points) nor to assign semantics to these datatypes (such as the impossibility of two sets being both intersecting and nonintersecting). Instead, the program synthesizer can reason purely about the abstract types specified in the DOMAIN schema, letting the constraints defined in the STYLE define the visual semantics. Thus, the program synthesizer can check if the generated code is valid by simply testing if the constraints defined in STYLE all evaluate to zero for the optimized diagram. This example captures an important aspect of our system design: the mapping defined by STYLE programs not only provides a superficial visual interpretation, but also assigns deeper mathematical meaning.

4.5.2 Functions

A natural concept to build on top of sets is mappings between sets. This example also illustrates the use of plugins (Section 4.4.3). We first add a `Map` type to the DOMAIN for sets (Section 4.5.1), as well as a constructor `From: Set * Set -> Map` specifying the domain and codomain of the map. Here, syntactic sugar

```
notation "f: A -> B"   "Map f; From(f, A, B)"
```

enables one to both declare and define the map via the concise, familiar notation `f: A -> B`. In Figure 4.19 we add predicates `Injection`, `Surjection`, and `Bijection` to the DOMAIN schema to illustrate some basic ideas about maps. The two different styles of illustration help ease the transition from thinking of mappings between discrete points to thinking of continuous mappings on the real line. To generate the discrete examples, we wrote a simple plugin (Section 4.4.3) that acts as "glue" between PENROSE and an external SMT solver (another example is shown in Figure 4.20). The compiler uses this plugin to expand the `Map` objects from Figure 4.19, *top* into specific instances of a new `Point` type, as well as a new predicate `(a, b) ∈ f` that expresses a map as an explicit list of domain/codomain pairs. For instance, the map in Figure 4.19 generates points `Point A0, A1, B0, B1, B2` with the two predicates `(A0, B1) ∈ f` and `(A1, B2) ∈ f`. A STYLE tailored to these types is used to generate diagrams in Figure 4.19, *left*; as in Figure 4.8, hue is optimized to enhance contrast between nearby points. In contrast, the continuous function diagrams in Figure 4.19, *right* do not require an external plugin, but instead constrain the degrees of freedom of a Bézier curve. Finally, Figure 4.20 shows how abstract function composition in SUBSTANCE is automatically translated into explicit composition of generated functions by the STYLE program without any SUBSTANCE writer effort.

4.5.3 Geometry

Classical geometric figures provide a good opportunity to examine how one can use different STYLE programs to change not only the superficial style of a diagram, but also its fundamental

```
type Set                                        -- Sets.dsl
predicate Intersecting : Set s1 * Set s2
predicate IsSubset : Set s1 * Set s2
predicate Not : Prop p
notation "A ⊂ B" ~ "IsSubset(A, B)"
notation "A ∩ B = ∅" ~ "Not(Intersecting(A, B))"
```

```
Set A, B, C, D, E, F, G                 F ⊂ C       -- Sets.sub
B ⊂ A                                   G ⊂ C
C ⊂ A                                   E ∩ D = ∅
D ⊂ B                                   F ∩ G = ∅
E ⊂ B                                   B ∩ C = ∅
```

```
forall Set x {                          -- Sets-Disks.sty
    x.shape = Circle { strokeWidth : 0.0 }
    x.text = Text { string : x.label }
    ensure contains(x.shape, x.text)
    encourage sameCenter(x.text, x.shape)
    layer x.shape below x.text
}
forall Set x; Set y
where IsSubset(x, y) {
    ensure contains(y.shape, x.shape)
    ensure smallerThan(x.shape, y.shape)
    ensure outsideOf(y.text, x.shape)
    layer x.shape above y.shape
    layer y.text below x.shape
}
forall Set x; Set y
where NotIntersecting(x, y) {
    ensure disjoint(x.shape, y.shape)
}
```

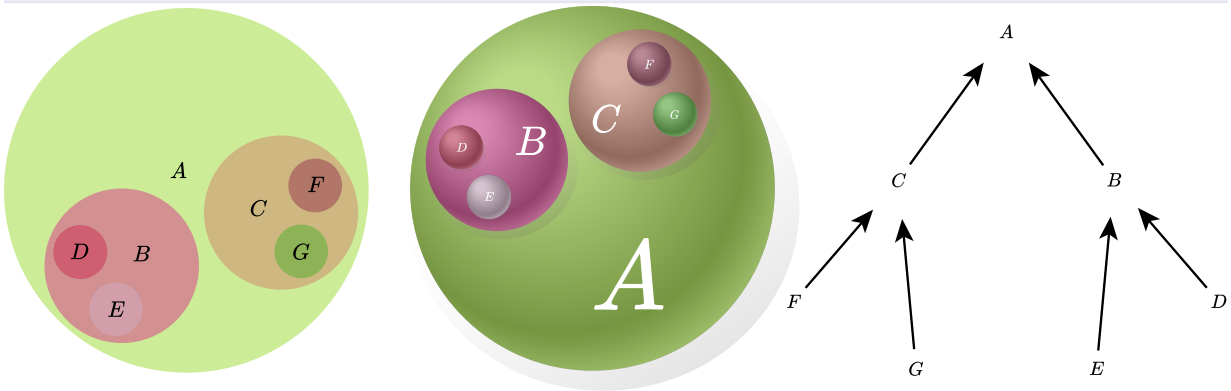


Figure 4.18: Here, some SUBSTANCE code is used to specify set relationships. Different STYLE programs not only tweak the visual style (e.g., flat vs. shaded disks), but allow one to use a completely different visual representation (e.g., a tree showing set inclusions). `Sets.sty` above describes the flat disk style.

```
-- Injection.sub
Set A, B
f: A -> B
Injection(f)
Not(Surjection(f))
```

```
-- Surjection.sub
Set A, B
f: A -> B
Surjection(f)
Not(Injection(f))
```

```
-- Bijection.sub
Set A, B
f: A -> B
Surjection(f)
Injection(f)
```

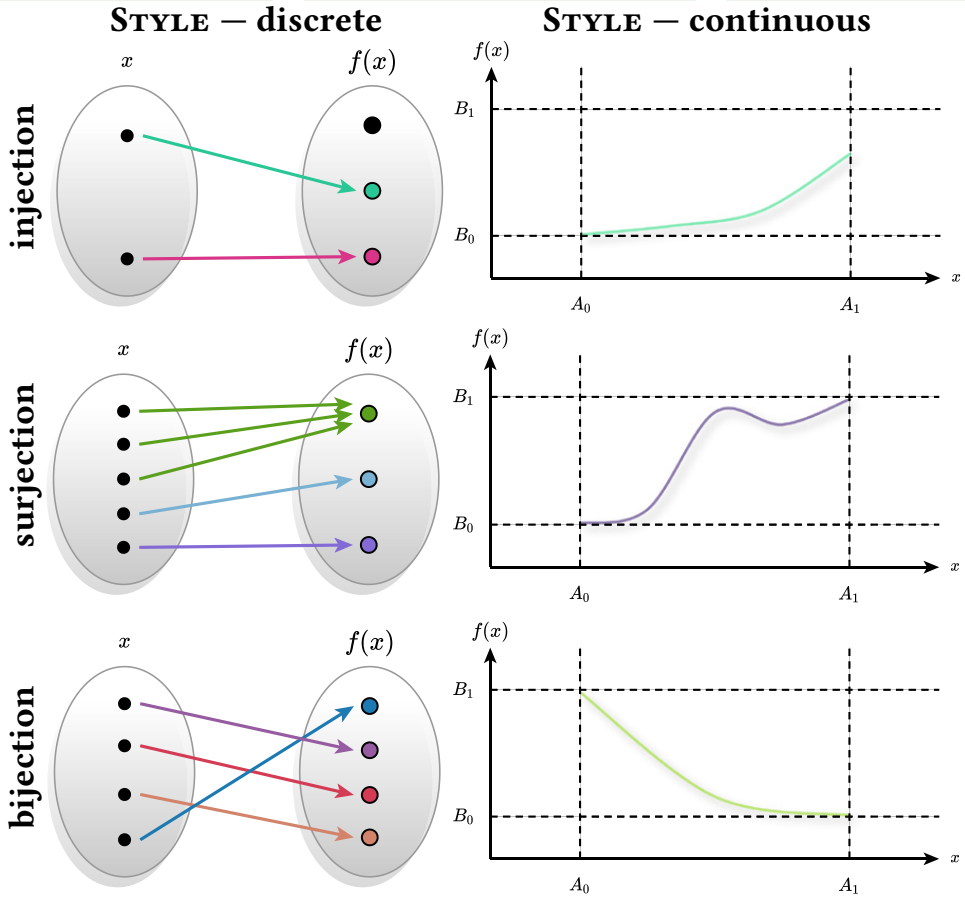


Figure 4.19: Different visual representations provide different ways of thinking about an idea. Here, the notion of injections, bijections, and surjections is illustrated in both discrete *(left)* and continuous *(right)* styles. In the former, functions with the desired properties are randomly generated by an SMT solver, allowing the user to learn from many different examples.

```
Set A, B, C
f: A -> B
g: B -> C
Injection(f)
Bijection(g)
Function gf = g(f)
```

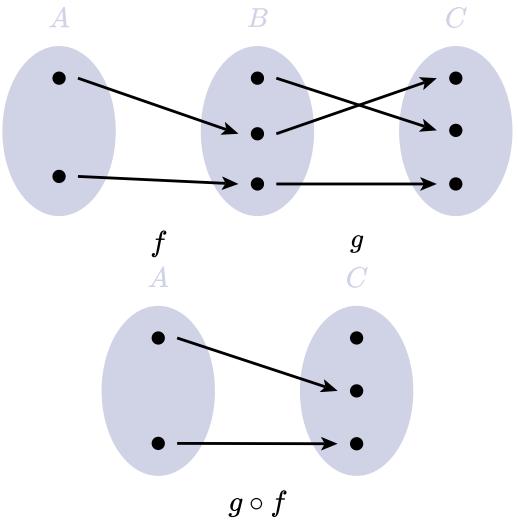
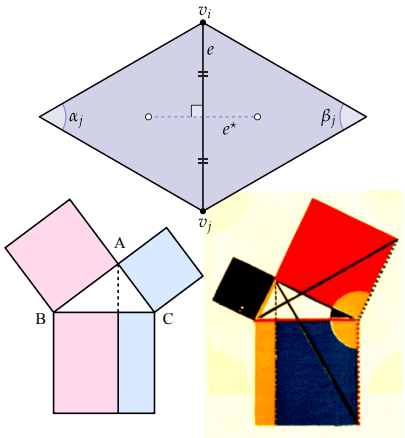


Figure 4.20: Here, abstract function composition is realized as explicit composition of functions produced via an SMT solver, illustrating the fact that the composition of an injection and a bijection is an injection.

visual representation. The familiar "two-column proof" exemplifies how, in mathematics, one can make geometric statements without referring to explicit quantities like coordinates and lengths. Likewise, compass-and-ruler constructions (dating back to the ancient Greeks) show how geometric figures can be specified with only relative constraints. These modalities are well-captured in the way we write SUBSTANCE and STYLE code for geometric diagrams, respectively. For instance, Figure 4.21, *top* gives a listing of geometric assertions that resemble the left column in a two-column proof. This would likely be a natural notation even for intermediate-level students. A bare-bones STYLE program for this domain (not shown) comprises basic statements very similar to those used in Figure 4.12, e.g., to express the orthogonality of two segments. (This approach is similar to the special-purpose geometry package *GCLC* [84]; however, here the domain of objects and visual representations are both extensible at the language level.)



Diagrams used as inspiration for the STYLEs in Figure 4.21.

One goal of PENROSE is to codify the subjective style choices made by professional illustrators so non-expert users can benefit from their expertise. Figure 4.21, *bottom* cascades on the bare-bones STYLE program to riff on styles from several existing sources (shown in inset), namely, *Byrne's Euclid* [30], *Wikipedia* [40], and a collection of illustrated notes on *discrete differential geometry* [42]. This figure also illustrates how we can "mix and match" different STYLE and SUBSTANCE programs. The bulk of these styles (∼500 lines) share the same baseline Style code; additional code for a specific style requires less than 100 more lines. To illustrate the Pythagorean theorem (right column), we also used cascading to add diagram-specific features (e.g., altitude and extension lines).

In the spirit of Hilbert's quip (Section 4.2), we can also swap out the basic visual representation of a given set of logical statements. For instance, any collection of geometric statements that does not assume the

```
Point A, B, C
-- define a right triangle
Triangle ABC := {A,B,C}
Angle θ := ∠(C,A,B)
Right(θ)
-- square each side
Point D, E, F, G, H, I
Square CBDE := [C,B,D,E]
Disjoint(CBDE, ABC)
Square BAGF := [B,A,G,F]
Disjoint(BAGF, ABC)
Square ACIH := [A,C,I,H]
Disjoint(ACIH, ABC
-- split hypotenuse area
Segment AK := Altitude(ABC,θ)
Point K := Endpoint(AK)
Segment DE := {D,E}
Point L
On(L, DE)
Segment KL := {K,L}
Perpendicular(KL, DE)
Rectangle BDLK := {B,D,L,K}
Rectangle CKLE := {C,K,L,E}
-- (plus additional objects
-- from Byrne's diagram)
```

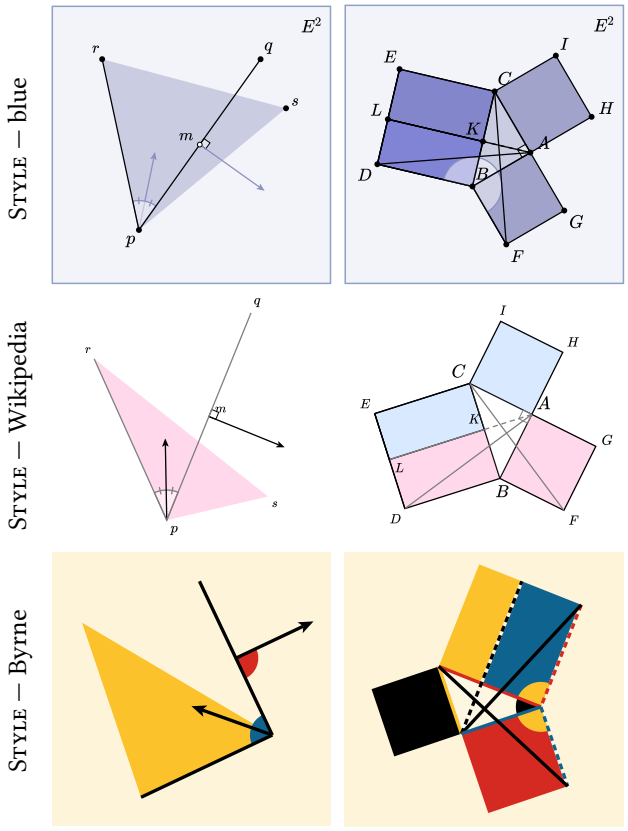


Figure 4.21: The cascading design of Style enables one to modify a base style with relatively little code. Here the two Substance programs from Figure 5.2 and the listing above are visualized in three different styles, all of which build on the same basic constraints and objectives.

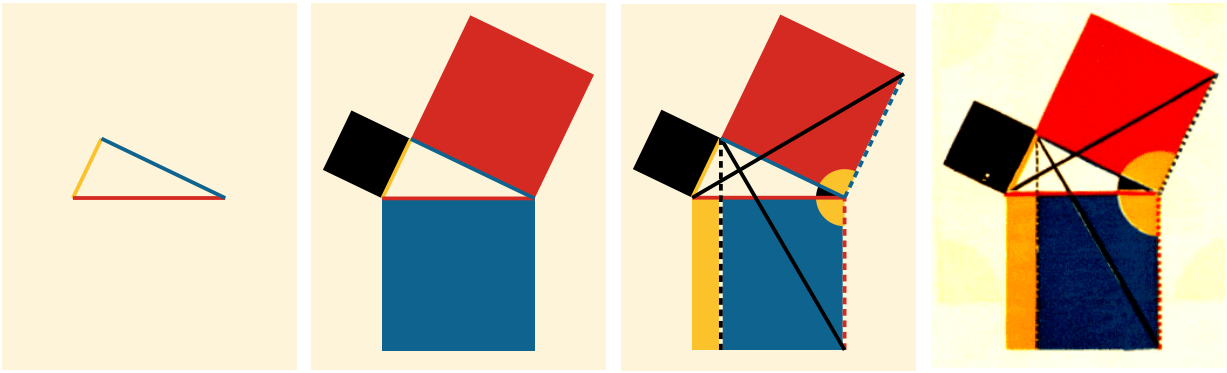


Figure 4.22: Once a complex diagram has been built, it can be easily broken into pieces or stages by, e.g., commenting out lines of SUBSTANCE code. Here we illustrate steps in Euclid's proof of the Pythagorean theorem, turning Byrne's static figure *(far right)* into a progressive "comic strip."

parallel postulate can be realized in several different geometries (Figure 5.2). To generate this figure, we wrote three STYLE programs that all match on the same patterns from a common "neutral geometry" DOMAIN schema. Swapping out these STYLE files then allows users to build intuition about spherical or hyperbolic geometry by exploring how a given figure (expressed via SUBSTANCE) differs from its Euclidean counterpart. Such an example could be further enriched by writing styles for different models of hyperbolic geometry (such as half-plane, hyperboloid, or Beltrami-Klein), each of which involves meticulous calculations.

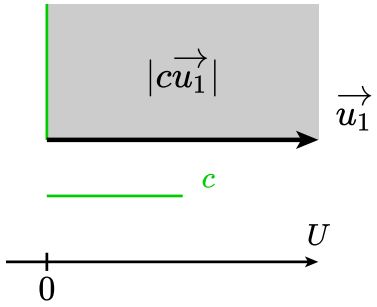
Finally, the diagram specification enables us to build "staged" diagrams, such as ones illustrating the steps of a proof. Figure 4.22 successively uncomments lines of SUBSTANCE code to produce a style of explanation common in textbooks and slides, a strategy which could easily be applied to (say) any two-column proof. In this example, the values of optimized variables are fixed by hand; an interesting question is how this might be done automatically.

4.5.4 Linear Algebra

In mathematics, complexity is built up by composing simple statements. The mapping defined by a STYLE program automatically lifts this *compositionality* to the visual setting. That is, it enables SUBSTANCE writers to compose logical statements to build up visual complexity without explicit effort from the STYLE programmer. A good analogy is procedural *L-systems* [138]. Although a graphics programmer can directly write code to recursively apply spatial transformations, it saves effort to first generate strings in a grammar, then systematically translate these strings into graphical transformations.

In PENROSE, examples from linear algebra demonstrate compositionality. The STYLE declaration on Section 4.3.3 of Figure 4.12 defines the visual icon for a vector (a 2D arrow). Suppose we now want to illustrate *linear maps*, denoted by f, which have two defining properties: linearity of vector addition ($f(u+v) = f(u) + f(v)$ for all vectors u, v) and homogeneity of scalar multiplication ($f(cu) = cf(u)$ for all vectors u and scalars c). Successive STYLE rules cascade on Figure 4.12 to define how these logical operations should be mapped to visual transformations. For example, application of a linear map f is represented by a literal 2×2 matrix-vector multi-

ply; the image vector $f(u)$ also inherits the color of the argument u. The map itself is visually represented by a labeled arrow, and the domain and target spaces by coordinate planes on either side. The STYLE programmer need not compose these directives explicitly; the compiler does the tedious job of translating SUBSTANCE statements (Figure 4.23, *top*) into a composition of graphical statements that define a diagram (Figure 4.23, *bottom*). Moreover, since the STYLE program faithfully represents the underlying mathematics, we observe the expected properties, e.g., the arrow for $f(u_1 + u_2)$ is the same as the arrow for $f(u_1) + f(u_2)$. Automatically checking consistency of the visual representation based on analysis of a richer DOMAIN schema would be an interesting topic for future work.



Finally, the inset (above) shows an alternative representation for vectors and scalars as signed and unsigned quantities (u_1 and c, *resp.*) on the number line. The benefit of a 1D representation is that the remaining dimension can be used to illustrate different concepts, in this case relating the magnitude of a product to an area. The ability to switch between representations can be pedagogically valuable, such as for transitioning from lower to higher mathematics.

4.5.5 Meshes

Polygonal meshes are ubiquitous in computer graphics, but illustrating meshes is often cumbersome due to the large number of elements involved, especially when illustrating meshes by hand or in GUI-based tools. In such cases, PENROSE can be useful not just for making illustrations, but also to inspect and debug user-defined data structures by attaching them to custom visualizations. A simple example is shown in Figure 4.24, where different regions of a mesh are specified via standard operators on a simplicial complex; this diagram also illustrates the definition of the simplicial *link* [18, Section 3.3]. Further examples in Figure 4.25 show how a user can quickly build intuition about this definition by drawing the link of a variety of different mesh subsets.

