SHF: SMALL: CREATING EFFECTIVE VISUAL DIAGRAMS USING PROGRAMMING LANGUAGES

PIs: Jonathan Aldrich, Keenan Crane, Joshua Sunshine

1 INTRODUCTION

Overview. Human knowledge is growing at an astounding rate: each day, academic researchers and industry practitioners across a wide variety of fields produce thousands of new research papers, legal documents, pieces of code, *etc.*—all examples of highly-structured data that express rich and intricate **logical relationships**. How can we keep up with the pace of information? Data visualization is a powerful tool for understanding *numerical, quantitative data* (banking information, web traffic, *etc.*) but does not provide an equally powerful solution for visualizing **abstract logical relationships**, where *quantity* may not be involved at all. Meaningful diagrams demonstrably improve human understanding, leading to more effective education, better decision making, and better task performance [**?**, **?**, **?**]; moreover, many domains (math, law, biology, *etc.*) already have a common "language" of diagrams [**?**, **?**, **?**, **?**, **?**, **?**]. Yet in practice such diagrams are often omitted due to the tremendous skill needed to produce them by hand.

Problem Statement and Scope. The goal of this proposal is to build fundamental tools needed to address the "next wave" of challenges in semantic data visualization. The aim is not to develop fixed visualization algorithms for specific types of information, but rather an extensible language-based framework that enables users to rapidly develop their own custom visualizations for structured information. As a concrete target, we focus primarily on the challenges associated with visualizing *mathematical expressions*, since this domain has a well-established syntax, and is an area where we can have immediate impact on both education and scientific communication. In particular, we are motivated by the following problem:

> Given an expression in standard mathematical notation,
> automatically generate a corresponding visual diagram.

In other words, a "visual LaTeX" which produces diagrams (rather than typesetting) as output. **Over the past two years we have taken significant steps toward this goal** by prototyping a system called PENROSE [**?**, **?**], which already handles basic diagrams (Figure 4), and provides clear evidence that our goal is achievable. Based on this preliminary work, our project mailing list has already accumulated over 900 subscribers, 800 stars on *GitHub*, and appeared as the 2nd-most upvoted story on Hacker News—there is a clear real-world need for tools that provide rapid translation of logical statements into visual diagrams. In order to establish a clear picture of what the specific needs look like, **we have also performed extensive interviews** with both novice and professional mathematical users, **and done a large-scale statistical study of diagram use** in documents on *arXiv*, which has helped to inform the research questions outlined in this proposal.

In the near-term we will address fundamental challenges across programming languages, numerical optimization, and visualization needed to turn this prototype into a full-featured tool that can have a real-world impact on scientific research and education. The longer-term motivation (beyond the 3–5 year window), is that the highly structured domain of mathematics provides essential scaffolding for building tools that can understand information from domains with less structured syntax (like law or biology); see further discussion in Section 2. The ultimate vision is to adapt this system to completely *unstructured* input—imagine, for instance, automatically illustrating any existing textbook, research paper, or legal document solely based on its content, without any additional user input. This vision is achievable, but only if we first develop the fundamental representations and algorithms needed to translate abstract relationships into concrete depictions.

Approach. The central hypothesis of our approach is that by allowing users to **easily encode how logical relationships should be translated into visual/graphical relationships**, we open the door to fundamentally

new opportunities not addressed by existing diagramming tools. The technical approach has two main components:

- A language-level specification of how logical relationships are translated into graphical constraints.
- Numerical optimization strategies that generate diagrams satisfying the given constraints.

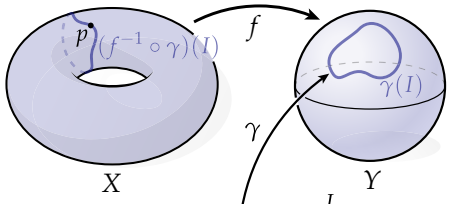
On top of this basic platform we can then easily build different tools which enable novice users to specify and interact with diagrams: either by (i) plain text notation that closely mirrors standard "textbook" mathematical notation, or (ii) interactive visual tools that allow users to generate and interact with diagrams while respecting important logical/semantic relationships (Figure 1). More advanced users can further develop custom visual representations tailored to a particular domain of knowledge (Figure **??**). In stark contrast, existing programmatic tools like *TikZ* or *Asymptote* require the user to provide explicit numerical coordinate values for every single graphical element (lines, points, labels, *etc.*). Even with higher-level tools like *Wolfram Alpha*, *MATLAB*, or *Maple* users must meticulously specify a particular function that should be plotted (*e.g.*, $f(x) := \frac{1}{2}x^2$) rather than generic relationships.

Use cases. Why do we need a fundamentally new approach to making diagrams? Existing tools are quite capable at addressing the needs of the print era (*e.g.*, generation of "one off", static diagrams), but do not address a much broader range of use cases relevant in the digital era. In particular:

- LARGE SCALE GENERATION — Consider a classroom setting where one wishes to generate hundreds of illustrated exercises for each student each semester. Using traditional graphical tools (such as *Illustrator*) or programmatic tools (such as *TikZ*) is simply not feasible due to the vast manual labor required. While existing online tutors can now randomly generate math- and word-problems, automatically generating accompanying figures is much harder, requiring custom code and a much higher level of expertise. By providing a bridge between language-based and graphical descriptions, Penrose opens the door to large-scale generation of diagrams (via, *e.g.*, existing techniques from program synthesis).
- EVOLUTION OF COLLECTIONS — After generating a collection of diagrams, it often needs to evolve over time. Consider for instance a publisher who needs to re-target thousands of articles or textbooks for a new delivery platform (*e.g.*, mobile or tablet rather than print) or has new publication demands (file formats, typographical specifications, *etc.*). This kind of diagram evolution is a major undertaking with traditional tools; with Penrose it is a matter of tweaking *Style* programs and re-compiling diagrams.
- INTERACTIVE EXPLORATION — Of course, the fact that the Penrose platform is built on top of language does not limit it to language-based interfaces. Unlike traditional digital illustrations, which just contain raw drawing commands, every diagram generated by Penrose has constraints "baked in" that allow a user to interactively adjust a drawing while preserving the *meaning* of the diagram. More broadly, the information encoded by Penrose allows users to explore the rich space of possible visual explanations for a given mathematical statement, providing alternative perspectives and enabling them to discover, *e.g.*, important special cases that may not have been apparent from a single static drawing (see Section **??**).

Figure 1 depicts an example of what a working system might look like. Importantly, the focus is not on expressions that already have a direct visual interpretation (like the graph of a function, which could easily be plotted using tools like *Mathematica* or *MATLAB*), but rather on *abstract* relationships that are not as easily to visualize (implication, equivalence, causality, containment, *etc.*). Though one can cite a litany of software packages that help visualize specific classes of logical relationships (*e.g.*, *Visio* [**?**] for flowcharts, or *GroupExplorer* [**?**]

```
TopologicalSpace X,Y
π1(X) = DirectProduct(Ints,Ints)
π1(Y) = TrivialGroup
I := [0,1] Subset Reals
ContinuousMap f : X -> Y
ContinuousMap gamma : I -> Y
gamma(0) = gamma(1)
eta1 = Image(gamma)
eta2 = PreImage(eta1,f)
p In eta1
```



for algebraic groups), the surest sign that there is no effective, general-purpose, and easily-accessible solution is that *most researchers and practitioners are still not including diagrams in their final work*. To give just one example, fewer than one-third of recent mathematics papers on *arXiv.org* contain any figures whatsoever—despite the fact that mathematicians regularly sketch diagrams to develop and understand their own work. Designing a system that is general, extensible, and accessible enough to be useful to real-world practitioners is not a superficial question about, *e.g.*, building a better graphical interface, but rather entails deep inquiry into questions about algorithmic visualization of logical concepts, and how such diagrams can be effectively expressed via formal language.

Interdisciplinary Team. The challenges raised in this proposal demand integration of insights from a variety of fields. Jonathan Aldrich brings expertise in programming languages, including the design of languages that, like PENROSE, are domain-specific [**?**]; the design of object models, modules, and composition mechanisms similar to those PENROSE will need [**?**, **?**]; and the design of modular metaprogramming mechanisms [**?**] that will be needed to make PENROSE extensible to new mathematical domains. Keenan Crane is an expert in algorithms from computational geometry and computer graphics—in particular his work on *fast distance transforms* [**?**] will play a key role in algorithms for diagram layout; he is also co-organizing a workshop on Visualizing Mathematics at ICERM and has himself done extensive work illustrating [**?**] and analyzing illustration techniques for abstract geometric concepts. Joshua Sunshine is an expert in the usability of programming languages. He has organized the popular *PLATEAU* workshop on the topic for the last few years and the human-centered techniques he developed to design and evaluate the *Plaid*, *Glacier*, and *Obsidian* programming languages will be applied in this proposal [**?**, **?**, **?**, **?**, **?**]. Finally, the student most likely to be funded by this work, Katherine Ye, has an extensive background in applied programming languages research [**?**, **?**, **?**]. She has also interned at Google Brain on the team for the *Distill* journal, where she designed and built interactive visualizations for a forthcoming publication [**?**] on the mathematics of optimization in deep learning.

