Penrose: From Mathematical Notation to Beautiful Diagrams

KATHERINE YE, Carnegie Mellon University
WODE NI, Carnegie Mellon University
MAX KRIEGER, Carnegie Mellon University
DOR MA'AYAN, Technion and Carnegie Mellon University
JENNA WISE, Carnegie Mellon University
JONATHAN ALDRICH, Carnegie Mellon University
JOSHUA SUNSHINE, Carnegie Mellon University
KEENAN CRANE, Carnegie Mellon University

```
Point p, q, r, s
Segment a := {p, q}
Segment b := {p, r}
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := {p, r, s}
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

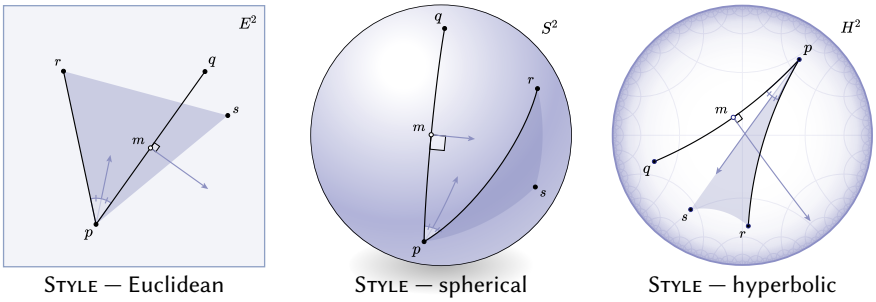


Fig. 1. Penrose is a framework for specifying how mathematical statements should be interpreted as visual diagrams. A clean separation between abstract mathematical objects and their visual representation provides new capabilities beyond existing code- or GUI-based tools. Here, for instance, the same set of statements *(left)* is given three different visual interpretations *(right)*, via Euclidean, spherical, and hyperbolic geometry. (Further samples are shown in Fig. 29.)

We introduce a system called Penrose for creating mathematical diagrams. Its basic functionality is to translate abstract statements written in familiar math-like notation into one or more possible visual representations. Rather than rely on a fixed library of visualization tools, the visual representation is user-defined in a constraint-based specification language; diagrams are then generated automatically via constrained numerical optimization. The system is user-extensible to many domains of mathematics, and is fast enough for iterative design exploration. In contrast to tools that specify diagrams via direct manipulation or low-level graphics programming, Penrose enables rapid creation and exploration of diagrams that faithfully preserve the underlying mathematical meaning. We demonstrate the effectiveness and generality of the system by showing how it can be used to illustrate a diverse set of concepts from mathematics and computer graphics.

CCS Concepts: • **Human-centered computing** → **Visualization toolkits**; • **Software and its engineering** → **Domain specific languages**.

Additional Key Words and Phrases: mathematical diagrams

Authors' addresses: Katherine Ye, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213; Wode Ni, Carnegie Mellon University; Max Krieger, Carnegie Mellon University; Dor Ma'ayan, Technion and Carnegie Mellon University, Haifa, Israel; Jenna Wise, Carnegie Mellon University; Jonathan Aldrich, Carnegie Mellon University; Joshua Sunshine, Carnegie Mellon University; Keenan Crane, Carnegie Mellon University.

0730-0301/2020/7-ART144
https://doi.org/10.1145/3386569.3392375

ACM Reference Format:
Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Trans. Graph.* 39, 4, Article 144 (July 2020), 16 pages. https://doi.org/10.1145/3386569.3392375

1 INTRODUCTION

"Mathematicians usually have fewer and poorer figures in their papers and books than in their heads."
—William Thurston

Effective communication of mathematical ideas is a major challenge for students, educators, and researchers. Though modern mathematics has a strong cultural bias toward formal language [Mashaal 2006], visualization and illustration undoubtedly have an equally profound impact on mathematical progress [Thurston 1998]. Yet the ability to translate abstract concepts into concrete illustrations is often limited to the select few individuals who have both a deep understanding of mathematics and an intimate knowledge of graphical tools. As a result, diagrams are rather scarce in mathematical writing—for instance, recent mathematical papers from *arXiv* have on average only one figure every ten pages. A central goal of this work is to lower the barrier to turning mathematical ideas into effective, high-quality visual diagrams. In the same way that TEX and LATEX have democratized mathematical writing by algorithmically codifying best practices of professional typesetters [Beeton and Palais 2016], Penrose aims to codify best practices of mathematical illustrators into a format that is reusable and broadly accessible.

Our approach is rooted in the natural separation in mathematics between abstract definitions and concrete representations. In particular, the Penrose system is centered around the specification of a *mapping* from mathematical objects to visual icons (Sec. 2). Such mappings are expressed via domain-specific languages (DSLs) that reflect familiar mathematical notation and can be applied to obtain new capabilities that are difficult to achieve using existing tools (Sec. 2.4). A key distinction is that Penrose programs encode a *family* of possible visualizations, rather than one specific diagram. Hence, effort put into diagramming can easily be reused, modified, and generalized. This approach has several broad-reaching benefits:

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

Beyond describing the implementation of Penrose, the purpose of this paper is to explore the general challenge of designing systems for diagram generation. We hence start with a discussion of goals and trade-offs that inform our system design (Sec. 2). Readers may also find it helpful to periodically refer to the more detailed but purely descriptive account of the system given in Sec. 3 and Sec. 4.

2 SYSTEM DESIGN

"One must be able to say at all times—instead of points, straight lines, and planes—tables, chairs, and beer mugs."
—David Hilbert

Our aim is to build a system for converting abstract mathematical ideas into visual diagrams. Choices about system design are guided by several specific goals, many of which are supported by interviews we did with users of diagramming tools [Ma'ayan et al. 2020]:

(1) Mathematical objects should be expressed in a familiar way.
(2) The system should not be limited to a fixed set of domains.
(3) It should be possible to apply many different visualizations to the same mathematical content.
(4) There should be no inherent limit to visual sophistication.
(5) It should be fast enough to facilitate an iterative workflow.
(6) Effort spent on diagramming should be generalizable and reusable.

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

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

Just as the occupant of Searle's "Chinese room" does not actually understand a foreign language [Cole 2014], a system designed this way need not perform deep reasoning about mathematics—it simply does a translation. We hence do not expect our system to solve all challenges of diagramming. Users are still responsible for, say,

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

Likewise, a system cannot be expected to solve hard computational or mathematical problems (*e.g.*, the halting problem or Fermat's last theorem) in order to construct diagrams. Yet despite this shallow level of reasoning, Penrose is able to generate quite sophisticated diagrams. In fact, even in the absence of such reasoning, naïve visualization often provides useful observations (Fig. 2).

The resulting system effectively models diagram generation as a compilation process, where the compilation target is a constrained optimization problem rather than (say) a binary executable or a static image. Once compiled, this problem can be used and *reused* to generate visual diagrams; Fig. 3 illustrates the high-level system flow. From a programming language point of view, a mapping expressed in this framework defines an *executable visual semantics*. That is, it gives a specific, visual, and computable interpretation to what were previously just abstract logical relationships.

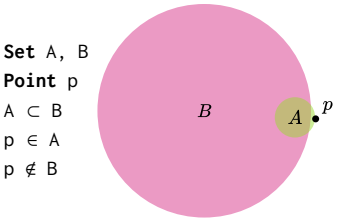


Fig. 2. An optimization-based approach has myriad benefits. Here a logically inconsistent program fails gracefully, providing visual intuition for *why* the given statements cannot hold.