To make these examples in PENROSE, we follow a pattern similar to the discrete function example (Section 4.5.2): generic mesh objects from the SUBSTANCE code are refined into specific instances of `Vertex`, `Edge`, and `Face` objects by an external plugin, which generates and optimizes a random triangle mesh. Since meshes are randomly generated, the plugin passes a random seed (from its STYLE arguments) to draw different pieces of the same mesh. For this example, we used an existing JavaScript-based mesh processing library [151] that was not designed ahead of time to interface with PENROSE. The benefit of generating these elements at the SUBSTANCE level (rather than returning, say, a static SVG image) is that they can continue to be styled and manipulated within PENROSE; the programmer does not have to edit extra graphics code or keep it compatible with the STYLE program. Likewise, programmers who adopt PENROSE

```
VectorSpace U, V
LinearMap f : U → V
Vector u1, u2, u3 ∈ U
Vector v1, v2, v3, v4 ∈ V
u3 := u1 + u2
v1 := f(u1)
v2 := f(u2)
v3 := f(u3)
v4 := v1 + v2
```

```
VectorSpace U, V
LinearMap f : U → V
Vector u1, u2 ∈ U
Vector v1, v2, v3 ∈ V
Scalar c
u2 := c * u1
v1 := f(u1)
v2 := f(u2)
v3 := c * v1
```

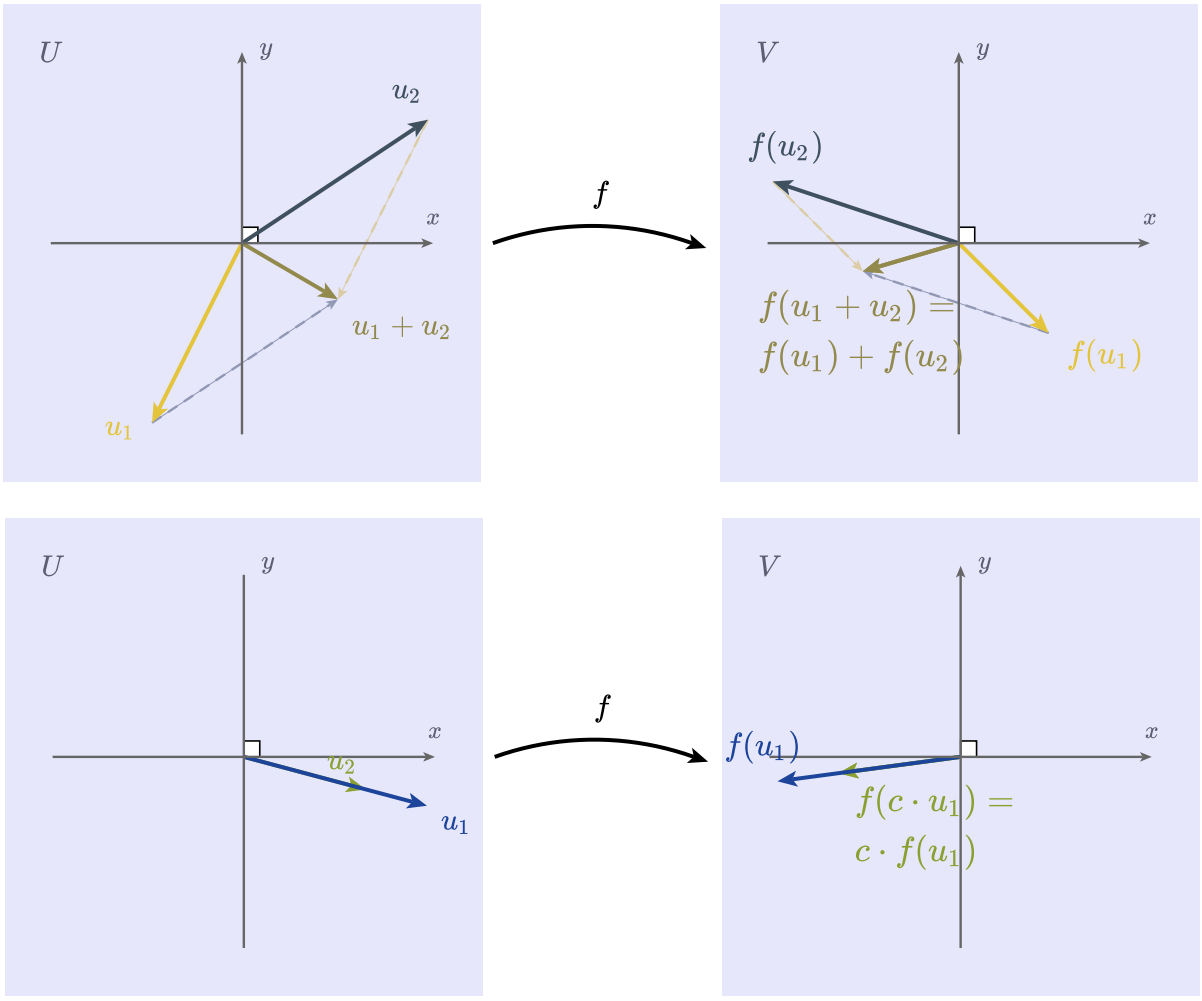


Figure 4.23: Composition of mathematical statements naturally translates into composition of graphical transformations with no explicit programmer effort. Here we compose linear maps, showing addition and scaling, to illustrate the two defining properties of linear maps.

```
SimplicialComplex K                    ClStE := Closure(StE)
Edge e ∈ K                             Subcomplex ClE ⊆ K
Subcomplex E ⊆ K                       ClE := Closure(E)
E := Closure(e)                        SimplicialSet StClE ⊆ K
SimplicialSet StE ⊆ K                  StClE := Star(ClE)
StE := Star(E)                         SimplicialSet LkE ⊆ K
Subcomplex ClStE ⊆ K                   LkE := SetMinus(ClStE, StClE)
```

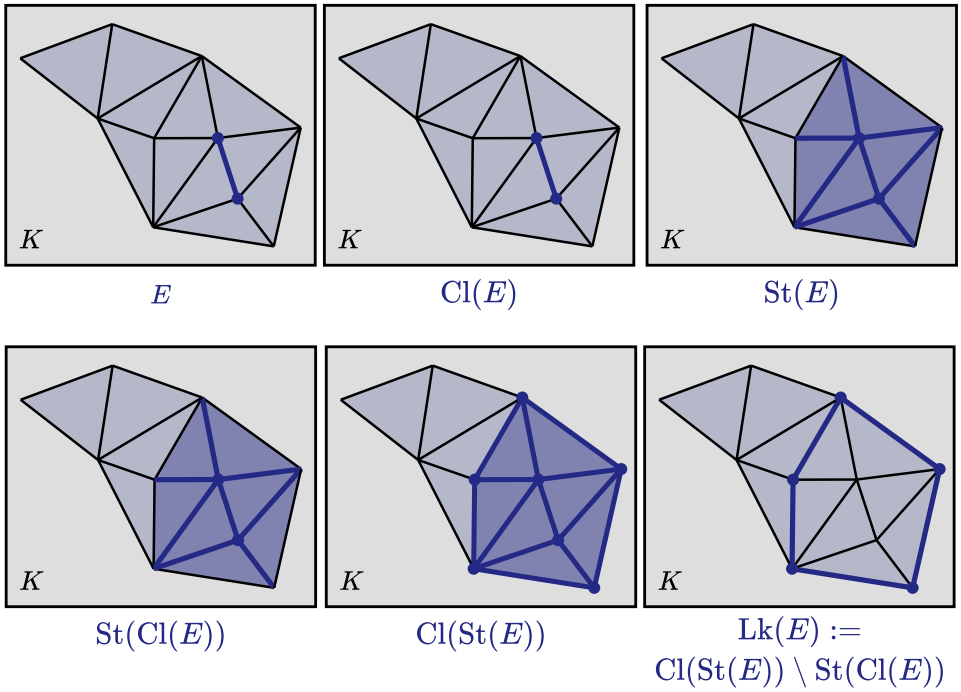


Figure 4.24: A language-based specification makes it easy to visually inspect data structures or assemble progressive diagrams with only minor changes to program code. Here we draw the simplicial *link* by building it up from simpler constituent operations.

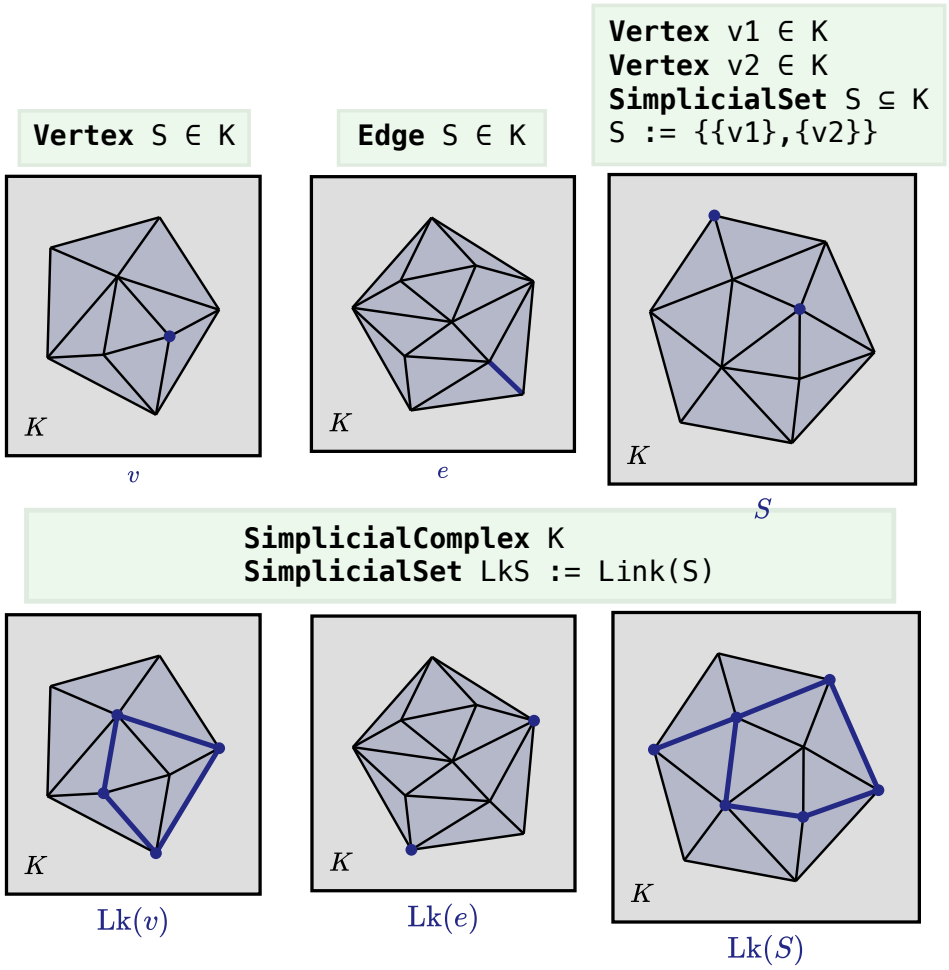


Figure 4.25: Domain-specific notation makes it easy to explore an idea by trying out many different examples. Here several subsets of a simplicial complex are specified *(top)* to explore the definition of the "link" *(bottom)*. An external plugin generates random example meshes, further enriching exploration.

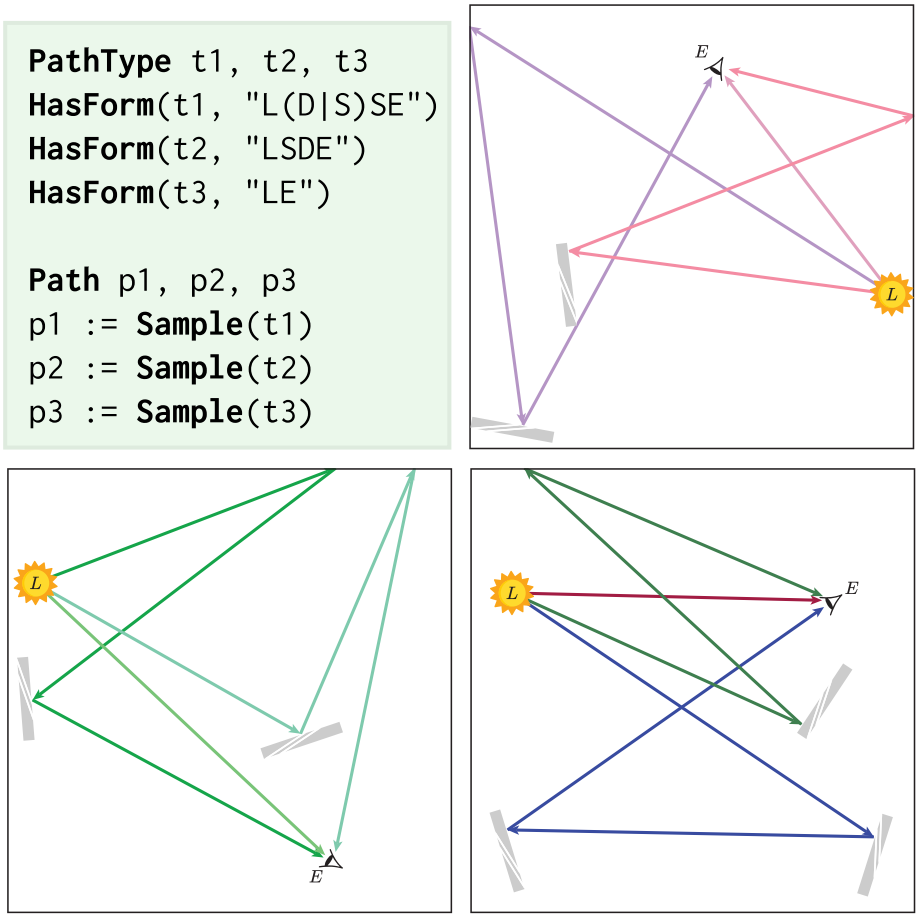


Figure 4.26: When drawing ray tracing diagrams by hand, it can be difficult to construct geometry that permits the desired path types. Here we jointly optimize path and scene geometry to match multiple path types simultaneously. Shown are several diagrams generated for the same program.

as a tool for visual debugging can benefit from system improvements while writing minimal code to attach their data structures to a visual representation.

4.5.6 Ray Tracing

Our final example constructs light path diagrams, which are often used to illustrate ideas in physically-based rendering. The SUBSTANCE code expresses types of light paths via Heckbert's regular expression notation. For instance, the expression `L(D|S)S*E` specifies a family of light paths that start at a light source (`L`), bounce off a diffuse or specular object (`S|D`), then bounce off zero or more specular objects (`S*`), then enter the "eye" or camera (`E`). One or more paths can then be declared that have a given form (Figure 4.26). The challenge in generating a diagram from such a specification is that there must be geometry in the scene that supports the given path(s). For instance, for a fixed eye, light, and mirror, there may simply be no path of the form `LSE`. Rather than meticulously constructing a valid scene by hand, we can use a simple STYLE program to jointly optimize the scene geometry and the light path by specifying constraints such as equality

of incoming and outgoing angles at a specular bounce. The set of objects in the scene is generated by a simple plugin that expands the regular expression into a set of compatible objects (e.g., a mirror for each specular bounce). This plugin also uses the canvas size to choose appropriate scene and path complexity according to the target output device (Figure 4.4). Diagrams for more intricate light transport phenomena could be generated by calling an actual ray tracer (such as *PBRT* [137]) to trace and rejection-sample paths by path type. The added value of generating the final diagrams with PENROSE is that initial path and scene geometry generated by the external code can be further optimized to meet other design goals, such as the canvas size. Additionally, the visual style of a large collection of diagrams (e.g., for a set of course notes) can easily be adjusted after the fact.

In our experience, PENROSE acts as a *nexus* for diagram generation. It connects disparate components, such as language-based specification and ray tracing, into a diagramming tool that provides the system-level benefits described in Section 4.1.

4.5.7 Large-Scale Diagram Generation

One goal for PENROSE is that effort spent on diagramming should be generalizable and reusable (Goal 6). To demonstrate the system's reuse potential, we developed a simple program synthesizer to automatically create any number of diagrams randomly sampled from a domain. Given a DOMAIN program, a STYLE program, and the number of diagrams (n) as input, the synthesizer analyzes the DOMAIN program to find the mathematical constructs in the domain, randomly creates n SUBSTANCE programs that contain these constructs, then compiles, optimizes, and renders the results. Figure 4.9 demonstrates an example of using the synthesizer to "autocomplete" an underspecified SUBSTANCE program by automatically enumerating all possible subset relationships, using information from the DOMAIN schema. Automatically generating diagrams at scale can also help users write better STYLE programs, since synthesizer can "fuzz" the space of possible SUBSTANCE programs to find corner cases.

To stress-test the system's performance and the reusability of STYLE and DOMAIN programs, we randomly generated 2000 SUBSTANCE programs from the sets domain (Section 4.5.1) in the flat disc style. PENROSE was able to create diagrams for all samples. Though 1058 of the 2000 programs had conflicting constraints due to randomness, the solver failed gracefully (as in Figure 4.3) and reached convergence.

4.5.8 Performance Evaluation

We hope to support an iterative workflow where the system's performance does not block the user's creative process. One possible goal is to generate most diagrams within ten seconds, since that threshold is deemed a "unit task" in cognitive science [126] and is about as long as similar authoring processes take, such as building a LaTeX document or compiling code. Even lower latency (< 500 ms) might enable new applications of PENROSE, since this threshold benefits users of data visualization, live programming, and other exploratory creative tools [106].

We have not focused on performance tuning, so a fully-interactive experience is not yet possible with PENROSE. With our current naïve implementation, PENROSE generated 70% of the figures in this chapter in under 10 seconds. However, some figures took significantly longer

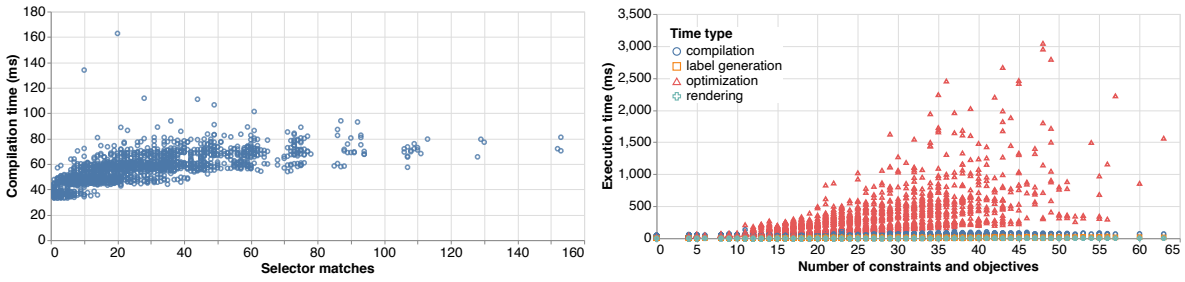


Figure 4.27: We evaluated the performance of the PENROSE compiler by running it on a large collection of programs, showing that the execution time of the compiler grows slowly as the number of selector matches increases *(left)*. To stress-test the system and collect timing information, we generated and visualized random SUBSTANCE programs of different sizes, revealing that optimization dominates the execution time *(right)*.

(e.g., Figure 5.2, Figure 4.6, and Figure 4.21), up to a few minutes. To assess the system's performance, we used diagrams generated in Section 4.5.7 to simulate arbitrary user inputs and measured the time to produce each diagram. To analyze the relative performance of different parts of the system, we separately timed the four stages in the layout engine (Section 4.4): compilation, optimization, label generation, and rendering. Timing was measured on a 2017 MacBook Pro; note that performance in our web-based IDE (Section 4.4.5) is slower due to the cost of editor services and communication with the browser. As shown in Figure 4.27 *(right)*, optimization dominates execution time, though the time to convergence grows slowly with the size of the optimization problem. The second slowest part is the compiler, though Figure 4.27 *(left)* shows that compilation time grows linearly with the number of selector matches, suggesting that the compiler scales well.

We are optimistic about our ability to improve the optimization speed, since we are currently using only a simple, generic solver that we implemented ourselves. (See Section 4.4.2 and Section 4.6 for further discussion.) In our experience, optimization often slowed by objectives that involve all pairs of a large collection of objects, especially for label placement, where all labels "repel" all others. Here one could apply standard acceleration strategies for n-body problems, such as the Barnes-Hut algorithm [8]. Moreover, though it may take time for diagrams to finish, the optimization-based approach provides near-instantaneous *feedback* for most diagrams by displaying the first few steps of the optimization process. These proto-diagrams typically provide enough information about the final layout that the user can halt optimization and continue iterating on the design.

4.6 Discussion and Future Work

Effectively communicating mathematical ideas remains a major challenge for students, educators, and researchers alike. PENROSE provides a step toward understanding the abstractions needed to build general-purpose diagramming tools that connect a concise specification of mathematical content with powerful tools for visual synthesis. In our experience, centering the design around *mappings* from logical to visual objects leads to a system that is both flexible and scalable. Moreover, providing a clean separation between content and presentation lays a foundation for

meaningful interaction techniques for making diagrams.

The system has several limitations that make interesting topics for future work. For example, the DOMAIN, SUBSTANCE, and STYLE languages are limited in what they can express. Thus, SUBSTANCE might be extended with more constructs from mathematical language, such as anonymous expressions, and STYLE might be extended to provide greater flexibility, e.g., via user-defined priorities on objectives. Additionally, the system currently supports only a fixed set of objectives, constraints, functions, graphical primitives, and renderers, as detailed in Section 4.4.6. However, our software architecture does not present major roadblocks to greater extensibility, such as enabling programmers to define constraints inline or emit output for other targets such as 3D or interactive platforms. The system also presents opportunities to study questions of usability, learnability, and debugging, such as the natural way that STYLE users might want to express spatial layout constraints and the learning curve for different classes of users.

The cost of optimization is the biggest bottleneck in our pipeline, as seen in Section 4.5.8, which is not surprising given that we currently use a "catch-all" solver. A nice feature of our design is that the program semantics provide rich information about the structure of the optimization problem. This structure should make it possible to adopt highly effective problem-specific optimization strategies of the kind described in Section 4.2.2 and Section 4.5.8, e.g., analyzing the computation graph to break down the diagram into smaller constituent pieces. Beyond finding local minima, we are excited about different design modalities enabled by exploring the constraint space defined by the STYLE program, such as sampling a diverse set of examples or interpolating between different layouts to create animations.

Finally, there is no reason that the basic design of PENROSE must be applied only to mathematical diagrams. Many other fields, such as law, chemistry, and biology, all deal with non-quantitative information comprised of intricate logical relationships. We believe that an extensible and expressive system for assigning visual interpretations to such information provides a powerful tool for seeing the big picture.

