Thesis Proposal

Developing conceptual understanding through interactive diagramming

Wode "Nimo" Ni

April 7, 2022

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.

Abstract

"Mental pictures" and "visual intuition" capture how people make sense of abstract concepts and see solutions to hard problems in a visual way. Learning research suggests that visual representations of knowledge are powerful tools for thought. Visual representations like diagrams enable more robust learning and flexible problem solving.

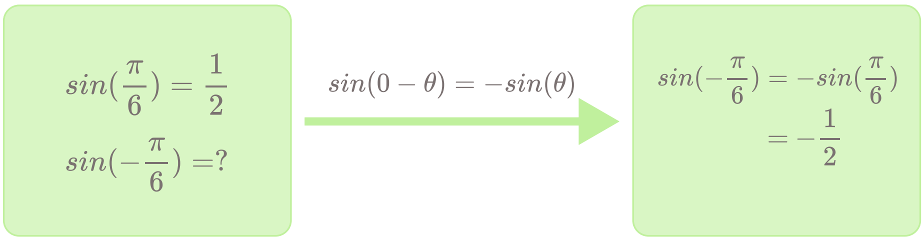
Existing diagramming tools often require hours of low-level tweaking of geometric primitives and do not capture the core task of diagramming: representing ideas visually. PENROSE is a diagramming platform that explicitly encodes visual representations in domain-specific languages. In this thesis proposal, I argue that this explicit encoding can be leveraged to (1) reduce the programming effort of producing diagrammatic problems at scale and (2) simplify the workflow of authoring interactive diagrams. The resulting diagrams also carry rich semantics, and I'll discuss how to use them to (3) provide useful, automated feedback to students.

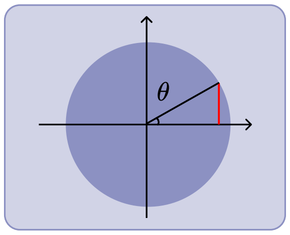
Prelude

Trigonometric identities are often presented as a big list of rules.

$\sin(0-\theta) = -\sin\theta$	$\sin(\frac{\pi}{2}-\theta) = +\cos\theta$	$\sin(\pi-\theta) = +\sin\theta$	$\sin(\frac{3\pi}{2}-\theta) = -\cos\theta$
$\cos(0-\theta) = +\cos\theta$	$\cos(\frac{\pi}{2}-\theta) = +\sin\theta$	$\cos(\pi-\theta) = -\cos\theta$	$\cos(\frac{3\pi}{2}-\theta) = -\sin\theta$
$\tan(0-\theta) = -\tan\theta$	$\tan(\frac{\pi}{2}-\theta) = +\cot\theta$	$\tan(\pi-\theta) = -\tan\theta$	$\tan(\frac{3\pi}{2}-\theta) = +\cot\theta$

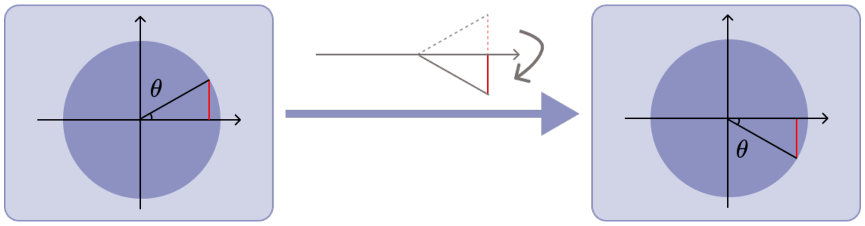
Students are often asked to solve problems by applying a subset of those rules, e.g., $sin(0-\theta) = -sin(\theta)$.


$sin(\frac{\pi}{6}) = \frac{1}{2}$
$sin(-\frac{\pi}{6}) = ?$
$sin(0-\theta) = -sin(\theta)$
$sin(-\frac{\pi}{6}) = -sin(\frac{\pi}{6})$
$= -\frac{1}{2}$



A useful visual representation of this concept is the unit circle: on a Cartesian plane with a circle of radius 1 centered at the origin, concrete values of trig functions are represented visually and rules are implicitly encoded as geometric transformations. For instance, the value of $sin(\theta)$ is the y-coordinate of a point on the circle, where the ray from the origin to the point forms angle θ with the x-axis.

To derive the identity rule visually, one only needs to note that $-\theta$ is a reflection about the x-axis, and observe that the y-coordinate is now a negative number. Instead of having to memorize a big set of rules, one can reduce this problem to a simple operation on the visual representation of a unit circle.



By translating the symbols to a visual representation, a unit circle, a student completely bypasses the tedious memorization of trig identities. While this is a much more retainable and robust representation for students, are we teaching representations like this to students? What does it take for students to internalize it?

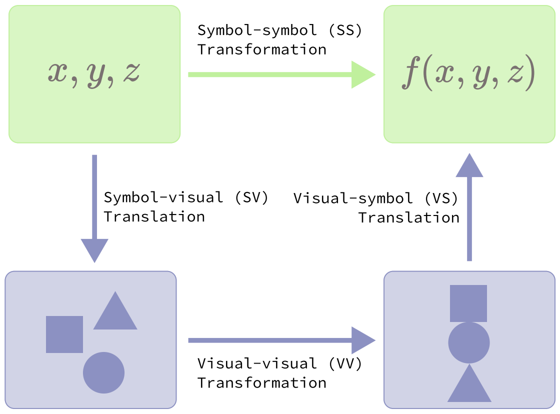
Chapter 1

Introduction

"Mental pictures" and "visual intuition" capture how people make sense of abstract concepts and see solutions to hard problems in a visual way. Hadamard described numerous examples of mathematicians doing exactly this in *The Mathematician's Mind* [23], later summarized by Alan Kay [28]:

> 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.

Learning research suggests that visual representations of knowledge are powerful tools for thought. Visual representations like diagrams enable more robust learning [39] and abstract and flexible problem solving [31, 32, 35]. Importantly, when people work with visuals, they build better conceptual understanding and more flexible mental models that go beyond memorized procedures [47].


Symbol-symbol (SS) Transformation
x, y, z
$f(x, y, z)$
Symbol-visual (SV) Translation
Visual-symbol (VS) Translation
Visual-visual (VV) Transformation

Let me use a diagram to capture this: the **grounding rectangle** represents two pathways to learning and problem solving: One can perform symbol-to-symbol transformations (SS, or "symbol pushing") or through an alternative diagrammatic pathway: a symbol-to-diagram translation (SV), a diagram-to-diagram transformation (VV), and finally a diagram-to-symbol translation (VS).

Research on expertise development suggests a need for substantial exposure involving repetition in varied contexts or deliberate practice [18] to acquire the perceptual chunks [19, 29] that support accurate interpretation and use of visual representations [31]. Through enough practice, learning the two paths in the grounding rectangle can produce better, more robust memory [13], learning [5, 39, 47], and future reasoning, both in providing flexibility and in supporting error recovery [33].

In reality, there seems to be an over-abundance of symbolic practice, continuing Kay's train of thought:

> The sad part of the diagram 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.

Diagrammatic practice is rare due to the significant cost of authoring diagrammatic problems. 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. In other words, these tools lack *representational salience*. As a result, the diagrams created by existing tools don't have semantics, as they are merely a collection of pixels and geometric blobs.

In prior work, colleagues and I built PENROSE, a diagramming platform that explicitly encodes visual representations in domain-specific languages (DSLs) [58]. In this thesis proposal, I argue that this explicit encoding can be leveraged to (1) reduce the programming effort of producing diagrammatic problems at scale and (2) simplify the workflow of authoring interactive diagrams. The resulting diagrams also carry rich semantics, and I propose to use them to (3) provide useful, automated feedback to students. My thesis statement summarizes the above:

Encoding visual representations in diagramming tools simplifies programming of interactive visual activities that provide students with automated feedback at scale.

The expected contributions of this work are:

1. *Need-finding studies* on challenges authors face.
2. *A platform of tools* based on the visual encoding of PENROSE for mass-production of diagrams (Chapter 3) and rapid authoring of interactive diagrams (Chapter 4).
3. *A theoretical framework* of the grounding rectangle, which guides the design of tools presented in this proposal.

Note. This proposal contains a mix of completed, in-progress, and proposed projects. In the rest of this document, proposed work will be marked in orange on the left margin.

Chapter 2

Understanding the diagramming process and encoding visual representations

Before diving into the educational context, it's important to understand why creating diagrams is hard in the first place. This chapter discusses an interview study on how domain experts including educators use diagramming tools [37], and briefly shows how this study informs the design of PENROSE, the technical basis for tools presented in this proposal.

2.1 How domain experts create diagrams and implications for tool design

Existing diagramming tools 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 [20]—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. This relatively limited support for diagramming in tools is in part because the process of diagramming is poorly understood. For instance, 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.

I conducted interviews with 18 domain experts from a wide variety of disciplines such as math, computer science, architecture, and education. The interviews reveal that diagrammers have diverse interactions with visual representations in both physical sketches and digital tools, including finding, creating, storing, and reusing representations.

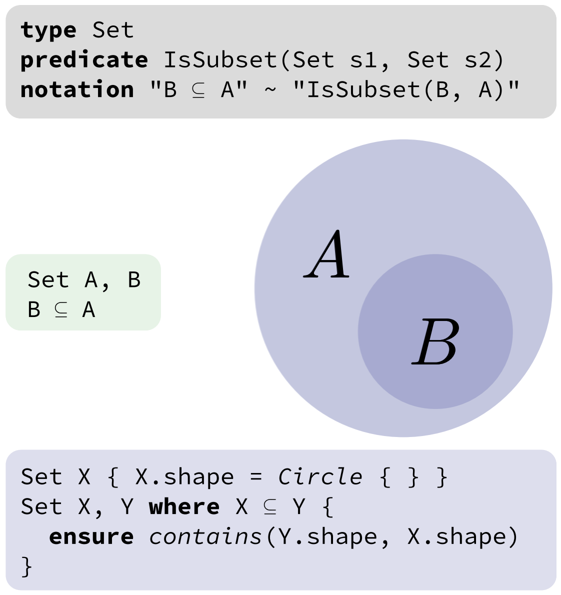
One implication of our results is the opportunity to design tools informed by the processes of diagramming, and practices that domain experts already use, making digital diagramming more intuitive and efficient. Here are four key opportunities for natural [43] diagramming tools that allow diagrammers 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.