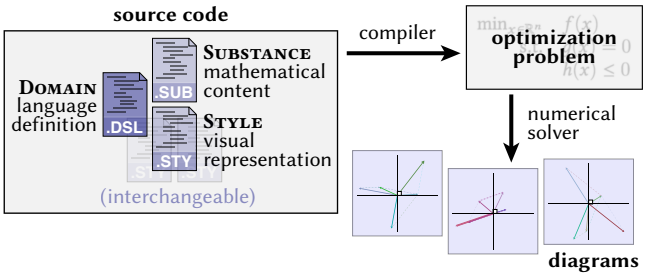


Fig. 3. High-level pipeline: a compiler translates mathematical statements and a chosen visual representation into a constrained optimization problem. This problem is then solved numerically to produce one or more diagrams.

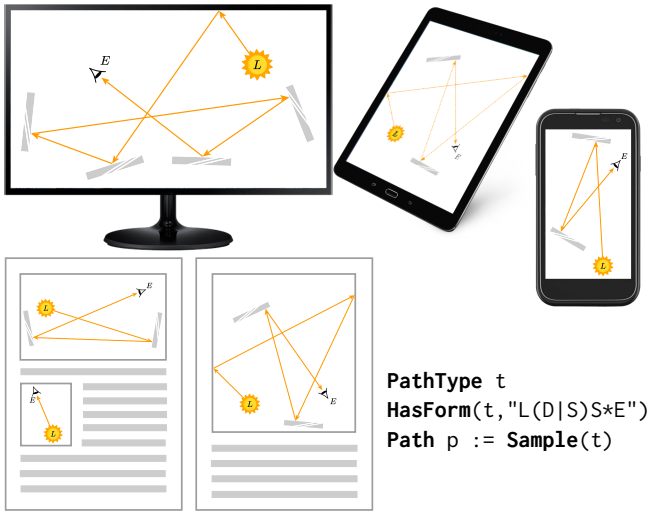


Fig. 4. By specifying diagrams in terms of abstract relationships rather than explicit graphical directives, they are easily adapted to a wide variety of use cases. Here we use identical Penrose code to generate ray tracing diagrams for several targets (Sec. 5.6). Though the arrangement and number of objects changes in each example, the meaning remains the same.

2.1 Language-Based Specification

A major decision in Penrose is to use programming languages to specify both mathematical objects (Sec. 2.1.2) and their visual representation (Sec. 2.1.3). Graphical (*e.g.*, sketch-based) specification would demand that users already know how to visualize abstract ideas, and it ties mathematical content to one specific visual representation, which conflicts with Goal 3. A language-based specification provides the level of abstraction needed to separate content from visualization. This technique supports Goal 1, since language is the most common means by which mathematical ideas are expressed. From a system design point of view, a language-based encoding provides a unified representation for identifying and transforming mathematical objects throughout the pipeline. Moreover, a language-based interface makes it easy for Penrose to provide a *platform* on which other diagramming tools can be built (as in Sec. 4.5 or Sec. 5.7). One trade-off is that a language-based approach requires users to express themselves in formal mathematical or computational language, making it more difficult for (say) visual artists and designers to contribute new representations.

A secondary decision is to split specification of mathematical content and visualization across two domain-specific languages: Substance and Style. A good analogy is the relationship between HTML [Berners-Lee and Connolly 1995], which specifies content,

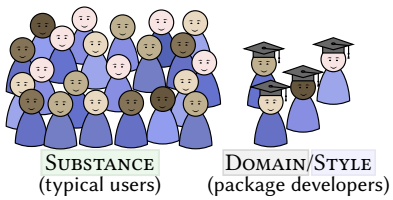


Fig. 5. Similar to the TEX ecosystem, most users need only use the Substance language, but can benefit from packages written by more expert Domain and Style programmers.

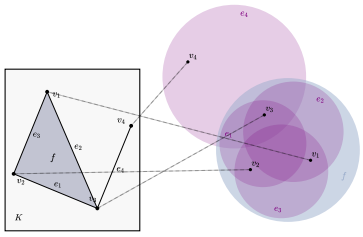


Fig. 6. One benefit of a unified framework is that different domains are easily combined. Here, two existing packages (for meshes and set theory) were combined to illustrate that a simplicial complex *(left)* is closed with respect to taking subsets *(right)*.

and CSS [Lie et al. 2005], which describes how it is rendered. A schema called Domain (analogous to XML or JSON schemas) defines the mathematical domain of interest, supporting Goal 2. In line with Goal 3, this division allows the same styles to be reused for different content, and likewise, the same content to be displayed in many different styles. Our goal is for this division to support an ecosystem where novice users can benefit from packages written by more experienced developers (Fig. 5). Finally, as in mathematics, the ability to adopt user-defined, domain-specific notation (Sec. 2.1.1) enables efficient expression of complex relationships in a way that is both concise and easy to understand [Kosar et al. 2012]. Encoding ideas directly in the idiom of a problem domain often results in better program comprehension than (say) a sequence of library calls in a general-purpose language [Van Deursen et al. 2000]. We discuss the scope and limitations of our languages in Sec. 6.

2.1.1 Mathematical Domain (Domain). One of our primary goals (Goal 2) is to build a unified system for diagramming, rather than to focus on specific domains (as in, say, *GraphViz* [Ellson et al. 2004] or *GroupExplorer* [Carter and Ellis 2019]). A unified design enables objects from different domains to coexist in the same diagram, often by doing little more than concatenating source files (Fig. 6). Moreover, effort put into (say) improving system performance or rendering quality is amortized across many different domains.

Users can work in any area of mathematics by writing so-called Domain schemas (Sec. 3.1) that define DSLs tailored to a given domain. This design also empowers users to adopt their own notation and conventions for modeling the domain. Use of domain- and user-specific notation reflects common practice in mathematical writing, where the meaning of a symbol is frequently overloaded depending on context. Importantly, a Domain schema is purely abstract: it does not define an internal representation for objects, nor does it give definitions to functions or predicates. This level of abstraction is crucial for Goal 3, since it allows multiple visual representations to later be applied to objects from the same domain.

2.1.2 Mathematical Content (Substance). To define the content of a diagram, one must be able to specify (i) the objects in the diagram, and (ii) relationships among these objects. In line with Goal 1, Substance uses concise assertions that resemble standard mathematical prose (see for example Fig. 7). Formally, it can model any domain expressible in a compositional language of types, functions, and predicates, which are the basic constructs found in conventional mathematical notation [Ganesalingam 2013]. Just as definitions are typically immutable in mathematics, Substance draws inspiration from strongly typed functional languages (such as *ML* [Milner et al. 1997]) where objects are stateless. This choice also simplifies system

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

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

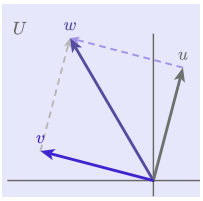


Fig. 7. Extensibility enables users to adopt conventions and notation *(center)* that reflect the way they naturally write mathematical prose *(left)*. Here, the resulting diagram *(right)* plays the role of the concluding statement.

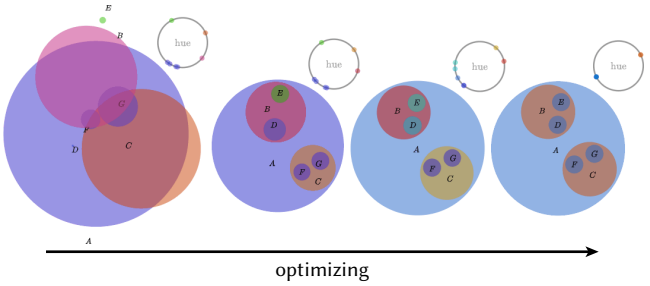


Fig. 8. An optimization-based approach makes it possible to jointly optimize visual attributes that are difficult to coordinate by hand. Here for instance we optimize color contrast according to the spatial proximity of adjacent disks *(left to right)*, ultimately discovering a two-color solution *(far right)*. The system can also be used to debug the optimization process itself—in this case by drawing the hue of each disk as a dot on a color wheel.