Chapter 5

EDGEWORTH: Diagrammatic Problem Authoring at Scale[1]

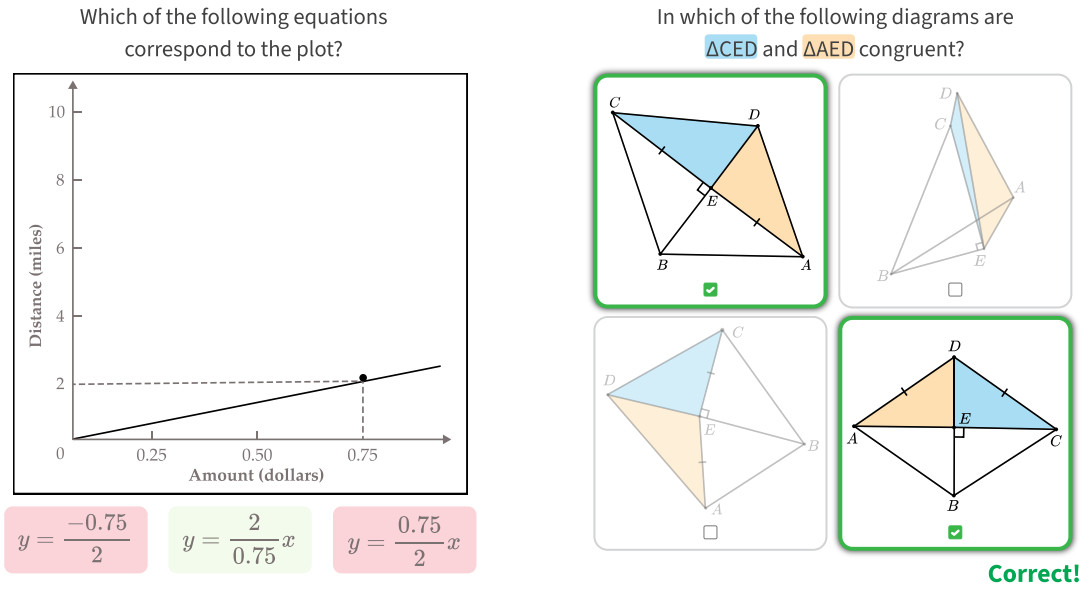


Figure 5.1: left: a translation problem that helps students discern the structure of linear equations (adapted from [88]). **right**: an EDGEWORTH generated problem that trains student to recognize diagram configurations [93] for triangle congruence.

5.1 Introduction

Effective use of visual representations requires a certain level of *representational fluency* that's achievable through deliberate practice and repetition [44, 125]. Recognizing how words, symbols,

[1] This chapter is adapted from Sections 1 to 5 of "Edgeworth: Efficient and Scalable Authoring of Visual Thinking Activities" [127].

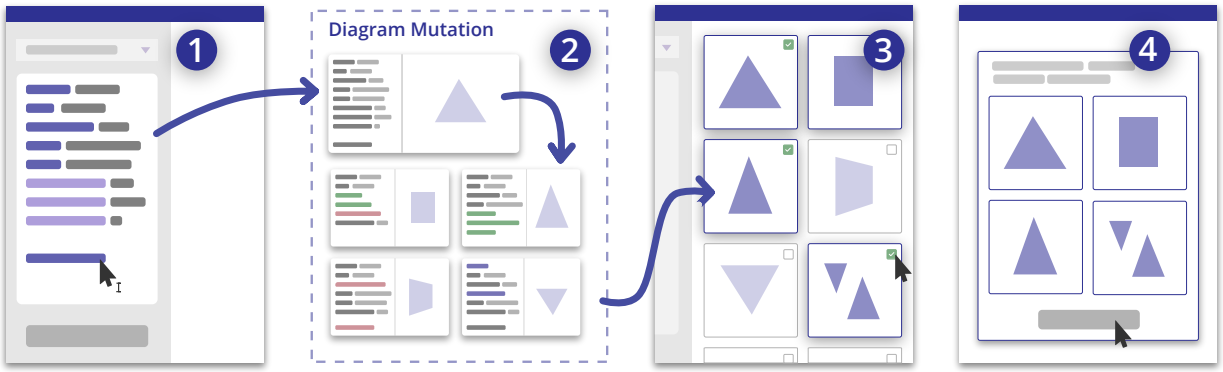


Figure 5.2: EDGEWORTH is a diagrammatic problem authoring tool that automatically generates diagram variations from a single diagram: the author creates an example diagram (1), then EDGEWORTH generates a myriad of diagram variations (2), from which the author selects diagrams (3) to form a diagrammatic multiple choice problem (4).

and diagrams relate to each other is an important first step of achieving fluency. Prior work has shown that these contrasting cases, i.e., discrimination and mapping, among representations significantly improve students' ability to translate among representations [87].

To train students' representational fluency, educators often create problem sets that involve numerous contrasting cases of a particular visual representation. For instance, Figure 5.1 shows two examples of *translation problems*, where the problem asks students to determine diagrammatic *examples* and *counterexamples* of a textual description and vice versa. Importantly, these examples and counterexamples have varying degrees of differences from the given diagram or text, and carefully picking examples on this spectrum has a big impact on learning [109].

Traditionally, educators author visual practice by drawing diagrams by hand. In formative interviews (Section 5.2), educators reported the vital role of visual practice in their instruction, but noted the tedium of authoring due to tool limitations, leading to fewer diagrams used than desired. Manual authoring can hardly keep up with the growth of STEM learners and demand for more visual practice.

As a first step towards scaling up visual practice authoring, we built EDGEWORTH, a diagrammatic problem generator. EDGEWORTH generates *translation problems*, an effective type of visual practice [87] that ask students to determine diagrammatic *examples* and *counterexamples* of a textual/symbolic description (Figure 5.1). To help authors get the most out of one diagram, EDGEWORTH contributes a "build once, generate many" authoring paradigm: Instead of manually editing diagrams to get variations, the author creates a single diagram and EDGEWORTH automatically generates diagram variations (Figure 5.2 1 2). The interaction design of EDGEWORTH allows the author to visually select diagram variations to rapidly form translation problems (Figure 5.2 3 4). Given the diversity of instructional contexts in STEM, we designed EDGEWORTH to be domain-agnostic: it uses a generic program mutation technique (Section 5.3.3) to change the author-provided diagram to produce variations.

In this chapter, we discuss formative interviews that drove the design of EDGEWORTH and then walk through the technical implementation of EDGEWORTH.

5.2 Formative interview

We conducted semi-structured interviews with 6 educators to understand how they author, use, and maintain diagrammatic problems. We recruited participants based on their background in education and usage of diagrams in their work. Selected participants work as secondary school teachers, university professors, teaching assistants, and competitive math coaches. All participants (P1–6) indicated that they have experience creating instructional material, authoring problems, and/or developing online courses that include visual content. Example interview questions include what roles diagrams play in the participant's educational materials, how students interact with diagrams, and how diagrams are authored and maintained.

Participants reported the usage of diagrams to build conceptual understanding and emphasized the need for deliberate practice to acquire representational fluency. Traditional educational materials, especially in higher education, tend to emphasize "*procedures, memorization, and symbolic manipulation*" (P6). Similarly, teachers such as P1 suffer from "*the curse of knowledge*" of teaching visual fluency: while teachers have internalized how to use visuals, they tend to "*under-train*" students and students therefore struggle to use visuals for problem-solving. As a result, students often become "*symbolically good*" and do not develop "*good conceptual understanding*" (P3). Visuals like diagrams and graphs provide alternative representations that help students "*develop intuition*" (P3) and "*become better problem-solvers*" (P4). To improve their instruction, all of our participants (P1–6) attempt to incorporate more diagrams than they currently have in their instructional materials. Some also ask students to draw, annotate, and explain diagrams (P1, P2, P6). P2 encourages students to learn "*multiple representations*" and makes diagrams central to their math and programming curricula. When students practice with diagrams, teachers also gain a richer assessment on students' level of understanding, and "*learned more from this [student-drawn diagram] than 10 similar problems without the pictures*" (P6).

Despite the widespread agreement among participants about the benefits of and need for diagrammatic practice, participants reported that tool limitations led to manual and repetitive authoring experience. Our participants therefore face a trade-off when authoring visual content: more visuals are beneficial for learning but are time-consuming to create and modify. When authoring practice problems, P1 struggled to "*create simple shapes by myself*" and always ended up repeatedly "*copy-pasting and searching online*". Similarly, P6 reported that they "*get online images for pre-made resources, but whenever I want something a little custom, it ' ll take a lot of time.*" To streamline the visual authoring process, P2 and P5 developed custom pipelines for authoring problem sets and quizzes using existing programming tools. Like the problems described by prior research on diagramming tool usability [108], these tools often lack support for "*high-level tweaking of my diagrams*" (P2) and "*are a pain to use because the language is not semantic and hard to use for non-programmers*" (P5). Participants showed us many examples of tedious changes necessary to create diagram variations.

From the results, we derived the following design requirements for tool design to address participants' needs:

D1 Address the need for practicing representational fluency

D2 Simplify the workflow for generating diagram variations

D3 Obviate the need to attend to low-level diagramming details

5.3 System Design of EDGEWORTH

EDGEWORTH realizes the design goals from Section 5.2 by: 1) providing a domain-agnostic workflow for rapidly authoring diagrammatic practice problems (**D1**), 2) automatically suggesting numerous diagram variations of a single example diagram and allowing the author to visually select from the variations (**D2**), and 3) fully automating the layout for all diagram variations (**D3**). Figure 5.3 walks through the user interface of EDGEWORTH, that encapsulates the ideas above.

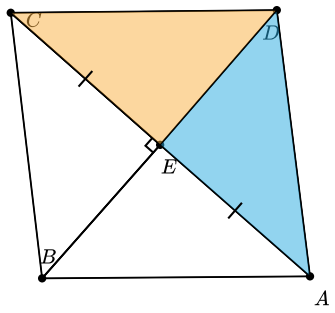
In Section 5.3.1, we demonstrate the workflow of EDGEWORTH by showing how to author an example diagrammatic problem in Euclidean geometry. We then describe EDGEWORTH's approach to diagram layout in Section 5.3.2 and how it generates diagram variation in Section 5.3.3.

5.3.1 Author Workflow

In this section, we use an example from high school geometry to demonstrate the process of creating a problem in EDGEWORTH, the user interface of which is annotated in Figure 5.3.

Create an example diagram

The author wants to write a problem about triangle congruence to assess students' understanding of the *Side-Angle-Side* (SAS) rule. They want to create a translation problem including one diagram where the SAS rule is satisfied and three others where it is not. The author first describes an example diagram (Figure 5.3b) where this rule is satisfied. They construct a scenario involving two triangles: $\triangle DEC$ and $\triangle DEA$ share one side DE and have two equal sides EC and EA. $\angle CEB$ indicates that AC and BD are perpendicular and therefore $\angle DEC = \angle DEA$. Therefore, $\triangle DEC$ and $\triangle DEA$ are congruent by the SAS rule. Given this description, EDGEWORTH lays out the diagram automatically (Figure 5.3f).



Select from EDGEWORTH-generated diagrams

Now the author can use EDGEWORTH to mutate the example diagram by clicking "Generate Variations" (Figure 5.3e). EDGEWORTH performs mutations on the example scenario and generates a grid of diagram *variations*, i.e., diagrams that differ from the example scenario in their content [2]. The grid is designed to give the author an overview of the mutation results, and

[2]Note that variations and distinct from alternate layouts, as they are different SUBSTANCE programs.

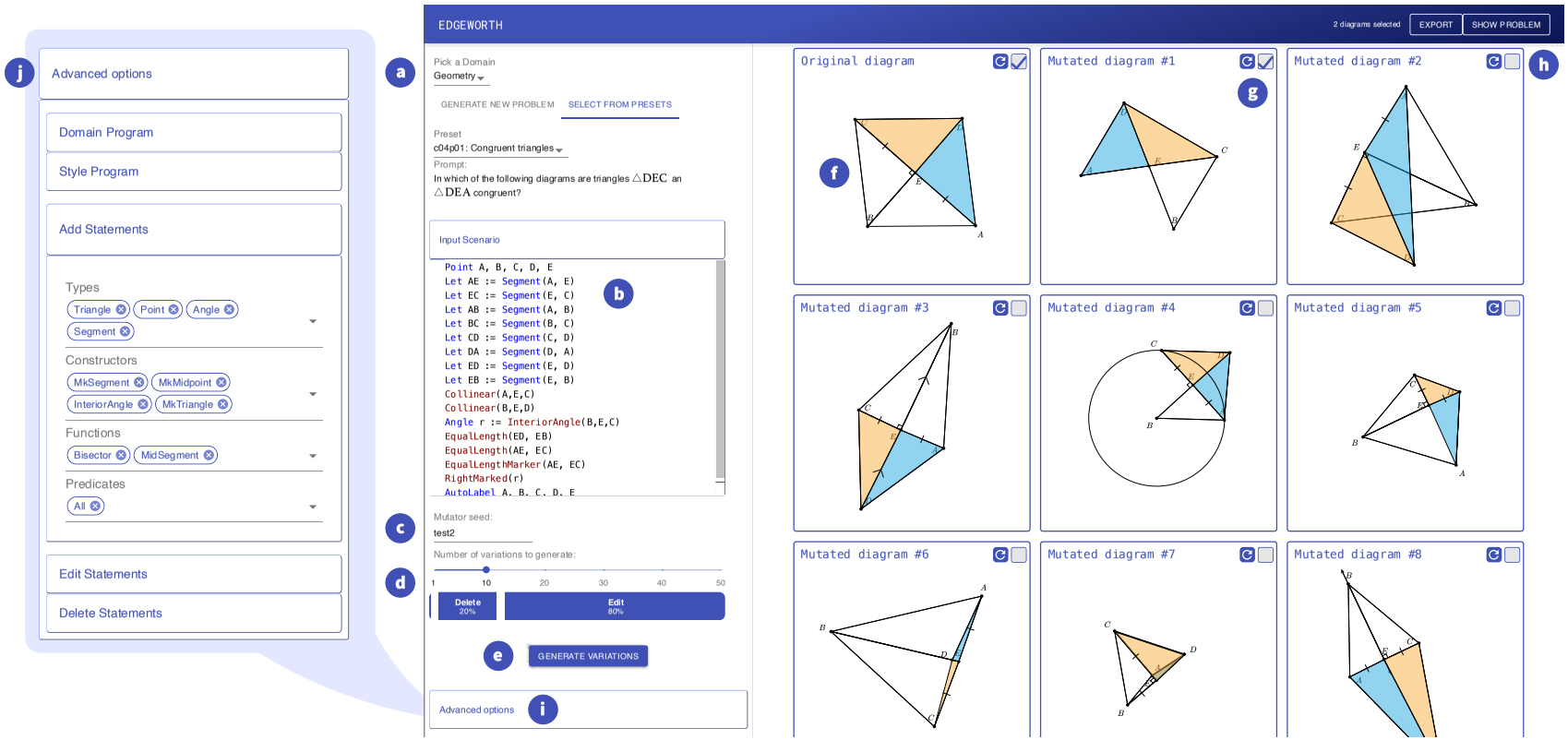


Figure 5.3: The user interface of EDGEWORTH. The author first provides a textual prompt (a) as an input scenario in SUBSTANCE notation (b). Then, clicking "Generate Variations" (e) generates the specified number of diagram variations (d) at random based on a string seed and weights on Add, Delete, or Edit mutations (c). In the diagram panel, the top-left diagram (f) corresponds to the input scenario and the rest are diagram variations generated by EDGEWORTH. The author can visually select diagrams (g) to assemble a diagrammatic multiple-choice problem (h). If needed, the author can fine-tune the mutator using "Advanced options" (i j).

diagrams are prominent in each cell to facilitate faster visual selection. The top-left cell in the grid will always display the original example diagram (Figure 5.3f), and the rest correspond to mutation results.

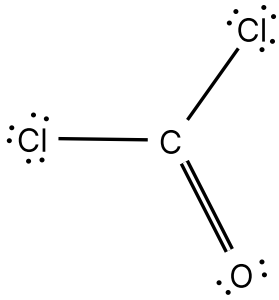
By inspecting each diagram in the grid, the author can determine if it is a good fit for their translation problem. If so, they click the top-right checkbox (Figure 5.3g) to include the diagram in the problem.

Preview and export the problem

After the author picks a sufficient number of diagrams (4 in this case), they can preview the translation problem by clicking "Show Problem" (Figure 5.3h), which displays an interactive multiple-choice widget. If the author is satisfied, they can click "Export" to download the diagrams and metadata to use the problem in their context. EDGEWORTH exports to Scalable Vector Graphics (SVG) images for static media, source programs for interactive use, and detailed mutation trace metadata for comprehensive analysis and reference purposes.

5.3.2 Diagram Notation and Layout

EDGEWORTH is built on PENROSE (Chapter 4). Compared with alternatives, PENROSE offers two distinct advantages: (1) a high-level diagram notation and (2) an automatic layout engine. As discussed in Chapter 4, a diagram in PENROSE consists of a textual description of the diagram content (SUBSTANCE) and a reusable layout stylesheet (STYLE). As a reminder of how the SUBSTANCE language works, Figure 5.4 shows the three kinds of SUBSTANCE statements: type statements (e.g., `Carbon c`) declare new objects; constructors (`Bond b1 := SingleBond(c, cl1)`) create new objects from existing objects; and predicates (`ZeroDots(c)`) indicate relations among objects.




```
Carbon c
Oxygen o
Chlorine cl1, cl2
Bond b1 := SingleBond(c, cl1)
Bond b2 := SingleBond(c, cl2)
Bond b3 := DoubleBond(c, o)
ZeroDots(c)
FourDots(o)
SixDots(cl1)
SixDots(cl2)
```

Figure 5.4: Diagram and SUBSTANCE notation for the Lewis structure of phosgene ($COCl_2$).

The current EDGEWORTH implementation builds on PENROSE's geometry, chemistry, and graph STYLE for diagram layout. Since the existing STYLE stylesheets are primarily used to generate a few human-written examples, they lack coverage for variations of SUBSTANCE descriptions required by EDGEWORTH. To this end, we improved STYLE, made new diagram examples, and added new standard library functions to PENROSE to accommodate EDGEWORTH.

EDGEWORTH is the first application of PENROSE that concurrently optimizes and renders a grid of multiple diagrams. Therefore, we have made significant updates to PENROSE to support EDGEWORTH's use case. To make EDGEWORTH a performant client-side web application for

interactive use, we have migrated from Haskell to TypeScript and made various performance improvements to efficiently run tens of layout optimization jobs in a single session. Compared to the state of PENROSE at the publication of Ye et al. [182], the development of EDGEWORTH has helped improve the performance of the system by 100×.

5.3.3 Program Mutation

EDGEWORTH generates diagram variations by mutating the example diagram written in SUBSTANCE. We purposely designed the system to include a small set of simple mutation operations. Since compilation errors in SUBSTANCE will not produce diagrams, all of the mutations are type-safe. Similar to generic tree-editing algorithms [55], EDGEWORTH supports 3 kinds of mutation operators: **Add**, **Delete**, and **Edit**.

Add appends a statement. **Delete** removes a statement and all other statements that refer to the original statement. **Edit** modifies a statement. There are several categories of **Edit** operations. Each **Edit** category contains a *guard* and an *action*. The guard checks if the operator is applicable to the given SUBSTANCE statement, and the action performs the mutation. For instance, **Replace Arguments** is only applicable when the current context has existing variables of the desired type.

- **Swap Arguments** reorders the arguments passed into a statement; e.g., if `A` and `B` are `Triangle`s:
  `Similar(A, B)` → `Similar(B, A)`
- **Replace Arguments** replaces the arguments passed into a statement with other arguments defined in scope; e.g., if `A, B, C, D` are `Point`s:
  `s := MkSegment(A, B)` → `s := MkSegment(C, D)`
- **Replace Function** replaces a statement with a different statement that takes the same arguments; e.g., if `T` is a `Triangle` and `E` is an `Angle`:
  `Equilateral(T)` → `Scalene(T)`
  `Segment s := Bisector(E)` → `RightAngleMarked(E)`

Algorithm 1 shows how the EDGEWORTH mutator works, at a high level. In addition to the input SUBSTANCE description p, EDGEWORTH also takes a number of user-defined configuration parameters: (1) a number of variations to generate (the number of times GENERATE is called); (2) a range of mutation counts per variation (the input variables ℓ and h), i.e., total number of mutations performed on the given SUBSTANCE description; (3) weights between 0 and 1 for the probabilities of picking **Add**, **Delete**, or **Edit** operations for every mutation (the input variables a, d, and e respectively); and (4) filter sets A, D, and E which limit the set of mutations that the **Add**, **Delete**, and **Edit** operations can produce.

Given an example diagram, EDGEWORTH performs several rounds of mutation generation. Each round results in a series of mutations that alter the input to produce a variation. The number of mutations (line 3) is bounded by the configuration parameters.

To generate a single mutation, EDGEWORTH makes a weighted choice (line 5) of the mutation kinds and enumerates all possible mutations for the chosen kind: **Add** enumerates all possible statements to add (line 7); **Delete** randomly deletes an existing statement (line 9); **Edit** enumerates all possible edits for all statements (line 11) and picks one of them randomly (line 12). The

Algorithm 1 The EDGEWORTH mutation algorithm.

```
1:  function GENERATE(p, ℓ, h, a, d, e, A, D, E)
2:      p′ ← p
3:      n ← uniform random integer between ℓ and h
4:      for i from 1 to n do
5:          x ← uniform random real between 0 and a + d + e
6:          if x < a then
7:              m ← RANDOMADD(A, p′)
8:          else if x < a + d then
9:              m ← RANDOMDELETE(D, p′)
10:         else
11:             s ← uniform random element of STATEMENTS(p′)
12:             m ← RANDOMEDIT(E, s)
13:         end if
14:         p′ ← MUTATE(p′, m)
15:     end for
16:     return p′
17: end function
```

randomness of EDGEWORTH is controlled by a single random generator seed.