2.2 The design of PENROSE

Informed by the results from the interview study, colleagues and I have developed PENROSE, a language-based diagramming platform [58]. The core PENROSE system addresses **representation salience** and **vocabulary correspondence**: it has first-class support for creating and reusing visual representations and translates familiar math-like notation into one or more possible 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 in math notation. STYLE explicitly specifies mappings from mathematical objects to visual icons.



```
type Set
predicate IsSubset(Set s1, Set s2)
notation "B ⊆ A" ~ "IsSubset(B, A)"
```

```
Set A, B
B ⊆ A
```

```
Set X { X.shape = Circle { } }
Set X, Y where X ⊆ Y {
  ensure contains(Y.shape, X.shape)
}
```

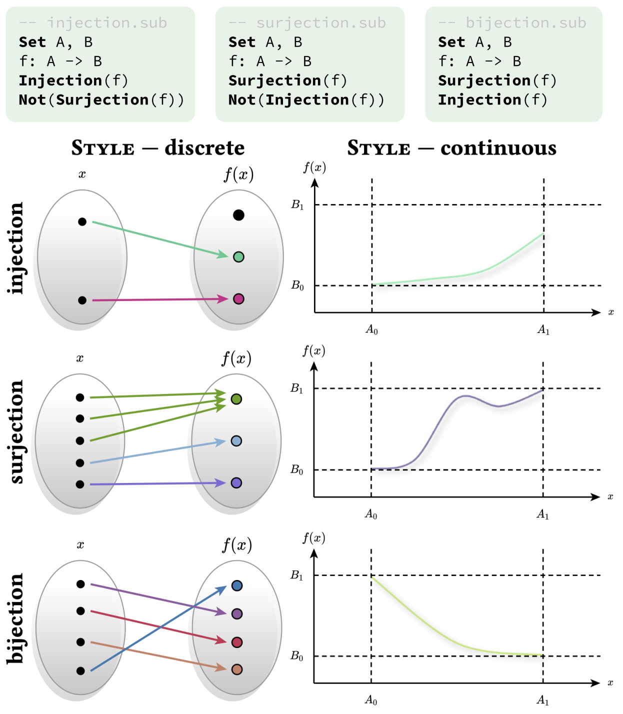
Instead of a limited focus on one specific domain (as in GraphViz [17] for graph theory or GroupExplorer [1] for group theory), PENROSE is extensible to user-defined domains of diagramming. Both SUBSTANCE and STYLE are parametrized by a DOMAIN schema that defines all possible objects (e.g., `Set`) and relations (e.g., `IsSubset`) in a particular domain, which can be used by associated SUBSTANCE and STYLE programs. In addition to user-extensibility, a formally encoded domain also enables automatic generation of PENROSE diagrams.

PENROSE compiles a **trio** of DOMAIN, SUBSTANCE, and STYLE into a constrained optimization problem defined by a set of graphical constraints (e.g., arrows that represent vectors should start from the origin). The optimization problem is in standard form, i.e., minimization of an objective function subject to equality and inequality constraints [9]. Such problems may be solved with many standard methods. PENROSE currently uses an exterior point method [27] 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.

[1]https://github.com/nathancarter/group-explorer

The design of PENROSE is driven by the design goals of reuse and scalability, and therefore is suitable for large-scale generation of visual content. The system is scalable and reusable in several dimensions:

- The optimization problem produced by PENROSE often has multiple solutions, and each point in the solution space corresponds to an alternative diagram. No program changes are required to generate these alternatives.
- For a visual representation encoded by a STYLE program, a wide range of notations (i.e., SUBSTANCE programs) can be visualized without any changes to STYLE. In the figure on the left, a single discrete STYLE program is used to visualize three SUBSTANCE programs that describe injective, surjective, and bijective functions.
- Conversely, multiple STYLE programs can be applied to the same SUBSTANCE program, generating alternative visual representations of the same underlying entities. The SUBSTANCE programs in the figure are also visualized by an alternative continuous STYLE.

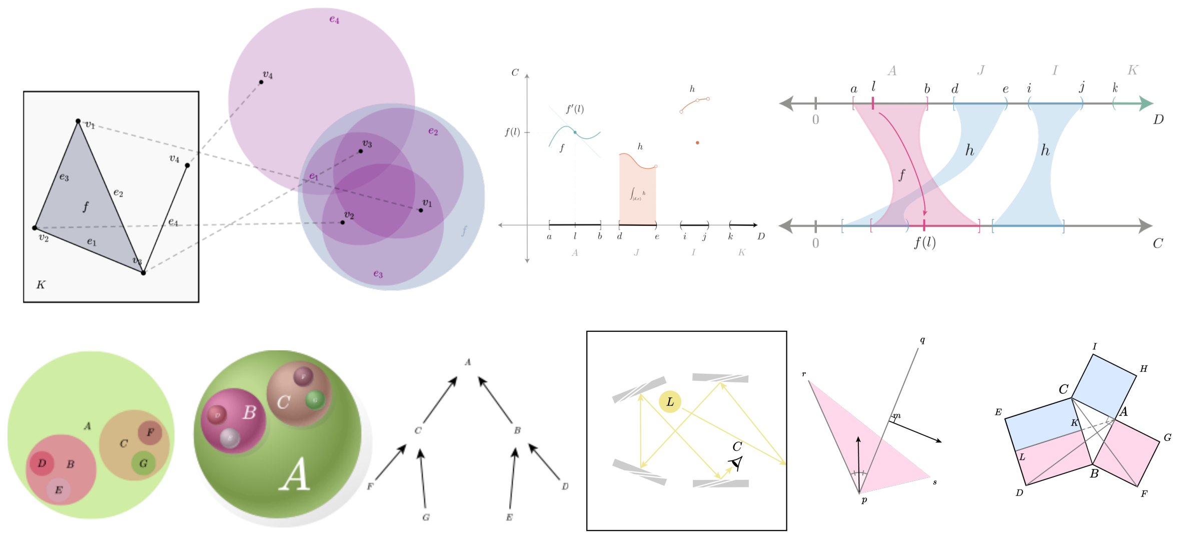


```
-- injection.sub
Set A, B
f: A -> B
Injection(f)
Not(Surjection(f))
```

```
-- 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)
```

With the extensible design, PENROSE can automatically generate diagrams from many different domains using familiar syntax. PENROSE-generated geometry, real analysis, ray-tracing, set theory, and algebra are shown below.



Chapter 3

EDGEWORTH: diagrammatic problem authoring at scale

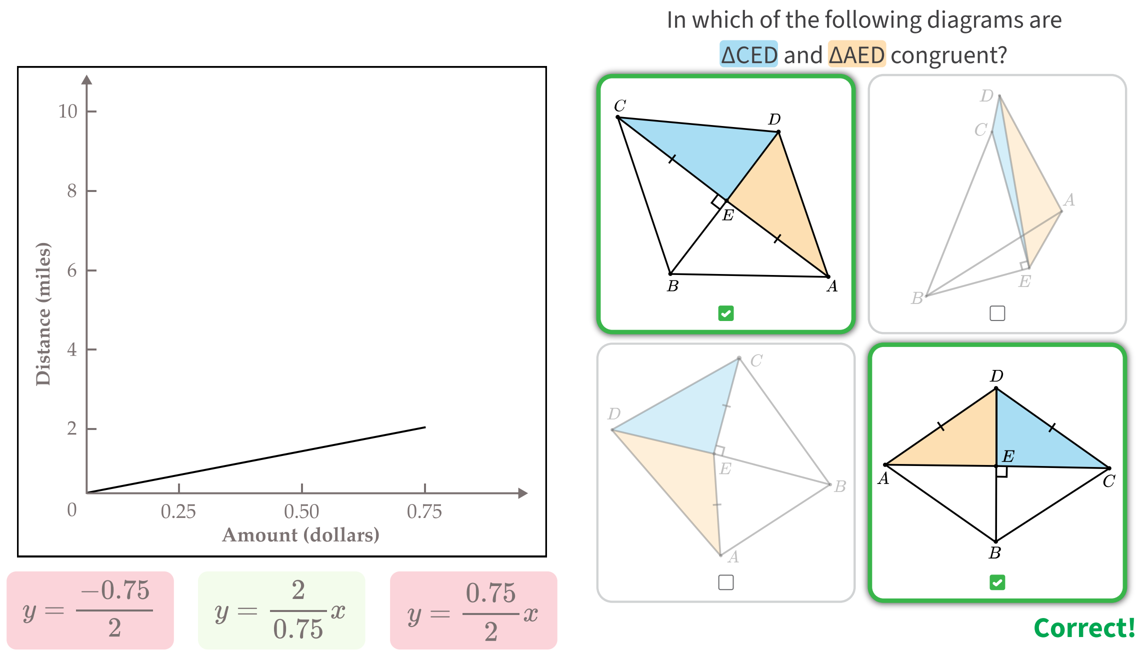


Figure 3.1: left: a translation problem that helps students discern the structure of linear equations (adapted from [30]). **right**: an EDGEWORTH generated problem that trains student to recognize diagram configurations [31] for triangle congruence.

Effective use of visual representations requires a certain level of *representational fluency* that's achievable through deliberate practice and repetition [16, 44]. Recognizing how words, symbols, 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 [30].

To train students' representational fluency, educators often create problem sets that involve numerous contrasting cases of a particular visual representation. For instance, Figure 3.1 shows two examples of *translation problems*, where the problem asks students to determine diagrammatic *instances* and *noninstances* of a textual description and vice versa. Importantly, these instances and noninstances have varying degrees of differences from the given diagram or text, and carefully picking examples on this spectrum has a big impact on learning [38].

Unfortunately, authoring diagrammatic problems remains a tedious, manual process. Our formative interviews show that educators have difficulties creating simple shapes using existing tools. Limited by the tools, educators don't have support to create enough problems and resort to copy-pasting and manual editing. They expressed desires for tools that understand the diagrams and allow them to make semantically-consistent tweaks to diagrams.

In this chapter, I discuss EDGEWORTH, a tool that supports an automated and scalable diagrammatic problem authoring workflow.