implementation, since the compiler can assume fixed definitions. A conscious design decision, in line with Goal 3, is to exclude all graphical data (coordinates, sizes, colors, *etc.*) from Substance—since its sole purpose is to specify *abstract relationships* rather than *quantitative data*. All such data is instead specified in Style or determined via optimization. This division relieves users from the burden of tedious and repetitive graphics programming, which can instead be factored out into reusable Style code.

Existing languages would be difficult to use in place of Substance since they lack the semantics needed to encode complex logical relationships and do not provide language-level extensibility. For instance, TEX [Beeton and Palais 2016] and *MathML* [Miner 2005] markup provide only enough information to translate plain text into mathematical glyphs; computer algebra systems like *Mathematica* and *Maple* have limited type systems or provide only a small set of fixed predicates (*e.g.*, asserting that a number is real). Conversely, the much richer languages used by automated theorem provers and proof assistants (such as *Coq* [Bertot and Castéran 2013] and *Lean* [de Moura et al. 2015]) are overkill for simply specifying diagrams. A custom language provides simple, familiar syntax and clear error messages. We do however adopt some ideas from *Coq*, such as the ability to customize syntactic sugar (Sec. 3.1).

2.1.3 Mathematical Visualization (Style). The meaning of a diagram is largely conveyed by relative relationships rather than absolute coordinates. Moreover, diagrams are often underconstrained: relationships needed to convey the intended meaning determine a *family* of possible solutions, rather than a single unique diagram. Style hence adopts a constraint-based approach to graphical specification in the spirit of *Sketchpad* [Sutherland 1964]: diagrams are built up from hard constraints that must be satisfied and soft penalties that are minimized (Sec. 3.3.6), then unspecified quantities are solved for via numerical optimization (Sec. 4). Though procedural definitions can still be used, the programmer need not provide absolute coordinates (as in imperative languages like *PostScript* or *OpenGL*). Though an implicit specification can make output hard to predict, part of the allure of Penrose is the potential to find interesting or surprising examples. Moreover, the approach yields more concise code; for instance, Style programs are much shorter than the SVG files they produce.

An alternative design might be to use an application programming interface (API), though historically APIs have been eschewed for specification languages for good reason. Language provides far more concise expression of complex relationships—imagine styling an entire website via the DOM API's `getElementById()` and `setStyle()` methods, versus a few short lines of CSS. Visual programming languages (like *LabView* [Elliott et al. 2007] or *Grasshopper* [McNeel et al. 2010]) might suffice for basic specifications (*e.g.*, vectors should be drawn as arrows), but don't scale to more complex concepts that are easily expressed via language [Burnett et al. 1995].

A key design challenge is identifying objects that appear in a Substance program. Objects in a given mathematical universe are distinguished not only by their type, but also by their relationships to other objects. A widely-used mechanism for specifying such relationships is through CSS-like *selectors*. Style adopts a similar mechanism that performs pattern matching on the types, functions, and predicates appearing in a Domain schema (Sec. 3.3.1).

2.2 Optimization-Based Synthesis

The second major design decision in Penrose is to use constrained optimization to synthesize diagrams satisfying a given specification (Sec. 4). This approach is again inspired by how people often draw diagrams by hand (*e.g.*, using GUI-based tools): visual icons are placed on a canvas and iteratively adjusted until no further improvements can be made. In difficult scenarios, a diagrammer may try several global arrangements before refining the final design, though typically no more than a few. Automating this process makes it easy to perform layout tasks that would be tedious by hand (Fig. 8).

There are good reasons to believe that an optimization-based approach can scale to very complex diagrams. First, attractive diagrams need not be optimal in a *global* sense—they should simply not permit obvious *local* improvements, such as text that could easily be moved closer to the item it labels. In fact, disparate local minima can provide useful examples that help build intuition (Fig. 9). Second, even sophisticated diagrams have surprisingly few degrees of freedom in comparison to many modern optimization problems (*e.g.*, tens or hundreds, versus thousands or millions). Finally, strategies employed by expert diagrammers can be applied to manage complexity, such as independently optimizing small components of a diagram (akin to nonlinear Gauss-Seidel), rather than optimizing all degrees of freedom simultaneously.

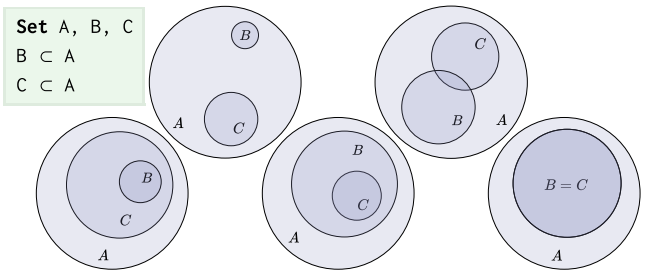


Fig. 9. A language-based design makes it easy to build tools on top of Penrose that provide additional power. Here we use standard techniques from program synthesis (Sec. 5.7) to automatically enumerate how the given relationships can be realized. Generating such examples helps to see important corner cases that might be missed when drawing diagrams by hand (where perhaps the top-left diagram most easily comes to mind).

In line with Goals 2 and 3, an optimization-based approach can be applied generically and automatically for any user-defined domain and visual representation, without requiring programmers to think about the details of the layout process. In our system, the optimization problem is defined using common-sense keywords (Sec. 3.3.6) in Style and chaining together basic operations (*e.g.*, arithmetic). Since the diagram specification is divorced from the details of the solver, optimization strategies can be changed and improved in future versions of the system while preserving compatibility with existing code. The main cost of an optimization-based approach is that it puts demands on system design "upstream": all expressions used to define a visual style must be differentiable. As discussed in Sec. 4.2, these requirements are largely satisfied via standard techniques (*e.g.* by using *automatic differentiation*).

In general, diagram optimization is a challenging problem in its own right, which we of course do not aim to solve conclusively in this paper. Currently, we just use a generic constrained descent solver (Sec. 4.2). However, we have been pleased to find that this simple approach handles a wide variety of examples from different domains without requiring domain-specific strategies.

2.3 Plugins

The final design decision in Penrose is to provide system-level extensibility via a *plugin* interface for calling external code in Substance and Style. Providing a plugin system is essential to enable users to integrate external code that is specialized to solve particular logical or graphical challenges. In fact, such interfaces for integrating external code are already provided by many systems (*e.g.*, TEX, *Adobe Illustrator*, and TikZ's plugin system for graph layout algorithms [Ellson et al. 2001]). The interface for Penrose plugins is designed to define a clear and simple boundary between the Penrose system and the plugin while enabling each component to focus on its strengths. A plugin can analyze and augment the set of abstract objects defined in Substance as well as analyze and augment the numerical information in Style. This simple interface allows plugins to be written in any language (or repurposed from other systems) and operate independently from the implementation details of Penrose. However, a plugin cannot change an existing Substance or Style program or directly generate static graphical content, since such plugins would abandon the benefits that Penrose provides, such as the ability to re-style content and avoid use of absolute coordinates. Fig. 4 illustrates how a simple plugin can make use of Substance and Style information to create "responsive" diagrams.

2.4 Related Systems

Here we consider how our system design relates to other systems that convert abstract mathematical ideas into visual diagrams. Other classes of tools, such as general-purpose drawing tools (*e.g.*, *Adobe Illustrator*) can also be used to make diagrams, though one quickly runs into barriers, such as for large-scale diagram generation or evolving the style of a large collection of existing diagrams. A broader discussion of related work can be found in a pilot study we did on how people use diagramming tools [Ma'ayan et al. 2020].