Timeline and Milestones. Work in this project is divided into three distinct "tracks," as depicted in Figure 2. Track I, which spans the majority of this period, focuses on development of domain-agnostic language and visualization facilities (Section 3 and Section 4). In Track II we will work with domain experts to prototype our system on several common mathematical domains (set theory, topology, group theory, *etc.*) which will ultimately constitute the *standard library* for PENROSE. Track III focuses on evaluation of the system according to several specific metrics (productivity, quality, expressivity, adoption) as outlined in Section 5. Parallel to this activity, we will engage in the Broader Impacts activities outlined in Section 6.

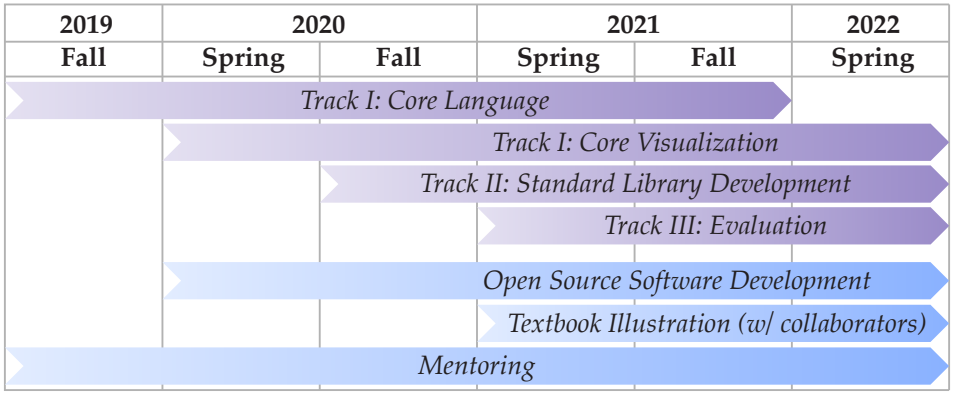


Figure 2: *Timeline of proposed activities. Research activites (top) in purple; broader impact activities (bottom) in blue.*

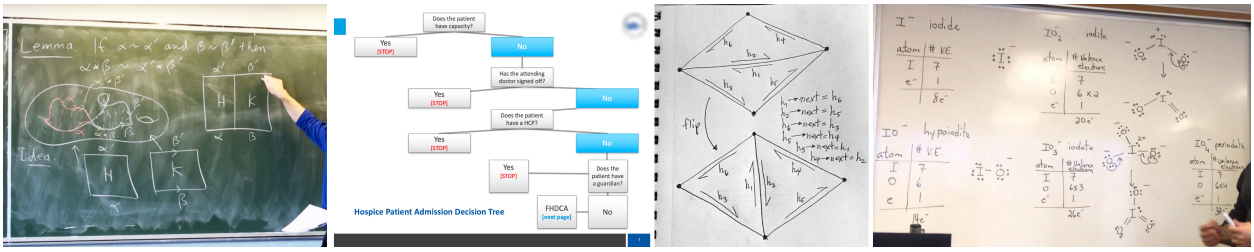


Figure 3: *Left to right: real-world examples of diagrams from mathematics, law, programming, and chemistry. In each instance, diagrams are used to understand* ***semantic or functional relationships****, rather than quantiative data. At present, most such diagrams must be sketched or built by hand and cannot be automatically generated.*

---

2 Broader Impacts—Vision

The rise of the Internet has driven ever more rapid creation and dissemination of highly structued information; making sense of this structure can have a tremendous impact on research, practice, and pedagogy. But how can we possibly keep up with the pace of data? Visualization is a powerful solution for combating this glut of information: studies have shown that the use of diagrams in programming has substantial benefits [**?**], and can likewise improve mathematical [**?**] and problem solving ability [**?**]. Such observations raise an obvious question: if seeing is so important, why are good diagrams so rare in technical writing? Fields medalist William Thurston often lamented this lack of visual exposition [**?**]:

> *"People have very powerful facilities for taking in information visually… On the other hand, they do not have a very good built-in facility for 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."*

Unfortunately, digital tools have not yet caught up to the growing need for compelling visual explanations—existing tools require considerable expertise and a substantial investment of time. Potentially valuable diagrams, such as those sketched on scratch paper or whiteboards, are therefore simply discarded or erased (Figure 3). Tools that ease creation, exploration and transmission of digital diagrams would hence open the floodgates for more rapid understanding.

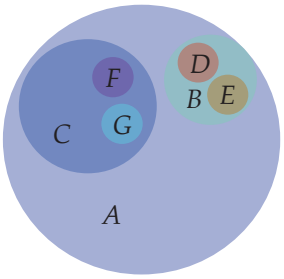
Current tools demand a great deal of low-level manual effort and have a steep learning curve. Moreover, existing paradigms for creating technical diagrams are almost entirely disconnected from the *semantics* of the concept being illustrated. Diagrams are therefore not only difficult to generate, but are also difficult to modify or explore. For example, the popular LaTeX package *TikZ* [**?**] requires the user to explicitly specify each display attribute, such as shape, color, position, *etc.* As a result, *TikZ* users have created collections of macros to help abstract away redundant tasks like drawing commutative diagrams [**?**] or Bayesian networks [**?**]. However, ad-hoc solutions lose the power of real programming languages—for instance, one could never hope to verify that such a diagram correctly represents the concept it is supposed to illustrate. At a more basic level, these highly specialized solutions cannot easily be generalized or adapted to new contexts. Graphical software like *Adobe Illustrator* or *PowerPoint* faces similar problems: since illustration is disconnected from semantics, drawing can be painfully manual and redundant. Moreover, when using a graphical interface users must commit to a specific visual representation and style, making it hard to reuse work or generalize to new examples. Tools like *Mathematica*, *matplotlib*, *Asymptote* [**?**], and *Desmos* serve as "generalized graphing calculators" that can plot fixed classes of concrete functions, but provide little facility for visualizing *abstract, logical relationships* and cannot easily be extended to accommodate user-specific data.

The main point of departure is that our goal is *not to build yet another domain-specific visualization tool*. Instead, we are developing a domain-agnostic **platform** that enables users to easily define custom semantics and visual representations, so that any kind of structured data can be rapidly converted into visual explanations. This system, dubbed PENROSE, enforces a clean separation between semantics and appearance via two extensible languages: SUBSTANCE and STYLE (akin to the separation between content and style provided by HTML and CSS). SUBSTANCE is a programming language for specifying domain-level semantics, completely independent of visual representation (for instance, an expression in standard mathematical notation). STYLE is a language for defining how domain-level semantics get translated into a visual representation—for instance, a Venn diagram or a graph (see Figure 4). As with traditional data visualization, the ability to visualize the semantic data via several *different* graphical representations provides a powerful way to understand information. More broadly, a language-based approach offers tremendous flexibility and generality, and is essential to achieving the long-term vision of illustrating *unstructured* data, since the vast majority of technical knowledge is currently encoded as text.

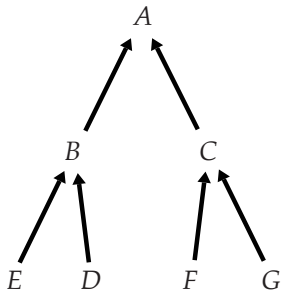
(a) A SUBSTANCE program defines **purely logical** relationships:

```
Set A, B, C, D, E, F, G
Subset B A
Subset C A
Subset D B
Subset E B
Subset F C
Subset G C
NoIntersect E D
NoIntersect F G
NoIntersect B C
```

(b) Two different STYLE programs then define two different ways to **visualize** these relationships:




```
Set x {
    shape = Circle { }
    ensure x contains x.label
}
NoIntersect x y {
    ensure x notOverlapping y
}
Subset x y {
    ensure y contains x
    ensure x smallerThan y
    ensure y.label outsideOf x
}
```




```
Set x { shape = Text{ } }
Subset x y {
    encourage y above x
    encourage x sameX y
    shape =  Arrow {
        start = x.shape
        end   = y.shape
}}
Set x, Set y {
    encourage x repel y
}
```

Figure 4: *Preliminary working prototype of our system. Novice users type purely logical statements* ***(a)*** *which are automatically visualized using distinct styles* ***(b)*** *developed by more expert users.*

On the whole, most users of our system *will* ***not*** *require any graphic design skill to create beautiful diagrams*. The widespread success of LaTeX is a clear indicator that this kind of language-based paradigm can enjoy wide adoption. Likewise, just as LaTeX revolutionized written technical communication by algorithmically codifying the best practices of professional typesetters (allowing authors to focus on *content*), we aim to revolutionize the way people visually communicate logical ideas. Our approach will also enable users to interact with knowledge in brand new ways: a static diagram designed for a textbook can be made interactive, deployed on the web, or even viewed in virtual reality by simply selecting a different STYLE; teachers and students can likewise swap out styles "on the fly" to illustrate the same concept from multiple perspectives. Lowering the barrier to creating *new* styles also encourages development of brand new visual representations of existing domains (that may presently be text-only).