3.1 Formative interviews

To inform the design of EDGEWORTH, I conducted semi-structured interviews of teachers, textbook authors, and learning engineers that have experience creating instructional material, authoring problems, or developing e-learning tools that include mathematical diagrams. The interview questions ask about what roles visuals play in the educational material, how they are connected to surrounding content, and how students interact with visual content. This formative study aims to understand the gaps between affordances of current tools and ideal workflows of content creators, and the need for automation and mass production of visual content. Participants are asked to share content they have created and discuss the creation process in detail.

3.1.1 Need for diagrammatic contents

Traditional educational materials, especially in higher education, tend to emphasize "*procedures, memorization, and symbolic manipulation*" (P6). As a result, students often become "*symbolically good*" and do not develop "*good conceptual understanding*" (P3). Visual content like diagrams and graphs provide alternative representations that help students "*develop intuition and become better problem solvers*" (P3). As a result, most of our participants include visuals in their instructional materials such as problem sets and lecture notes. Some also ask students to draw, annotate, and explain diagrams (P1, 2, 6).

3.1.2 Visual content in instruction

P2 would deliberately encourage students to learn "*multiple representations*" and made diagrams central to their math and programming curricula. For instance, a typical problem set in a pre-calculus class will have either problems with diagrams, or students have to "*produce a drawing if no pictures are given.*" To improve students' "*visual fluency,*" P1 incorporates visualization tools such as GeoGebra in their middle school math classroom. Students draw in in-class activities and homework assignments. P1 found that students "*benefit from using a diagram,*" but also refer to the "*curse of knowledge*" when teaching students how to use diagrams: students need deliberate practice to use visuals successfully, but existing instruction tends to under-train them. When students practice with visuals, instructors like P6 also gain richer feedback of students' level of understanding: "*when the students drew diagrams to figure out the answer. I actually learned more from this than 10 similar problems without the picture.*"

3.1.3 Need for better tools

Many aspects of content authoring can be repetitive and iterative. Authors typically create "*multiple similar examples*" (P1) and "*variants of classroom examples in problem sets*" (P6). In addition, they iterate on their content often, even "*re-do my courses every semester*" (P2) in some cases. Participants face a trade-off when authoring visual content: visuals are much harder and time-consuming to create, especially when variations and frequent changes are required in instructional materials, which is often the case. When authoring practice problems, P1 struggles to "*create simple shapes by myself*" and always ends up "*copy-pasting and searching online*" repeatedly. As a result, P1 tends to "*reuse [a few of my existing diagrams. Thinking about having to do it all over again for another class is just too much.*" To make visual content authoring time efficient, P2 and P5 developed a custom pipeline for authoring problem sets and quizzes using programming tools. Similar to problems described in [37], 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)

3.2 Mutation-based diagram generation

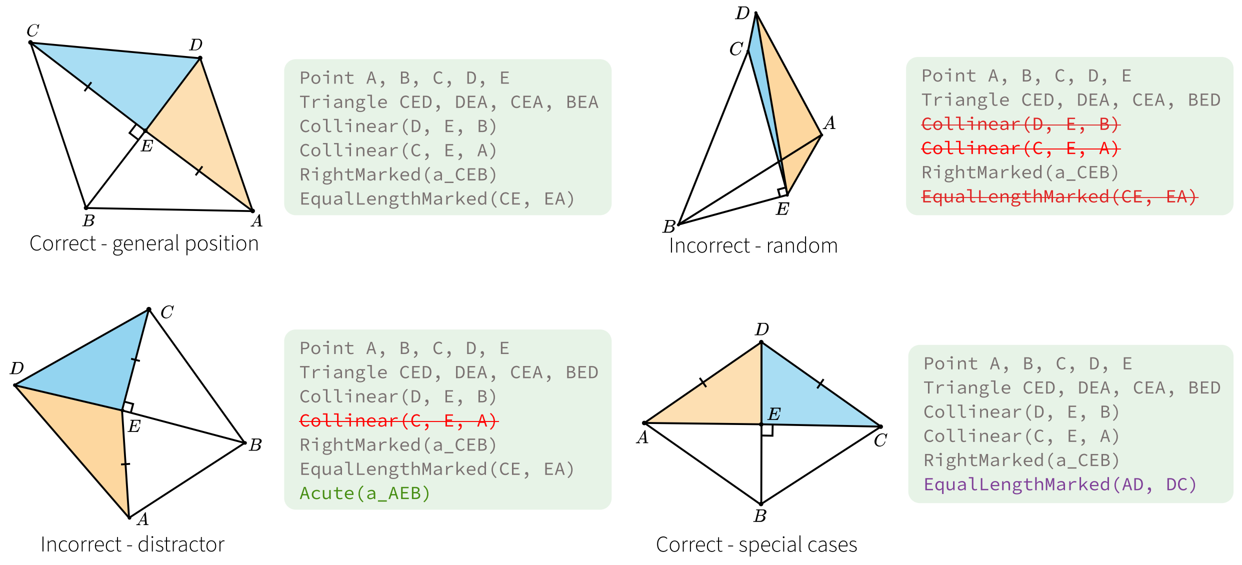
Educators simply don't have enough time to produce good-looking diagrams, not to mention the amount and variety of diagrams required for training students to be fluent in visual representations. Therefore, a key pain point to automate is diagram generation. Importantly, the generated diagrams have to be meaningful and need to include contrasting cases of the same subject matter.

In EDGEWORTH, I propose to **generate a large pool of diagrams by mutating the SUBSTANCE program in a PENROSE trio**. Compared with general-purpose programming languages, PENROSE DSLs have unique advantages: DOMAIN is a meta-language that precisely defines the available program constructs in SUBSTANCE, which helps define the mutation search space. Moreover, it's easier to make sense of mutation on SUBSTANCE, because it corresponds to the domain-specific vocabulary of diagram authors.

At a high level, the EDGEWORTH mutator takes in a PENROSE trio and a small configuration file, and simply generates an arbitrary number of SUBSTANCE programs. Constrained by the configuration, EDGEWORTH mutates the SUBSTANCE program (*prompt program*) by applying a series of program mutations to get a *mutant* SUBSTANCE program. For each mutant, the system then uses the original STYLE and DOMAIN programs to render a diagram.

Since SUBSTANCE is a small declarative language, EDGEWORTH uses a set of pre-defined, high-level mutation operations listed below.

- **Add** Appends a statement to the SUBSTANCE program.
- **Delete** Removes a random statement from the SUBSTANCE program.
- **Cascading Delete** Removes a random statement and all other references to that statement.



```
Point A, B, C, D, E
Triangle CED, DEA, CEA, BEA
Collinear(D, E, B)
Collinear(C, E, A)
RightMarked(a_CEB)
EqualLengthMarked(CE, EA)
```

```
Point A, B, C, D, E
Triangle CED, DEA, CEA, BED
~~Collinear(D, E, B)~~
~~Collinear(C, E, A)~~
RightMarked(a_CEB)
~~EqualLengthMarked(CE, EA)~~
```

```
Point A, B, C, D, E
Triangle CED, DEA, CEA, BED
Collinear(D, E, B)
~~Collinear(C, E, A)~~
RightMarked(a_CEB)
EqualLengthMarked(CE, EA)
Acute(a_AEB)
```

```
Point A, B, C, D, E
Triangle CED, DEA, CEA, BED
Collinear(D, E, B)
Collinear(C, E, A)
RightMarked(a_CEB)
EqualLengthMarked(AD, DC)
```

Figure 3.2: Four example classes of mutants generated by EDGEWORTH. **Top-left**: the original prompt program, representing a general correct instance. **Bottom-Left**: an incorrect noninstance that only slightly differs from the original prompt semantically. **Top-right**: an incorrect noninstance that differs significantly from the prompt. **Bottom-right**: a correct instance that's a corner-case of the prompt.

- **Swap Arguments** Reorders the arguments passed into a statement. e.g., if `A` and `B` are `Triangles`:

  `Similar(A, B)` → `Similar(B, A)`
- **Swap-In Arguments** Replaces the arguments passed into a statement with other arguments defined in scope. e.g., if `A, B, C, D` are `Points`:

  `s := MkSegment(A, B)` → `s := MkSegment(C, D)`
- **Replace Statement Name** Replaces a statement with a different statement that takes the same type of arguments and has the same return type. e.g., given that `T` is a `Triangle`:

  `Right(T)` → `Obtuse(T)`
- **Type Change** Replaces a statement with a new one that takes the same number and type of arguments, but does not necessarily return a value of the same type. e.g., if `E` is an `Angle`:

  `Segment s := Bisector(E)` → `Right(E)`

These mutations are all done safely at the level of the abstract syntax tree (AST) and EDGEWORTH maintains a local context and symbol table, so operations will not introduce errors. The configuration file contains a set of rules to filter down the search space by statement types and specify the kinds of mutations allowed.

Authoring contrasting cases require different classes of diagrams: those that correctly correspond to the textual/symbolic description and others that don't. Importantly, nearest neighbors of the prompt program seem to have great education values, i.e., "near misses" and "near hits." Knowing the correctness of a mutant also helps with automated grading of problems.

Although EDGEWORTH generates syntactically valid mutants, the system doesn't know whether a mutant is semantically consistent with the prompt a priori. Currently, the system uses the graphical constraints to determine semantic consistency. Specifically, it uses an energy-based heuristic by performing **cross-instance energy evaluation (CIEE)** of each mutant. Suppose EDGEWORTH mutates the prompt program P to mutant P' and generates a diagram D' from P'. the system can compute the cross-instance energy of D' by 1) checking if all of the constraints generated from P are met by D' and 2) run the objective function defined by P on D' and check if the D' is at a local minimum. In other words, the PENROSE optimizer determines if the diagram D' generated from mutated program P' is a good fit for the prompt program.