Users can specify filter sets under the "Advanced options" section of the UI, shown in Figure 5.3 i j. The filters default to "All," which indicates that the mutator may change any statement in the example diagram. While this precise configuration may be useful, we ended up not using them in our evaluation (Chapter 6) and instead achieving our results using only EDGEWORTH's simpler core set of configuration options, i.e., weights on mutation operators.

5.4 Limitations

5.4.1 Domains of instruction

As shown in Section 5.3.3, the design of the EDGEWORTH mutator is domain-agnostic, as the mutation operators do not require any domain-specific knowledge to produce mutants. However,improving STYLE requires domain expertise. Therefore, future EDGEWORTH authors may not have the technical background or the time to invest in a new STYLE program, which might prohibit them from using EDGEWORTH if the domain is not well supported by the PENROSE ecosystem. However, as noted in Chapter 4, the effort to build PENROSE stylesheets for new domains is only necessary once per domain and not once per diagram or problem.

5.4.2 Numerical and textual variations

EDGEWORTH cannot produce numerical and textual variations like traditional problem generators [2, 142] do. It is, however, possible to build this functionality on top of EDGEWORTH to

produce further problem variations.

5.4.3 Usability of UI components

The design presented in Section 5.3 focuses on generating diagram variations and selecting diagram mutants to create problems. We use the SUBSTANCE language and PENROSE's textual interface without modification. Any limitations of SUBSTANCE and its UI are inherited by EDGEWORTH. We use standard Material UI elements[3] to allow users to configure EDGEWORTH (e.g., a standard text box for changing diagram variations in Figure 5.3C). While these components might be usable as-is, they are not designed explicitly for the problem authoring workflow.

5.5 Translation Problem Dataset

EDGEWORTH's mutation-based approach is domain-agnostic: it simply applies generic program mutations on any SUBSTANCE program. Through collecting a dataset of translation problems in Euclidean geometry, general chemistry, and discrete mathematics, we evaluate if this approach is expressive enough for different instructional contexts in STEM. We selected these three domains because they have wide audiences in K-12 and higher education, making them rich sources for existing instructional material. Each domain also has a canonical visual representation that is explicitly taught to students. Therefore, students can benefit from visual practice in these domains.

We choose problems from existing textbooks or online courses and follow the procedure outlined in Section 5.3.1 to recast each problem.

5.5.1 Summary Statistics

We reproduced 31 problems in total. Since creating the example diagram (Section 5.3.1) took the most time in this process, we report statistics on the example diagrams here.

On average, EDGEWORTH's diagram notation is compact and simple. The description for example diagrams are 14.7 lines of code ($\sigma = 4.57$) and 109.9 tokens ($\sigma = 48.6$). In contrast, the average SVG source of these same diagrams have 454.7 lines of code ($\sigma = 184.3$) and 1290.4 tokens ($\sigma = 650.4$). This indicates that EDGEWORTH provides a concise and compact textual representation of diagrams across all three domains.

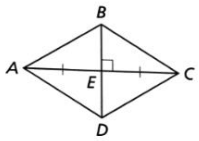
5.5.2 Euclidean Geometry

We sample 17 Euclidean geometry problems from Holt *Geometry* [27], a high school geometry textbook. Figure 5.5 (left) shows an example problem. The textbook uses a consistent visual style of predominantly black line segments and dots with text labels. Most diagrammatic problems are presented as one diagram followed by one or more multiple-choice problems. We've recast the problems as diagrammatic translation problems.

For this domain, we build on the existing geometry stylesheet from PENROSE [182, Section 5.3] for diagram layout. There are many different types of entities in geometry, and we found

[3] https://mui.com/

Use the diagram for Items 1 and 2.



1. Which of these congruence statements can be proved from the information given in the figure?

   (A) $\triangle AEB \cong \triangle CED$ (C) $\triangle ABD \cong \triangle BCA$

   (B) $\triangle BAC \cong \triangle DAC$ (D) $\triangle DEC \cong \triangle DEA$

2. What other information is needed to prove that $\triangle CEB \cong \triangle AED$ by the HL Congruence Theorem?

   (F) $\overline{AD} \cong \overline{AB}$ (H) $\overline{CB} \cong \overline{AD}$

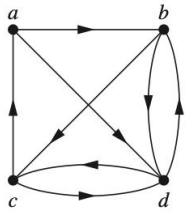
   (G) $\overline{BE} \cong \overline{AE}$ (J) $\overline{DE} \cong \overline{CE}$

Choose the correct Lewis structure for HCN.

- ○ H – C = N
- ○ H = C = N
- ◉ H – C ≡ N
- ○ H – C – N

In Exercises 18–23 determine whether the directed graph shown has an Euler circuit. Construct an Euler circuit if one exists. If no Euler circuit exists, determine whether the directed graph has an Euler path. Construct an Euler path if one exists.

18.



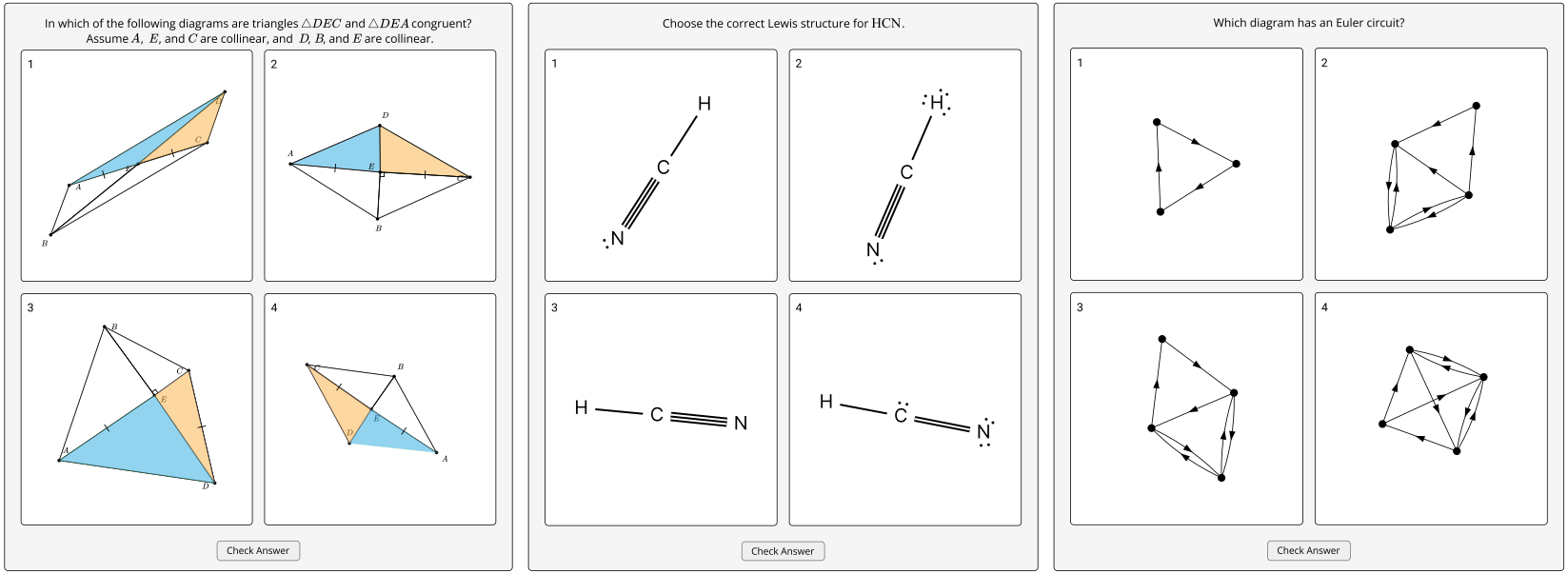


Figure 5.5: We used EDGEWORTH to recast real-world problems as diagrammatic translation problems. Left: Determine if triangles are congruent. Middle: Identify the correct Lewis structure for hydrogen cyanide. Right: Identify graphs with Euler circuits.

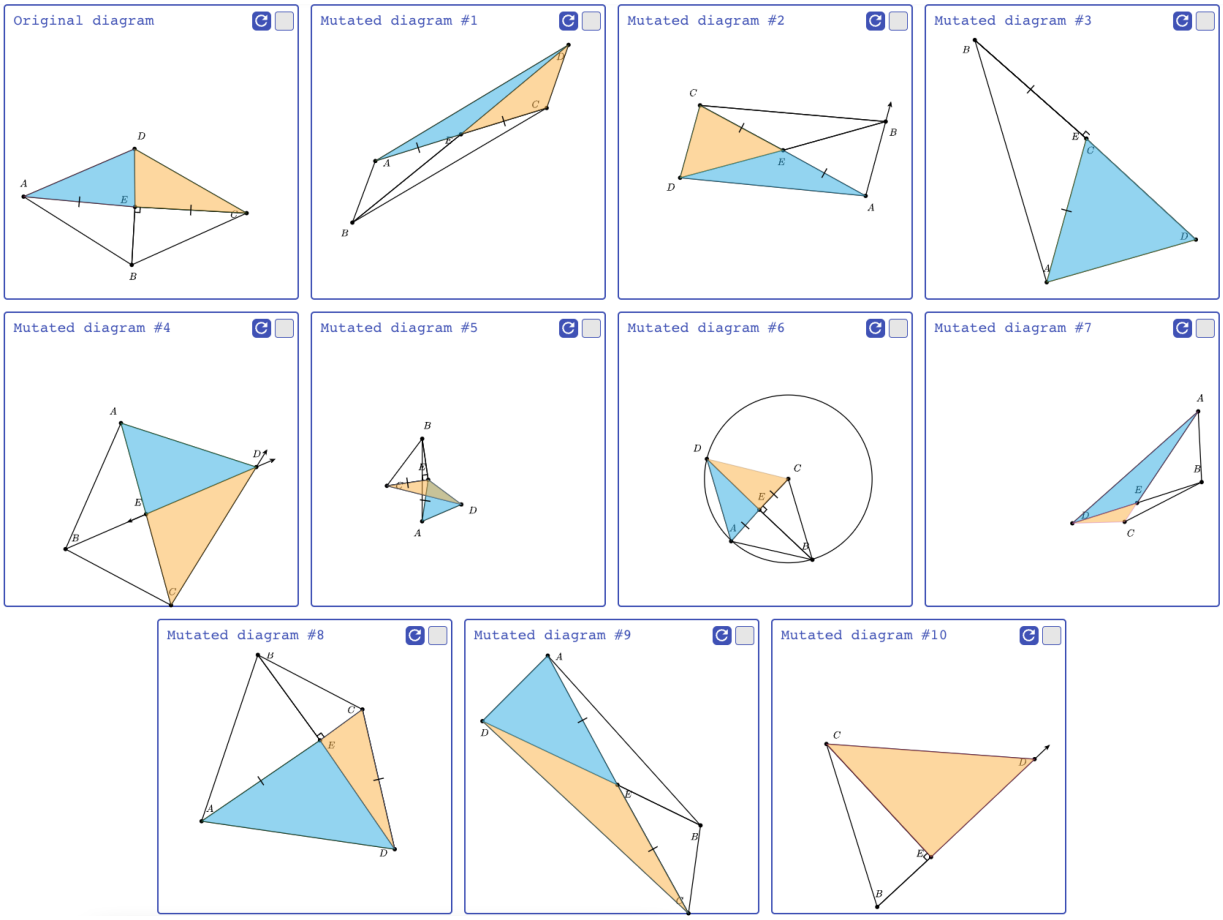
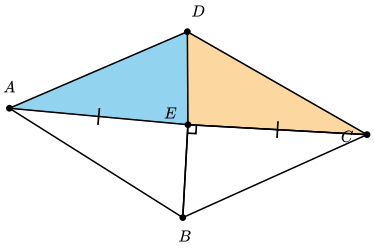


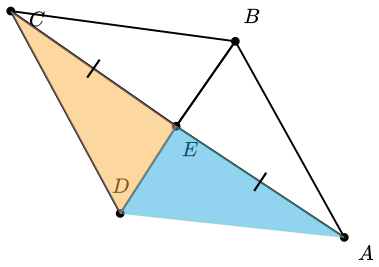
Figure 5.6: The first ten diagram variations generated by EDGEWORTH for the problem shown in Figure 5.5 (left).

that **Add** tend to introduce elements to the diagram that obviously do not pertain to the question prompt. Through some trial-and-error, we settled on weighting deletions 20%, edits 80%, and adds 0% to achieve reasonable results. The reason we weight edits higher than deletions is that many of our geometry problems ask about specific named points, and deletions can make the diagram invalid by removing points that are mentioned in the prompt.

We use the problem in Figure 5.5 (lower-left) to demonstrate how EDGEWORTH generates variations that are meaningful as problem options. In correct diagram to the prompt (Option 2), $\triangle DEC$ and $\triangle DEA$ are congruent by the Side-Angle-Side rule. In particular, they share a side (DE), the sides AE and EC appear to have equal length and are marked as such with a tick, and $\angle DEA$ and $\angle DEC$ are both right angles and therefore equal.



Option 4 in Figure 5.5 involves mutating the scenario by removing the right angle marker which makes it impossible to prove that $\angle DEA$ and $\angle DEC$ are equal. This is an example of the **Delete** mutation described in Section 5.3.3. The angle appears to be a right angle in Option 3, so this option might serve as a good distractor for students still learning the distinction between the appearance of angles and their markings.



Option 3 involves mutating Option 2 by editing which sides have equal length. In Option 3, sides CD and AE are equal instead of AE and CE. This is an example of the **Replace Arguments** mutation described in Section 5.3.3. A student might incorrectly select Option 3 if they believed in a Side-Angle congruence rule, where a single angle and single side being equal could prove congruence. Finally, in Option 1 $\angle CEB$ neither is marked as a right angle nor appears as a right angle. A student might incorrectly select Option 1 if they believed in a Side-Side congruence rule, where two sides being equal could prove congruence.

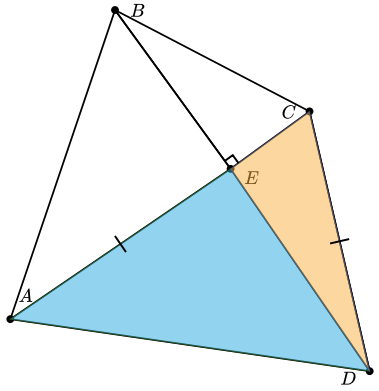


Figure 5.6 shows the first 10 variations EDGEWORTH generated from the example diagram. To create our problem, shown in Figure 5.5 (left), we selected the original diagram and two incorrect variations (numbers 1 and 8), plus another variation in an extended pool (number 16). As shown in Figure 5.6, there are many other viable answer choices in the first 10 variations. Many of the diagrams involve extra details that are irrelevant to the problem, like the circle in number 6 or the vector above point B in number 2. These extra details can be pedagogically useful for teaching students to filter irrelevant information in the domain. Some of the other diagrams are very obviously incorrect, like number 10 which doesn't show a blue triangle, or number 7 where the blue triangle is much larger than the orange triangle; these can be useful for building confidence when students are first learning.

5.5.3 General Chemistry: Lewis Structures

We chose 7 chemistry problems on Lewis structure from an online General Chemistry 1 course [132]. These problems test students' understanding of how atoms bond together based on formal charges. The module introduces students to the *octet rule*: the tendency of main group atoms to form enough bonds to obtain eight valence electrons. Lewis structure diagrams show bonds among atoms and valence electrons on atoms typically following the octet rule.

We extend the existing PENROSE chemistry stylesheet to include notation and layout rules for Lewis structures. To permit incorrect diagrams, the chemistry stylesheet must not enforce the octet rule. It does specify that an atom can have any number of bonds and that it can have 0, 2, 4, or 6 valence electrons. These specifications cover all problem scenarios in this Lewis structure module.[4] In accordance with stylistic conventions in the field, EDGEWORTH automatically lays out atoms, bonds, and electrons to maximize bond angles and repel electrons from bonds. For molecules involved in all 7 problems, the layout algorithm produces high-quality diagrams without any manual manipulation needed from the author.

To configure EDGEWORTH for this domain, we weight edits 100%. We exclusively weight on edits because we observed that variations of molecules never add or delete atoms and bonds. Although valence electrons may be added or deleted, they are modeled as predicates that can be edited to change the number of electrons for an atom (e.g., `ZeroDots(H)` $\rightarrow$ `TwoDots(H)`) via a **Replace Function** mutation operation.

Figure 5.5 (middle) shows an EDGEWORTH Lewis structure problem for hydrogen cyanide. In the original problem, the third option is correct. The EDGEWORTH version shows the correct diagram as option 1. Incorrect choices for this problem can be generated via mutation. For instance, if a student forgets that nitrogen must have eight surrounding electrons, they might choose the bottom-left option, which was generated by removing the valence electrons around nitrogen. Or, if a student does not know that hydrogen should only have two electrons instead of eight, they might select the top-right choice, which was generated by mutating the number of electrons around hydrogen from zero to six. Finally, if a student does not know that free electrons should be minimized, they might pick the bottom-right diagram, which was generated by mutating up the number of electrons around carbon and nitrogen and changing the triple bond to a double bond.

5.5.4 Discrete Math: Graphs

We draw 7 graph theory problems from the "Graphs" chapter of *Discrete Mathematics and Its Applications* [148, Chapter 10]. We model our visual representation after the style used in the textbook because students are already accustomed to recognizing graph diagrams in this style. We created a new PENROSE stylesheet for four subdomains of graphs (directed vs not, and multigraph vs not). For each of these subdomains, EDGEWORTH automatically lays out graph nodes, edges, loops, arrows, and labels in configurations that minimize confusing overlap of diagram elements.

To configure EDGEWORTH for the graph domain, we weight additions 50%, deletions 40%, and edits 10%. We disfavor edits in this domain because most of them are not useful: **Replace**

[4] "Odd electron molecules are very rare and cannot achieve full octets of electrons around atoms because of the odd number of electrons." [132]

Function is inapplicable for any of our graph subdomains, and **Swap Arguments** only applies to directed graphs. **Replace Arguments** is meaningful, but most desirable mutations for graphs are better represented by the addition or deletion of edges and nodes. For instance, a bipartite graph can become not-so by adding edges, or a strongly-connected graph can become not-so by deleting edges.

Figure 5.5 (right) shows an EDGEWORTH problem asking which of four directed graphs have an Euler circuit [5]. The example scenario shown as option 4 does not have an Euler circuit, as can be seen by observing that the sum of a's in-degree and out-degree is odd. In contrast, for option 3 generated by deleting edge (a, d), every node has an even sum of in-degree and out-degree, and indeed there does exist an Euler circuit. This condition on degree is only sufficient for undirected graphs, though; the diagram in the top-right is generated by flipping edge (b, c) from the bottom-left diagram, but does not have an Euler circuit, thwarting the simple degree counting heuristic. Finally, the simple diagram in the top-left is generated by deleting d and trivially has an Euler circuit.

5.6 Summary

The chapter introduces EDGEWORTH, a tool designed to aid the authoring of diagrammatic problems for educators in STEM. Recognizing the importance of representational fluency–the ability to understand and translate between different forms of representation such as diagrams, symbols, and text–EDGEWORTH addresses the challenge educators face in manually creating and maintaining visual content for instructional purposes. By allowing educators to create a single example diagram, EDGEWORTH automatically generates numerous variations, simplifying the process of developing diagrammatic problem sets.

The chapter also covers the system design of EDGEWORTH and its program mutation techniques for generating diagram variations. EDGEWORTH's approach is domain-agnostic, making it adaptable across various instructional contexts. To show EDGEWORTH's flexibility and expressiveness, we gather a translation problem dataset from Euclidean geometry, general chemistry, and discrete mathematics.

[5] An Euler circuit is a path in a graph that starts and ends at the same vertex and visits every edge exactly once.

Chapter 6

Evaluating EDGEWORTH[1]

In this chapter, we discuss three studies to evaluate various aspects of EDGEWORTH (Chapter 5). To effectively scale up visual practice authoring, EDGEWORTH must support a diverse set of instructional domains, generate high-quality diagrams consistently, and allow educators to author real-world problems. In Sections 6.1 to 6.3, we evaluate EDGEWORTH by answering the following research questions on these qualities:

- **Reliability** (RQ3.1): Can EDGEWORTH reliably generate translation problems with relatively few variations required?
- **Efficiency** (RQ3.2): comparing with a conventional drawing tool, are authors more efficient at making translation problems using EDGEWORTH?
- **Ecological validity** (RQ3.3): Do real-world instructors consider EDGEWORTH-generated translation problems to be useful?

First, we evaluated the reliability of EDGEWORTH by labeling 310 diagram variations from translation problem dataset (Section 5.5) by hand. With high inter-rater reliability, the result shows that EDGEWORTH can reliably generate diagrams that constitute valid four-choice translation problems, when constrained to 10 variations per problem.

Second, we performed a user study to measure authors' efficiency at creating translation problems using EDGEWORTH, compared with a conventional drawing tool. The results show that once authors make a correct diagram, they are about 3 times faster at making diagrammatic options for translation problems using EDGEWORTH compared to Google Drawings.

Finally, we conducted walkthrough demonstrations with 9 educators that have experience creating problems. The goal of the demonstrations was to obtain feedback on the ecological validity of EDGEWORTH-generated problems and the usefulness of EDGEWORTH in general. Overall, these experts found EDGEWORTH-generated problems to contain pedagogically useful variations and high visual quality. They provided detailed feedback on individual diagram variations and suggested how EDGEWORTH might fit into their instructional contexts.

[1]This chapter is partly adapted from Sections 6 to 7 of "Edgeworth: Efficient and Scalable Authoring of Visual Thinking Activities" [127].

6.1 Reliability Evaluation (RQ3.1)

EDGEWORTH's approach involves random mutations. The mutation operations are type-safe, but type-safety does not prevent degenerate diagram layouts. For instance, `Point A, B` followed by `Triangle t := MkTriangle(A, A, B)` will typecheck. However, since the triangle described in this scenario involves the `Point A` twice, EDGEWORTH will produce a line segment, not a triangle from this scenario. Are EDGEWORTH suggestions dominated by these nonsensical scenarios? In this section, we evaluate whether EDGEWORTH can reliably suggest diagrams that are valid answer options to multiple-choice translation problems (RQ3.1).