Looking beyond mathematics, insights generated by this work have the potential for transformative change across a variety of industries in the United States. For instance, consulting with lawyers from disparate areas of law we discovered that informal diagrams are often created to aid understanding of complex legal documents (legal statutes, contracts, *etc.*), yet such diagrams are currently sketched by hand or created using low-level tools like *PowerPoint* rather than taking advantage of structured information found in the documents themselves. Likewise, software developers sketch complex diagrams in order to develop or debug code [**?**], but such diagrams are typically discarded since there is no easy way to embed them in the code itself. Existing code visualization tools, such as *Python Tutor* [**?**] and *VisuAlgo* [**?**] are typically limited to common data structures like arrays and trees, or only visualize small pre-chosen examples and not real working code. By instead expending this effort on developing reusable visualizations of logical, semantic data (math, law, code, *etc.*), one can greatly improve the rate and depth with which knowledge can be acquired and communicated across many different human activities.

3 LANGUAGE DESIGN

Major Research Questions. The core goal of the PENROSE language is to provide customizable visual semantics for mathematical expressions. Achieving this goal requires answering a number of high-level programming language research questions:

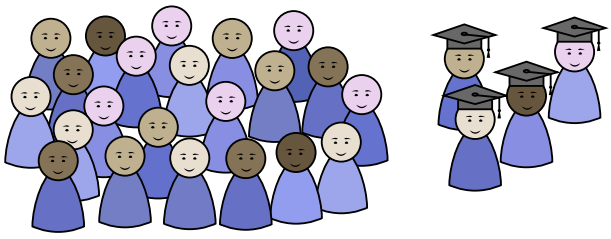
- How can we design a system that is extensible to multiple domains of mathematics, and within a given domain allows users to mix and match mathematical expressions with reusable visualization styles?
- How can we design a source language for writing mathematical expressions that is *natural* enough for students while also being *expressive* enough for domain experts?
- How can we design a language that supports composable styles that are easy to reason about and support deliberately nondeterministic diagram generation?
- How can language tools help users effectively write well-formed mathematical expressions and graphical styles?

TODO: Do we want to add "change to styling affects multiple diagrams" as separate from "mix and match" above?

In the rest of this section, we will outline these research questions in more detail, describe through a running example how our planned approach meets our research goals, and describe the research to be done in the project.

System Design in Support of Generality. The primary goal of Penrose is generality: we wish to allow programmers to describe diverse mathematical expressions from a variety of domains, and visualize them using diverse styles. Another principal goal is reusing mathematical and visualization expertise, thus enabling users who are just learning about mathematics to build on libraries of sophisticated mathematical concepts as well as rich and insightful visualization styles.

Our proposed system design supports these goals by providing three languages: ELEMENT, a language for defining the primitives of a mathematical domain; SUBSTANCE, a language for describing mathematical expressions in a given domain, and STYLE, a language for describing a visualization style for a given domain. Once an ELEMENT domain is selected, users can visualize any SUBSTANCE program in that domain using any STYLE. Typically, an ELEMENT library will be defined by a mathematical expert who knows the language semantics of Penrose well; many ELEMENT libraries are provided with the platform and defining new ones is relatively uncommon. Likewise, STYLE libraries will be defined by visualization experts with knowledge of a given domain of mathematics, and general familiarity with Penrose, but who may not have programming language expertise. SUBSTANCE programmers can be written by anyone who knows or is learning a mathematical domain; they can reuse the semantic expertise of the ELEMENT writer and the visualization expertise of the style writer without having such expertise themselves.



SUBSTANCE *(typical users)*

ELEMENT / STYLE *(expert users)*

Figure 5: *Most users of our system (left) need only interact with the* SUBSTANCE *language, which allows them to get a diagram by simply typing expressions in familiar mathematical notation. Expert users (right) can extend the system to new mathematical domains and visual representations via the* ELEMENT *and* STYLE *languages, respectively.*

Natural Yet Expressive Mathematical Syntax. Penrose is intended to be usable with minimal training, for example in a classroom setting. Therefore, the syntax used to describe mathematics should be natural, easy to write, and well-supported by a wide variety of text editors. It should also be independent of any visualization elements, since we want to be able to describe the same mathematical construction in multiple diagrams without duplicating the construction itself. Yet it should also be highly expressive, allowing users to describe mathematical objects from a wide variety of domains along with a rich set of relationships between those objects.

The following SUBSTANCE code, describing a set of objects and relationships in the domain of linear algebra, illustrates our language design:

$$
\begin{aligned}
&\texttt{VectorSpace}\ U\\
&\texttt{Vector}\ u_1, u_2, u_3, u_4, u_5 \in U\\
&u_3 = u_1 + u_2\\
&u_5 = u_3 + u_4
\end{aligned}
$$

The notation is largely plain text, which allows end users to use any text editor to compose SUBSTANCE programs. Above, we show what a program might look like in an editor that has support for unicode characters such as $\in$, performs syntax highlighting (e.g. bolding keywords such as **`VectorSpace`** and italicizing variables such as U), and is able to render simple notation such as `u_1` into a subscripted variable u_1. The syntax follows typical mathematical conventions: for example **`VectorSpace`** U declares an object U of type **`VectorSpace`**.

On a technical level, SUBSTANCE programs consist of statements s, each of which is a variable declaration (with a variable name and type), predicate (with a predicate name and expression arguments), or equality (with a variable on the left and an expression on the right). Expressions consist of variables and function application:

$$
\begin{aligned}
s &::= \mathit{type}\ x \mid \mathit{pred}(\bar{e}) \mid x = e\\
e &::= x \mid \mathit{func}(\bar{e})
\end{aligned}
$$

In the program above, for example, the declaration **`Vector`** $u_1, u_2, u_3, u_4, u_5 \in U$ is syntactic sugar for the declaration of 5 variables, and 5 predicates of the form $in(u_1, U)$. The expression $u_1 + u_2$ is syntactic sugar for application of the vector addition function, i.e. $AddV(u_1, u_2)$. This yields a small grammar that can nevertheless describe a very wide set of mathematical objects from a variety of domains, along with relationships (expressed via predicates, equality, and functions) among those objects.

In the planned research, we will apply the natural programming research method [**?**] to explore refinements and alternatives to the syntax described in this example, helping to ensure that students and other novice users are able to write SUBSTANCE programs with little training.

Modeling Mathematical Domains. The ELEMENT language allows mathematicians to define the kinds of objects, functions, and relationships that are relevant to a given domain of mathematics. Our goal is to support a wide variety of domains, including domains defined by independent users. We seek to model the structure of objects and relationships in the domain–for example, that adding two vectors produces another vector—but not their detailed semantics (e.g. Penrose is not intended to support proofs about mathematical objects; that role is well supported by proof assistants such as Coq [**?**]). We also want to support notation specific to a domain.

Here are the sample ELEMENT definitions for the example program above:

```
type VectorSpace
type Vector
predicate In(v : Vector, V : VectorSpace)
operator AddV(v1 : Vector, v2 : Vector)
StmtNotation "Vector a ∈ U" → "Vector a; In(a, U)"
StmtNotation "v1 + v2" → "AddV(v1,v2)"
```

Here, we define constructors for the mathematical types in a domain (e.g. Vector and Vectorspace), operators (e.g. Vector addition), and predicates (e.g. a Vector may be In a VectorSpace). The example also shows a lightweight definition of a more natural syntax for adding Vectors and asserting that a Vector is in a VectorSpace.

One challenge with the sketched syntax extension facility shown above is modularity: two syntax extensions written in two separate ELEMENT libraries may conflict if a single SUBSTANCE program uses both libraries. We propose to build on our prior work that introduced modular, type-based extensible parsing [**?**], which allows library developers to define expression-based syntax extensions that, by construction, cannot conflict. A complementary keyword-based modular language extension approach by Schwerdfeger *et al.* [**?**] will be useful to support extensible top-level mathematical declarations.

STYLE Language Design. The STYLE language transforms mathematical objects into abstract graphical objects along with constraints and objectives over them. It should be compositional, so that one style can build on another, and so that users can make additions to a style that customize it for a particular diagram. It is important for STYLE to express nondeterminism, so that a single style can be used to generate multiple diagram variants from a single mathematical expression, where appropriate.

Here is an example of our STYLE language, designed to generate a diagram from the SUBSTANCE code above:

```
Vector v
with VectorSpace U
where v ∈ U {
  v.shape = Arrow {
    start = U.shape.center
  }
  ensure contains(U.shape, v.shape)
  encourage nearHead(v.shape, v.text)
}

Vector u
with Vector v,w; Vectorspace U
where u = v + w; u,v,w ∈ U {
  u.shape.end = v.shape.end + w.shape.end
  ...
}
```

In our design, STYLE pattern matches over mathematical objects and relationships defined in SUBSTANCE. For example, the first three lines indicate that we should match every Vector v in the SUBSTANCE program, such that there exists some VectorSpace U and some relationship $v \in U$. The matching semantics is universal for the declarations on the first line (we match for every possible v) but existential for the "with" and "where" clauses (we match nondeterministically for some U and some $v \in U$ but if there is more than one match, we don't take the cross product). This semantics gives the STYLE writer control over what is matched, by putting more or fewer variables in the first line.