There are three main kinds of systems that convert an abstract form of input (*e.g.*, an equation or code) into a visual representation. Language-based systems, such as *TikZ* [Tantau [n. d.]] (which builds on TEX), are domain-agnostic and provide significant flexibility for the visual representation. Their use of "math-like" languages influenced the design of Substance. However, existing systems do not aim to separate mathematical content from visual representation. For instance, TikZ is domain- and representation-agnostic because it requires diagrams to be specified at a low level (*e.g.*, individual coordinates and styles) making programs hard to modify or reuse. Moreover, since there are only shallow mathematical semantics, it becomes hard to reason about programs at a domain level.

Plotting-based systems, like *Mathematica* and *GeoGebra* [Hohenwarter and Fuchs 2004] enable standard mathematical expressions to be used as input and automatically generate attractive diagrams. Just as a graphing calculator is easy to pick up and use for most students of mathematics, these tools inspired us to provide a "tiered" approach to Penrose that makes it accessible to users with less expertise in illustration (Fig. 5). However, much like a graphing calculator, the visual representations in these systems are largely "canned," and the set of easily accessible domains is largely fixed. For instance, *Mathematica* does not permit user-defined types, and to go beyond system-provided visualization tools, one must provide low-level directives (in the same spirit as tools like *TikZ*).

Finally, systems like *graphviz* [Ellson et al. 2004], and *Geometry Constructions Language* [Janičić 2006] translate familiar domain-specific language into high-quality diagrams. Here again, the domains are fairly narrow and there is little to no opportunity to expand the language or define new visualizations. Yet the convenience and power of such systems for their individual domains inspired us to build a system with greater extensibility. More broadly, while all these systems share some design goals with ours, a key distinction is that Penrose is designed from the ground up as an extensible *platform* for building diagramming tools, rather than a monolithic end-user tool.

3 LANGUAGE FRAMEWORK

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

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

A *package* consisting of a Domain, and one or more compatible Style programs, can be used to illustrate Substance programs from a given domain (Fig. 3). Though some starter packages are provided for the examples discussed in Sec. 5, the real power of Style and Domain is that they enable Penrose to be easily extended. In this section we illustrate these languages via the running example of a linear algebra package (Figures 10 through 12). Formal grammars for the three languages are given in supplemental material.

3.1 The Domain Schema

A Domain *schema* describes a domain of mathematics by defining the objects and notation that can be used by associated Substance and Style programs. A partial example for linear algebra is shown in Fig. 10 (the full schema is provided in supplemental material). The `type` lines define the available object types, `function` lines define the domain and codomain for the set of available functions (where * denotes a Cartesian product), and `predicate` lines define the possible relationships among objects, including unary predicates. Importantly, a Domain schema is purely abstract: it does not define a specific representation for objects, nor does it define bodies for functions or predicates. For instance, we do not say here that a vector is encoded by a list of coordinates, nor do we write an addition operation on such coordinates. A concrete visual interpretation of these definitions is given by a Style program (Sec. 3.3). Types can be given fields via *constructors*. For instance, the line

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

assigns fields `min` and `max` to an `Interval`, which can be accessed from a Substance or Style program (*e.g.*, to assert a relationship between endpoints). Subtyping via the syntax `Subtype <: Type` facilitates generic programming. Finally, `notation` lines define optional syntactic sugar that can simplify code (*e.g.*, in Fig. 11).

3.2 The Substance Language

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

Fig. 11 shows an example in which Substance code specifies properties and relationships for a pair of vectors. Importantly, these statements do not induce any kind of numerical evaluation. For instance, no coordinates are assigned to `x1` in order to make it unit—in fact, the vector space `X` does not even have an explicit dimension. Instead, statements specify persistent and purely symbolic relationships that provide cues for visualization; specific coordinates and attributes are later determined by the layout engine (Sec. 4). The

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

Fig. 10. A Domain schema specifies the building blocks available in a given mathematical domain, as well as any associated syntactic sugar. This schema (abbreviated) enumerates some basic constructs from linear algebra.

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

(unsugared)

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

(sugared)

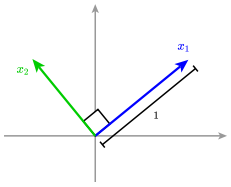


Fig. 11. When used with the Style defined in Fig. 12, this Substance code (with or without syntactic sugar) produces the diagram shown at right.

final lines specify label strings to be used by the Style program, here in TEX notation. Fig. 11, *center* shows a "sugared" version of this program using notation defined in the Domain schema (Fig. 10). Users can write programs either way, depending on the capabilities of their editor (*e.g.*, support for Unicode input).

3.3 The Style language

Rule

```
forall Type t {      Selector
  t.field = expression;
}                    Declaration
```

Style specifies how expressions in a Substance program are translated into graphical objects and relationships. It is a declarative specification language that shares many features with CSS. The core principle is to sketch out basic *rules* (*e.g.*, visual icons for basic types) and then refine these rules via *cascading* (Sec. 3.3.2). Each rule uses a *selector* to pattern match on objects and relationships appearing in Substance code (Sec. 3.3.1). A sequence of *declarations* then specifies a corresponding visualization, *e.g.*, by emitting graphical primitives or enforcing constraints. Each declaration either assigns a *value* to a *field* (Sections 3.3.5 and 3.3.3) or specifies a *constraint* or *objective* (Sec. 3.3.6). An example is shown in Fig. 12, which defines part of the style used for the Substance program in

Fig. 11 (a full Style is given in the supplemental material). We will use this example to highlight the basic features of the language.

3.3.1 Selectors. A selector uses pattern matching to specify which objects will be styled by a rule. Unlike regular expressions, selectors do not match literal strings of Substance code, but rather objects and relationships in the context defined by this code. A simple example is a selector that matches every instance of a type, indicated by the `forall` keyword. For instance, Line 1 matches all vector spaces. In subsequent declarations, `U` refers to the vector space `X` from the Substance program. The `where` keyword restricts matches to objects that satisfy one or more relationships; *e.g.*, Line 37 matches all pairs of orthogonal vectors. One can also match by name using backticks; *e.g.*, Line 48 matches only the vector `x2`. Selectors could be enriched in the future to allow other statements from first-order logic (such as ∃, disjunctions, and conjunctions).

3.3.2 Cascading. A cascading mechanism allows rules to be refined for more specialized objects or relationships. For example, the selector in Line 48 matches a specific vector, refining an earlier rule that applies to all vectors. Rule precedence is determined by order in the Style file, and later rules can refer to any previously defined *field* (Sec. 3.3.3). The `override` keyword (Line 49) hints that a rule will modify an existing field, otherwise the compiler issues a warning.

3.3.3 Fields. The visual representation of an object is specified by creating *fields* that are assigned *values* (Sec. 3.3.5). For instance, in Lines 17–23 a field called `u.arrow` is created and assigned an expression describing an arrow. Fields are created on assignment and can have any name not conflicting with a reserved word. Fields not naturally associated with a single object can also be assigned locally to a rule. For instance, Line 38 is used to draw a right angle mark between any pair of orthogonal vectors. Every object automatically has fields `name` and `label` storing its Substance name and label string (*resp.*), as well as any field created via a constructor (Sec. 3.1).

3.3.4 Properties. Style provides built-in graphical primitives (circle, arrow, *etc.*) with a fixed set of *properties*. Like fields, properties can be assigned values (as in Lines 38–44). If a value is not assigned, it will be assigned a default value, possibly a *pending value* (Sec. 3.3.5). For example, an arrow might be black by default, whereas the width of a box might be optimized (akin to *flexible space* in TEX).