6.1.1 Methods

The goal of EDGEWORTH is to generate enough diagram variations to assemble a four-choice multiple-choice problem for a given prompt. To this end, we use the following classification scheme for diagram variations: a variation can be a **Correct** or **Incorrect** answer to the prompt, or **Discard**ed because the diagram is invalid for missing key components or lacking readability.

For RQ3.1, we define "relatively few variations" to be 10 diagrams, and consider EDGEWORTH to have generated a translation problem in n variations if at that point we have (possibly including the original diagram) at least one **Correct** diagram, at least one **Incorrect** diagram, and in total at least four diagrams that are either **Correct** or **Incorrect**.

We used EDGEWORTH to generate 10 diagrams per problem for all 31 problems in the translation problem dataset (Section 5.5), which yielded 310 diagrams in total. To evaluate this coding scheme, we randomly sampled 2 problems from each of our 3 domains, for 60 generated diagrams total. The first two authors each coded all 60 of those sample diagrams, after which we calculated the Cohen's κ [38] statistic. Then with the assumption that our coding scheme has reasonable inter-rater reliability, at least one author [2] coded all remaining diagrams, allowing us to determine the number of our prompts for which EDGEWORTH was able to successfully generate a multiple-choice problem. The coding results are included in supporting files.

6.1.2 Results

Reliability of Problem Generation

For RQ3.1, we found that EDGEWORTH generated valid multiple-choice problems for 27/31 prompts within 10 variations, and for 30/31 problems within 20 variations. For each of these four failures with 10 variations, EDGEWORTH did generate at least four **Correct** examples, but we had to **Discard** all the other diagrams, leaving no **Incorrect** examples. For the one remaining failure with 20 variations, EDGEWORTH never succeeded even after we increased the number of variations to 50.

	Correct	**Incorrect**	**Discard**	*total*
geometry	52	54	64	170
chemistry	3	54	13	70
discrete	28	25	17	70
total	85	133	94	310

Table 6.1: Distribution of diagram variation classes.

Distribution

The original diagram is a **Correct** answer for every prompt, except for the two Euler circuit prompts, in which the original diagram is **Incorrect**. For EDGEWORTH-generated variations, the full distribution of classes is shown in Table 6.1.

The chemistry domain had a far smaller proportion of **Correct** variations than the other two domains because the only way for a variation to be **Correct** is for it to coincidentally be identical to the original diagram. Interestingly, in the other two domains, there were about the same number of **Correct** and **Incorrect** variations.

In the geometry domain, **Discard**ed diagrams were primarily either diagrams missing elements referred to in the question prompt, or diagrams that were visually degenerate (e.g., everything compressed into a single line). In chemistry, we **Discard**ed diagrams where the molecule was disconnected. Finally, in the graph domain, we **Discard**ed diagrams in which some nodes were labeled and others were unlabeled (i.e., EDGEWORTH had inserted new unlabeled nodes when all nodes in the original diagram were labeled).

Inter-rater Agreement

We sampled two problems per domain from the problems collected in Section 5.5 to evaluate inter-rater agreement (six problems or sixty diagrams in total, 19% of the dataset). We found perfect agreement on that sample, so $\kappa = 1$.

6.2 Experimental Evaluation of Authoring Efficiency (RQ3.2)

To answer RQ3.2, we conduct an experiment that compares EDGEWORTH against a conventional drawing tool in translation problem authoring tasks. In this section, we describe the experimental setup and findings.

6.2.1 Study Design

Participants

We recruited 16 participants through advertisement in the university community (e.g. emails and Slack channels). Participants were screened to have some past experience using digital drawing

[2]The study is conducted jointly with authors of Ni et al. [127].

Domain	Task 1 (Prompt 1)	Task 2 (Prompt 1)	Task 3 (Prompt 2)	Task 4 (Prompt 2)
Chemistry	Google Drawings	EDGEWORTH	Google Drawings	EDGEWORTH
Chemistry	EDGEWORTH	Google Drawings	EDGEWORTH	Google Drawings
Geometry	Google Drawings	EDGEWORTH	Google Drawings	EDGEWORTH
Geometry	EDGEWORTH	Google Drawings	EDGEWORTH	Google Drawings

Table 6.2: Participants were divided into 4 groups by the tools they used and diagramming domains of the tasks. Each row corresponds to the task sequence of one of the groups. Participants used both EDGEWORTH and Google Drawings to author problems for two prompts in chemistry or geometry (Figure 6.2).

tools. All participants reported that they have used Google Drawings and/or equivalent tools to make diagrams in the past.

Tasks

We selected four problem prompts from the translation problem dataset (Section 5.5), two from the chemistry domain and two from geometry, shown in Figure 6.1.

We segmented the authoring of the first correct diagram and subsequent incorrect diagrams in the tasks. This segmentation allows us to separately measure the authoring efficiency of creating the example scenario (Section 5.3.1) and creating counterexamples (Section 5.3.1). For participants who used EDGEWORTH, we were particularly interested in the upfront cost of making the first SUBSTANCE diagram in the PENROSE editor.

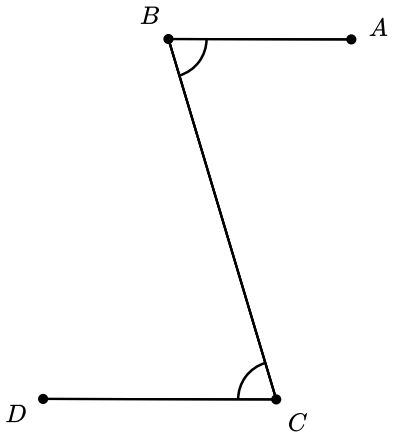
For each task, the participant were given (1) a textual problem prompt and (2) an example diagram (i.e., a correct response to the prompt). Participants were then given up to 20 minutes to complete each task, which involve two segments: (a) **correct segment**: participants first re-created one example visually similar to the given diagram and then (b) **incorrect segment**: made up to 10 incorrect diagrams by editing the diagram produced in sub-task (a). Each sub-task is time-bounded to 10 minutes. If the participant failed to produce 1 correct diagram in the first segment, they were provided with one so they could continue to the next segment. Each participant completed two problem prompts in chemistry or geometry.

Experimental Design

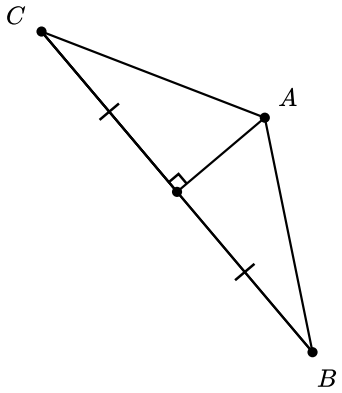
The study was a within-subject design, where participants were divided into four groups by the ordering of tools they use and diagramming domains of their tasks. Participants used both EDGEWORTH and Google Drawings to author diagrams in a random counterbalanced order. Participants were further randomly assigned into one of two subgroups: one subgroup made chemistry diagrams and the second subgroup made geometry diagrams. Table 6.2 summarizes the four groups that resulted from the tool and domain assignments. Each group had four participants.

In the 90-minute study session, each participant was given two problem prompts in total, each repeated twice for EDGEWORTH and Google Drawings, so four tasks in total. For instance, a participant in the chemistry-drawing group (the first row in Table 6.2) would spend up to 10 minutes making 1 correct diagram (correct segment) and then up to 10 minutes to make incorrect diagrams of CH_2O using Google Drawings (incorrect segment) first, and then another 20 minutes

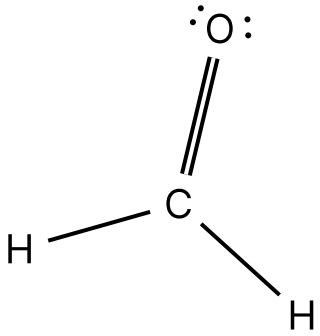
Prompt 1: In which of the following diagrams is there a perpendicular bisector for segment BC?



Prompt 2: In which of the following diagrams is there a perpendicular bisector for segment BC?



Prompt 1: Which of the following diagrams shows the correct Lewis structure for CH_2O?



Prompt 2: Which of the following diagrams shows the correct Lewis structure for HNO_3?

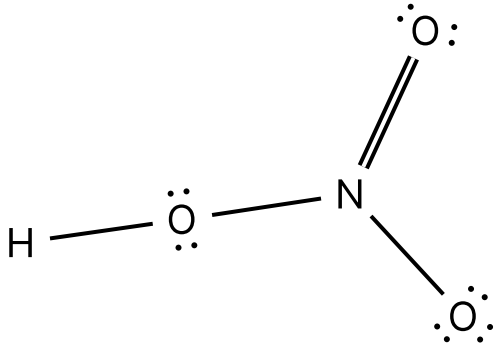
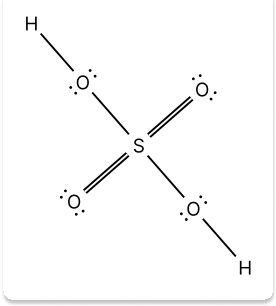


Figure 6.1: Tasks used in the EDGEWORTH experimental evaluation. Each participant is given a textual prompt and a correct diagram to this prompt at the beginning of each task. They are asked to first re-produce the correct diagram using the designated tool in the correct segment, and then edit this diagram to produce up to 10 incorrect diagrams to the prompt in the incorrect segment.

```
Hydrogen h1, h2
Sulfur s1
Oxygen o1, o2, o3, o4
Bond b2 := SingleBond(h1, o1)
Bond b3 := SingleBond(h2, o2)
Bond b4 := SingleBond(o1, s1)
Bond b5 := SingleBond(o2, s1)
Bond b6 := DoubleBond(o3, s1)
Bond b7 := DoubleBond(o4, s1)
ZeroDots(h1)
ZeroDots(h2)
ZeroDots(s1)
FourDots(o1)
FourDots(o2)
FourDots(o3)
FourDots(o4)
```




```
Point A, B, C, D
Let BC := Segment(B, C)
Angle CAB := InteriorAngle(C, A, B)
Angle CDB := InteriorAngle(C, D, B)
Angle ACB := InteriorAngle(A, C, B)
Angle CBD := InteriorAngle(C, B, D)
EqualAngleMarker(ACB, CBD)
RightMarked(CAB)
RightMarked(CDB)
Let AB := Segment(A, B)
Let AC := Segment(A, C)
Let CD := Segment(C, D)
Let DB := Segment(D, B)
EqualLengthMarker(AC, DB)
EqualLength(AC, DB)
ParallelMarker(AB, CD)
Parallel(AB, CD)
Point M
On(M, BC)
Let c := Circle(M, B)
OnCircle(c, C)
AutoLabel A, B, C, D
```

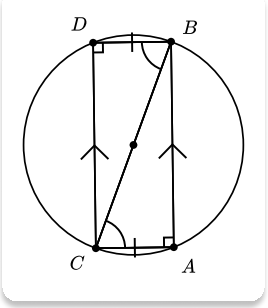


Figure 6.2: Participants were provided both Google Drawings and SUBSTANCE examples throughout the study. The SUBSTANCE code (left) was given in the EDGEWORTH tasks and a Google Drawings file that visually resembles the PENROSE output (right) was given for the Google Drawings tasks.

on the same prompt using PENROSE for the correct segment and EDGEWORTH for the incorrect segment. After that, this participant would repeat the same for HNO_3.

At the start of each study session, participants were given 5-minute tutorials of EDGEWORTH and Google Drawings, in which they were guided to draw either a right triangle or the Lewis structure of O_2. The ordering of tutorials match the counterbalanced ordering of Google Drawings and EDGEWORTH. Throughout the session, participants had access to one Google Drawings example and one SUBSTANCE example. Figure 6.2 shows the chemistry and geometry examples. The examples are samples from the translation problem dataset (Section 5.5) that are visually more complex than the actual study tasks. We provided them to the participants as an authoring aid so that they can copy elements from the examples to save time, analogous to the real-world experience of copying and pasting online examples reported in Section 5.2.

Participants received no more instructions during the tasks. The experimenter only observed the participant and used a stopwatch to measure the time on task. After completing each task, participants completed a survey that asked them if they agree with the following statements on a 5-point Likert scale:

- I would use this problem for a class that I teach.
- The problem is pedagogically useful (i.e., students will benefit from doing this problem).
- The diagrams in the problem are of high visual quality.

The study took about 90 minutes per participant, using a provided MacBook Pro with the latest version of Chrome installed. The study sessions were audio-recorded and transcribed. All participants were compensated $25 Amazon gift cards for their time.

6.2.2 Results

Table 6.3 shows the average total time, diagrams produced, and time per diagram for all participants. The **Diagram Ct** column in Table 6.3 shows how many diagrams participants produced in each segment of all tasks on average. Any number lower than 1 for correct segments and 10

for incorrect segments indicates that the corresponding participant did not complete the segment. All participants authored 10 incorrect diagrams within 10 minutes using EDGEWORTH for both domains. In the geometry group, 6 out of 8 participants (11 out of 16 total segments) failed to do so using Google Drawings for at least one segment. In the chemistry group, 1 out of 8 participants failed for both incorrect segments. All participants were able to complete the correct diagram for both chemistry prompts using both tools. For the geometry tasks, all participants produced one correct diagram in the first segment using Google Drawings, but 3 failed using PENROSE.

Domain	Segment	Tool	Total Time	Diagram Ct	Time/Diagram
Chemistry	correct	PENROSE	144.19s	1.00	144.19s
		Google Drawings	231.81s	1.00	231.81s
	incorrect	EDGEWORTH	150.13s	10.00	15.01s
		Google Drawings	440.63s	9.38	51.56s
Geometry	correct	PENROSE	390.94s	0.81	390.94s
		Google Drawings	228.50s	1.00	228.50s
	incorrect	EDGEWORTH	257.25s	10.00	25.73s
		Google Drawings	549.38s	7.38	100.90s

Table 6.3: Summary of Average Time, Diagram Count, and Time Per Diagram by Domain for both chemistry and geometry domains, and two segments of each task (Section 6.2.1).

A two-way repeated measures ANOVA was conducted to examine the within-subject effects of tool (EDGEWORTH vs. Google Drawings) and task (correct vs. incorrect) on various outcomes. The analysis included task completion time and other performance metrics across the domains of chemistry and geometry.

For the incorrect segments, the analysis revealed significant differences in task completion times between the tools used, visualized in Figure 6.3 (right). In the chemistry domain, participants completed tasks significantly faster (almost 3 times faster) using EDGEWORTH (M = 150.13s, SD = 59.15s) compared to Google Drawings (M = 440.63s, SD = 113.04s), as indicated by the significant main effect of tool, $F(1,7) = 53.33, p = 0.0002$. There was no significant effect of the task itself, $F(1,7) = 1.29, p = 0.293$, suggesting that the difficulty of the tasks was consistent regardless of the tool used. Similarly, in the geometry domain, participants also completed tasks significantly faster (similarly, almost 3 times faster) with EDGEWORTH (M = 257.25s, SD = 139.55s) compared to Google Drawings (M = 549.38s, SD = 91.28s), with a significant effect of the tool, $F(1,7) = 90.97, p < 0.0001$. There was a marginal effect of task, $F(1,7) = 3.64, p = 0.098$, indicating a trend towards task differences that did not reach statistical significance.

For the correct segments, the analysis showed mixed results on PENROSE's performance depending on the domain, illustrated in Figure 6.3 (left). In the chemistry domain, participants completed tasks significantly faster using PENROSE (M = 144.19s, SD = 79.22s) compared to Google Drawings (M = 231.81s, SD = 129.78s), as indicated by the significant main effect of tool, $F(1,7) = 6.65, p = 0.037$. There was no significant effect of the task itself, $F(1,7) = 0.99, p = 0.353$. In the geometry domain, however, Google Drawings outperformed PENROSE, with participants completing tasks faster using Google Drawings (M = 228.5s, SD = 71.74s)

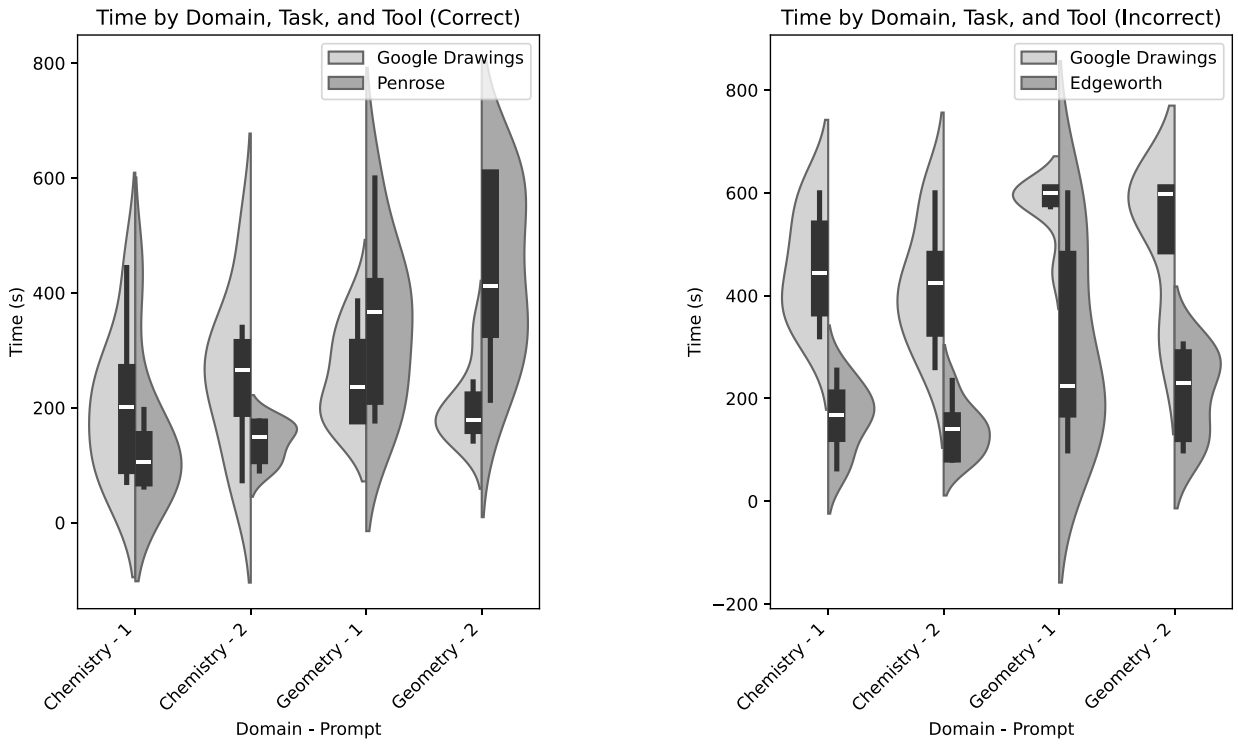


Figure 6.3: Violin plots showing the distribution of time-on-task for both correct (**Left**) and incorrect (**Right)** segments of tasks. The shape of the violins represents a smoothed approximation of the data distribution, with wider sections representing higher density. The embedded box plots within the violins show the median (white line) and inter-quartile range (thick black bar), with the whiskers (thin black lines) extending to the data range.

compared to PENROSE (M = 390.94s, SD = 149.74s), as indicated by the significant main effect of tool, $F(1, 7) = 15.95, p = 0.005$. Again, there was no significant effect of the task itself, $F(1, 7) = 0.28, p = 0.611$.

In the per-task survey, summarized in Table 6.4, participants provided feedback on the tasks across two domains and both tools. For the chemistry tasks, EDGEWORTH was rated highly across all survey items, with participants expressing a strong likelihood of using the problems in their classes (M = 4.19), finding them pedagogically useful (M = 4.25), and rating the visual quality of the diagrams as excellent (M = 4.50). In contrast, Google Drawings received lower ratings in chemistry, particularly in terms of visual quality (M = 2.81). In the geometry domain, EDGEWORTH also was rated well, with participants finding it useful (M = 4.06) and visually acceptable (M = 3.38), although the ratings were slightly lower compared to chemistry. Google Drawings in geometry was rated lower across all dimensions, with middling scores for usefulness (M = 3.50) and visual quality (M = 3.44). Overall, EDGEWORTH consistently out-rates Google Drawings, particularly in terms of the visual quality of the diagrams and pedagogical usefulness, especially in the chemistry tasks.

We conducted a Multivariate Analysis of Variance (MANOVA) on the survey data to quantitatively assess the impact of the tool (EDGEWORTH vs. Google Drawings) and the domain (chemistry vs. geometry) on three dependent variables corresponding to the survey questions. The results showed a significant effect of the tool on the combined dependent variables, with

Domain	Tool	Would Use	Useful	High Quality
Chemistry	EDGEWORTH	4.19	4.25	4.50
	Google Drawings	3.63	3.94	2.81
Geometry	EDGEWORTH	3.94	4.06	3.38
	Google Drawings	3.38	3.50	3.44

Table 6.4: Survey responses for chemistry and geometry tasks using EDGEWORTH and Google Drawings. Higher numbers (visualized in green hue) indicates positive responses and lower numbers (yellow and red hue) negative responses.

Wilks' lambda indicating that the choice of tool had a statistically significant influence on the survey responses, $F(3, 59) = 3.3995$, $p = 0.0235$. The domain (chemistry vs. geometry) did not have a significant effect on the combined dependent variables, $F(3, 59) = 0.7550$, $p = 0.5239$. A significant intercept observed in the analysis, $F(3, 59) = 217.9321$, $p < 0.0001$, suggests that the overall mean response across all groups was significantly different from zero, indicating that participants generally provided positive ratings across all survey items. In summary, EDGEWORTH was perceived more favorably across the three survey questions compared to Google Drawings.