For each matched mathematical object, one or more *abstract graphical objects* can be produced. In the example, we generate an Arrow to represent the vector v, and we constrain the arrow's tail to be at the origin of the VectorSpace. The abstract graphical objects are identified by newly introduced fields of the mathematical object—v.shape in the example. This allows later patterns to identify and extend (or override) a graphical object with new characteristics. For example, the second match statement finds a vector u where $u = v + w$. It adds a constraint setting the end of the u vector to be the addition of the ends of the v and w vectors.

We can add constraints (which must be satisfied) and objectives (which the solver tries to optimize). For example, the built-in contains predicate ensures that the vector is shown completely within the vectorspace. With nearHead, we encourage the label for the arrow (v.text, which defaults to the name of the vector in SUBSTANCE) to be near the head of the arrow.

These graphical objects are *abstract* in that some characteristics are left unspecified, and may be nondeterministically be chosen by the constraint solving back end. For example, the solver may determine the end point of all arrows except those constrained by vector addition.

TODO: show output

Since STYLE interprets mathematical objects from SUBSTANCE, specifying corresponding graphical objects as well as constraints and objectives over them, we say that STYLE provides a *graphical semantics* for SUBSTANCE programs.

TODO: should we discuss the target of STYLE as a language itself? e.g. differentiability, extensibility, enable prodi

Semantics and Tooling. As with any computer language, the semantics of ELEMENT, SUBSTANCE, and STYLE are important to get right. A well-defined semantics ensure that the language behaves in a predicable way and facilitates the construction of useful tools. We plan to define a type system suggested by the examples above. The ELEMENT typing rules primarily introduce new types, functions, operators, and predicates into the typing environment. The SUBSTANCE and STYLE rules use that environment in the standard way to ensure that SUBSTANCE variable declarations, constraints, and expressions are well-formed, and to check that STYLE patterns are well-formed.

TODO: a typing rule would be *completely* standard–but is it still a good idea to provide one to give the user an id

We will explore potential typing rules for the body of STYLE patterns; these rules are interesting because STYLE bodies can add fields to objects, but only to objects matched by the patterns. Using linear types, or perhaps abstract interpretation, to check STYLE bodies may be appropriate since the types of objects change as patterns are applied. This kind of checking has the potential to get complicated, and it is important that the typechecker provide error messages that are easy to understand for users who are experts in visualization but perhaps not in programming languages. Our goal will be to provide error messages when there is definitely a problem with the Style body definitions, but deferring errors to run-time checks when static analysis cannot be certain that there is an issue. We will leverage PI Aldrich's experience with gradual type systems [**?**] and gradual verification [**?**], which similarly mix static and dynamic checking in order to provide a good end-user experience.

We will also define a dynamic semantics describing the mathematical objects produced by SUBSTANCE programs, specifying how STYLE patterns match against those objects, and defining how graphical objects are produced when a STYLE rule is triggered.

These semantics can also be used to facilitate editor tools such as syntax highlighting and code completion. This is not a major part of our research plan, however, we will build editor tools opportunistically as needed

to facilitate the educational use of Penrose as well as are user-focused evaluation studies.

4 VISUALIZATION

4.1 GOALS FOR VISUALIZATION

How does a structured approach to diagram specification help with diagram visualization? The language-based representation of diagrams outlined in Section 3 opens the door to visualization techniques that are simply not possible with existing tools. The basic reason has to do with how diagrams are specified: in a graphical tool like *Adobe Illustrator* a user specifies low-level graphical primitives (points, lines, *etc.*) using cursor clicks; likewise, existing language-level tools like *TikZ* and *Asymptote* use low-level commands to specify the placement and style of individual elements. This data provides no information about the *semantics* of these graphical objects: what do they mean, and how are they related? As a result, it is essentially impossible to decide whether (for instance) a subsequent modification of a diagram is "valid," *i.e.*, does it still convey the same meaning as the original diagram? In stark contrast, the rich semantic information provided by *Penrose* syntax furnishes highly sophisticated editing and exploration of diagrams at the graphical level.

Concept Exploration. As outlined below, each Penrose diagram can be interpreted as a *constraint space*. The ability to *explore* this space provides a great deal of power relative to conventional diagrams. In particular, we will develop numerical and graphical algorithms that enable the following diagram exploration paradigms:

- *(Design Galleries)* Deeper understanding of mathematical concepts often comes form seeing many disparate examples. A constraint-based design makes it possible to randomly sample a "design gallery" of candidate diagrams, which can be iteratively evolved by sampling points near the user's favorite candidate. This paradigm has been shown to work well even for novice users with no graphical expertise [**?**].
- *(Interactive Manipulation)* By interactively adjusting elements of a diagram *while respecting constraints*, users can develop a sense of which relationships are fundamental and which are merely superficial or coincidental features of a particular diagram. This kind of exploration is critical for avoiding misconceptions—an elementary example is the statement "A, B, and C are sets; B and C are subsets of A." In the absence of visualization tools, a reader might imagine (or even draw) a diagram where circles representing B and C are both contained in A, and B and C are disjoint. Of course, nothing in this statement forbids a situation where, *e.g.*, B is also contained in C Figure 10.
- *(Special Cases)* Special examples (*e.g.*, diagrams with a high degree of symmetry) are essential for revealing "corner cases" in a mathematical definition. The constraint-based framework makes it possible to *automatically* construct such examples by optimizing the eigenvalues of the constraint Jacobian (see Section 4.2 for further discussion). Alternatively, eigenvalue optimization can be used to ensure that a diagram represents the general case (*e.g.*, making triangles scalene rather than isosceles or equilateral), helping again to avoid misconceptions.
- *(Equivalence)* Showing that two statements or objects are *equivalent* is one of the cornerstones of mathematical education and understanding [**?**]. Penrose enables users to explore questions of equivalence by testing whether or not there are continuous transitions between different diagram instances; here, manual exploration can be assisted via sophisticated *motion planning* algorithms from robotics [**?**].

Note that these exploration strategies are available only because we have an explicit and highly structured mapping between logical relationships and graphical constraints—information that is not available in even in today's most sophisticated visual or programmatic diagramming tools. Moreover, since these strategies

depend on nothing more than the constraint functions encoded by the *Style* code, they are completely generic and can be immediately applied to any new domain defined by the user, without the need for sophisticated high-level reasoning.

Dynamic Visualization Contemporary digital media increasingly demands diagrams for applications beyond the printed page. The rich metadata available in the Penrose framework makes it possible to generate *dynamic* diagrams in a wide variety of contexts. In particular:

- *(Interactive Diagrams)* The importance of interactive exploration has been outlined above; to facilitate such interaction we will develop high-performance numerical solvers that allow real-time generation and modification of constrained diagrams (as outlined in Section 4.2).
- *(Conceptual Dependency)* An effective way to communicate dependency relationships among objects or definitions is to translate these dependencies into temporally "staged" figures where each piece of a diagram is added in a semantically meaningful order (as might be done in a PowerPoint presentation). A nice feature of our language-based specification is that this dependency graph is already implicit in any given *Substance* program. Since a given graph typically has many different linear orderings, we will also explore ways to generate groupings and orderings that help convey logical meaning (*e.g.*, semantically similar objects appearing at the same moment in the sequence).
- *(Animation)* Animated diagrams (as might appear in an educational *YouTube* video) help to illustrate ideas like equivalence or relationships among different special cases, as discussed above. The constraint-based framework makes it easy to implement common animation editing interfaces such as constrained *keyframing*; combining these standard techniques with powerful language-level editing (*e.g.*, adding or removing statements from SUBSTANCE code at different moments in time) will make it easy to generate rich animations suitable for, *e.g.*, classroom or conference presentations.
- *(Debugging)* Debugging of computer programs is most typically achieved via text-based debuggers, which can make it difficult to track the evolution of intricate data structures–yet many abstract data structures can be naturally associated with one or more visualizations. In the later phases of this project, we will explore how the SUBSTANCE/ELEMENT/STYLE paradigm might be used to help visualize program execution (and where it needs to be augmented or adapted).

Note that *none* of these applications are easily achieved via traditional diagramming tools, since even tools that provide animation facilities (such as *Adobe Flash*) cannot draw on semantic information to ensure that diagrams remain *meaningful*, nor use semantics to infer a logical ordering for the presentation of a diagram. Likewise, some data visualization tools provide facilities for interaction, animation, exploration, *etc.*, but can generally only deal with numerical, *quantitative* data rather than non-quantitative logical relationships.