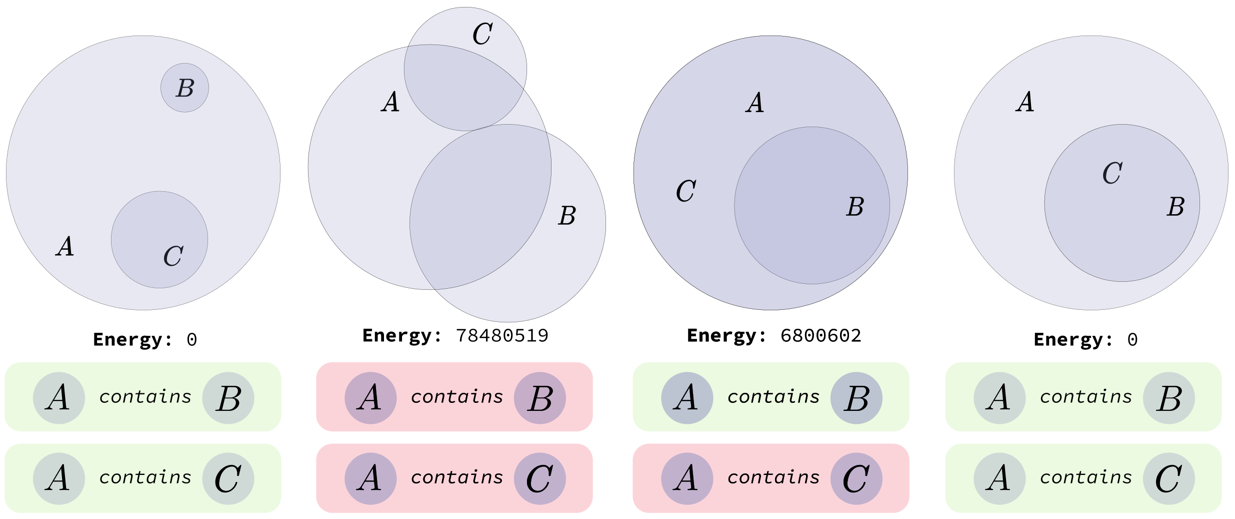


Figure 3.3: An example of CIEE, where the leftmost is the prompt program P and its corresponding diagram D. The overall energy value is 0. The right three are instances of P', on which the constraints from the prompt are evaluated. The middle two are *semantically inconsistent* with the prompt and have high energy values. The rightmost is *semantically consistent* with the prompt and therefore has an energy value of 0.

CIEE robustness. CIEE showed some promise on a limited set of examples, but further evaluation is needed to see the robustness of this heuristic. For instance, how much does this method depend on the qualities of the STYLE program that defined the constraints? If the mutant significantly differs from the prompt (e.g., missing most identifiers from the prompt), is this heuristic still useful?

DOMAIN language extensions. None of the predefined mutations listed above carry any mathematical semantics because DOMAIN doesn't contain enough information. For instance, many mathematical predicates have reflexive, symmetric, transitive, and substitution properties, but DOMAIN only encodes basic type definitions. For a more precise notion of correctness, I plan to extend DOMAIN to model such properties, and use them in the EDGEWORTH mutator, possibly together with CIEE, to generate higher quality mutants.

3.3 Preliminary evaluation of the EDGEWORTH mutator

In preliminary work, we evaluated the system by recreating problems in a middle-school geometry textbook [10]. We examined all 53 diagrammatic problems in the chapter review sections and picked a representative subset of 24 problems and implemented them in EDGEWORTH. For each textbook problem, the prompt is reframed so that the diagram accompanying the original problem can be considered a correct answer. Then, we consider possible answers to the posed question, including “distractors”—answers that are designed to tempt students with limited conceptual understanding—as well as correct “special cases”. We then describe the original diagram as a prompt SUBSTANCE program and pass it to EDGEWORTH, which generates numerous answers to the original problem.

We examined sets of 20 diagrams generated by EDGEWORTH based on various prompt programs. Each program was mutated 1-3 times and the correctness of each diagram was determined manually. We found that while EDGEWORTH easily generates a variety of correct and incorrect diagrams, careful selection of configuration parameters was often required to get more “interesting” diagrams (correct special cases, distractors).

Usability of the EDGEWORTH configuration. As noted above, results from the EDGEWORTH mutator are sensitive to the configuration. For instance, a statement in the SUBSTANCE program might be particularly more suitable for mutations than others (perhaps because it contributes to the correctness of the problem.) Under- or over-specifying mutations in the configuration might lead to a pool of “noisy” diagrams. I propose to conduct a light-weight case study with a handful of problems to identify the key to successful configurations. With that insight, we can either improve the configuration format or explore other modes of interaction.

3.4 Mutation paths as problem templates

After generating a pool of diagrams, the author then picks a subset of them for a single *problem instance*. For each generated diagram, EDGEWORTH keeps a record of the series of mutations performed on the prompt program (*mutation path*) to the mutant. For a multiple choice problem with four options, there will be four mutation paths from the prompt. Together these paths form a *problem template*.

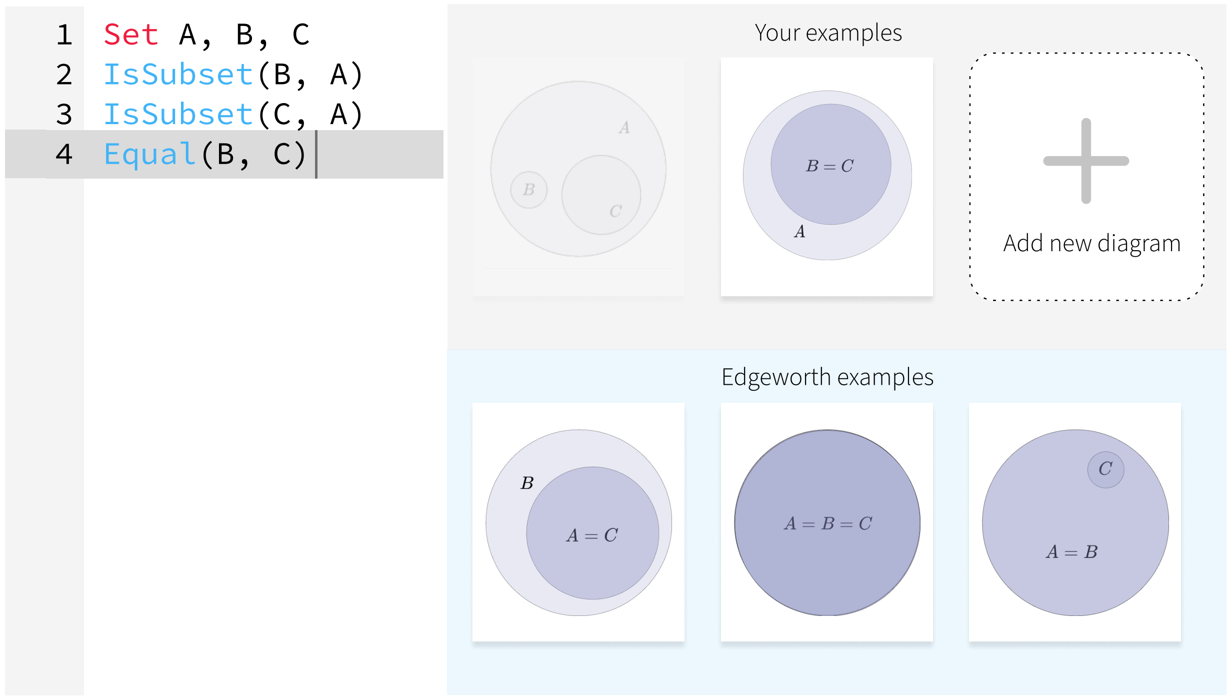
A problem template is specific to a prompt, so it’s unlikely to be reusable for generating other problem instances. However, many problems might share the same instructional goal such as teaching students the conditions for the Hypotenuse-Leg (HL) theorem. While the choice of names and diagram design may differ, the core structure is the same: two instances of congruent triangles satisfying HL and two noninstances of non-congruent triangles. Encoding this information can further scale up problem authoring.

I propose to **investigate common structures among problem templates and encode them as problem template specifications that are generalizable to multiple prompts**.

3.5 By-example workflow for authoring at scale

With the EDGEWORTH mutator, the primary mode of interaction is picking examples from the mutant pool and editing the configuration to narrow down the search space. In some cases (see Section 3.3), the author might want to write a few examples from scratch, or prefer to manually make slight tweaks to examples in the mutant pool. I propose to **create a programming-by-**

example workflow, where the author manually creates a few diagrams and EDGEWORTH generates a bigger pool of diagrams with similar properties.



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

Figure 3.4: User interface mock-up of a by-example workflow in EDGEWORTH.

For example, if EDGEWORTH fails to generate a pool of useful diagrams, the author can manually create a few examples by directly editing the prompt program. In Figure 3.4, the author adds the `Equal(B, C)` predicate. Their intent is to include the edge case of proper subsets in this problem, where some of the subset relations are actually equality. EDGEWORTH generates a set of similar examples that add `Equal` predicates with existing identifiers in different ways.

The addition of `Equal(B, C)` is effectively a user-generated mutation, and EDGEWORTH needs to understand this mutation to generate similar instances. Currently, the EDGEWORTH synthesizer matches a series of author edits to predefined mutations. Once the synthesizer finds a path, it can then inform the mutator to generate examples with similar properties (i.e., including the edge case of equal sets).

Generalized mutation paths Similar to templates, the by-example workflow also requires a generalizable encoding of mutations. I plan to experiment with a few possible formats such as (1) another mutator configuration and (2) mutation paths with "holes."

3.6 Evaluation

Usability study of EDGEWORTH. I propose to evaluate the usability of EDGEWORTH by recruiting authors to perform content authoring tasks with the EDGEWORTH prototype. For example, the participants may be asked to author a problem set of 10 diagrammatic problems. The goal of this study will be to identify missing features, usability problems, and opportunities for simplification. The study may include several rounds with increasingly high-fidelity prototypes. After each round, I will refine the design and implement the next prototype. Here are some possible research questions:

- What are the key design considerations for diagrammatic problem authoring? How do they fit with the features of EDGEWORTH?
- How do authors prefer to work with EDGEWORTH? When do they opt to write a configuration file and generate many diagrams? When do they use the by-example workflow? Do they mix the two workflows?
- How does the experience compare to their existing tools? How can EDGEWORTH incorporate useful parts of them?

3.7 Related work

3.7.1 Using contrasting cases to improve representational fluency

Representational fluency refers to the ability to quickly understand a visual representation and to use it to solve domain-specific tasks [47]. To become representationally fluent, an important first step is to identify meaningful aspects of a particular representation. Kellman et al. [30] 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 [24].