3.3.5 Values and Expressions. Atomic *values* can be combined to form *expressions*. For instance, Lines 10–12 assign values, whereas Lines 6–9 assign composite expressions involving inline computation. Line 26 specifies value via a *path*, *i.e.*, a sequence of expressions separated by `.` characters; such assignments are made by reference. Expressions can also access values from plugins (Sec. 4.3). A very important construct is *pending values*, denoted by a `?` as in Line 20. This line specifies that the location of the arrow endpoint is not fixed and will be automatically determined by the solver (Sec. 4).

3.3.6 Constraints and Objectives. Constraints and objectives describe how pending values should behave. In particular, the `ensure` keyword defines a hard constraint that the diagram must satisfy. For instance, Line 45 specifies that two orthogonal vectors must be drawn at right angles. The `encourage` keyword specifies a relationship that should be satisfied as much as possible. For instance,

```
1  forall VectorSpace U {                -- LinearAlgebra.sty
2     U.originX = ? -- to be determined via optimization
3     U.originY = ? -- to be determined via optimization
4     U.origin = (U.originX, U.originY)
5     U.xAxis = Arrow { -- draw an arrow along the x-axis
6           startX : U.originX - 1
7           startY : U.originY
8           endX   : U.originX + 1
9           endY   : U.originY
10       thickness   : 1.5
11          style   : "solid"
12          color   : Colors.lightGray
13    } -- (similar declarations omitted for the y-axis)
14 }
15 forall Vector u, VectorSpace U -- match any vector
16 where In(u, U) {               -- in some vector space
17    u.arrow = Arrow {
18       startX : U.originX
19       startY : U.originY
20         endX : ?
21         endY : ?
22       color : Colors.mediumBlue
23    }
24    u.text = Text {
25         string : u.label -- label from Substance code
26          color : u.arrow.color  -- use arrow's color
27              x : ?
28              y : ?
29    }
30    u.start = (u.arrow.startX, u.arrow.startY)
31    u.end = (u.arrow.endX, u.arrow.endY)
32    u.vector = minus(u.arrow.end, u.arrow.start)
33    encourage near(u.text, u.end)
34    ensure contained(u.end, U.shape)
35 }
36 forall Vector u, Vector v
37 where Orthogonal(u, v) {
38    local.perpMark = Curve {
39         pathData : orientedSquare(u.shape, v.shape,
40                                   U.origin, const.perpLen)
41         strokeWidth : 2.0
42         color : Colors.black
43         fill : Colors.white
44    }
45      ensure equals(dot(u.vector, v.vector), 0.0)
46 }
47 ... -- (similar rule omitted for Unit)
48 Vector `x2`  {
49    override `x2`.shape.color = Colors.green;
50 }
```

Fig. 12. The Style program defining the visual style used in Fig. 11, *right*. Note that this Style program can be reused for many different Substance programs in the same domain.

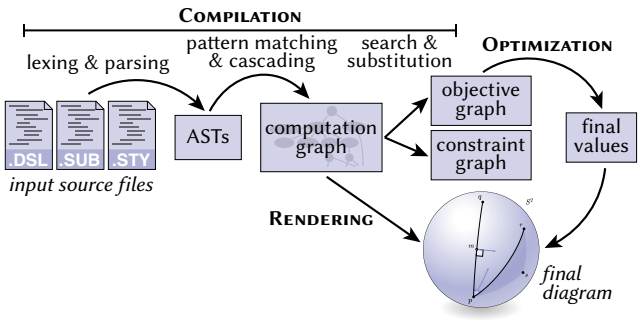


Fig. 13. Pipeline view of the layout engine. Rather than a single static image, compilation yields an optimization problem that can be solved and re-solved to produce many diagrams, or (in principle) used in an interactive tool.

Line 33 asks that the label for a vector be placed close to its endpoint. These expressions are translated into constraints and energy functions that make up a numerical optimization problem (Sec. 4.2).

4 LAYOUT ENGINE

The layout engine translates Penrose code into images (Fig. 13). There are two main stages: a compiler (Sec. 4.1) translates code into an optimization problem that describes possible diagrams, then a solver (Sec. 4.2) produces solutions to this problem. These values are used to render the final diagram (Sec. 4.4). For simplicity, the goal is to automatically produce one static diagram, but the same pipeline could be extended to support capabilities like interaction.

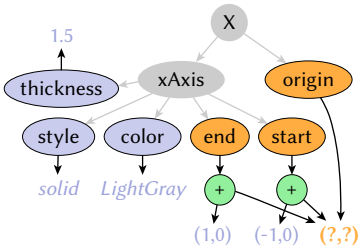
4.1 Compiler

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

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

4.1.2 Computational Graph. The ASTs are combined to define a *computational graph* that encodes operations that define the final diagram (Fig. 14). To build this graph, we apply a standard pattern matching and cascading procedure: iterate over rules in the Style program, find tuples of Substance variables that match the selector pattern, then modify the graph according to the declarations within the matched rule. For example, when the first selector `VectorSpace U` from Fig. 12 matches the variable `X` from Fig. 11, we add nodes to the graph that encode the axes of this vector space. In general, declarations could also remove nodes from the graph or connect previously added nodes. Once this transformation is complete, we have replaced all abstract mathematical descriptions with concrete graphical representatives. All that remains is to determine pending values (*i.e.*, those marked by a `?`) and those values that depend on them, which will be done by the solver.

Fig. 14. Applying the mapping defined by Style code to a Substance program yields a graph that describes how to draw the diagram—here, for part of Fig. 11. Some values are known (in blue), whereas others (in orange) depend on unknowns that must be determined via optimization.



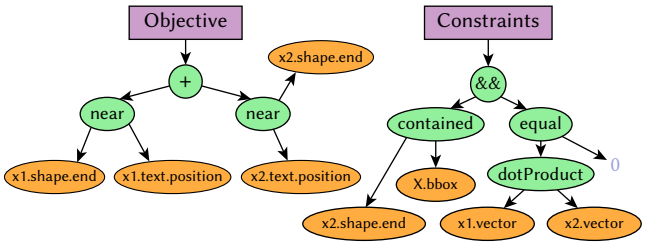


Fig. 15. The computation graph is further expanded to produce graphs representing the objective and constraint space for our optimization problem. From there, we can easily use automatic differentiation to obtain derivatives. This figure depicts part of the optimization graph for Fig. 11.

4.1.3 Optimization Graphs. To encode the optimization problem, we collect terms from the computational graph into an objective and constraint graph (Fig. 15). Each `ensure` and `encourage` statement is then replaced by the corresponding mathematical expression. For instance, `ensure equal(x,y)` is translated into the constraint $x - y = 0$, which the solver seeks to enforce exactly, whereas `encourage equal(x,y)` becomes the objective $(x - y)^2$, which the solver seeks to minimize as much as possible. The overall constraint set is the intersection of all constraints, and the overall objective is a sum of objective terms. Currently Penrose provides a fixed set of constraints and objectives, though it would be straightforward to extend Style to allow user-defined inline expressions.

4.2 Solver

The optimization graphs produced by the compiler describe an optimization problem in standard form, *i.e.*, minimization of an objective function subject to equality and inequality constraints [Boyd and Vandenberghe 2004, Section 4.1]. Such problems may be solved with many standard methods. We currently use an *exterior point method* [Yamashita and Tanabe 2010] 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 (Sec. 2.2). Moreover, the exterior point mehod is an appropriate choice since (i) a feasible starting point is typically not known (Fig. 16), and (ii) by converting constraints into progressively stiffer penalty functions, we can use descent algorithms that do not directly support constrained optimization. In particular, we use L-BFGS with a line search strategy suitable for nonsmooth objectives [Lewis and Overton 2009]. Given the rich structure of our optimization graphs, which can be linked back to program semantics, there are plenty of opportunities to improve this generic strategy, such as decomposing

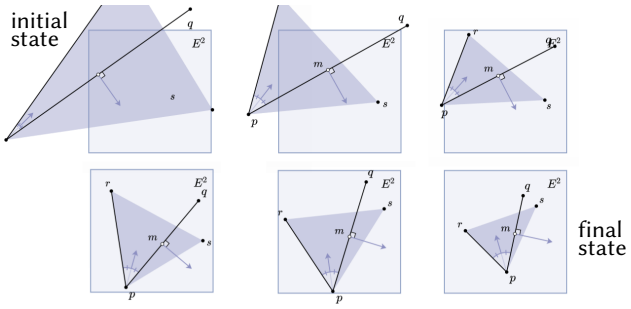


Fig. 16. Our solver can lay out diagrams even if we do not initially know how to satisfy all the constraints. Here we show several steps of optimization.