4.2 VISUALIZATION—TECHNICAL APPROACH

From a visualization point of view, the core functionality needed by our proposed system is translation of logical relationships defined at the language level into a concrete visual representation of those relationships, *i.e.*, a diagram. The value provided by PENROSE is that a designer can focus on developing a high-level design specification, rather than having to worry about low-level system design issues (such as choices of numerical optimization strategies or graphical rendering algorithms). The main challenge technical is developing computational strategies that are *flexible* and *scalable* enough to deal with arbitrary user-specified diagrams, especially in a real-time interactive environment. To address this challenge we will we will apply modern techniques from optimization theory, computational geometry, and computer graphics, as outlined below.

The proposal focuses on challenges that are universal to many different types of diagrams—questions about how to effectively visualize a particular domain of mathematical object (*e.g.*, graphs, or groups, or

manifold) are largely up to the designer of a particular STYLE program. For certain classes of objects one can lean on highly specialized algorithms—consider for instance *GraphViz* [**?**], *Group Explorer* [**?**], *Tensorboard* [**?**], or *SBOL Visual* [**?**], which are custom tailored to specific objects from graph theory, abstract algebra, machine learning, and synthetic biology. However, the fixed-function approach is not well suited to many of our system's goals (such as extensibility, or the ability to switch visual representations); moreover, a single diagram often needs to illustrate interactions among several different *kinds* of objects (*e.g.*, a graph *and* a group). A more holistic approach also facilitates generation of more cognitively effective diagrams—for instance, English speakers demonstrate greater comprehension and task performance when the overall left-to-right flow of a diagram follows temporal or logical ordering [**?**]. Systems for cognitively effective visualization and analysis of *quantitative* data have been built for a variety of problems including engineering diagrams [**?**] and general relational databases [**?**]; this proposal investigates complementary questions about how to automatically generate cognitively effective diagrams for purely logical (*i.e.*, non-quantitative) relationships—here we draw inspiration from work done in psychology [**?**], the philosophy of mathematics [**?**] and mathematics education [**?**].

Diagrams as Constrained Optimization Problems. To address all of the challenges outlined above we adopt a unifying paradigm: the compilation target for a PENROSE program (consisting of SUBSTANCE, STYLE, and ELEMENT code) is not a final binary executable, but rather a *constrained optimization problem* that can be subsequently solved via any sufficiently capable numerical optimization package. For instance, the logical relationship "A contains B" might get mapped to the constraint "disk A covers disk B", or alternatively, "node B is below node A" (see Figure 4 for an example), statements that must be resolved by constructing coordinate values for a specific diagram instance that satisfies these relationships. This distinction is important: a Penrose diagram is not a single static image, but rather a collection of constraints and objectives that together determine the allowable arrangement of graphical objects on a canvas.

Differentiable Diagrams. Modern optimization algorithms typically require derivatives of constraint and objective functions in order to evaluate the update direction (currently, for instance, we use an *exterior point method* [**?**], which requires first derivatives). Efficient evaluation of derivatives is challenging, since the objective function for a complex diagram can be extremely complex, involving a large heterogeneous collection of terms and perhaps even calls to external libraries. On the other hand, for design exploration we do not need to exactly evaluate derivatives, especially when far from the solution. We will therefore explore how to efficiently approximate derivatives in both space and in time, by replacing complex geometry with a progressively finer proxy (*e.g.*, via a *bounding volume hierarchy*) as we approach the final solution. This strategy effectively captures the strategy of human illustrators, who "rough out" a diagram before making low-level adjustments to spacing, kerning, *etc.*. Given the heterogeneous nature of our objectives, we will also explore how program introspection (*e.g.*, monitoring of derivative magnitudes) can be used to guard against difficult objective terms—just as a human illustrator might intermittently omit a problematic piece of a figure while trying to establish an initial global layout.

Scheduled Optimization. Since deterministic calculations may depend on the output of an optimization problem (and vice versa), we must infer a *schedule* for optimization from dependencies in the given program. To aid in this process, each graphical attribute in a STYLE program is internally tagged as either "varying" or "fixed." Initially all variables are "varying"; any attribute whose value is determined by a fixed constant or the output value of a deterministic computation is then tagged as "fixed." (These tags can be inferred from a dependency graph, and do not need to be explicitly specified by the user.) All remaining varying variables become the independent degrees of freedom during optimization. Deterministic computation can be implemented in the native Style language, or (in more specialized scenarios) through an interface with the host language, so that existing "black-box" visualization software may be integrated with PENROSE.

Local Approximation. For complex diagrams, even high-performance optimization may not be sufficient to facilitate real time interaction. Here we will explore strategies for local approximation and caching that allow users to quickly *approximate* nearby diagrams; once a rough solution is found, the optimizer can be used to hone in on the final solution. The choice of approximation scheme must be adapted to the particular modality of interaction; for instance, working with linear extrapolation or a low-dimensional eigenbasis (*à la* [**?**]) may be suitable for manipulation of individual elements, whereas interpolation of scattered samples (*à la*) may be more suitable for exploring more global design trade offs.

Geometric Predicates. Generation of any diagram demands repeated evaluation of *geometric predicates*, *i.e.*, queries like "is A inside B?" or "where do X and Y intersect?" Descent-based optimization techniques also require *derivatives* of basic geometric quantities, such as the distance between two shapes. Moreover, unlike many problems in, *e.g.*, computational geometry or computational physics which may only need to identify and resolve interactions between shapes at isolated points, diagramming often involves making judgements about overall, *integrated* quantities, *e.g.*, "how *much* do two shapes overlap?" or "what's the *average* distance between two shapes?" Together with the fact that we wish to facilitate real-time interaction, these requirements demand intelligent strategies for quickly approximating geometric predicates and their derivatives. For simple shapes, such as circles or lines, such predicates can often be evaluated accurately and efficiently using special-purpose algorithms from computational geometry [**?**]. In general, however, objects appearing in diagrams can be highly complex (*e.g.*, curved Bézier paths, font glyphs, or 2D projections of 3D polyhedra), or may lack an analytical description altogether (*e.g.*, raster images rather than vector graphics). We will explore how different representations of graphical primitives facilitate effective energy and derivative evaluation; in particular, we have already started exploring a hybrid strategy involving (i) polygonal approximation of complex curves; (ii) hierarchical *bounding volume* approximation of complex shape arrangements, and (iii) Eulerian *level set representations* [**?**] for raster graphics. An important feature of all these representations is that they have a natural *multi-resolution* interpretation, *i.e.*, the accuracy of the approximation can be easily adapted according to (say) the proximity between two shapes, or the stage of optimization (*e.g.*, initial layout vs. final refinement). To support optimization of common geometric predicates (containment, intersection, mutual distance, *etc.*) as well as more sophisticated geometric predicates (such as tangency or transversal intersection) we must also augment these representations with fast evaluation of approximate derivatives, or integrals of these derivatives over regions of interest. Here we can take advantage of a powerful idea from differential geometry called *Stokes' theorem* [**?**] to significantly reduce the dimensionality of such calculations.

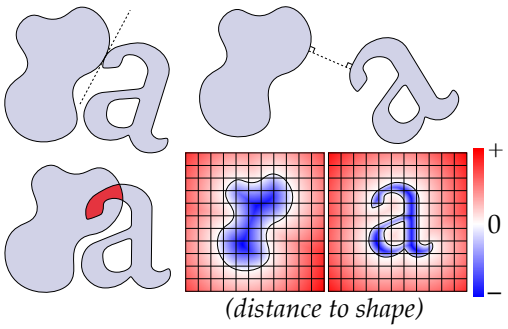


Figure 6: *To optimize diagrams, we must repeatedly evaluate sophisticated geometric predicates (tangency, intersection, closest point,* etc.*) and their derivatives—even for intricate, user-specified primitives. Signed distance functions (bottom right) sampled onto a regular grid provide an efficient, unified solution.*

Symmetry and Invariance. To the degree that it is possible, our system should produce a warning when a user-defined visual style does not faithfully represent the abstract object it is meant to illustrate—a simple example is shown in Figure 7. How can we automatically detect this kind of error? One way is to check that *invariants* of the abstract object are preserved by *symmetries* of its visual representation. For instance, in the case of Figure 7, the function mapping the two points a, b to the vector v is *antisymmetric* in its two arguments: reversing the order will flip the sign of the resulting vector. In contrast, two of the candidate visual representations (a line segment with either zero or two arrowheads) exhibits *symmetry* rather than antisymmetry when endpoints are exchanged, *i.e.*, the graphical primitive remains the same. Though it im-

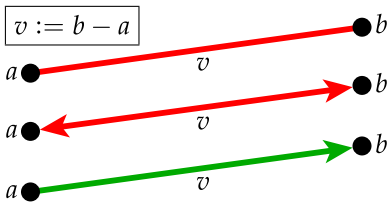


Figure 7: *A poorly chosen graphical representation may betray basic properties of an abstract object. As an elementary example, a directed vector v is poorly represented by a line segment or a segment with two arrowheads, but well-represented via a single arrowhead. We can check for such errors by comparing geometric symmetries to program invariants.*

possible to guarantee that no such error will occur, we can guard against many common errors by tagging each atomic graphical primitive with its associated symmetries (reflectional, rotational, *etc.*) and verifying their correspondence (or non-correspondence) with basic program invariants, such as order of arguments. This strategy can be further enhanced by inferring the symmetry of composite graphical primitives from the spatial transformations used to construct them (*e.g.*, two orthogonal copies of a double-headed arrow will exhibit additional rotational symmetry). For user-defined graphical objects that do not have a clear syntactic expression (*e.g.*, bitmap images rather than SVG primitives), we can apply tools from 3D geometry processing to automatically detect symmetries [**?**, **?**, **?**].