In addition to mapping between representations, Marton [38] 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 [12].

3.7.2 Multiplicity of examples and problem generation

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 [46]. Equally important is research indicating the importance of active learning [11, 15] and repeated practice [18, 54] that occurs within varied contexts [45, 49] and involves direct explanatory feedback [30].

As a result, a number of authoring tools exist for large-scale production of examples and practice problems. Intelligent Tutoring Systems (ITS) are automated curricula that often include worked examples and practice problems that are customized to individual students. The Cognitive Tutor Authoring Tools (CTAT) is an ITS authoring platform [3]. CTAT has a "Mass Production" feature that lets the user create a problem template and insert problem-specific values via a spreadsheet [4]. The ASSISTment builder allows authors to "variablize" numerical values in problem templates for automatic generation [48]. In the computer-aided education literature, a number of systems were also proposed for problem generation [1, 21, 22] In both lines of work, most systems don't tackle the problem of diagram generation—they mostly generate symbolic problems and examples. Notably, Gulwani et al. [22] generated ruler-and-compass geometry constructions automatically, which is a significantly narrower domain than what EDGEWORTH targets.

Chapter 4

From encoding to semantics-preserving interactivity

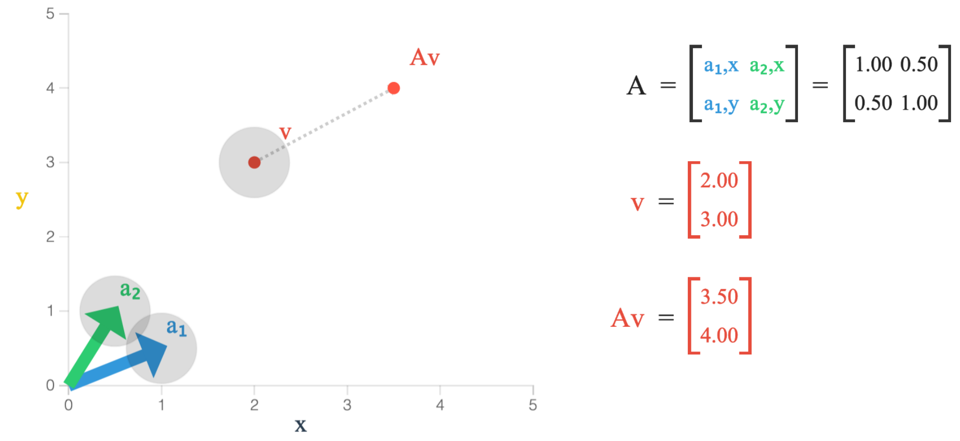
Diagrams live in the context of surrounding text, overlaid annotations, and human gestures. The web opens up opportunities for even richer in-context interaction. In education, though students spend more time on digital platforms, they often see diagrams that are presented exactly as before: blurry, static, and ornamental. In addition to their values as an external, static representation of knowledge, diagrams are also beneficial when people learn *with*, instead of *from* them [55]. Prior work shows interacting with visual representations has unique benefits to learning [2, 6]. 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.** Meaningful interaction with diagrams helps students move from passive recognition to active synthesis of visual representations [34].

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 [36, 40]. 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. In this section, I investigate how to build interactive diagram activities that are automatically derived from PENROSE diagrams and easily created without extensive programming efforts. 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**.

4.1 Motivating example

Consider the first diagram in a popular explorable explanation piece ”Eigenvectors and Eigenvalues Explained Visually [1].” The diagram is one of a series of interactive diagrams showing the visual properties of eigenvalues and eigenvectors: it shows a visual interpretation of matrices as linear transformations: matrix A with columns a_1 and a_2 transforms v to Av. In the diagram, a_1, a_2 and v are all draggable.



Seeing what varies and what doesn't is an important form of *feedback* that fosters conceptual understanding. The reader gains an initial understanding of how columns of A impact Av's value through interacting with the diagram: dragging any of a_1, a_2 and v affects the position of Av.

In the original code repository [2], the authors wrote about a hundred lines of JavaScript with D3.js to make the first diagram. Although D3.js and Angular already provide significant support, it's still a lot of work to handle mouse down/up/hover events, and to keep track of intermediate values during dragging.

To reproduce this diagram in PENROSE, one can write a simple SUBSTANCE program in the linear algebra domain [58, Section 5.4].

```
Vector a_1, a_2, v
Matrix A := columns(a_1, a_2)
Vector Av = multiply(A, v)
AutoLabel All
```

With the core system, the trio generates a static SVG diagram. Under the hood, every `Vector` is represented visually as an arrow starting at the origin (a_1, a_2), or a single point (v). They are all degrees of freedom (DOF) in the optimization problem. In other words, both the x and y-components of the arrow-end of a_1, a_2, and the point representing v are free to move on the canvas. Following the original design of the explorable, the system surfaces the DOFs as draggable points. Whenever the user drags the end of one of the arrows, the optimizer takes the new position as a part of the final solution, and solves the rest of the optimization problem. Effectively, by using this simple and generalizable strategy, which I will discuss in the following

[1] `https://setosa.io/ev/eigenvectors-and-eigenvalues/`
[2] `https://github.com/vicapow/explained-visually/tree/master/client/explanations/eigenvectors-and-eigenvalues`

sections, the system can reproduce the interactive design using the PENROSE trio for a static diagram *without a single line of code added.*

4.2 Semantics-preserving interactivity as feedback

Section 4.1 is an example of a set of interactive behaviors that can be automatically derived from a PENROSE trio without any additional programming. Specifically, the example leverages how PENROSE encodes visual semantics: PENROSE compiles a program trio to computational and optimization graphs with degrees-of-freedom (DOF) [58, Section 4.1.2-3]. Degrees-of-freedom determine a diagram instance in PENROSE. They are "free" variables within the computational graph and non-constant root nodes in the optimization graph. DOFs are the key to generate a family of diagrams: by manipulating DOFs, the optimizer solves for different diagrams that satisfy the constraint set defined by the trio. In other words, DOFs are a concise representation for interaction. In this section, I use *dragging* as a case study and show a few ways of manipulating the DOFs in a semantics-preserving manner.

As a reasonable default, the system can find positional properties in the DOFs and make them draggable. In Section 4.1, the relevant STYLE blocks define a simple computational graph for the SUBSTANCE program, where `a_1.data`, `a_2 .data`, and `v.data` are DOFs. Figure 4.1 shows the graph for `a_1`'s properties. To accomplish the interactivity in the example, the system can analyze the computational graph to find DOFs and their aliases, i.e., child nodes that are assigned values of the DOFs. For instance, `a_1.data` is a DOF and `a_1.icon.end` references `a_1.data`. In contrast, `Av.end` is not made draggable because it's not a DOF nor an alias in the computational graph: its value is computed by `matmul(a_1.data, a_2.data)`.

```
Vector v {
  v.data  = (?, ?)
  v.icon = Arrow {
    start : (0, 0)
    end   : v.data
  }
  v.text = Text { string : v.label }
  ensure near(v.text, v.icon)
}
Matrix A
where A := columns(a_1, a_2) {
  A.data = [a_1.data; a_2.data]
}
Vector Av = multiply(A, v) {
  override Av.data = matmul(A.data, v.data)
}
```

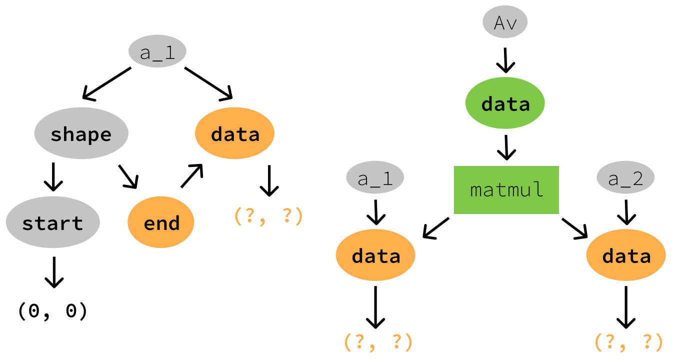


Figure 4.1: Left: relevant blocks in the linear algebra STYLE program for Section 4.1. **Right**: computational graphs for a_1 and Av, where the data field for the former is optimized and that for the latter is computed.

Once exposed as draggable properties, the user can now change the values of positional DOFs by dragging shapes around. However, since their interaction is situated in an optimization problem, it's important to discuss how an optimizer influences this interaction and manipulates the rest of the diagram in a semantics-preserving way. In Section 4.1, dragging `a_1.icon.end` and `a_2.icon.end` works as intended because they are independent from each other: they don't participate in the same constraints in the computational graph. However, this is not the right interaction for DOFs that participate in the same constraints, which is often the case. In this section, I give two example optimization strategies for supporting semantics-preserving drag.

4.2.1 Follow the cursor

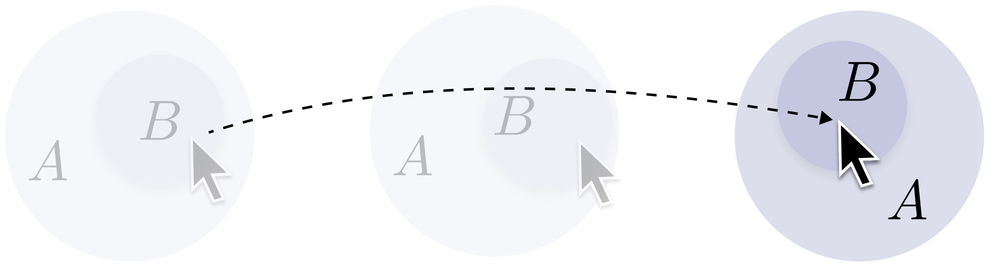


Figure 4.2: Dragging a subset, B, in a Venn diagram in an intuitive and semantics-preserving way, where B is always under the cursor and $B \subset A$ is always held true.

Consider the example in Figure 4.2, which shows a simple Venn diagram of sets A and B where $B \subset A$. The underlying rule of this visual representation is that a subset is always visually contained in the superset. An interactive diagram should clearly reveal this rule by keeping this containment relationship true at all times. For instance, if a student drags B to the right, the diagram should change such that A still contains B. Importantly, the interaction should be natural, and also make the feedback very clear: as the student is dragging B, B must stay under the cursor, and the rest of the diagram should incrementally move with B to maintain the containment relationship.