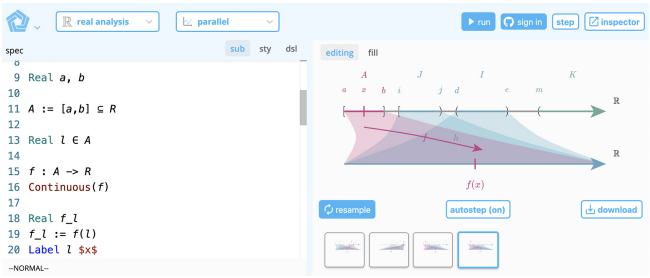


Fig. 17. Our system supports integration with web-based applications. Here a Penrose IDE provides automatic syntax highlighting and autocomplete for any user-defined domain.

the problem into smaller pieces that can be independently optimized, or employing an SMT solver to find a feasible initial state.

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

4.2.2 Failures and warnings. Since our language framework is quite general, a programmer might define difficult or impossible optimization problems. Hence, we can't guarantee that Penrose produces a valid diagram. However, the system can provide feedback by simply printing an error message if any of the constraint values are nonzero. The resulting invalid diagram might even provide useful visual intuition for why the Style program failed (Fig. 2).

4.3 Plugins

A plugin is a piece of external code, written in any language, that is given information from a specific pair of Substance and Style files and can produce more Substance and Style information in specific files for Penrose to use. A plugin is run when making diagrams with a particular Style. A Style may declare the plugins to be called at the top of the file with the syntax `plugin "myPlugin" (args)`, which states that the plugin `myPlugin` should be run with the given argument list. When a diagram is generated, the plugin is given the Substance program as a JSON file, as well as the parameters given in Style as command-line arguments. The plugin can output new Substance code as a text file and/or a set of values for the fields of any Substance variable, encoded as a JSON file. The Substance code generated by a plugin is appended to the existing Substance program, and the values generated by the plugin can be accessed in Style via the syntax `myPlugin[variable][field]`. Note that a plugin is run exactly once, prior to execution of all Penrose code. Therefore, the values generated by a plugin are not optimized by the layout engine, so plugin code does not have to be differentiable. For examples of plugin use, see Sec. 5.2 and Sec. 5.5.

4.4 Rendering

In this paper we focused on generating 2D vector graphics, but in principle nothing about our system design limits us to this particular target. For instance, the constraint-based approach is just as suitable for, say, generating arrangements of 3D objects that can be rendered via photorealistic ray tracing [Pharr et al. 2016], or even constrained interactive diagrams that could be used in virtual reality. In our current implementation, graphical primitives are translated to SVG-native primitives via *React.js* [Facebook 2020] and labels are postprocessed from raw TEX to SVG paths using MathJax [Cervone 2012]. Since Penrose code is typically quite concise, we embed it as metadata into the SVG, easing reproducibility. We also embed Substance names as tooltips to improve accessibility.

4.5 Development Environment

To facilitate development, we built a web-based IDE (Fig. 17) that highlights the potential for high-level diagramming tools built on Penrose. For instance, since the Domain grammar has a standard structure, the IDE can provide features like autocomplete and syntax highlighting for any domain. We are optimistic that the design choices made in Sec. 2 will support the use of Penrose as a platform for building diagramming tools beyond the use cases in this paper.

4.6 Implementation

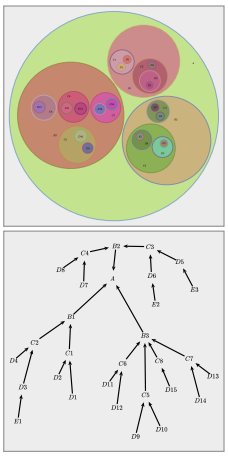
The Penrose system is written in Haskell and the rendering frontend is written in Typescript. We wrote our own solver using the Haskell library *ad* [Kmett 2015] to perform automatic differentiation. We provide one output target and renderer (SVG), together with a fixed set of graphical primitives that are loosely based on SVG (*e.g.*, circles and paths), plus features that SVG users commonly add by hand, like arrowheads. We also provide a fixed set of objectives and constraints for specifying spatial layout, such as shape containment and adjacency relationships, and other functions for performing spatial queries, such as computing bounding boxes and pairwise distances. Sustained use by a community of users might point the way to a standard library. The system has been open-sourced here: `github.com/penrose/penrose`

5 EXAMPLES AND EVALUATION

Our main goal for Penrose was to create a system that can automatically generate diagrams from many different domains using

familiar syntax. Here we examine our design by exploring examples from a variety of common domains in mathematics and computer graphics; we also do some basic performance analysis (Sec. 5.8). All examples in the paper were generated automatically and were not manipulated by hand. However, due to small errors when converting our SVG output to the PDF format required by `pdflatex`, minor manual editing was required (*e.g.* fixing broken transparency).

5.1 Sets



A simple example that illustrates many principles of our system design is the domain of basic set theory—Fig. 18 shows a complete listing for one of three possible styles. Notice here the complete absence of explicit coordinates in both the Substance and Style code. The other two Style programs (not shown) either improve the visual styling, or shift to a different representation where subset inclusion is indicated by a tree-like drawing rather that overlapping disks. Different representations are especially helpful for different types of examples—for instance, disks must shrink exponentially for deeply nested subsets, whereas a tree diagram remains easy to read (see inset, *left*). One limitation highlighted by this example is that the constraints and objectives appearing in Style are not yet extensible within the language itself—for instance, the statement `ensure contains(y.shape,x.shape)` translates to a fixed function on the centers and radii of the two disks.

This example also demonstrates the benefit of a more explicit type system, rather than, say, interpreting raw mathematical strings as in TEX. In particular, Fig. 9 shows how a Domain schema can be used with program synthesis techniques (Sec. 5.7) to automatically enumerate different *logical* instantiations of the given Substance code. To make this example, there was no need to model sets as an explicit datatype (*e.g.* a list of points) nor to assign semantics to these datatypes (such as the impossibility of two sets being both intersecting and nonintersecting). Instead, the program synthesizer can reason purely about the abstract types specified in the Domain schema, letting the constraints defined in the Style define the visual semantics. Thus, the program synthesizer can check if the generated code is valid by simply testing if the constraints defined in Style all evaluate to zero for the optimized diagram. This example captures an important aspect of our system design: the mapping defined by Style programs not only provides a superficial visual interpretation, but also assigns deeper mathematical meaning.

5.2 Functions

A natural concept to build on top of sets is mappings between sets. This example also illustrates the use of plugins (Sec. 4.3). We first add a `Map` type to the Domain for sets (Sec. 5.1), as well as a constructor `From: Set * Set -> Map` specifying the domain and codomain of the map. Here, syntactic sugar

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

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

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

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

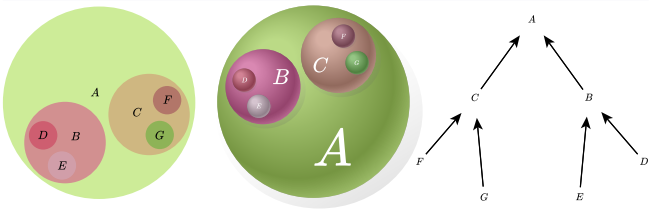


Fig. 18. Here, some Substance code is used to specify set relationships. Different Style programs not only tweak the visual style (*e.g.*, flat vs. shaded disks), but allow one to use a completely different visual representation (*e.g.*, a tree showing set inclusions). `Sets.sty` above describes the flat disk style.