6.2.3 Discussion

The results show a trade-off between the time taken to create correct diagrams using PENROSE and the efficiency of generating incorrect variations using EDGEWORTH. Participants might spend more time on the initial correct diagram using PENROSE, but are significantly and consistently faster at making incorrect diagrams using EDGEWORTH than Google Drawings. On average, participants were 3–4$\times$ faster using EDGEWORTH. In geometry, the initial correct diagram took more time with PENROSE (390.94s per diagram versus 228.50s with Google Drawings), but the time per incorrect diagram was much lower with EDGEWORTH (25.73s) compared to Google Drawings (100.90s).

The initial investment in the first SUBSTANCE program differs depending on the language complexity and layout consistency. Generally speaking, the PENROSE chemistry domain is simpler to learn than the PENROSE geometry domain. The chemistry domain has a simpler grammar consisting of atoms, bonds, and valance electrons. The layout for chemistry diagrams is also more stable and consistent. In contrast, the PENROSE geometry domain includes many predicates among points, line segments, lines, rays, angles, and so on. The geometry STYLE is also less polished than that of chemistry. We observed that participants were sometimes confused by bad layouts produced by PENROSE, and doubted the correctness of their SUBSTANCE programs. The timing data in the correct segment shows the difference: participants were $1.7\times$ faster to make the correct diagram using PENROSE on average in chemistry, but $1.7\times$ slower in geometry.

6.3 Expert Walkthrough Demonstration and Feedback (RQ3.3)

The intended users of EDGEWORTH are educators who create problems. These users are very important to the education system since other teachers make use of their problems. Therefore, we

ID	Occupation	Years of Experience	Domain(s)
E1	MOOC Course Designer	7	Chemistry
E2	Liberal Arts College Professor	4	Chemistry, Geometry
E3	Community College Professor	30	Chemistry
E4	Liberal Arts College Professor	11	Graphs
E5	Research University Professor	17	Graphs
E6	Research University Professor	5	Graphs
E7	Middle School Teacher	5	Geometry
E8	Undergraduate Teaching Assistant	3	Geometry, Graphs
E9	High School Teacher	11	Geometry, Graphs

Table 6.5: Demographics of walkthrough demonstration participants.

recruited educators who created visual practice problems in multiple domains and educational settings to evaluate ecological validity of EDGEWORTH-generated problems (RQ3.3). While an expert survey may suffice for rating problem quality, we opted for walkthrough demonstration, based on prior research on evaluation methods by Ledo et al. [101], to gather additional qualitative feedback on the value of having the toolkit in their day-to-day work.

6.3.1 Participants and Procedure

We recruited domain expert educators of chemistry, geometry, and graph theory. Experts were invited based on their extensive teaching experience in the domain and past experience in *authoring* diagrammatic content. In contrast to the criteria in the formative study (Section 5.2), this study selected participants based on their domain-specific expertise in authoring problems. Recruited educators came from a wide range of institutions, including Massive Open Online Courses (MOOC) platforms, liberal arts colleges, community colleges, research universities, and secondary schools. The average teaching experience among the 9 expert educators (E1 – E9) was 10.33 years, with a standard deviation of 8.39 years, highlighting a broad range of teaching experience. One of the participants is the original author of the chemistry problems reproduced in the translation problem dataset (Section 5.5). Table 6.5 summarizes the demographic information for 9 expert educators (E1–9) who participated in the study.

Each expert participated in a 60- to 90-minute session via video conferencing, which was recorded with their consent. At the start of each session, we demonstrated the workflow of EDGEWORTH end-to-end, as described in Section 5.3.1, on one problem outside of the expert's domain. For the remainder of the session, we asked the expert to assemble problems from the EDGEWORTH output of two to four problem prompts randomly sampled from the translation problem dataset (Section 5.5) in their domain. Per prompt, the expert rated 10 diagram variations based on the categories described in Section 6.1.1. In addition, we asked participants to provide more granular feedback on diagram quality. After rating the diagram variations, they were asked to pick diagrams to assemble a four-choice diagrammatic translation problem. After the problem was assembled and shown on the interface, we asked (1) if they would use the problem in their instruction and (2) how they would author the diagram using their own workflow. The full study protocol is included in supporting files.

6.3.2 Ecological Validity of Generated Problems

Overall, experts were happy with the problems they assembled with EDGEWORTH-generated diagrams. Experts (E1–9) indicated that they would use all of the problems they created using EDGEWORTH in their coursework. Other experts said they would use EDGEWORTH-generated problems "*early in the learning process*" (E3) and "*as a warm up exercise at the start of the next lecture*" (E4). In addition, expert said these problems could be used to review previously introduced concepts. For example, E3 found the diagram variations that break the octet rule to be useful for "*after you've also introduced expanded octet or non-octet-rule things.*" Experts plan to use EDGEWORTH-generated problem to "*focus on things that students struggle with*" (E3) and when introducing concepts that are "*all about visualization*" (E5) such as planarity of graphs. E7 asked to see all problems we gathered in the translation problem dataset (Section 5.5) and was excited to them in their class because they were "*going to be covering everything [on the list].*" In addition to just asking students to select correct diagrams, E3 also pointed out that by prompting students to "*tell me what is wrong rather than just which is the correct one,*" the problem can be used to "*dive deeper.*" Similarly, E4 proposed to use EDGEWORTH problems as "*an interactive warm-up for reviewing the last lecture, where students vote on and explain why a diagram is correct.*" E7 even plans to use EDGEWORTH as "*a creative instead of assessment piece*" and "*have the students be the teacher . . . playing this role more, they get better at tests, because they understand what the test makers are doing.*"

6.3.3 Expert Feedback

Experts provided positive qualitative feedback on EDGEWORTH

Experts reacted positively to EDGEWORTH. They found EDGEWORTH to be a "*perfect fit*" (E1, E6, E8) for generating multiple-choice problems, especially "*low-stake*" (E2, E3, E5, E6, E8, E9) quizzes that "*incentivize [students] to keep up with the class*" (E8). Experts said the automatic layout of EDGEWORTH "*draws things really fast*" (E5), "*saves you the time of drawing multiple structures*" (E3), and produces "*beautiful*" (E4, E7) diagrams. Comparing with their existing tools, EDGEWORTH is a "*nice time-saver*" (E3) and the translation problems they authored during the session would take an "*enormous amount of work*" (E4), "*infinitely longer than this took*" (E6).

Notably, experts pointed out that EDGEWORTH aids creativity by promoting "*recognition over recall*" (E6). Specifically, EDGEWORTH helps with "*the thinking about how to come up with the graphs*" and simplifies the diagram layout such that "*you just generate some mutations that you click refresh until it looks nice*" (E6). E2 liked that "*it can come up with different possibilities than the ones that would be immediately apparent to me.*"

In addition, experts commented that EDGEWORTH can enable them to give students more practice. For instance, E4 noted that "*there's a feedback loop where . . . if I had a really good tool for generating nice multi-choice questions, then I could envision doing that much more frequently.*"

Importantly, in the context of student authoring problems themselves, E7 thinks that lowering the barrier of problem authoring help students "*feel they have ownership in their learning as well as sharing their ownership with other students in the class.*"

Experts used visual selection to express diverse standards on diagram quality

When rating diagram mutants, experts agreed with diagram ratings of Section 6.1, but expressed unique standards for selecting answer choices (Section 6.3.1). Since experts had different standards, they selected different diagrams to assemble problems. This suggests EDGEWORTH's use of visual selection met experts' needs.

One group of experts (E2, E4, E6, E7, E8, E9) preferred to maintain a balanced mix of answer choices, "*at least one that's obviously correct, at least one that's obviously incorrect, and then ... two where you have to think about that a little bit*" (E6). One rationale was to "*make sure [the problem] is challenging enough, but also has some things that are accessible to students that haven't completely mastered the material*" (E4). Another was to teach "*the process of elimination*" (E7, E9). Another group of experts (E1, E3, E5) had much higher standards for including a mutant in a multiple-choice problem. For example, E1 preferred problems to contain one correct answer and multiple distractor options that are "*less obvious*" such that students won't "*pattern match without looking at the details.*"

On a problem that E2 accepted 7 out of 10 mutants as good incorrect options, E3 discarded 8 out of 10 because they "*violated the octet rule in egregious or blatantly egregious way.*" However, E3 said whether the octet rule can be broken depends on "*where students are in the course.*"

The difference of standards is highly individual. From E2's knowledge of "*colleagues [who] only give difficult distractors*" and "*certain profs [who] are legendary for having really hard multiple choice,*" they guessed that harder problems "*motivate the students to try harder,*" but also pointed out that it "*only works for certain students in my experience.*" E3 stated that their choice in diagrams "*hinges upon my perception of whether students will automatically disqualify something,*" which they admitted is "*a certain premise or bias.*" In E3's words: "*Wow, it's really tough to ... completely take off the instructor hat.*"

This comment reflects the concept of *expert blind spot* in learning sciences literature, where experts fail to "*understand the processes of novices who are struggling to understand new ideas during their constructive learning process*" [124].

Experts selected isomorphic diagrams to build conceptual understanding

EDGEWORTH sometimes produces isomorphic diagrams, i.e., diagrams with identical content but different layouts. These diagrams occur when EDGEWORTH's mutations have no net impact on the example diagram, e.g., the mutator removes an edge from a graph and adds it back. Surprisingly, experts found value in these isomorphic diagrams. In their geometry course, E2 said that their textbook's diagrams "*get drawn the same way over and over again. And some students get stuck into thinking that the concept is only communicated when the diagram is drawn [exactly] that way.*" When assembling a problem about the HCN molecule, E3 compared two isomorphic variations, and picked one over another because "*it's drawn the opposite ... which is interesting and I think students are going to get it wrong.*" Similarly, E1 finds isomorphic diagrams to be useful for "*molecules with resonance structures.*" E5 found isomorphic planar graphs to be particularly useful because students find them "*painstaking to visualize when they just started.*" E5 planned to use EDGEWORTH to "*draw a graph that doesn't look like it could be planar first, but then untangle it to show that the graph is actually planar.*"

6.4 Limitations of the Studies

We discuss some limitations to the studies presented in this chapter.

6.4.1 Ecological Validity

The studies primarily focused on specific instructional domains, including chemistry, geometry, and graph theory. While these domains were chosen from the translation problem dataset, the studies are limited by the scope of the dataset itself. The performance and effectiveness of EDGEWORTH might vary when applied to other fields that require different types of visual problem representations, which were not explored in this research.

Although the expert walkthroughs provided valuable feedback on the ecological validity of EDGEWORTH-generated problems, the artificial nature of the study environment might not fully capture the complexities and constraints of real-world educational settings. The experts' feedback were based on hypothetical scenarios and short-term interactions with the tool, which may not fully reflect the challenges and demands of long-term usage in a classroom or curriculum development context.

6.4.2 Tool

Participants in the user study were provided with brief tutorials on using EDGEWORTH and Google Drawings, which may not have been sufficient for them to become fully proficient with these tools. The learning curve associated with PENROSE and EDGEWORTH, especially their unique approach to generating diagrammatic problems, might have influenced the results, particularly in the efficiency evaluation. Participants with more extensive experience or training in using either tool might exhibit different levels of efficiency and satisfaction than those observed in the study.

Both PENROSE and EDGEWORTH are still under development, and the studies were conducted on particular versions of them. As they evolve, with potential updates and improvements, the findings presented here might become outdated. Future research would need to re-evaluate the tool's performance and usability in light of new changes.

6.4.3 Authoring Speed vs. Problem Quality

Participants in the user study described in Section 6.2 were not screened for their prior experience nor expertise in diagrammatic problem authoring. In fact, a majority of them are undergraduate students. Therefore, their judgment of what make a good problem (or lack thereof) might influence the task performance data reported in Section 6.2. Theoretically, to get through the tasks, participants could pick incorrect diagrams at random using EDGEWORTH, or make minimal edits to the correct diagram to get alternatives using Google Drawings. To address this limitation of the user study, we conducted expert demonstration walkthroughs Section 6.3 to gain a deeper understanding of the ecological validity of EDGEWORTH-generated problems.

During the user study, we did in fact observe participants taking shortcuts in both conditions. Here we show some examples of them and contrast them with the experts' opinions. For instance, Figure 6.4 (top) shows P4's pattern of making small edits to the correct diagram to quickly

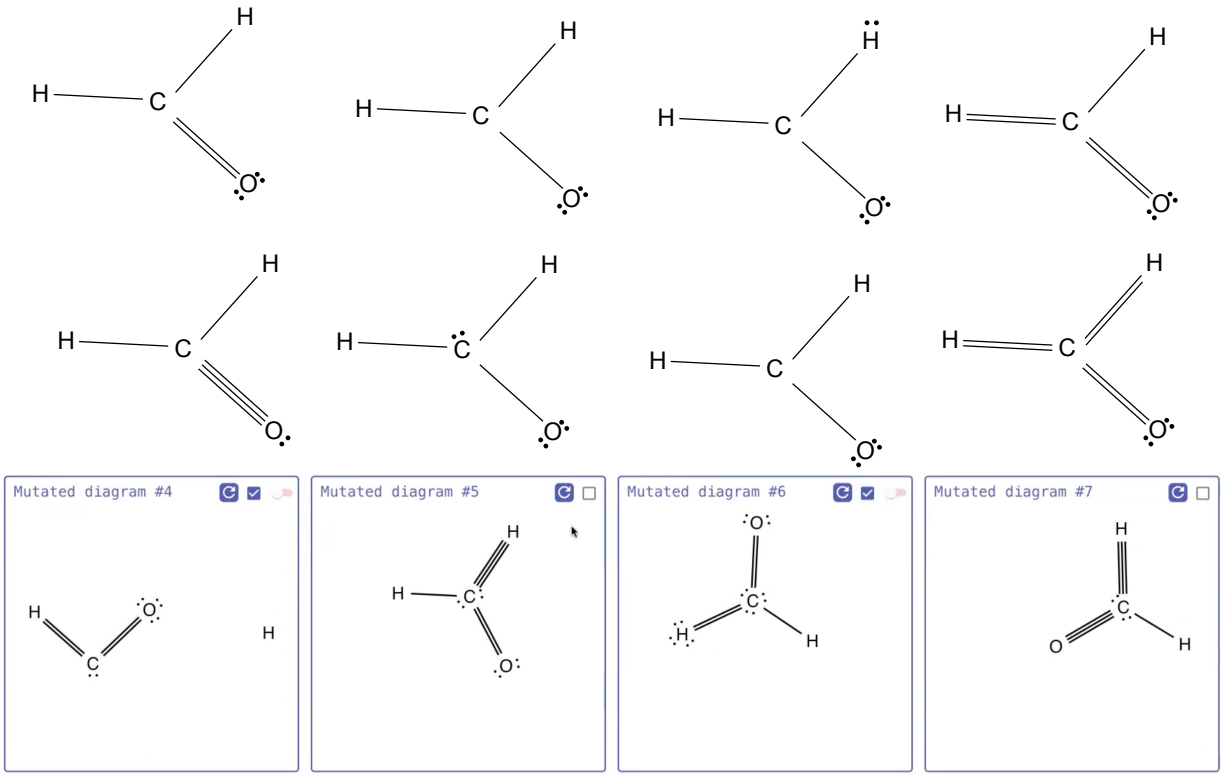


Figure 6.4: Screenshots of Google Drawings *(top)* and EDGEWORTH selections *(bottom)* of diagrams by P4 of the user study (Section 6.2). They are instances of "shortcuts" participants took when using both tools, avoiding large layout edits *(top)* in Google Drawings and selecting counterexamples seemingly at random in EDGEWORTH *(bottom)*.

produce incorrect diagrams. These edits avoids moving many diagram components around while maintaining a good layout. The similar layouts of atoms and bonds contrast experts' feedback on isomorphic diagrams (Section 6.3.3, e.g., "*some students get stuck into thinking that the concept is only communicated when the diagram is drawn [exactly] that way.*" by E2). As another example for EDGEWORTH, Figure 6.4 (bottom) shows two diagrams selected by a participant as suitable incorrect answers for chemistry prompt 1 (*Which of the following diagrams shows the correct Lewis structure for* CH_2O*?*) in Figure 6.1. Per their feedback in Section 6.3.3, E3 would not accept "Mutated Diagram #6" in the figure as a good incorrect options, because it "*violated the octet rule in egregious or blatantly egregious way.*" [3] Notably, as we discussed in Section 6.3.3, experts did not agree on a single standard of high problem quality among themselves.

Overall, while participants took shortcuts in both user study conditions, we do not know how much impact their quality standards have on the task performance. Importantly, there is not a single standard for "good" judgment of problem quality, as the expert demonstration walkthroughs showed a diversity of standards among experienced educators. Therefore, future research is needed to tease out (1) what is an acceptable quality standard for diagrammatic translation problems and (2) the effect of problem quality on authoring speed of diagrammatic problems.

[3]The Carbon (C) atom in the diagram has 6 valance electrons, 2 double bonds, and one single bond, which is way beyond the expected 8 electrons and bonds combined per the octet rule.

6.5 Summary

This chapter presents an evaluation of EDGEWORTH through three studies focusing on its reliability, efficiency, and ecological validity. The research questions addressed are whether EDGEWORTH can reliably generate translation problems with minimal variations (RQ3.1), if it enhances authoring efficiency compared to conventional tools (RQ3.2), and whether educators find the generated problems useful in real-world contexts (RQ3.3).

The first study (Section 6.1) assessed the reliability of EDGEWORTH by analyzing 310 diagram variations across 31 problems. The results indicated that EDGEWORTH successfully generated valid multiple-choice problems for most prompts within 10 diagram variations, demonstrating its reliability in producing consistent and usable outputs (RQ3.1).

The second study (Section 6.2) compared the efficiency of authoring translation problems using EDGEWORTH versus Google Drawings. The findings revealed that participants were significantly faster when using EDGEWORTH (RQ3.2).

The final study (Section 6.3) involved expert educators who provided feedback on the ecological validity of EDGEWORTH-generated problems. The educators found the problems to be pedagogically useful and expressed interest in using EDGEWORTH in their instructional practices. Their feedback also emphasized EDGEWORTH's potential to save time and enhance creativity in problem design (RQ3.3).

Overall, these studies demonstrate that EDGEWORTH is a reliable and efficient tool for authoring educational problems, with strong support from educators for its application in diverse instructional contexts.

Chapter 7

Conclusion and Future Work

7.1 Summary of contributions

This thesis makes several significant contributions to the study of diagramming, diagramming tools, and educational technology, including:

- An interview study of the diagramming process, providing detailed insights into how experts across different domains create and use conceptual diagrams. This study documents how expert from a diverse set of domains author diagrams and identifies key challenges in existing diagramming tools (Chapter 3).
- A natural diagramming framework that specifies four dimensions of diagramming tool design opportunity that seamlessly and naturally translate diagrammers' high-level ideas to illustrative and effective diagrams (Section 3.4).
- PENROSE, a novel system for creating diagrams from plain-text descriptions (Chapter 4). PENROSE allows authors to encode domain-specific visual representations and automatically lays out diagrams, bridging the gap between abstract ideas and their visual representation.
- EDGEWORTH, a tool built atop PENROSE, aimed at automating the generation of multiple-choice diagrammatic translation problems (Chapter 5).
- A dataset of translation problems that includes real-world diagrammatic problems in graph theory, chemistry, and Euclidean geometry (Section 5.5).
- Empirical evidence supporting the reliability, efficiency, and ecological validity of EDGEWORTH in educational contexts (Chapter 6).

7.2 Future work

We now discuss potential future directions for PENROSE, EDGEWORTH, and diagramming tool research in general.

7.2.1 Composable visual representations

Over the years of building diagrams using PENROSE and EDGEWORTH, we have observed that visual representations in different domains often share common visual components and layout patterns. Further, common visual techniques are widely used in diagramming to convey domain-independent concepts, such as using varying opacity or line weight to highlight parts of a diagram, maintaining layout consistency across multiple diagrams to form a visual narrative, or using sliders or other widgets to drive real-time physical simulations in interactive webpages. It seems natural to separate out these common patterns into their own components, suggesting that PENROSE's existing reusability of visual representations in STYLE does not provide sufficient flexibility for the needs of digital diagrammers.

In the current version of PENROSE, authors can reuse geometric and layout primitives to create new STYLE programs, and users consume these programs by writing different SUBSTANCE programs with them. Each STYLE program is standalone and self-contained, meaning that everything from the styling of points to the color palettes must be defined within that program. In practice, this means that common visual design patterns are copied and pasted between STYLE files. Additionally, it is common for individual diagrams within a domain to have customized visual elements to draw focus or illustrate a concept. Currently, the only way to override the domain-wide visual style in PENROSE is by using workarounds that involve more copying/pasting code in STYLE. These two limitations result in repetitive and lengthy programs that require high effort to edit and maintain, even for expert PENROSE users.

While code duplication and multiple versions of STYLE may be manageable on a small scale, we envision building a broader ecosystem of diagrams and this requires more flexible reuse mechanisms. We suggest **composability** as the main design goal for improving PENROSE. The existing layout primitives are an example of composability: authors can reuse and *combine* multiple primitives to form new layout problems. Looking forward, we plan to allow diagrammers to create *modules* of visual components and layout patterns. Through this mechanism, an author can draw together multiple different modules they need for their own diagram. And these modules can themselves be composed from other modules: for instance, a module for visualizing complex analysis might make use of lower-level modules for visualizing a coordinate plane and plotting curves, but build on top of that with domain-specific visuals for singularities in holomorphic functions. In addition to user-defined modules, there are also opportunities to build domain-independent visual techniques, such as individual object-level highlighting or annotations, into our languages or as standard library modules.

We believe this composable approach will open up new possibilities for diagrammers to collaborate and create more flexible, reusable, and expressive visual representations. Going forward, we plan to survey existing compose mechanisms such as modules, type systems, and package ecosystems to inform our design for PENROSE.