Unfortunately, when using the current PENROSE optimizer, dragging either A or B yields counterintuitive results: the optimizer changes arbitrary properties, including the manipulated ones. This is because it optimizes all DOFs simultaneously. In Figure 4.3, it moves both A and B to satisfy the containment constraint. This behavior adds noise to the feedback, and may confuse the student.

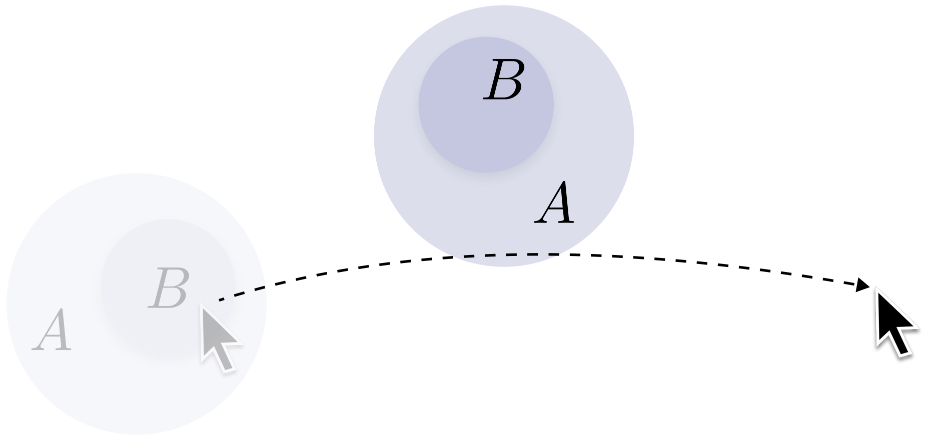


Figure 4.3: Dragging a subset, B, in a Venn diagram in semantics-preserving but counterintuitive way, where $B \subset A$ held true but the shapes appear in random locations.

To enable intuitive interactivity, the system can analyze the computational graph again to derive the right behavior. We can achieve this behavior by "locking" the DOFs, treating them as constants in the optimizer. Specifically, when a student manipulates DOFs or its aliases, the system locks these DOFs and optimizes the rest as usual. When the student interacts with an object (i.e., dragging to change `x` and `y` of a `Circle`), the system yields the control to the student completely and locks the manipulated properties during optimization. The visual effect is that all other parts of the diagram "follow" the student interaction.

4.2.2 Freeze the world

Locking the manipulated property is not the only way to maintain the visual semantics. Instead of limiting the optimizer, we could also limit the interaction so they see the effect of changing one or multiple shape properties under constraints. When the student interacts with a shape, the optimizer keeps all other properties locked and continuously uses the energy function to "guide" the student. The techniques involved are different from Section 4.2.1. In this case, the student is playing the role of the optimizer, i.e., changing DOFs, while the optimizer only sends feedback to make sure the interaction is semantic. The visual effect is a constrained interaction where the student can only make semantically-valid moves.

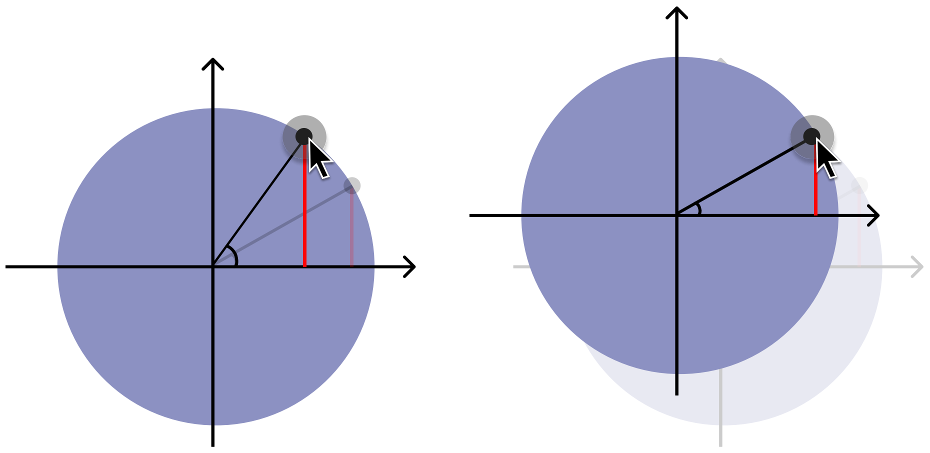


Figure 4.4: The behavior of dragging a point along the unit circle depends on the optimization strategy. **Left**: "Follow the cursor" shifts the entire diagram to follow the point and doesn't correspond to the mathematical semantics. **Right**: "Freeze the world" should be the correct optimization strategy, where the point only moves along the circle, and nothing else changes in the diagram.

For instance, Figure 4.4 shows a diagram of the unit circle. A natural interaction is to drag the point along the unit circle to see how the values of trig functions change. In this case, the red line shows the value of sin. If the optimizer naively follows the cursor, Figure 4.4 (right) would be the result, where the rest of the diagram is translated to stay in a valid layout. Instead, it's much more desirable to "freeze the world" and constrain the student input within the feasible region——along the unit circle (Figure 4.4 left).

Together, these two strategies cover a wide range of drag behaviors that are traditionally difficult and time-consuming to implement. Note that these two strategies are not necessarily mutually exclusive. In fact, the system may have a set of default rules for or let the author specify the strategy on a per-DOF basis. For instance, an instructor might apply "freeze the world" to show students the valid positions of a component in a diagram, while applying "follow the cursor" to the rest of the components to show alternative layouts of the diagram.

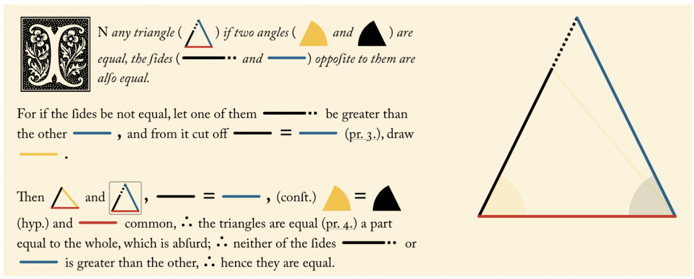
Encoding optimization strategies. If the author wants to control the optimization strategy, they will need an encoding to do so. Because STYLE already has language constructs for matching on shapes, a STYLE language extension may be suitable for specifying static strategies per shape, e.g., a shape should always follow the cursor when dragged. However, the current design of STYLE may not be suitable for deciding strategies dynamically if needed, e.g., a shape follows the cursor in a certain region of the diagram, and freezes the world on the boundary.

4.3 Highlighting and annotation as feedback

As demonstrated in Chapter 3, diagram understanding is a vital step towards representational fluency. A significant part of diagram understanding maps to learning the translational semantics of a diagram, i.e., which shape represents what math object. While EDGEWORTH helps students practice the mapping between a particular visual representation and symbols, I propose to **provide on-demand, inter-representational feedback by utilizing the translational semantics of a PENROSE trio**.

4.3.1 Inter-representational highlighting

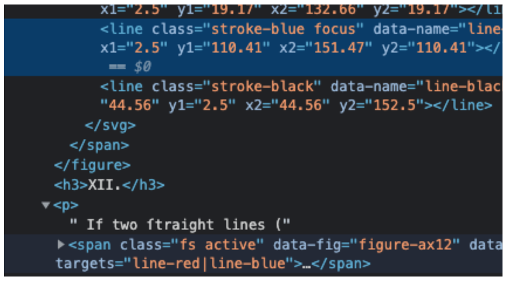
Students' exposure to visual representations is often limited by traditional media like textbooks and in-person lectures. The mapping between symbolic and visual representations is often scarcely presented via prose, gesture, and carefully designed worked examples. Web-based materials show a much more pervasive use of on-demand highlighting to build up inter-representational connections. However, there's also a non-trivial authoring burden to meticulously annotate the HTML document and the diagram with CSS classes:



I*N any triangle* ( ) *if two angles* (*and*) *are equal, the ſides* (*and*) *oppoſite to them are alſo equal.*

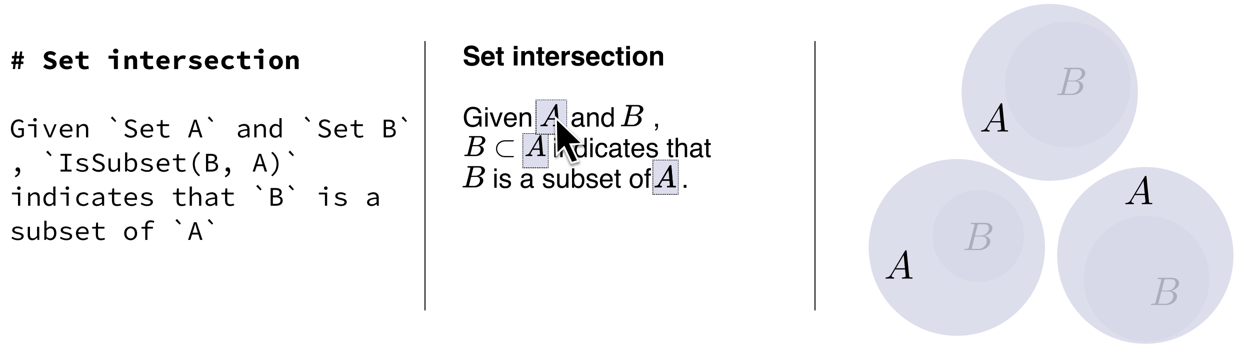
For if the ſides be not equal, let one of them be greater than the other , and from it cut off = (pr. 3.), draw .