enables one to both declare and define the map via the concise, familiar notation `f: A -> B`. In Fig. 19 we add predicates `Injection`, `Surjection`, and `Bijection` to the Domain schema to illustrate some basic ideas about maps. The two different styles of illustration help ease the transition from thinking of mappings between discrete points to thinking of continuous mappings on the real line. To generate the discrete examples, we wrote a simple plugin (Sec. 4.3) that acts as "glue" between Penrose and an external SMT solver (another example is shown in Fig. 20). The compiler uses this plugin to expand the `Map` objects from Fig. 19, *top* into specific instances of a new

```
-- 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)
```

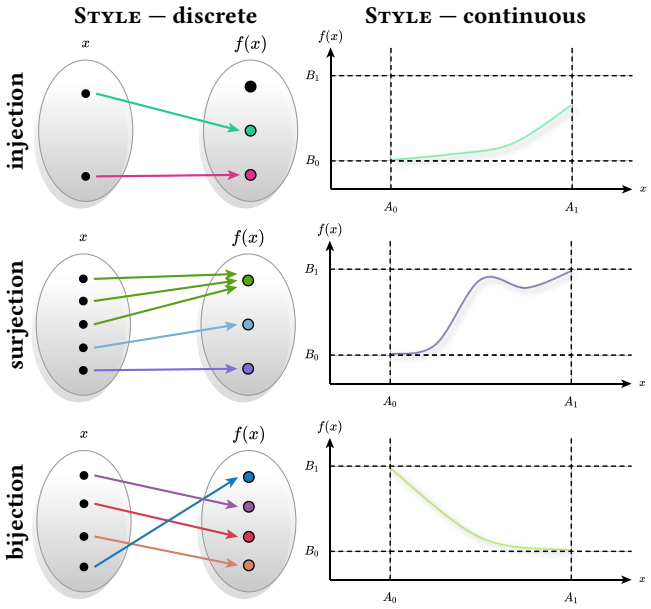


Fig. 19. Different visual representations provide different ways of thinking about an idea. Here, the notion of injections, bijections, and surjections is illustrated in both discrete *(left)* and continuous *(right)* styles. In the former, functions with the desired properties are randomly generated by an SMT solver, allowing the user to learn from many different examples.

`Point` type, as well as a new predicate `(a, b) ∈ f` that expresses a map as an explicit list of domain/codomain pairs. For instance, the map in Fig. 19 generates points `Point A0, A1, B0, B1, B2` with the two predicates `(A0, B1) ∈ f` and `(A1, B2) ∈ f`. A Style tailored to these types is used to generate diagrams in Fig. 19, *left*; as in Fig. 8, hue is optimized to enhance contrast between nearby points. In contrast, the continuous function diagrams in Fig. 19, *right* do not require an external plugin, but instead constrain the degrees of freedom of a Bézier curve. Finally, Fig. 20 shows how abstract function composition in Substance is automatically translated into explicit composition of generated functions by the Style program without any Substance writer effort.

5.3 Geometry

Classical geometric figures provide a good opportunity to examine how one can use different Style programs to change not only the superficial style of a diagram, but also its fundamental visual representation. The familiar "two-column proof" exemplifies how, in mathematics, one can make geometric statements without referring to explicit quantities like coordinates and lengths. Likewise, compass-and-ruler constructions (dating back to the ancient Greeks)

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

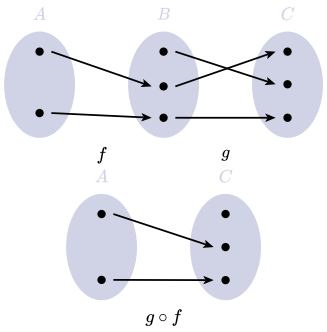
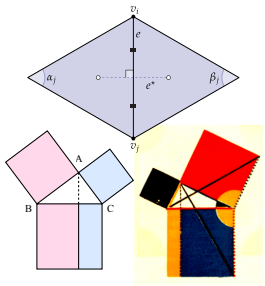


Fig. 20. Here, abstract function composition is realized as explicit composition of functions produced via an SMT solver, illustrating the fact that the composition of an injection and a bijection is an injection.

show how geometric figures can be specified with only relative constraints. These modalities are well-captured in the way we write Substance and Style code for geometric diagrams, respectively. For instance, Fig. 21, *top* gives a listing of geometric assertions that resemble the left column in a two-column proof. This would likely be a natural notation even for intermediate-level students. A bare-bones Style program for this domain (not shown) comprises basic statements very similar to those used in Fig. 12, *e.g.*, to express the orthogonality of two segments. (This approach is similar to the special-purpose geometry package *GCLC* [Janičić 2006]; however, here the domain of objects and visual representations are both extensible at the language level.)



Diagrams used as inspiration for the Styles in Fig. 21.

One goal of Penrose is to codify the subjective style choices made by professional illustrators so non-expert users can benefit from their expertise. Fig. 21, *bottom* cascades on the bare-bones Style program to riff on styles from several existing sources (shown in inset), namely, *Byrne's Euclid* [Byrne 1847], *Wikipedia* [Commons 2006], and a collection of illustrated notes on *discrete differential geometry* [Crane et al. 2013]. This figure also illustrates how we can "mix and match" different Style and Substance programs. The bulk of these styles (~500 lines) share the same baseline Style code; additional code for a specific style requires less than 100 more lines. To illustrate the Pythagorean theorem (right column), we also used cascading to add diagram-specific features (*e.g.*, altitude and extension lines).

In the spirit of Hilbert's quip (Sec. 2), we can also swap out the basic visual representation of a given set of logical statements. For instance, any collection of geometric statements that does not assume the *parallel postulate* can be realized in several different geometries (Fig. 1). To generate this figure, we wrote three Style programs that all match on the same patterns from a common "neutral geometry" Domain schema. Swapping out these Style files then allows users to build intuition about spherical or hyperbolic geometry by exploring how a given figure (expressed via Substance) differs from its Euclidean counterpart. Such an example could be further enriched

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

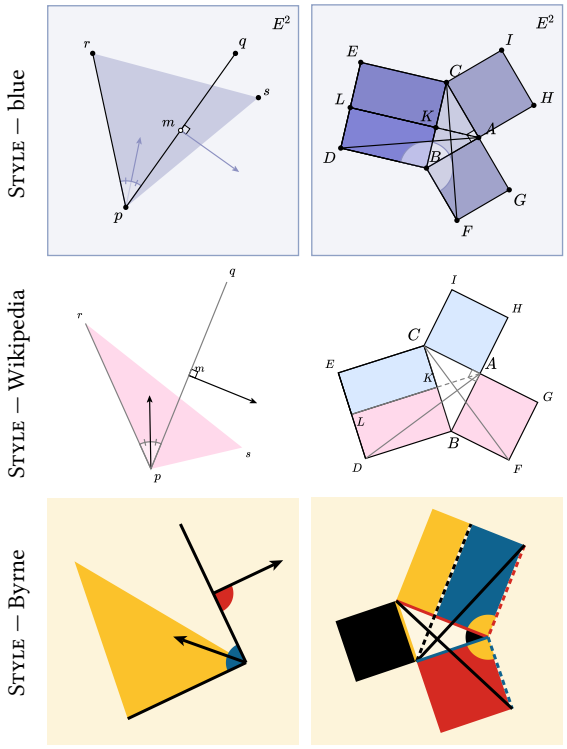


Fig. 21. The cascading design of Style enables one to modify a base style with relatively little code. Here the two Substance programs from Fig. 1 and the listing above are visualized in three different styles, all of which build on the same basic constraints and objectives.