7.2.2 Knowledge-infused problem variation

EDGEWORTH provides a mixed-initiative [4] workflow: authors focus on specifying the content and the general direction of variations through the example scenario, while EDGEWORTH fully automates the details of variation generation and layout. The evaluation studies presented in

Chapter 6 showed that this workflow improves authoring speed and can produce useful diagrams to educators already. In this section, we focus on the current state of EDGEWORTH's outputs and propose future work for improving problem quality.

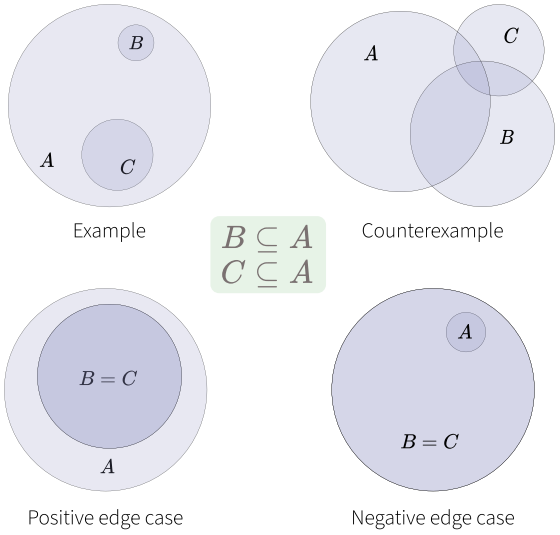
As discussed in Section 6.3, experts used terms like "*obviously incorrect*" (E6) and "*less obvious*" (E1) to characterize the quality of problem options in a multiple-choice translation problem. Based on their feedback, we divide these options into four categories: given a set of mathematical statements describing logical entities and their relationships, a diagram can be associated with them in one of the following ways:

• **Example**: the diagram represents the math statements, i.e., all the statements hold true in the diagram.

• **Counterexample**: the diagram clearly violates the math statements, i.e., one or more statements are false in the diagram.

• **Positive edge case**: the diagram is an example of the math statements, but contains extraneous entities and/or more specialized relationships.

• **Negative edge case**: the diagram is a counterexample, but only requires a few changes to become an example.



Using EDGEWORTH, the author creates an example scenario and EDGEWORTH's mutator generates a set of diagrams. When these diagrams don't satisfy the needs of the author (e.g., missing counterexamples that are important for an educational goal), the author can only generate more variations and hope to get better ones. For example, suppose an author would like to create problems that test students' knowledge of improper subsets, especially the fact that if $A \subseteq B$, $A = B$ is allowed. Using the EDGEWORTH, the author first creates a SUBSTANCE program and clicks "Generate Diagrams."

```
Set A, B, C
IsSubset(B, A)
IsSubset(C, A)
```

Ideally, EDGEWORTH should generate a set of examples of the subset relations that include the edge cases of $A = B$, $A = C$, or $B = C$, and counterexamples of $B \not\subseteq A$ or $C \not\subseteq A$. However, those particular mutated programs are extremely unlikely to be generated by EDGEWORTH. The default EDGEWORTH output for this scenario is show in Figure 7.1. There are useful counterexamples, but none of the diagrams include edge cases such as:

Figure 7.1: A screenshot of the EDGEWORTH interface, after generating examples for a translation problem focusing on improper subsets. The first pool of mutants aren't suitable for this problem.

```
Set A, B, C
IsSubset(B, A)
IsSubset(C, A)
Equal(B, C)
```

In our experience, it is not uncommon forEDGEWORTH to miss important edge cases.

In future work, we propose to work with Large Language Models (LLMs) to generate high-quality edge cases. Since LLMs are trained on the text of the entire internet, they may contain enough knowledge to suggest pedagogically usefual positive and negative edge cases.

To turn these conceptual edge cases into diagrams, PENROSE needs SUBSTANCE programs. Therefore, we will first test LLMs capability to generate SUBSTANCE programs.

Consider the case of a teacher authoring the example scenario (Section 5.3.1): imagine the teacher specifying the diagram in natural language and an augmented version of EDGEWORTH will prompt an LLM to generate the example diagram in SUBSTANCE. In preliminary work, we tested this use case and showed that GPT-4 does not do a good job of generating low-level visual code like SVG [83]. In contrast, when prompted carefully, GPT-4 can generate SUBSTANCE programs which yield correct and legible diagrams.

Assuming reliable SUBSTANCE generation capability, an LLM may use the author's inputs (i.e., example scenario SUBSTANCE and diagram and diagram choices in the mutant pool) together with its embedded domain knowledge to generate pedagogically useful edge cases. EDGEWORTH may use a mix of the existing mutation algorithm (Section 5.3.3) and an LLM to generate SUBSTANCE programs, for a balance of examples/counterexamples and edge cases. To iterate on the mutant pool, the author picks multiple diagrams in the pool and the LLM can be prompted again with the author's choices in its context to future generate more diagrams based on the

author's need.

We note a few challenges with the aforementioned approach. First, LLMs may need help on generating SUBSTANCE code because SUBSTANCE programs are few in number comparing with other languages in LLMs' training set. In addition to prompt-engineering, future work can experiment with LLM *agents* [179] so that authors can provide more granular input and feedback to the model. Second, code and natural language may be insufficient to produce high-quality problems, and future work should try leveraging the visual output of EDGEWORTH. The recent rise of visual question-answering (VQA) datasets and benchmarks shows a growing interest in improving LLMs visual reasoning capabilities [107, 13, 56, 110]. While LLMs' ability to reason with just images is still unclear [139], all diagrams produced using EDGEWORTH have both symbolic (i.e., SUBSTANCE, STYLE, and DOMAIN) and visual (i.e., the output SVGs) representations. Future research should investigate how to incorporate both representations in prompting, fine-tuning, and potentially pre-training of LLMs so that models will be capable of producing high-quality diagrams for all desirable diagram classes.

7.2.3 Interactive diagrams

Diagrams live in the context of surrounding text, overlaid annotations, and human gestures. The web opens up opportunities for even richer in-context interaction. In education, though students spend more time on digital platforms, they often see diagrams that are presented exactly as before: pixelated, static, and ornamental. In contrast with a static diagram, a semantics-preserving interactive diagram allows students to rapidly explore alternatives, understand the underlying rules of a visual representation, and receive instant feedback on their actions [94]. Meaningful interaction with diagrams helps students move from passive recognition to active synthesis of visual representations [96].

Sadly, interactive diagrams are scarce in the wild. Most interactive documents require authors to be proficient in general-purpose programming and have decent knowledge in handling low-level events like mouse down/up, hover, etc. As a result, a simple interactive diagram often takes up 100s of lines-of-code and can be hard to debug [121, 102]. Additionally, because interactive diagrams change a lot, authors often need to reason about a collection of diagrams, making the task even harder.

PENROSE and EDGEWORTH elevate the semantics of diagrams from low-level primitives to mathematically meaningful notations. Specifically, PENROSE encodes both the translational semantics of how notations are translated to diagrams, and the visual semantics of how shape primitives relate to each other expressed as constraints. By exploiting both, we can automatically support semantics-preserving interactive diagrams. One promising direction of future work is to investigate how to build interactive diagram activities that are automatically derived from PENROSE diagrams without extensive programming effort. In short, I propose to (1) simplify programming interactive diagrams and (2) provide students with rich, automated feedback by leveraging the encoding of visual representations.

7.3 Concluding remarks

Curiously, building authoring tools for rich, interactive diagrams, narratives, and learning activities seems just the right amount of material for a second dissertation,[1] or a full-time job.

[1] In the spirit of Barik [7]

Bibliography

[1] S. Ainsworth, V. Prain, and R. Tytler. “Drawing to learn in science”. *Science* 333.6046 (2011), pp. 1096–1097 (cit. on pp. 6, 21).

[2] V. Aleven, B. M. McLaren, J. Sewall, and K. R. Koedinger. “The cognitive tutor authoring tools (CTAT): Preliminary evaluation of efficiency gains”. *International Conference on Intelligent Tutoring Systems*. Vol. 4053 LNCS. Jhongli, Taiwan: Springer, 2006, pp. 61–70 (cit. on pp. 9, 65).

[3] V. Aleven, J. Sewall, B. M. McLaren, and K. R. Koedinger. “Rapid authoring of Intelligent Tutors for real-world and experimental use”. *Proceedings - Sixth International Conference on Advanced Learning Technologies, ICALT 2006* 2006 (2006), pp. 847–851 (cit. on p. 9).

[4] J. E. Allen, C. I. Guinn, and E. Horvtz. “Mixed-initiative interaction”. *IEEE Intelligent Systems and their Applications* 14.5 (1999), pp. 14–23 (cit. on p. 88).

[5] E. Andersen, S. Gulwani, and Z. Popovic. “A trace-based framework for analyzing and synthesizing educational progressions”. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’13. New York, NY, USA: Association for Computing Machinery, Apr. 2013, pp. 773–782 (cit. on p. 9).

[6] R. Arnheim. *Visual Thinking*. en. University of California Press, 1969 (cit. on pp. 5, 7).

[7] T. Barik. “Error Messages as Rational Reconstructions”. English. Ph.D. United States – North Carolina: North Carolina State University (cit. on p. 92).

[8] J. Barnes and P. Hut. “A hierarchical O (N log N) force-calculation algorithm”. *nature* 324.6096 (1986), pp. 446–449 (cit. on p. 56).

[9] D. Barrett, F. Hill, A. Santoro, A. Morcos, and T. Lillicrap. “Measuring abstract reasoning in neural networks”. *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. PMLR, 2018, pp. 511–520 (cit. on p. 10).

[10] J. Barwise. “Heterogeneous reasoning”. en. *Conceptual Graphs for Knowledge Representation*. Berlin, Heidelberg: Springer, 1993, pp. 64–74 (cit. on p. 6).

[11] J. Barwise and J. Etchemendy. “Visual information and valid reasoning”. *Philosophy And The Computer*. 2019, pp. 160–182 (cit. on p. 6).

[12] B. Beeton and R. Palais. “Comm. of math. with TEX”. *Visible Language* 50.2 (2016) (cit. on pp. 27, 31).

[13] J. Belouadi, A. Lauscher, and S. Eger. *AutomaTikZ: Text-Guided Synthesis of Scientific Vector Graphics with TikZ*. 2024 (cit. on p. 91).

[14] J. Bender and M. Marrinan. *The Culture of Diagram*. Stanford: Stanford University Press, 2010 (cit. on p. 5).

[15] H. R. Bernard and H. R. Bernard. *Social research methods: Qualitative and quantitative approaches*. Sage, 2013 (cit. on p. 13).

[16] T. Berners-Lee and D. Connolly. *Hypertext markup language-2.0*. 1995 (cit. on p. 30).

[17] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013 (cit. on p. 31).

[18] E. D. Bloch. *A first course in geometric topology and differential geometry*. Springer Science & Business Media, 1997 (cit. on p. 50).

[19] A. Blum and T. Mitchell. "Combining labeled and unlabeled data with co-training". *Proceedings of the Annual ACM Conference on Computational Learning Theory* (1998), pp. 92–100 (cit. on p. 3).

[20] E. Bobek and B. Tversky. "Creating visual explanations improves learning". *Cognitive Research: Principles and Implications* 1.1 (Dec. 2016), pp. 1–14 (cit. on p. 6).

[21] A. Borning. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory". *ACM Trans. Program. Lang. Syst.* 3.4 (Oct. 1981), pp. 353–387 (cit. on p. 8).

[22] M. Bostock, V. Ogievetsky, and J. Heer. "D3 data-driven documents". *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2301–2309 (cit. on p. 6).

[23] J. C. Bowman and A. Hammerlindl. "Asymptote: A vector graphics language". *TUGboat: The Comm. of the TEX Users Group* 29.2 (2008), pp. 288–294 (cit. on p. 8).

[24] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004 (cit. on p. 40).

[25] V. Braun and V. Clarke. "Using thematic analysis in psychology". *Qualitative research in psychology* 3.2 (2006), pp. 77–101 (cit. on p. 14).

[26] Bret Victor. *Drawing dynamic visualizations*. 2013 (cit. on p. 8).

[27] E. B. Burger, D. J. Chard, E. J. Hall, P. A. Kennedy, S. J. Leinwand, F. L. Renfro, D. G. Seymour, and B. K. Wattis. *Holt geometry*. Holt, Rinehart and Winston, 2007 (cit. on p. 66).

[28] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. "Scaling up visual programming languages". 28.3 (1995), pp. 45–54 (cit. on p. 32).

[29] B. Buxton. *Sketching user experiences: getting the design right and the right design*. Morgan kaufmann, 2010 (cit. on p. 20).

[30] O. Byrne. *The first six books of the Elements of Euclid: in which coloured diagrams and symbols are used instead of letters for the greater ease of learners*. 1847 (cit. on p. 47).

[31] S. K. Card. *Readings in information visualization: using vision to think*. eng. The Morgan Kaufmann series in interactive technologies. San Francisco, Calif: Morgan Kaufmann Publishers, 1999 (cit. on p. 5).

[32] P. A. Carpenter, M. A. Just, and P. Shell. "What one intelligence test measures: A theoretical account of the processing in the Raven progressive matrices test". *Psychological Review* 97.3 (1990), pp. 404–431 (cit. on p. 10).

[33] N. Carter and R. Ellis. *Group explorer version 3.0*. manual. Waltham, MA, 2019 (cit. on p. 30).

[34] D. Cervone. “MathJax: a platform for mathematics on the Web”. *Notices of the AMS* 59.2 (2012), pp. 312–316 (cit. on p. 42).

[35] M. T. Chi and R. Wylie. “The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes”. *Educational Psychologist* 49.4 (Oct. 2014), pp. 219–243 (cit. on p. 7).

[36] P. P. Chik and M. L. Lo. “Simultaneity and the enacted object of learning”. *Classroom Discourse and the Space of Learning*. Vol. 9781410609. Routledge, Apr. 2004, pp. 89–112 (cit. on p. 7).

[37] R. Chugh. “Prodirect manipulation: bidirectional programming for the masses”. *Proceedings of the 38th international conference on software engineering companion*. ACM, 2016, pp. 781–784 (cit. on p. 23).

[38] J. Cohen. “A coefficient of agreement for nominal scales”. *Educational and psychological measurement* 20.1 (1960), pp. 37–46 (cit. on p. 73).

[39] D. Cole. “The chinese room argument”. *The stanford encyclopedia of philosophy*. Metaphysics Research Lab, Stanford University, 2014 (cit. on p. 28).

[40] W. Commons. *Illustration to euclid’s proof of the pythagorean theorem*. 2006 (cit. on p. 47).

[41] R. Coulon, G. Dorfsman-Hopkins, E. Harriss, M. Skrodzki, K. E. Stange, and G. Whitney. “On the importance of illustration for mathematical research”. *Notices of the American Mathematical Society* 71.01 (Jan. 2024), p. 1 (cit. on p. 2).

[42] K. Crane, F. de Goes, M. Desbrun, and P. Schröder. “Digital geometry processing with discrete exterior calculus”. *ACM SIGGRAPH 2013 courses*. ACM, 2013 (cit. on p. 47).

[43] L. Deslauriers, L. S. McCarty, K. Miller, K. Callaghan, and G. Kestin. “Measuring actual learning versus feeling of learning in response to being actively engaged in the classroom”. *Proceedings of the National Academy of Sciences of the United States of America* 116.39 (Sept. 2019), pp. 19251–19257 (cit. on p. 7).

[44] A. A. diSessa and B. L. Sherin. “Meta-representation: an introduction”. *The Journal of Mathematical Behavior* 19.4 (Oct. 2000), pp. 385–398 (cit. on p. 58).

[45] Elijah Meeks. *Third wave data visualization*. 2018 (cit. on p. 8).

[46] C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen. “National instruments LabVIEW: a programming environment for laboratory automation and measurement”. *JALA: J. of Assoc. for Lab. Auto.* 12.1 (2007), pp. 17–24 (cit. on p. 32).

[47] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum. “Learning to infer graphics programs from hand-drawn images”. *Advances in neural information processing systems 31*. Curran Associates, Inc., 2018, pp. 6059–6068 (cit. on p. 21).

[48] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. “Graphviz and dynagraph − static and dynamic graph drawing tools”. *Graph drawing software*. Springer, 2004, pp. 127–148 (cit. on pp. 8, 24, 30, 33).

[49] S. E. Embretson. “A Cognitive Design System Approach to Generating Valid Tests: Application to Abstract Reasoning”. *Psychological Methods* 3.3 (1998), pp. 380–396 (cit. on p. 10).

[50] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. "The State of the Art in Language Workbenches". en. *Software Language Engineering*. Vol. 8225. Cham: Springer International Publishing, 2013, pp. 197–217 (cit. on p. 24).

[51] K. A. Ericsson. "The Influence of Experience and Deliberate Practice on the Development of Superior Expert Performance." *The Cambridge handbook of expertise and expert performance*. New York, NY, US: Cambridge University Press, 2006, pp. 683–703 (cit. on p. 7).

[52] S. M. Ervin. "Designing with diagrams: a role for computing in design education and exploration". *The Electronic Design Studio, The MIT Press, Cambridge, Massachusetts* (1990), pp. 107–122 (cit. on p. 6).

[53] S. M. Ervin. "Designing with diagrams: a role for computing in design education and exploration". *The Electronic Design Studio, The MIT Press, Cambridge, Massachusetts*. Cambridge, Massachusetts: The MIT Press, 1990, pp. 107–122 (cit. on pp. 6, 23).

[54] Facebook. *React: A JavaScript library for building user interfaces*. 2020 (cit. on p. 42).

[55] J. R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. "Fine-grained and accurate source code differencing". *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, Inc, 2014, pp. 313–323 (cit. on p. 64).

[56] B. Fatemi, J. Halcrow, and B. Perozzi. *Talk like a Graph: Encoding Graphs for Large Language Models*. Oct. 2023 (cit. on p. 91).

[57] J. C. Flanagan. "The critical incident technique." *Psychological bulletin* 51.4 (1954), p. 327 (cit. on p. 13).

[58] M. Ganesalingam. "The language of mathematics". 2013 (cit. on p. 31).

[59] M. J. Gierl and T. M. Haladyna. *Automatic item generation: Theory and practice*. Routledge, 2012 (cit. on p. 9).

[60] M. J. Gierl and H. Lai. "The Role of Item Models in Automatic Item Generation". *International Journal of Testing* 12.3 (July 2012), pp. 273–298 (cit. on p. 9).

[61] M. Gleicher and A. Witkin. "Drawing with constraints". *The Visual Computer 1994 11:1* 11.1 (Jan. 1994), pp. 39–51 (cit. on p. 24).

[62] V. Goel. *Sketches of thought*. MIt Press, 1995 (cit. on p. 20).

[63] T. R. G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". *Journal of Visual Languages & Computing* 7.2 (June 1996), pp. 131–174 (cit. on pp. 12, 24).

[64] M. D. Gross. "The electronic cocktail napkin − a computational environment for working with design diagrams". *Design studies* 17.1 (1996), pp. 53–69 (cit. on p. 22).

[65] S. Gulwani. "Example-based Learning in Computer-aided STEM Education". *Commun. ACM* 57.8 (Aug. 2014), pp. 70–80 (cit. on p. 9).

[66] S. Gulwani, W. R. Harris, and R. Singh. "Spreadsheet data manipulation using examples". *Communications of the ACM* 55.8 (2012), pp. 97–105 (cit. on p. 23).

[67] S. Gulwani, V. A. Korthikanti, and A. Tiwari. “Synthesizing geometry constructions”. *ACM SIGPLAN Notices* 46.6 (June 2011), pp. 50–61 (cit. on p. 9).

[68] S. Gulwani, O. Polozov, and R. Singh. “Program Synthesis”. English. *Foundations and Trendső in Programming Languages* 4.1-2 (July 2017), pp. 1–119 (cit. on p. 21).

[69] J. Hadamard. *The Mathematician’s Mind: The Psychology of Invention in the Mathematical Field*. en. Princeton University Press, 1945 (cit. on p. 1).

[70] D. F. Halpern, A. Graesser, and M. Hakel. “Learning principles to guide pedagogy and the design of learning environments”. *Association for Psychological Science* (2007) (cit. on p. 7).

[71] P. Hanrahan. “Systems of thought”. *EuroVis 2009 keynote address* (2009), pp. 10–12 (cit. on p. 11).

[72] R. L. Harris. “Information graphics: A comprehensive illustrated reference”. Oxford University Press, 2000, p. 100 (cit. on p. 5).

[73] M. Hegarty and M. Kozhevnikov. “Types of visual − spatial representations and mathematical problem solving”. *Journal of Educational Psychology* 91.4 (1999), pp. 684–689 (cit. on p. 6).

[74] B. Hempel and R. Chugh. “Semi-automated SVG programming via direct manipulation”. *Proceedings of the 29th annual symposium on user interface software and technology*. Uist ’16. New York, NY, USA: ACM, 2016, pp. 379–390 (cit. on p. 21).

[75] Y. Hiroshi and T. Tanabe. “A Primal-Dual Exterior Point Method for Nonlinear Optimization”. *http://dx.doi.org/10.1137/060676970* 20.6 (Nov. 2010), pp. 3335–3363 (cit. on p. 40).

[76] H. Holling, J. P. Bertling, and N. Zeuch. “Automatic item generation of probability word problems”. *Studies in Educational Evaluation* 35.2 (2009), pp. 71–76 (cit. on p. 9).

[77] L. F. Hornke and M. W. Habon. “Rule-Based Item Bank Construction and Evaluation Within the Linear Logistic Framework”. *Applied Psychological Measurement* 10.4 (Dec. 1986), pp. 369–380 (cit. on p. 10).

[78] T. Hottelier, R. Bodik, and K. Ryokai. “Programming by manipulation for layout”. *Proceedings of the 27th annual ACM symposium on user interface software and technology*. Uist ’14. New York, NY, USA: ACM, 2014, pp. 231–241 (cit. on pp. 23, 24).