Avoiding & Promoting Special Configurations. How can one avoid generating diagrams that are not representative of the typical case, or conversely, how can one automatically identify interesting examples? Such examples are pervasive in mathematical writing, since both the general case *and* important special cases help to build intuition about a given object (Figure 8).

• **(General Position)** In computational geometry, the idea of "special cases" is formalized via the notion of *general position*—loosely speaking, an arrangement of geometric primitives is in general position if any small perturbation of the configuration would change the status of a binary predicate such as "two circles are tangent" or "three points are on a common line." Since PENROSE is an extensible language, the full set of predicates cannot be known *a priori*—instead, we will allow a STYLE programmer to define what it means for a diagram to be generic by providing *constraints that should be **far** from being satisfied*. PENROSE will then automatically transform such constraints into objective functions that discourage non-generic configurations. For instance, given a constraint equation of the form $f(x) = 0$, PENROSE might add an objective term $\frac{1}{|f(x)|^2}$ which encourages the residual of the constraint equation to be *large* rather than small. However, such constraints need not be expressed in terms of low-level mathematical expressions—for instance, to discourage three points a, b, c from sitting on a common line, a STYLE programmer might simply write, `avoid collinear(a,b,c)`. (More specialized relationships will inevitably have to be coded, though such functions can themselves be subsequently incorporated into reusable and redistributable libraries.)

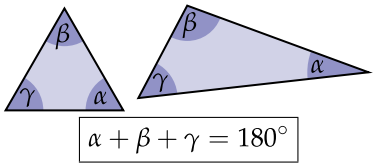


Figure 8: *When illustrating a statement like "the angles in any triangle sum to* 180°*," a highly symmetric example (left) may mislead the reader into believing the statement does not hold in the general case (right). This proposal develops strategies for automatically avoiding misleading examples.*

• **(Special Cases)** In many situations, special configurations of geometric objects occur precisely at the moment where constraints exhibit "abnormal" behavior, *e.g.*, when (i) the constraints themselves fail to be differentiable, or when (ii) the constraint derivative fails to be *regular*. We will develop *universal strategies* for automatically discouraging or encouraging such special cases based on analytical properties of the constraint function and its derivatives, since these properties can be automatically inferred from the program and do not require special knowledge from the user. As a concrete example, consider a function $f : \mathbb{R}^m \to \mathbb{R}^n$ which defines the constraints in an optimization problem via the condition $f(x) = 0$. The *constraint Jacobian* is simply the collection of all partial derivatives of this function, which can be encoded as an $m \times n$ matrix J. Points where J is *(row) rank-degenerate* tend to correspond to configurations that exhibit interesting "symmetries" (in a very general sense), since at

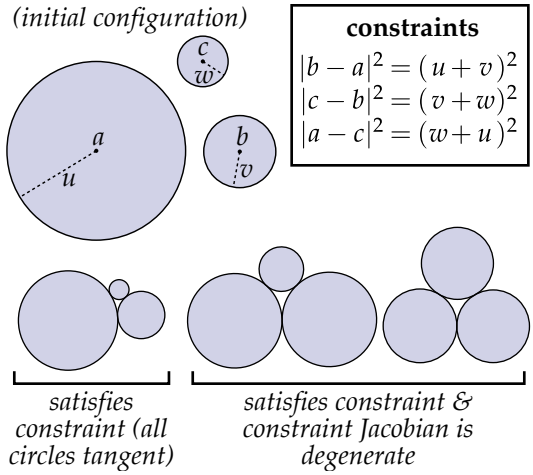


Figure 9: *Representing abstract objects as diagrams also helps one* identify *interesting special cases. Here we prototype a general strategy for finding diagrams with complete or partial symmetry (right) by minimizing the rank of the* constraint Jacobian.

these points there will be two equal and opposite directions along which the configuration can change without violating the constraint. One can therefore encourage or discourage special cases by optimizing the eigenvalues λ_i of the matrix $A := JJ^T$, which quantify *how degenerate* the Jacobian is. Since the gradient be expressed generically as $2e^T J' J^T e$ (where $Ae = \lambda e$), the system needs only second derivatives of the constraint functions in order to identify special configurations of *any* diagram, *i.e.*, **no domain-specific knowledge is required**. Such derivatives can either be supplied by the programmer, or automatically computed via numerical or automatic differentiation. In Figure 9 we implement this strategy for a simple toy example where three circles are constrained to be tangent; by minimizing one or more eigenvalues of A we get diagrams with an increasing degree of symmetry.

• **(Generating Diverse Examples)** Many different diagrams can be used to illustrate the same logical or mathematical idea—how can we be sure we've represented the full landscape of possibilities? For sufficiently small diagrams one can exhaustively enumerate the possibilities: for instance, in Figure 10 we use PENROSE to generate all possible classes of visualizations for the statement "B and C are subsets of A" (where A, B, and C are generic sets); to get a diagram in each class we simply enumerate the complementary containment relations (*e.g.*, "C is a subset of B," "B is not a subset of C," *etc.*) and use constrained optimization to generate an instance from each class. For larger diagrams, even just *enumerating* all possible constraints becomes prohibitively expensive, yet here we can still develop strategies that efficiently generate a diverse collection of examples. The first challenge is obtaining a sampling of the constraint space that provides good "coverage" of the constraint space. The second challenge is classifying samples into distinct categories. We will address the first challenge by applying tools from sampling theory, namely *Markov chain Monte Carlo methods*, which seek to spread samples uniformly over a domain while ensuring that sampling is "ergodic," *i.e.*, that every state can potentially be reached. The second challenge will be addressed by applying tools from *computational topology*—in particular, we can say that two examples are "equivalent" if the diagrams representing these examples are *homotopic*, *i.e.*, if there exists a continuous interpolation of diagram parameters that at each moment respects our constraints. A simple but often effective heuristic for finding such a path (used in computational geometry [**?**]) is to start with a linear path, which is then recursively subdivided while projecting each point onto the constraint set. By simultaneously optimizing the *length* of this path, one also obtains a smooth trajectory suitable for producing animated diagrams. Clusters of diagrams that are mutually homotopic then provide distinct examples that can be presented to the user. In this way we get a relationship between [equivalence classes of mathematical objects] and [homotopy classes of diagrams] (the former being coarser than the latter); since we can reason about the latter algorithmically, we obtain a very general strategy to help automatically *classify* instances of the object itself—a task that typically must be carried out "by hand" by domain experts.

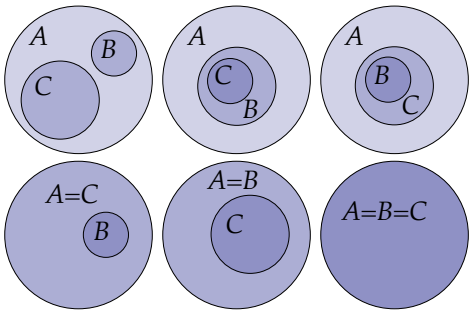


Figure 10: *Even for a statement as simple as "$B, C \subset A$" there are numerous possible diagrams (here enumerated by our* PENROSE*prototype). We will develop strategies for automatically generating and selecting a small representative set of diagrams.*

Graphical Interfaces. Finally, although the basic premise of PENROSE is that users should *not* require graphical expertise in order to produce high-quality diagrams, a rich array of intuitive graphical tools can nonetheless be built on top of the core language-based platform. We will explore the use of PENROSE in a graphical environment, targeting the list of use cases given at the beginning of this section. One prototype example, in an educational context, is already shown in Figure 11.

5 EVALUATION

PENROSE is a domain-agnostic platform and therefore by its nature abstract. We also have ambitious adoption goals:we expect PENROSE to be used by students, educators, illustrators, textbook authors, mathematicians, scientists, and programmers. The quality attributes we expect of it are therefore multi-layered and multi-faceted. We focus our evaluation on five attributes that we believe are crucial: *needs fitness, productivity, diagram quality, code quality, expressiveness.*

Needs fitness. We want Penrose to fill a real-world need. In particular, we want to know: 1) is there demand for a mathematical diagramming tool? 2) what are the crucial features of such a tool? Since this needs assessment is crucial we've already completed this part of the evaluation.