Then and , = , (conſt.) = (hyp.) and common, ∴ the triangles are equal (pr. 4.) a part equal to the whole, which is abſurd; ∴ neither of the ſides or is greater than the other, ∴ hence they are equal.



```
          x1="2.5" y1="19.17" x2="132.66" y2="19.17"></li
          <line class="stroke-blue focus" data-name="line
          x1="2.5" y1="110.41" x2="151.47" y2="110.41"></
           == $0
          <line class="stroke-black" data-name="line-blac
          "44.56" y1="2.5" x2="44.56" y2="152.5"></line>
        </svg>
      </span>
    </figure>
    <h3>XII.</h3>
  ▾<p>
      " If two ftraight lines ("
    ▸<span class="fs active" data-fig="figure-ax12" data
    targets="line-red|line-blue">…</span>
```

If an online textbook or website uses diagrams generated by PENROSE, the author may leverage the translational semantics to automatically provide on-demand highlights. For instance, suppose an author writes an visual explanation in markdown with interleaving SUBSTANCE symbols in the prose. The system can automatically generate diagrams by extracting the SUBSTANCE symbols and provide highlights for all subsequent references to the same symbols. Since PENROSE can generate alternative diagrams in the same visual presentations, the highlighting can also provide contrasting cases of a particular entity across diagram instances.



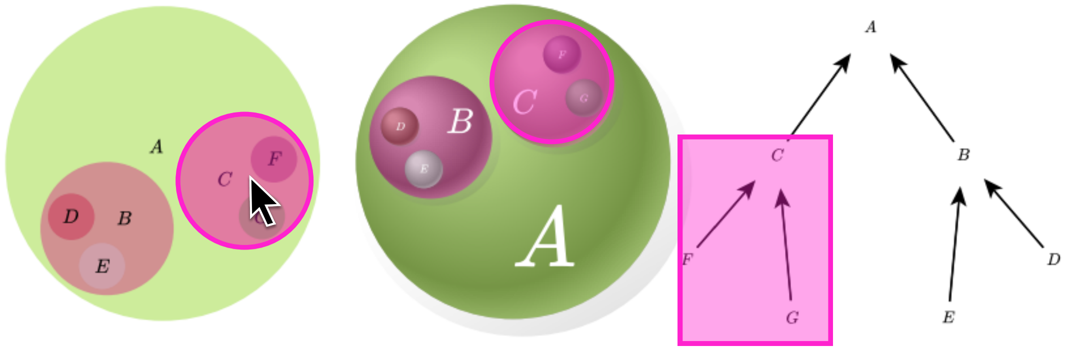
```
# Set intersection

Given `Set A` and `Set B`
, `IsSubset(B, A)`
indicates that `B` is a
subset of `A`
```

Set intersection

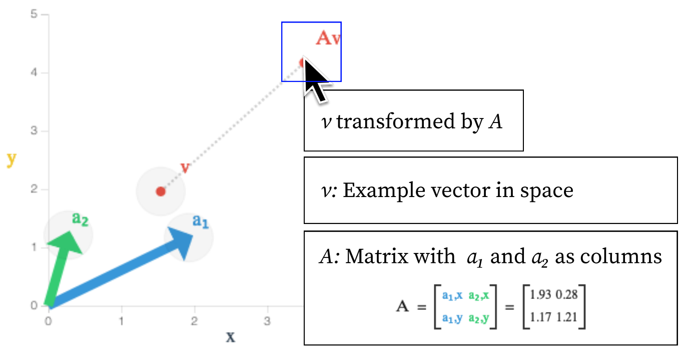
Given A and B, $B \subset A$ indicates that B is a subset of A.

Building connections among multiple visual representations also improve learning [47]. Because a PENROSE trio is representationally salient, one can swap among alternative STYLE programs to get diagrams that visualize the same symbols. Because the SUBSTANCE program stays the same, the same strategy also works for highlighting diagram parts across multiple visual representations.



4.3.2 Documentation and program slices as tooltips

```
-- First column of `A` [a_1.data]
Vector a_1
-- Second column of `A` [a_2.data]
Vector a_2
-- Example vector in space [v.data]
Vector v
-- Matrix with `a_1` and `a_2` as columns
   [A.data]
Matrix A := columns(a_1, a_2)
-- `v` transformed by `A`
Vector Av = multiply(A, v)
AutoLabel All
```

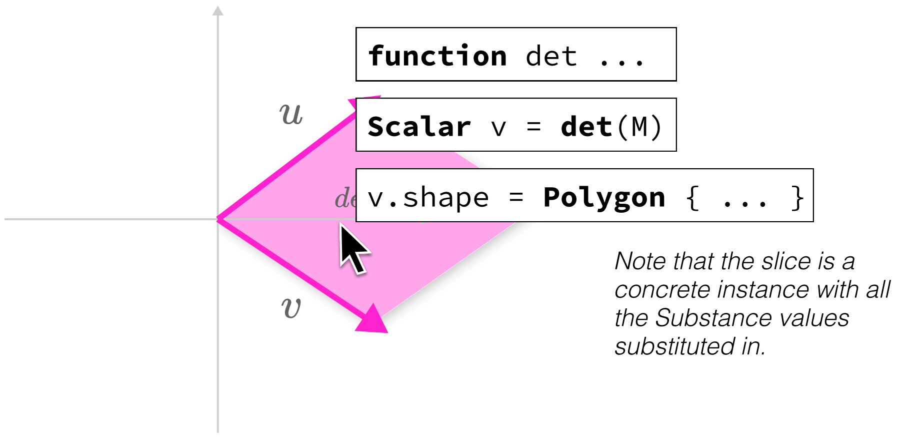

5
4
3
2
1
0
0
1
2
3
y
x
Av
v
a_2
a_1
v transformed by A
v: Example vector in space
A: Matrix with a_1 and a_2 as columns

$$A = \begin{bmatrix} a_1.x & a_2.x \\ a_1.y & a_2.y \end{bmatrix} = \begin{bmatrix} 1.93 & 0.28 \\ 1.17 & 1.21 \end{bmatrix}$$

In technical documents, symbols and acronyms are often defined once and used everywhere else. To help readers understand them, tools like ScholarPhi and Nota [25, 57] use tooltips to aid readers. In real world publications, authors augment math equations for better readability, too. Diagrams use even more symbolism and can be hard to understand. We propose a lightweight markup language in the form of SUBSTANCE documentation for authoring simple *diagram aug-*

mentation. Similar to Idyll [14], the markup language has a markdown-like syntax, but allows splices of SUBSTANCE variables and runtime values. In the frontend, we analyze the SUBSTANCE values in each snippet, and trace all related snippets based on variable references. For instance, the snippet about Av refers to both A and v, so they appear on the tooltip stack.

The translational semantics also involve how DOMAIN, SUBSTANCE, and STYLE programs relate to each other. Therefore, STYLE and DOMAIN can also be valuable sources of feedback: the STYLE program encodes the visual semantics, and the DOMAIN program captures the grammar of notations. A slice of a PENROSE trio traces the origin of a graphical primitive to the DOMAIN, SUBSTANCE, STYLE programs. For instance, without any authoring burden, the system can display slices of the program trio based on object selection. Alternatively, the proposed markup language may be extended to DOMAIN and STYLE, and the system can render inline documentations in all three languages.



```
function det ...
```

```
Scalar v = det(M)
```

```
v.shape = Polygon { ... }
```

Note that the slice is a concrete instance with all the Substance values substituted in.

4.4 Evaluation

To evaluate the discussed interactive techniques, I plan to conduct comparative case studies between feature-full modern JavaScript libraries (e.g. D3.js) and PENROSE. Research questions for this study include:

- Does the PENROSE-based system simply programming interactive diagrams?
- Are the interactive features comparable to the hand-written examples?
- How expressive is our grammar of interactivity?
- When does the approach break down?

In general, I expect that our system can cover common, important interactive features with significantly less manual effort. In the studies, I plan to collect both quantitative (e.g., lines-of-code, time taken) and qualitative data about authoring interactive diagrams using JS library versus our system. Currently, the candidate pool of examples include:

- Worked examples and explorable explanations:
    - A Gentle Introduction to Graph Neural Networks: `https://distill.pub/2021/gnn-intro/`

    - Explained visually: `https://setosa.io/ev/`
    - Explorable explanations: `https://explorabl.es/`
    - Gallery of concept visualization: `https://conceptviz.github.io/`
- Online textbooks and curricula:
    - Seeing theory: `https://seeing-theory.brown.edu/`
    - Immersive math: `http://immersivemath.com/ila/index.html`
    - Mathigon: `https://mathigon.org/`
    - Physically-based rendering: `https://pbr-book.org/`
    - Brilliant: `https://brilliant.org/`

4.5 Related work

4.5.1 Grammars for interactivity

Pioneered by the grammar of graphics [56], researchers in data visualization developed a rich set of tools based on an explicit encoding of the mapping from data to visual primitives [8, 51, 52]. Notably, Vega-Lite [52] is a grammar for interactive data visualization. One key to the Vega-Lite grammar is "selection," because the underlying data doesn't change during interaction. In diagramming, this assumption is not always true. The basic building blocks are mostly "manipulation" of shapes in relation with the underlying representation. Satyanarayan et al. [53] give an extensive review of data visualization authoring tools, including those that support interactivity.

In addition, the information/data visualization literature also contributed taxonomies and conceptual frameworks of interactivity. For instance, Yi et al. [59] propose 7 general categories of interactive techniques in information visualization: Select, Explore, Reconfigure, Encode, Abstract/Elaborate, Filter, and Connect. Similarly, Heer and Shneiderman [26] describe a taxonomy of interactive dynamics for visual analysis, which was presented as a list of verbs, too. These frameworks and taxonomies are useful to build upon. Again, the high-level concepts involved for interactive diagrams can change significantly from them because of the differences between diagrams and data/information visualization [37],

4.5.2 Constraint-based interactivity

In the HCI literature, there's a long line of work on authoring interactive user interfaces using constraints. For instance, Garnet [41] and Amulet [42] use dataflow constraints to build highly interactive UIs. In these systems, the author declaratively specifies constraints on the relationship among graphical elements in terms of data dependencies (i.e. $D = f(l_1, l_2, \dots, l_n)$, and the interactivity is handled by a constraint solver at runtime. Thinglab [7] supports simultaneous equations for building simulations. At its core, the PENROSE system is a combination of dataflow constraints (computations) and simultaneous equations (constraints and objectives). Therefore, many of the techniques from this line of work may apply to supporting interactivity in PENROSE, such as efficient incremental constraint solving [50].