[79] C. D. Hundhausen and J. L. Brown. “What you see is what you code: a “live” algorithm development and visualization environment for novice learners”. *Journal of Visual Languages & Computing* 18.1 (2007), pp. 22–47 (cit. on p. 23).

[80] N. Hurst, W. Li, and K. Marriott. “Review of automatic document formatting”. *Proceedings of the 9th ACM symposium on document engineering*. ACM, 2009, pp. 99–108 (cit. on p. 24).

[81] E. L. Hutchins, J. D. Hollan, and D. A. Norman. “Direct manipulation interfaces”. *Human-computer interaction* 1.4 (1985), pp. 311–338 (cit. on pp. 22, 24).

[82] J. Jacobs, J. Brandt, R. Mech, and M. Resnick. “Extending manual drawing practices with artist-centric programming tools”. *Proceedings of the 2018 CHI conference on human factors in computing systems*. ACM, 2018, p. 590 (cit. on p. 23).

[83] R. Jain, W. Ni, and J. Sunshine. “Generating domain-specific programs for diagram authoring with large language models”. *Companion proceedings of the 2023 ACM SIGPLAN international conference on systems, programming, languages, and applications: Software for humanity*. SPLASH’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 70–71 (cit. on p. 90).

[84] P. Janii. “GCLC-a tool for constructive euclidean geometry and more than that”. *Int. Con. on math. Soft.* Springer, 2006, pp. 58–73 (cit. on p. 47).

[85] A. Kay. *Doing with images makes symbols*. 1987 (cit. on p. 1).

[86] R. H. Kazi, T. Grossman, H. Cheong, A. Hashemi, and G. W. Fitzmaurice. “DreamSketch: Early stage 3D design explorations with sketching and generative design.” *UIST*. 2017, pp. 401–414 (cit. on p. 21).

[87] P. J. Kellman and C. M. Massey. “Perceptual Learning, Cognition, and Expertise”. *Psychology of Learning and Motivation - Advances in Research and Theory*. Vol. 58. Academic Press, Jan. 2013, pp. 117–165 (cit. on p. 59).

[88] P. J. Kellman, C. M. Massey, and J. Y. Son. “Perceptual learning modules in mathematics: Enhancing students’ pattern recognition, structure extraction, and fluency”. *Topics in Cognitive Science* 2.2 (Apr. 2010), pp. 285–305 (cit. on pp. 1, 7, 58).

[89] M. B. Kery, A. Horvath, and B. Myers. “Variolite: Supporting exploratory programming by data scientists”. *Proceedings of the 2017 CHI conference on human factors in computing systems*. Chi ’17. New York, NY, USA: ACM, 2017, pp. 1265–1276 (cit. on p. 21).

[90] M. B. Kery and B. A. Myers. “Exploring exploratory programming”. *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, 2017, pp. 25–29 (cit. on p. 21).

[91] E. Kmett. *ad: automatic differentiation*. 2015 (cit. on p. 43).

[92] K. R. Koedinger. “Emergent properties and structural constraints: Advantages of diagrammatic representations for reasoning and learning”. *Proc. AAAI Spring Symposium on Reasoning with Diagrammatic Representations*. 1992, pp. 154–169 (cit. on p. 6).

[93] K. R. Koedinger and J. R. Anderson. “Abstract planning and perceptual chunks: Elements of expertise in geometry”. *Cognitive Science* 14.4 (Oct. 1990), pp. 511–550 (cit. on pp. 1, 58).

[94] K. R. Koedinger, J. Kim, J. Z. Jia, E. A. McLaughlin, and N. L. Bier. “Learning is Not a Spectator Sport: Doing is Better than Watching for Learning from a MOOC”. *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*. L@S ’15. New York, NY, USA: Association for Computing Machinery, Mar. 2015, pp. 111–120 (cit. on p. 91).

[95] T. Kosar, M. Mernik, and J. C. Carver. “Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments”. *Empirical software engineering* 17.3 (2012), pp. 276–304 (cit. on p. 30).

[96] D. R. Krathwohl and L. W. Anderson. *A taxonomy for learning, teaching, and assessing: A revision of Bloom’s taxonomy of educational objectives*. Longman, 2009 (cit. on pp. 11, 91).

[97] G. Kurdi, J. Leo, B. Parsia, U. Sattler, and S. Al-Emari. “A Systematic Review of Automatic Question Generation for Educational Purposes”. *International Journal of Artificial Intelligence in Education* 30.1 (Mar. 2020), pp. 121–204 (cit. on p. 9).

[98] D. Kurlander and S. Feiner. “Inferring constraints from multiple snapshots”. *ACM Trans. Graph.* 12.4 (Oct. 1993), pp. 277–304 (cit. on p. 24).

[99] J. A. Landay and B. A. Myers. *Interactive sketching for the early stages of user interface design*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994 (cit. on p. 21).

[100] J. H. Larkin and H. A. Simon. “Why a Diagram is (Sometimes) Worth Ten Thousand Words”. *Cognitive Science* 11.1 (Jan. 1987), pp. 65–100 (cit. on pp. 1, 6, 7, 14, 17).

[101] D. Ledo, S. Houben, J. Vermeulen, N. Marquardt, L. Oehlberg, and S. Greenberg. “Evaluation strategies for HCI Toolkit research”. *Conference on Human Factors in Computing Systems - Proceedings* 2018-April (Apr. 2018) (cit. on p. 81).

[102] C. Letondal, S. Chatty, W. G. Phillips, F. André, and S. Conversy. “Usability requirements for interaction-oriented development tools”. *Psychology of Programming* (2010), pp. 12–26 (cit. on p. 91).

[103] A. S. Lewis and M. L. Overton. “Nonsmooth optimization via BFGS”. *SIAM J. Optimiz* (2009), pp. 1–35 (cit. on p. 41).

[104] H. W. Lie, B. Bos, C. Lilley, and I. Jacobs. *Cascading style sheets*. Pearson India, 2005 (cit. on p. 30).

[105] H. Limerick, D. Coyle, and J. W. Moore. “The experience of agency in human-computer interactions: a review”. English. *Frontiers in Human Neuroscience* 8 (2014) (cit. on p. 22).

[106] Z. Liu and J. Heer. “The effects of interactive latency on exploratory visual analysis”. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2014) (cit. on p. 55).

[107] P. Lu, H. Bansal, T. Xia, J. Liu, C. Li, H. Hajishirzi, H. Cheng, K.-W. Chang, M. Galley, and J. Gao. *MathVista: Evaluating Mathematical Reasoning of Foundation Models in Visual Contexts*. Jan. 2024 (cit. on p. 91).

[108] D. Ma’ayan, W. Ni, K. Ye, C. Kulkarni, and J. Sunshine. “How Domain Experts Create Conceptual Diagrams and Implications for Tool Design”. *Conference on Human Factors in Computing Systems - Proceedings* 20 (Apr. 2020) (cit. on pp. 11, 60).

[109] F. Marton. “Sameness and Difference in Transfer”. *Journal of the Learning Sciences* 15.4 (2006), pp. 499–535 (cit. on pp. 7, 59).

[110] A. Masry, D. X. Long, J. Q. Tan, S. Joty, and E. Hoque. *ChartQA: A Benchmark for Question Answering about Charts with Visual and Logical Reasoning*. Mar. 2022 (cit. on p. 91).

[111] R. E. Mayer. “The promise of multimedia learning: using the same instructional design methods across different media”. *Learning and instruction* 13.2 (2003), pp. 125–139 (cit. on p. 11).

[112] R. E. Mayer. “Multimedia learning”. *Psychology of Learning and Motivation*. Vol. 41. Academic Press, Jan. 2002, pp. 85–139 (cit. on pp. 1, 3, 6).

[113] S. McDirmid. “Living it up with a live programming language”. *Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems and applications*. Oopsla ’07. New York, NY, USA: ACM, 2007, pp. 623–638 (cit. on p. 23).

[114] S. McDirmid and W. C. Hsieh. “SuperGlue: Component Programming with Object-Oriented Signals”. en. *ECOOP 2006 Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2006, pp. 206–229 (cit. on p. 23).

[115] M. McKeon. “Harnessing the Information Ecosystem with Wiki-based Visualization Dashboards”. *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1081–1088 (cit. on p. 22).

[116] R. McNeel et al. “Grasshopper: generative modeling software for Rhino” (2010) (cit. on p. 32).

[117] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of standard ML*. 1997 (cit. on p. 31).

[118] R. Miner. “The importance of MathML to mathematics communication”. *Notices of the AMS* 52.5 (2005), pp. 532–538 (cit. on p. 31).

[119] D. Moritz, C. Wang, G. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. “Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco”. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019) (cit. on p. 22).

[120] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. “The Lean theorem prover”. *International conf. on automated deduction*. Springer, 2015, pp. 378–388 (cit. on p. 31).

[121] B. A. Myers. “Separating application code from toolkits: Eliminating the Spaghetti of call-backs”. *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991*. New York, New York, USA: ACM Press, 1991, pp. 211–220 (cit. on p. 91).

[122] B. A. Myers, J. F. Pane, and A. Ko. “Natural programming languages and environments”. *Communications of the ACM* 47.9 (2004), pp. 47–52 (cit. on pp. 12, 19).

[123] B. A. Myers, A. Lai, T. M. Le, Y. S. Yoon, A. Faulring, and J. Brandt. “Selective undo support for painting applications”. *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. Chi ’15. New York, NY, USA: ACM, 2015, pp. 4227–4236 (cit. on p. 21).

[124] M. J. Nathan, K. R. Koedinger, M. W. Alibali, et al. “Expert blind spot: When content knowledge eclipses pedagogical content knowledge”. *Proceedings of the third international conference on cognitive science*. Vol. 644648. 2001 (cit. on p. 83).

[125] M. J. Nathan, A. C. Stephens, D. K. Masarik, M. W. Alibali, and K. R. Koedinger. “Representational fluency in middle school: A classroom study”. *Proceedings of the twenty-fourth annual meeting of the North American chapter of the International Group for the Psychology of Mathematics Education*. Vol. 1. ERIC Clearinghouse for Science, Mathematics and Environmental Education~…, 2002, pp. 462–472 (cit. on p. 58).

[126] A. Newell. *Unified theories of cognition*. Harvard University Press, 1994 (cit. on p. 55).

[127] W. Ni, S. Estep, H.-S. Harriman, K. R. Koedinger, and J. Sunshine. “Edgeworth: Efficient and Scalable Authoring of Visual Thinking Activities”. *Proceedings of the Eleventh ACM Conference on Learning @ Scale*. L@S ’24. New York, NY, USA: Association for Computing Machinery, July 2024, pp. 98–109 (cit. on pp. 58, 72, 74).

[128] W. Ni, K. Ye, J. Sunshine, J. Aldrich, and K. Crane. “Substance and Style: domain-specific languages for mathematical diagrams”. *Domain-specific language design and implementation*. DSLDI’17. 2017 (cit. on pp. 13, 22).

[129] C. Olah. *Understanding LSTM networks*. 2015 (cit. on p. 11).

[130] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer. “Live functional programming with typed holes”. *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 14 (cit. on p. 23).

[131] S. Oney, B. A. Myers, and J. Brandt. “Euclase: A live development environment with constraints and FSMs”. *2013 1st International Workshop on Live Programming (LIVE)*. May 2013, pp. 15–18 (cit. on p. 24).

[132] Open Learning Initiative. *General chemistry 1*. 2024 (cit. on p. 70).

[133] F. G. Paas and J. J. Van Merriënboer. “Variability of Worked Examples and Transfer of Geometrical Problem-Solving Skills: A Cognitive-Load Approach”. *Journal of Educational Psychology* 86.1 (1994), pp. 122–133 (cit. on p. 7).

[134] H. Pashler, P. M. Bain, B. A. Bottge, A. Graesser, K. Koedinger, M. McDaniel, and J. Metcalfe. *Organizing Instruction and Study to Improve Student Learning*. Tech. rep. Washington, DC: NCER, IES,, U.S. Department of Education, 2007 (cit. on p. 7).

[135] R. Patel, S. Sanghavi, D. Gupta, and M. S. Raval. “CheckIt - A low cost mobile OMR system”. *TENCON 2015 - 2015 IEEE region 10 conference*. 2015, pp. 1–5 (cit. on p. 9).

[136] C. S. Peirce. *Collected papers*. eng. Cambridge: Harvard University Press, 1931 (cit. on p. 6).

[137] M. Pharr, W. Jakob, and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016 (cit. on pp. 42, 55).

[138] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012 (cit. on p. 49).

[139] P. Rahmanzadehgervi, L. Bolton, M. R. Taesiri, and A. T. Nguyen. *Vision language models are blind*. July 2024 (cit. on p. 91).

[140] M. A. Rau. “Conceptual learning with multiple graphical representations: Intelligent tutoring systems support for sense-making and fluency-building processes”. en. PhD thesis. Carnegie Mellon University, 2013 (cit. on pp. 1, 3, 7).

[141] M. A. Rau. “Conditions for the Effectiveness of Multiple Visual Representations in Enhancing STEM Learning”. en. *Educational Psychology Review* 29.4 (Dec. 2017), pp. 717–761 (cit. on p. 7).

[142] L. Razzaq, J. Patvarczki, S. F. Almeida, M. Vartak, M. Feng, N. T. Heffernan, and K. R. Koedinger. “The ASSISTment builder: Supporting the life cycle of tutoring system content creation”. *IEEE Transactions on Learning Technologies* 2.2 (2009), pp. 157–166 (cit. on pp. 9, 65).

[143] C. Reas and B. Fry. “Processing: Programming for the media arts”. *AI & SOCIETY* 20.4 (Sept. 2006), pp. 526–538 (cit. on p. 8).

[144] S. P. Reiss, Q. Xin, and J. Huang. “SEEDE: Simultaneous execution and editing in a development environment”. *Proceedings of the 33rd ACM/IEEE international conference*

on automated software engineering. Ase 2018. New York, NY, USA: ACM, 2018, pp. 270–281 (cit. on p. 23).

[145] C. Richards. "The Fundamental Design Variables of Diagramming". en. *Diagrammatic Representation and Reasoning*. London: Springer London, 2002, pp. 85–102 (cit. on p. 23).

[146] Y. Riche, N. Henry Riche, K. Hinckley, S. Panabaker, S. Fuelling, and S. Williams. "As we may ink?: Learning from everyday analog pen use to improve digital ink experiences". *Proceedings of the 2017 CHI conference on human factors in computing systems*. Chi '17. New York, NY, USA: ACM, 2017, pp. 3241–3253 (cit. on p. 20).

[147] D. Rohrer and K. Taylor. "The shuffling of mathematics problems improves learning". *Instructional Science* 35.6 (Nov. 2007), pp. 481–498 (cit. on p. 7).

[148] K. H. Rosen. *Discrete mathematics & applications*. McGraw-Hill, 1999 (cit. on p. 70).

[149] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. Thompson, M. Brehmer, and Z. Liu. "Critical reflections on visualization authoring systems". *IEEE Transactions on Visualization and Computer Graphics* 26.1 (Jan. 2020), pp. 461–471 (cit. on p. 8).

[150] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. "Declarative interaction design for data visualization". *UIST 2014 - Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, 2014, pp. 669–678 (cit. on p. 6).

[151] R. Sawhney and K. Crane. *geometry-processing.js: A fast and flexible geometry processing library*. 2017 (cit. on p. 50).

[152] H. L. Schnackenberg, H. J. Sullivan, L. F. Leader, and E. E. Jones. "Learner preferences and achievement under differing amounts of learner practice". *Educational Technology Research and Development* 46.2 (1998), pp. 5–16 (cit. on p. 7).

[153] B. Sheil. "Environments for exploratory programming". *Datamation* 29.7 (1983), pp. 131–144 (cit. on p. 21).

[154] S.-J. Shin, O. Lemon, and J. Mumma. "Diagrams". *The Stanford Encyclopedia of Philosophy*. Winter 2018. Metaphysics Research Lab, Stanford University, 2018 (cit. on p. 6).

[155] B. Shneiderman. "Direct manipulation: a step beyond programming languages". *Computer* 16.8 (1983), pp. 57–69 (cit. on p. 22).

[156] R. Singh, S. Gulwani, and S. Rajamani. "Automatically Generating Algebra Problems". *Proceedings of the AAAI Conference on Artificial Intelligence* 26.1 (2012), pp. 1620–1628 (cit. on p. 9).

[157] J. Strömberg. "Integrating constraints with a drawing CAD application". *Stockholm University* (2006) (cit. on p. 24).

[158] I. E. Sutherland. "Sketchpad a man-machine graphical communication system". *Simulation* 2.5 (1964), R–3 (cit. on pp. 8, 24, 31).

[159] M. Suwa and B. Tversky. "What do architects and students perceive in their design sketches? A protocol analysis". *Design studies* 18.4 (1997), pp. 385–403 (cit. on p. 21).

[160] S. L. Tanimoto. "VIVA: a visual language for image processing". 1.2 (June 1990), pp. 127–139 (cit. on p. 23).

[161] T. Tantau. *The TikZ and PGF packages. Manual for version 3.0.0*. manual. Dec. 2013 (cit. on p. 8).

[162] W. P. Thurston. "On proof and progress in mathematics". *Bulletin of the American Mathematical Society* 30.2 (Apr. 1994), pp. 161–177 (cit. on p. 2).

[163] C. D. Tippett. "What recent research on diagrams suggests about learning with rather than learning from visual representations in science". *International Journal of Science Education* 38.5 (Apr. 2016), pp. 725–746 (cit. on p. 6).

[164] L. Torvalds and J. Hamano. "Git: Fast version control system". *URL http://git-scm. com* (2010) (cit. on p. 18).

[165] E. R. Tufte and P. R. Graves-Morris. *The visual display of quantitative information*. Vol. 2. Graphics press Cheshire, CT, 1983 (cit. on p. 5).

[166] B. Tversky. "Diagrams". *Information Design*. Routledge, 2017 (cit. on p. 5).

[167] B. Tversky. "Visualizing Thought". *Topics in Cognitive Science* 3.3 (July 2011), pp. 499–535 (cit. on p. 5).

[168] A. Van Deursen, P. Klint, and J. Visser. "Domain-specific languages: An annotated bibliography". *ACM Sigplan Notices* 35.6 (2000), pp. 26–36 (cit. on pp. 24, 30).

[169] K. VanLehn. "The Behavior of Tutoring Systems". *International Journal of Artificial Intelligence in Education* 16.3 (Jan. 2006), pp. 227–265 (cit. on p. 9).

[170] B. Victor. *Up and Down the Ladder of Abstraction: A systematic approach to interactive visualization*. 2011 (cit. on p. 22).

[171] M. Voelter and V. Pech. "Language modularity with the MPS language workbench". *2012 34th International Conference on Software Engineering (ICSE)*. June 2012, pp. 1449–1450 (cit. on p. 24).

[172] K. Wang and Z. Su. "Automatic generation of Raven's progressive Matrices". *IJCAI International Joint Conference on Artificial Intelligence*. Vol. 2015-Janua. 2015, pp. 903–909 (cit. on p. 10).

[173] D. Weitekamp, E. Harpstead, and K. R. Koedinger. "An Interaction Design for Machine Teaching to Develop AI Tutors". *Conference on Human Factors in Computing Systems - Proceedings* (Apr. 2020) (cit. on p. 9).

[174] E. Wiese. "Toward Sense Making with Grounded Feedback". en. Thesis. July 2018 (cit. on p. 1).

[175] Wikipedia contributors. *Diffie-Hellman key exchange — Wikipedia, The Free Encyclopedia*. 2019 (cit. on p. 11).

[176] L. Wilkinson. "The Grammar of Graphics". *Handbook of Computational Statistics* (2012), pp. 375–414 (cit. on pp. 6, 22).

[177] E. Williger, C. Lilley, D. Schulze, B. Brinza, D. Storey, and A. Bellamy-Royds. *Scalable vector graphics (SVG) 2*. Candidate Recommendation. W3C, Oct. 2018 (cit. on p. 8).

[178] J. O. Wobbrock and J. A. Kientz. "Research contributions in human-computer interaction". *interactions* 23.3 (Apr. 2016), pp. 38–44 (cit. on pp. 3, 4).

[179] Y. Wu, Y. Fan, S. Y. Min, S. Prabhumoye, S. McAleer, Y. Bisk, R. Salakhutdinov, Y. Li, and T. Mitchell. *AgentKit: Structured LLM Reasoning with Dynamic Graphs*. July 2024 (cit. on p. 91).

[180] Y. Yang, D. Sanyal, J. Michelson, J. Ainooson, and M. Kunda. “Automatic Item Generation of Figural Analogy Problems: A Review and Outlook” (Jan. 2022) (cit. on p. 10).

[181] K. Ye, K. Crane, J. Aldrich, and J. Sunshine. “Designing extensible, domain-specific languages for mathematical diagrams”. *Off the Beaten Track*. OBT’17. Jan. 2017 (cit. on pp. 11, 13).

[182] K. Ye, W. Ni, M. Krieger, D. Ma’ayan, J. Wise, J. Aldrich, J. Sunshine, and K. Crane. “Penrose: From Mathematical Notation to Beautiful Diagrams”. *ACM Transactions on Graphics* 39.4 (July 2020), 144:144:1–144:144:16 (cit. on pp. 26, 64, 66).

[183] Y. S. Yoon and B. A. Myers. “A longitudinal study of programmers’ backtracking”. *2014 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, 2014, pp. 101–108 (cit. on p. 21).

[184] Y. S. Yoon and B. A. Myers. “Supporting Selective Undo in a Code Editor”. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. May 2015, pp. 223–233 (cit. on p. 21).

[185] J. Zhang and D. A. Norman. “Representations in distributed cognitive tasks”. *Cognitive science* 18.1 (1994), pp. 87–122 (cit. on p. 6).