To assess the demand, we analyzed the 22,170 papers published on arXiv in September 2018. We found that only 75% of these papes contained at least one graphical figure. However, only 39% of mathematics papers contain figures and the papers that contain figures, average less than half as many as non-mathematical papers. This indicates a real need to assist mathematicians in making visualizations for their papers. In addition, our very preliminary project has already accumulated over 900 email list subscribers, 800 stars on *GitHub*, and appeared as the 2nd-most upvoted story on Hacker News. We consider this interest evidence of real-world need for tools that provide rapid translation of logical statements into visual diagrams.

To better understand *what* we should design we conducted an interview study of 24 frequent diagrammers from a diverse pool of expert diagrammers. The participants were mathematicians, textbook authors, computer scientists, data/science journalists, and teachers. This interview questions were broadly focused on the participants diagramming experience and goals with particular emphasis on the strenghts and weaknesses of the tools they currently used. PENROSE and its design were not mentioned, to avoid biasing participants toward PENROSE's features.

Five themes emerged from the interviews. Each of these ideas were brought up by at least 5 participants organically and were not in response to a leading question. Each theme is connected to one of the design decisions we made in PENROSE:

1. Participants reuse representation and stylistic details across many diagrams in the same domain. *Design implication*: STYLE libraries can be shared and imported.
2. Participants often need to make the same conceptual change to every diagram in a collection. They are frustrated with this process in existing tools. *Design implication*: SUBSTANCE and STYLE are seperate languages. A single change in a STYLE program can be applied automatically to anySUBSTANCE program in the relevant domain.
3. Participants dont like the tedium involved in dealing with existing programmatic tools like TikZ. *Design implication*: PENROSE programs are written at a much higher-level of abstraction. Many details can be omitted by the programmer and optimized by the system.
4. Participants often speak about mapping from abstract objects to visual/graphical elements (e.g. vector→arrow, set→circle). *Design implication*: The pattern matching in our STYLE programs performs exactly this mapping and should therefore feel natural to our participants **[?]**.
5. Many participants sketch early drafts of diagrams by hand primarily in order to experiment with alternative layouts of the graphical objects in their diagrams. *Design implicatoin*: These participants used their brains, plus paper and pen to optimize the layout. PENROSE performs this optimization automatically. It can also present multiple alternative layouts for the same mathematical notation.

We plan to follow up this interview study with a widely-distributed survey to gather more quantitative data. We also expect to continue interviewing experts throughout the design of Penrose to ensure that its design is grounded in real-world needs.

Productivity. We want PENROSE users to be able to create diagrams in our system *efficiently*, relative to standard tools such as *TikZ*, *Adobe Illustrator*, or *Mathematica*. To be successful our system will need to be both easily learnable and usable. We will therefore employ human-computer interaction (HCI) techniques heavily in both the refinement and evaluation of PENROSE's design.

During this refinement process we will periodically ask one of CMU's many HCI experts to provide expert, systematic usability advice using Nielsen's heuristic evaluation criteria [**?**] or *Cognitive Dimensions of Notations* [**?**]. In addition, we will conduct design exercises with potential users. In particular, since PENROSE is a programming system, we will employ the *natural programming* methodology [**?**] to understand how people think about visual programming tasks. We will then use this knowledge as guidance on how to design PENROSE's syntax and semantics so they are as *natural* to our users. We have successfully used this methodology to help design the *Obsidian* domain-specific language for blockchain programming [**?**].

To evaluate the productivity of PENROSE we will conduct controlled user experiments using traditional designs [**?**, **?**]. We have recently conducted controlled experiments to evaluate the *Plaid* and *Glacier* programming systems [**?**, **?**, **?**, **?**]. We will consider both novice and expert users, asking them to reproduce existing visualizations using both our tool and standard tools such as *TikZ*, *Adobe Illustrator*, or *Mathematica*. The primary output measure will be time on task, but we will also collect many qualitative measures like affinity for the tools, understanding of diagram semantics, and learning barriers.

Diagram quality. To achieve its mission, PENROSE needs to be capable of creating beautiful and useful diagrams. Therefore, we will perform a series of formal case studies in which we will recreate diagrams created by professional illustrators and compare the results [**?**]. Here we will use established textbooks, such as *Visual Group Theory* [**?**], as a baseline, and conduct a "blind taste test" using a platform like *Amazon Mechanical Turk* to evaluate both aesthetic and informative qualities of generated diagrams. In addition, we will heuristically evaluate the output using well-known design principles [**?**].

It is important to note that PENROSE is a platform for creating diagrams and the diagrams it outputs can only be as good as the inputted STYLE programs. Our evaluation will therefore compare PENROSE at its best to professional illustrators at their best.

Code quality. PENROSE has a programmatic interface and the languages it employs are domain-specific. The quality of the code that is produced using PENROSE will have significant impact on the maintainability and extensibility of PENROSE diagrams. We will evaluate PENROSE's code quality using the same case study discussed above to evaluate diagram quality. Since we are evaluating code quality and diagram quality simultaneously this ensures that we cannot maximize one at the expense of the other [**?**]. The textbook we choose will have programmatically created diagrams to provide an avenue for comparison. We will use lines of code and traditional code complexity metrics as crude quantitative measures of quality. We will qualitatively evaluate the code for saliency of the mathematical semantics, saliency of the visual design, and abstraction level. In addition, we will conduct simple debugging and extension exercises on both codebases to evaluate maintainability.

We intend to recruit expert mathematical illustrators to provide feedback on the PENROSE output. This third-party feedback will reduce bias in the evaluation. For example, these illustrators could compare code-writing product and process as compared to the *TikZ* code they traditionally use. They can provide qualitative feedback like "the PENROSE SUBSTANCE code looks concise" or "I would like to write styles like

this, not like that" or "it doesn't look like you can reproduce this specific kind of diagram." These experts can also provide feedback on the diagram quality (see above). Nathan Carter, author of *Visual Group Theory* [**?**], has agreed to provide this feedback (see attached letter of collaboration) and we will continue to consult other textbook authors and mathematical illustrators.

Expressiveness. To gauge the expressiveness of PENROSE we will develop STYLE programs for a wide variety of mathematical diagrams. To ensure breadth, we will select diagrams based on a well-developed taxonomy of mathematics or mathematical diagrams. For example, we might create two diagrams from each subcategory of Wikimedia's mathematical diagrams category [**?**]. Alternatively, we might create five diagrams from a popular college textbook in each of the ten "core subjects" in the Mathematics Association of America's Core Subject Taxonomy for Mathematical Sciences Education [**?**]. Does our architecture allow, in principle, all kinds of diagrams in this taxonomy to be created? Or did we make a blunder that prevents certain kinds of diagrams? The extensibility of our system will be crucial to this evaluation since we will need to make use of external tools to derive certain diagrams.

---

6 BROADER IMPACTS—EDUCATION, OUTREACH, AND MENTORING

6.1 EDUCATION

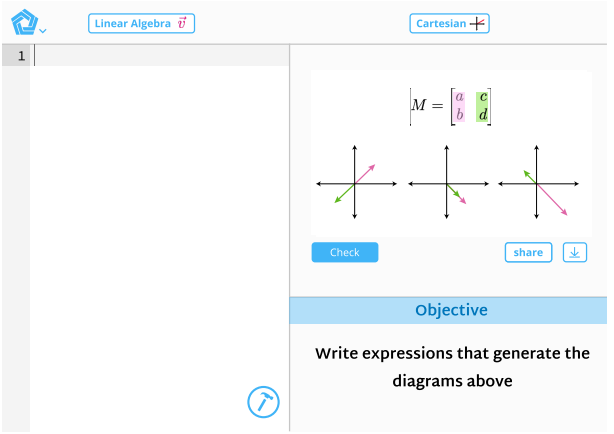
Abstract mathematical and computational thinking is extremely valuable across many domains, including some of the fastest growing and most societally important. However, the vast majority of students never learn these concepts and find the relevant courses intimidating. We aim to apply lessons from learning science combined with the technology we are building into Penrose to alleviate this situation. In particular, we intent to automatically generate visual practice problems to enhance learning.

The highly-cited Knowledge-Learning-Instruction (KLI) framework provides research-based principles to guide instruction decision making [**?**]. The framework suggests that concept learning can be effectively catalyzed with practice classifying, identifying, or generating examples and non-examples of the concept. In many cases, a visual representation of a concept is simpler and more intuitive for a learner [**?**]. PENROSE can easily generate examples of abstract concepts, as specified by a SUBSTANCE program, and users can then classify these.