Bibliography

[1] Umair Z Ahmed, Sumit Gulwani, and Amey Karkare. Automatically Generating Problems and Solutions for Natural Deduction. In *IJCAI '13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1968–1975, 8 2013. 3.7.2

[2] Shaaron Ainsworth, Vaughan Prain, and Russell Tytler. Drawing to learn in science. *Science*, 333(6046):1096–1097, 8 2011. 4

[3] Vincent Aleven, Bruce M. McLaren, Jonathan Sewall, and Kenneth R. Koedinger. The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4053 LNCS:61–70, 2006. 3.7.2

[4] Vincent Aleven, Jonathan Sewall, Bruce M. McLaren, and Kenneth R. Koedinger. Rapid authoring of Intelligent Tutors for real-world and experimental use. *Proceedings - Sixth International Conference on Advanced Learning Technologies, ICALT 2006*, 2006:847–851, 2006. 3.7.2

[5] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. *Proceedings of the Annual ACM Conference on Computational Learning Theory*, pages 92–100, 1998. 1

[6] Eliza Bobek and Barbara Tversky. Creating visual explanations improves learning. *Cognitive Research: Principles and Implications*, 1(1):1–14, 12 2016. 4

[7] Alan Boming. Thinglab–A Constraint-Oriented Simulation Laboratory. *Stanford Univ. report STANCS-79-746*, 1979. 4.5.2

[8] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. 4.5.1

[9] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004. 2.2

[10] Edward B Burger, David J Chard, Earlene J Hall, Paul A Kennedy, Steven J Leinwand, Freddie L Renfro, Dale G Seymour, and Bert K Wattis. *Holt geometry*. Holt, Rinehart and Winston, 2007. 3.3

[11] Michelene T.H. Chi and Ruth Wylie. The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes. *Educational Psychologist*, 49(4):219–243, 10 2014. 3.7.2

[12] Pakey P.M. Chik and Mun Ling Lo. Simultaneity and the enacted object of learning.

In *Classroom Discourse and the Space of Learning*, volume 9781410609, pages 89–112. Routledge, 4 2004. 3.7.1

[13] James M. Clark and Allan Paivio. Dual coding theory and education. *Educational Psychology Review 1991 3:3*, 3(3):149–210, 9 1991. 1

[14] Matthew Conlen and Jeffrey Heer. IdylL: A markup language for authoring and publishing interactive articles on the web. In *UIST 2018 - Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 977–989. Association for Computing Machinery, Inc, 10 2018. 4.3.2

[15] Louis Deslauriers, Logan S. McCarty, Kelly Miller, Kristina Callaghan, and Greg 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):19251–19257, 9 2019. 3.7.2

[16] Andrea A. DiSessa. Metarepresentation: Native Competence and Targets for Instruction. *http://dx.doi.org/10.1207/s1532690xci2203_2*, 22(3):293–331, 2010. 3

[17] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz— Open Source Graph Drawing Tools. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2265 LNCS:483–484, 2001. 2.2

[18] K Anders Ericsson. The Influence of Experience and Deliberate Practice on the Development of Superior Expert Performance. In *The Cambridge handbook of expertise and expert performance.*, pages 683–703. Cambridge University Press, New York, NY, US, 2006. 1, 3.7.2

[19] Fernand Gobet. Chunking models of expertise: implications for education. *Applied Cognitive Psychology*, 19(2):183–204, 3 2005. 1

[20] T. R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2): 131–174, 6 1996. 2.1

[21] Sumit Gulwani. Example-based learning in computer-aided STEM education. *Communications of the ACM*, 57(8):70–80, 2014. 3.7.2

[22] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. *ACM SIGPLAN Notices*, 46(6):50–61, 6 2011. 3.7.2

[23] Jacques Hadamard. *The Mathematician's Mind.* Princeton University Press, 12 1997. 1

[24] D F Halpern, A Graesser, and M Hakel. Learning principles to guide pedagogy and the design of learning environments. *Association for Psychological Science*, 2007. 3.7.1

[25] Andrew Head, Kyle Lo, Dongyeop Kang, Raymond Fok, Sam Skjonsberg, Daniel S Weld, and Marti A Hearst. Augmenting scientific papers with just-in-time, position-sensitive definitions of terms and symbols. *Conference on Human Factors in Computing Systems - Proceedings*, 5 2021. 4.3.2

[26] Jeffrey Heer and Ben Shneiderman. Interactive Dynamics for Visual Analysis. *Queue*, 10 (2):30–55, 2 2012. 4.5.1

[27] Yamashita Hiroshi and Takahito Tanabe. A Primal-Dual Exterior Point Method for Nonlinear Optimization. *http://dx.doi.org/10.1137/060676970*, 20(6):3335–3363, 11 2010. 2.2

[28] Alan Kay. Doing with images makes symbols, 1987. 1

[29] Philip J. Kellman and Christine M. Massey. Perceptual Learning, Cognition, and Expertise. In *Psychology of Learning and Motivation - Advances in Research and Theory*, volume 58, pages 117–165. Academic Press, 1 2013. 1

[30] Philip J. Kellman, Christine M. Massey, and Ji Y. Son. Perceptual learning modules in mathematics: Enhancing students' pattern recognition, structure extraction, and fluency. *Topics in Cognitive Science*, 2(2):285–305, 4 2010. 3.1, 3, 3.7.1, 3.7.2

[31] Kenneth R. Koedinger and John R. Anderson. Abstract planning and perceptual chunks: Elements of expertise in geometry. *Cognitive Science*, 14(4):511–550, 10 1990. 1, 1, 3.1

[32] Kenneth R. Koedinger and Atsushi Terao. A Cognitive Task Analysis of Using Pictures To Support Pre-Algebraic Reasoning. *Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society*, pages 542–547, 4 2019. 1

[33] Kenneth R. Koedinger, Martha W. Alibali, and Mitchell J. Nathan. Trade-Offs Between Grounded and Abstract Representations: Evidence From Algebra Problem Solving. *Cognitive Science*, 32(2):366–397, 3 2008. 1

[34] David R Krathwohl and Lorin W Anderson. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Longman, 2009. 4

[35] Jill H. Larkin and Herbert A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11(1):65–100, 1 1987. 1

[36] Catherine Letondal, Stéphane Chatty, W Greg Phillips, Fabien André, and Stéphane Conversy. Usability requirements for interaction-oriented development tools. *Psychology of Programming*, pages 12–26, 2010. 4

[37] Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. How Domain Experts Create Conceptual Diagrams and Implications for Tool Design. *Conference on Human Factors in Computing Systems - Proceedings*, 20, 4 2020. 2, 3.1.3, 4.5.1

[38] Ference Marton. Sameness and Difference in Transfer. *Journal of the Learning Sciences*, 15(4):499–535, 2006. 3, 3.7.1

[39] Richard Mayer. *Multimedia Learning*. Cambridge University Press, 7 2020. 1, 1

[40] Brad A Myers. Separating application code from toolkits: Eliminating the Spaghetti of callbacks. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991*, pages 211–220, New York, New York, USA, 1991. ACM Press. 4

[41] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *Computer*, 23(11):71–85, 1990. 4.5.2

[42] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, and Bruce D. Kyle. The amulet environment: New models for effective user

interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997. 4.5.2

[43] Brad A Myers, John F Pane, and Amy Ko. Natural programming languages and environments. *Communications of the ACM*, 47(9):47–52, 2004. 2.1

[44] Mitchell J Nathan, Ana C Stephens, D K Masarik, Martha W Alibali, and Kenneth R Koedinger. Representational fluency in middle school: A classroom study. In *Proceedings of the twenty-fourth annual meeting of the North American chapter of the International Group for the Psychology of Mathematics Education*, volume 1, pages 462–472. ERIC Clearinghouse for Science, Mathematics and Environmental Education˜. . . , 2002. 3

[45] Fred G.W.C. Paas and Jeroen J.G. Van Merriënboer. Variability of Worked Examples and Transfer of Geometrical Problem-Solving Skills: A Cognitive-Load Approach. *Journal of Educational Psychology*, 86(1):122–133, 1994. 3.7.2

[46] Harold Pashler, Patrice M Bain, Brian A Bottge, Arthur Graesser, Kenneth Koedinger, Mark McDaniel, and Janet Metcalfe. Organizing Instruction and Study to Improve Student Learning. Technical report, NCER, IES,, U.S. Department of Education, Washington, DC, 2007. 3.7.2

[47] Martina A Rau. *Conceptual learning with multiple graphical representations: Intelligent tutoring systems support for sense-making and fluency-building processes*. PhD thesis, Carnegie Mellon University, 2013. 1, 1, 3.7.1, 4.3.1

[48] Leena Razzaq, Jozsef Patvarczki, Shane F. Almeida, Manasi Vartak, Mingyu Feng, Neil T. Heffernan, and Kenneth R. Koedinger. The ASSISTment builder: Supporting the life cycle of tutoring system content creation. *IEEE Transactions on Learning Technologies*, 2(2): 157–166, 2009. 3.7.2

[49] Doug Rohrer and Kelli Taylor. The shuffling of mathematics problems improves learning. *Instructional Science*, 35(6):481–498, 11 2007. 3.7.2

[50] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience*, 23(5):529–566, 5 1993. 4.5.2

[51] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative interaction design for data visualization. In *UIST 2014 - Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pages 669–678, New York, NY, USA, 2014. ACM. 4.5.1

[52] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 1 2017. 4.5.1

[53] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John Thompson, Matthew Brehmer, and Zhicheng Liu. Critical reflections on visualization authoring systems. *IEEE Transactions on Visualization and Computer Graphics*, 26(1): 461–471, 1 2020. 4.5.1

[54] Heidi L. Schnackenberg, Howard J. Sullivan, Lars F. Leader, and Elizabeth E.K. Jones.

Learner preferences and achievement under differing amounts of learner practice. *Educational Technology Research and Development*, 46(2):5–16, 1998. 3.7.2

[55] Christine 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):725–746, 4 2016. 4

[56] Leland Wilkinson. The Grammar of Graphics. *Handbook of Computational Statistics*, pages 375–414, 2012. 4.5.1

[57] Will Crichton. A New Medium for Communicating Research on Programming Languages. In *11th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2020)*, Virtual, 2020. 4.3.2

[58] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Transactions on Graphics*, 39(4):16, 7 2020. 1, 2.2, 4.1, 4.2

[59] Ji Soo Yi, Youn Ah Kang, John T. Stasko, and Julie A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 11 2007. 4.5.1