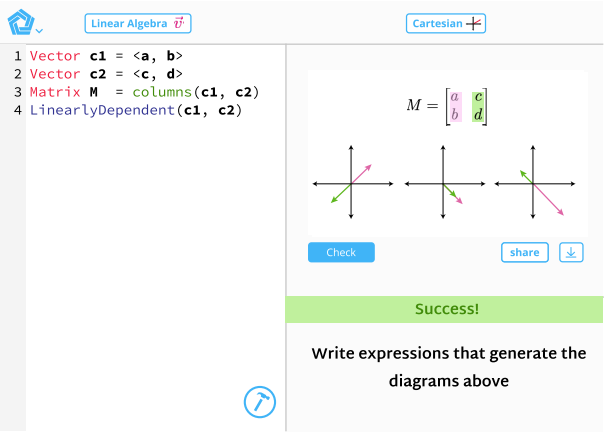
A recent meta-analysis of the learning science literature found that adding multimedia content, like diagrams, is the single most effective intervention to instruction. [**?**]. This lesson has registered with many producers of passive instructional materials (e.g. textbook chapters, slide decks, MOOC multimedia lessons). These often include both symbolic and diagrammatic representation to teach abstract concepts. However, the associated practice problems are rarely diagrammatic. For example, the Khan Academy videos on linear algebra includes some diagrams in the lessons, but most lessons include no practice problems at all, and only a small fraction of those few problems are visual in any way. PENROSE, as you might expect from a diagram generation platform, will always generate visual problems.

Finally, learning notation presents a significant barrier to learners in mathematical domains. For example, most novice students have an easier time answering story problems than even simple arithmetic equivalents (e.g. 7 = 2x - 3) [**?**]. Many experts forget or significantly underestimate this notation learning challenge because the notation, once learned, feels simple and intuitive [**?**], but the challenge is real. Since the SUBSTANCE language embodies notation, it can provide an avenue for deliberate notation practice [**?**].

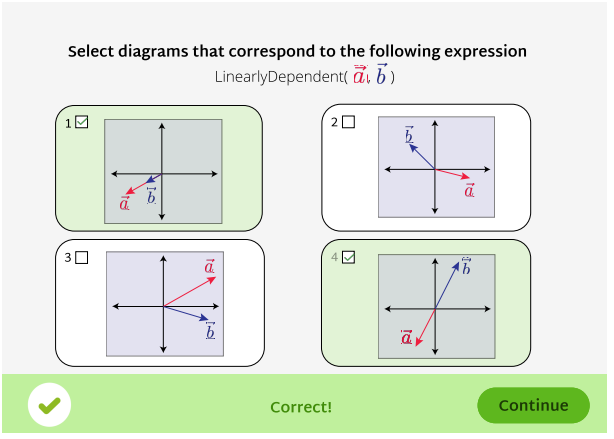
As part of this grant proposal, we propose to develop Penrose Tutor which generates problems to facilitate mathematical concept and notation learning. For example, imagine we are teaching the concept of linear independence from linear algebra. We can generate diagrams from a SUBSTANCE program that



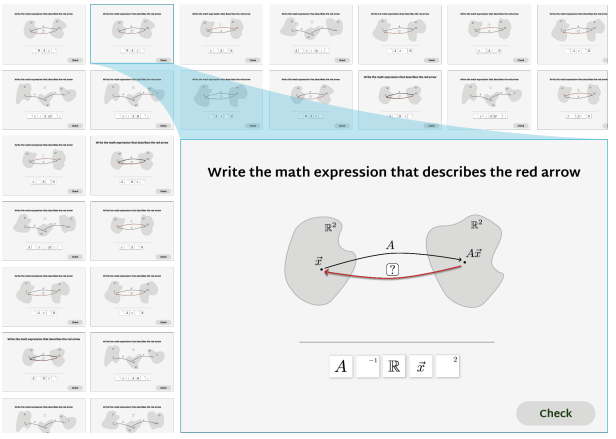
(a) *Free-response problem*



(b) *Succesfully completed problem (a)*



(c) *Diagramatic multiple-choice problem*



(d) *A student's problem is selected from generated problem instances.*

includes linear independence and we can ask the learner to write the notation that corresponds to the diagrams (Figures 11a and 11b). We can also generate diagrams that are examples and non-examples of linear independence and ask the learner to select the examples (Figure 11c). We can teach multiple concepts at the same time by asking the learner to match candidate SUBSTANCE programs to resulting diagrams. All of these problem types can be scaled up to more complex mathematical statements that are composed from multiple concepts (e.g. a single SUBSTANCE program that includes both linear independence and vector addition). These more complicated statements can be used to teach component concepts and practice fluency in context. This is analogous to the difference between practicing a sentence and practicing a word or phrase.

The problem types described above are enabled by the technologies underlying the PENROSE system. The SUBSTANCE program closely models real mathematical notation. The domain is defined by an ELEMENT program which defines all of the mathematical objects and the relationships among them. In preliminary work, we have developed a system to automatically generate pseudo-random SUBSTANCE programs from an input element program. We can then apply our STYLE library to these programs to generate diagrams that correspond to each random SUBSTANCE program, each of which corresponds to a mathematical statement. Each of these programs and diagrams can be used to automatically to produce all of the problem types described above. In this way, we can generate millions of problems for each domain defined in our system (Figure 11d). We can select problems that are appropriate to each learner based on the strengths and weaknesses they have exhibited on previous problems. We can evaluate student output either by comparing the student program to the program used to generate the diagram. In more complex cases with multiple correct solutions, we can use the optimization engine to determine whether the energy of the student solution is optimal.

6.2 OUTREACH

PENROSE is an open source project and already available for download and use on Github. During this course of this grant, we will develop a web interface for PENROSE to allow users to create diagrams and new visual styles without the need to download or deploy the system. The web interface will also allow users to store and share their STYLE programs to build and leverage PENROSE's user community (as does the current "bl.ocks" community for the *D3* visualization platform).

We have already corresponded with two textbook authors—Nathan Carter of *Visual Group Theory* [**?**] and Richard Zach, founder of the open-source, modular, collaboratively authored *Open Logic Project* [**?**]. We intend to use PENROSE to develop new diagrams for both of these books. This activity has two purposes: first, to evaluate the expressiveness, code quality, and diagram quality of PENROSE as discussed in Section 5. Second, have direct influence on the thousands of teachers and students who use these books. In addition, as part of our expressiveness evaluation, we plan to create diagrams for many domains of mathematics. In particular we will work with the Wikipedia community to explain mathematics to the massive user base of that online encyclopedia, and explore ways PENROSE might be integrated with Wikipedia (as TEX is currently).

---

7 RESULTS FROM PRIOR NSF SUPPORT

Jonathan Aldrich was the PI of NSF award CCF-1116907, "SHF:Small:Foundations of Permission-Based Object-Oriented Languages" ($500K, August 2011–July 2015). **Intellectual Merit:** This project developed a new programming language, Plaid, that incorporated permissions natively into the type system and runtime of the language. Permissions are type-like annotations that specify how an object may be aliased, and how the various aliases may use the object. Two applications of permissions were demonstrated: 1) the ability to safely change the interface, behavior, and representation of an object at run time and 2) automatic parallelization. This work resulted in 10 major conference or journal publications [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. **Broader Impacts:** This grant contributed to the education of 7 Ph.D. students, including 2 women, in

addition to several undergraduate and masters students. The grant also supported the development and release of the open-source languages and tools Plural and Plaid [**?**, **?**]. Plural was used to teach typestate concepts in courses at Carnegie Mellon.

Keenan Crane is the PI of NSF award #1717320, "RI: Small: Collaborative Research: Computational Representations for Design and Fabrication of Developable Surfaces" ($250K, August 1, 2017–July 31, 2020). **Intellectual Merit:** This grant studies fundamental representations and algorithms for the geometry of flat sheet materials (like plywood or sheet metal) that bend without stretching, using methods from the area of *discrete differential geometry (DDG)*. Since the start of the funding period (approximately three months ago) we have made significant developments toward (i) a new variational method for cutting curved surfaces into flat pieces while minimizing material stress; (ii) a geometric flow that approximates a given curved surface by one that is piecewise developable (suitable, *e.g.*, for 5-axis milling), and (iii) a new technique for "programming" curvature into flat material via special patterns that can be fabricated via laser cutter; these three results are currently being prepared for submission. **Broader Impacts:** During the award period the PI has carried out several of the proposed BI activities including (i) development of an open source JavaScript framework for 3D geometry processing (which can be found in the `geometry-processing-js` repository on *GitHub*); mentoring four undergraduates in research, including one female student and one underrepresented minority; development of new course material in DDG; organizing an AMS Short Course on DDG, and closely related to this proposal, co-organizing a workshop on mathematical visualization. This grant also supported the PI's work an overview article on DDG [**?**] recently published in the *Notices of the American Mathematical Society*, which is the world's most widely-read mathematics journal.

Joshua Sunshine is the PI of NSF award #1560137, "REU Site: Interdisciplinary Software Engineering" ($359K, February 2016–January 2019). **Intellectual Merit:** The students in the program are mentored by 14 faculty members in many research areas: program repair, software architecture, self-adaptive systems, requirements engineering, software ecosystems, collaborative environments, API usability, program analysis, variational computing, usable security engineering, programming language design and many more. They have been awarded 12 prizes in ACM student research competitions. The total publications are too numerous to list here, but three papers directly involved the PI [**?**, **?**, **?**]. **Broader Impacts:** This REU site has broadened access to computing research. REUSE focuses on broadening participation in computing research including female and minority groups underrepresented in computing (URMs). Since the program's inception, the program has include 65 students, including 35 female students and 20 URM students.