by writing styles for different models of hyperbolic geometry (such as half-plane, hyperboloid, or Beltrami-Klein), each of which involves meticulous calculations. To show that these examples were not cherry-picked, Fig. 29 depicts a gallery of samples.

Finally, the diagram specification enables us to build "staged" diagrams, such as ones illustrating the steps of a proof. Fig. 22 successively uncomments lines of Substance code to produce a style of explanation common in textbooks and slides, a strategy which could easily be applied to (say) any two-column proof. In this example, the values of optimized variables are fixed by hand; an interesting question is how this might be done automatically.

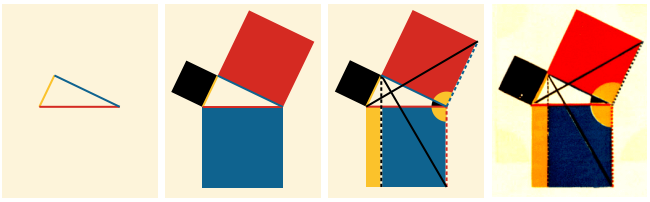
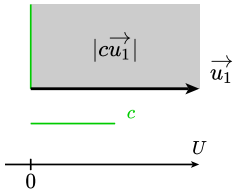


Fig. 22. Once a complex diagram has been built, it can be easily broken into pieces or stages by, *e.g.*, commenting out lines of Substance code. Here we illustrate steps in Euclid's proof of the Pythagorean theorem, turning Byrne's static figure *(far right)* into a progressive "comic strip."

5.4 Linear Algebra

In mathematics, complexity is built up by composing simple statements. The mapping defined by a Style program automatically lifts this *compositionality* to the visual setting. That is, it enables Substance writers to compose logical statements to build up visual complexity without explicit effort from the Style programmer. A good analogy is procedural *L-systems* [Prusinkiewicz and Lindenmayer 2012]. Although a graphics programmer can directly write code to recursively apply spatial transformations, it saves effort to first generate strings in a grammar, then systematically translate these strings into graphical transformations.

In Penrose, examples from linear algebra demonstrate compositionality. The Style declaration on Line 23 of Fig. 12 defines the visual icon for a vector (a 2D arrow). Suppose we now want to illustrate *linear maps*, denoted by f, which have two defining properties: linearity of vector addition ($f(u+v) = f(u) + f(v)$ for all vectors u, v) and homogeneity of scalar multiplication ($f(cu) = cf(u)$ for all vectors u and scalars c). Successive Style rules cascade on Fig. 12 to define how these logical operations should be mapped to visual transformations. For example, application of a linear map f is represented by a literal 2×2 matrix-vector multiply; the image vector $f(u)$ also inherits the color of the argument u. The map itself is visually represented by a labeled arrow, and the domain and target spaces by coordinate planes on either side. The Style programmer need not compose these directives explicitly; the compiler does the tedious job of translating Substance statements (Fig. 23, *top*) into a composition of graphical statements that define a diagram (Fig. 23, *bottom*). Moreover, since the Style program faithfully represents the underlying mathematics, we observe the expected properties, *e.g.*, the arrow for $f(u_1 + u_2)$ is the same as the arrow for $f(u_1) + f(u_2)$. Automatically checking consistency of the visual representation based on analysis of a richer Domain schema would be an interesting topic for future work.



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

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

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

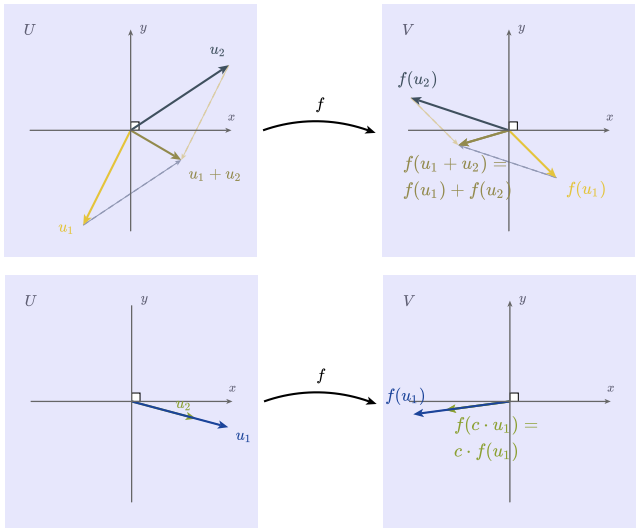


Fig. 23. Composition of mathematical statements naturally translates into composition of graphical transformations with no explicit programmer effort. Here we compose linear maps, showing addition and scaling, to illustrate the two defining properties of linear maps.

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

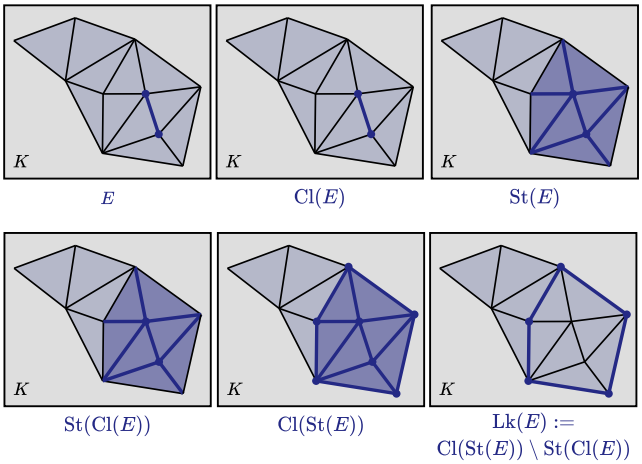


Fig. 24. A language-based specification makes it easy to visually inspect data structures or assemble progressive diagrams with only minor changes to program code. Here we draw the simplicial *link* by building it up from simpler constituent operations.

5.5 Meshes

Polygonal meshes are ubiquitous in computer graphics, but illustrating meshes is often cumbersome due to the large number of elements involved, especially when illustrating meshes by hand or in GUI-based tools. In such cases, Penrose can be useful not just for making illustrations, but also to inspect and debug user-defined data structures by attaching them to custom visualizations. A simple example is shown in Fig. 24, where different regions of a mesh are specified via standard operators on a simplicial complex; this diagram also illustrates the definition of the simplicial *link* [Bloch 1997, Section 3.3]. Further examples in Fig. 25 show how a user can quickly build intuition about this definition by drawing the link of a variety of different mesh subsets.

To make these examples in Penrose, we follow a pattern similar to the discrete function example (Sec. 5.2): generic mesh objects from the Substance code are refined into specific instances of `Vertex`, `Edge`, and `Face` objects by an external plugin, which generates and optimizes a random triangle mesh. Since meshes are randomly generated, the plugin passes a random seed (from its Style arguments) to draw different pieces of the same mesh. For this example, we used an existing JavaScript-based mesh processing library [Sawhney and Crane 2017] that was not designed ahead of time to interface with Penrose. The benefit of generating these elements at the Substance level (rather than returning, say, a static SVG image) is that they can continue to be styled and manipulated within Penrose; the programmer does not have to edit extra graphics code or keep it compatible with the Style program. Likewise, programmers who adopt Penrose as a tool for visual debugging can benefit from system improvements while writing minimal code to attach their data structures to a visual representation.

5.6 Ray Tracing

Our final example constructs light path diagrams, which are often used to illustrate ideas in physically-based rendering. The Substance code expresses types of light paths via Heckbert's regular expression notation. For instance, the expression `L(D|S)S*E` specifies a family of light paths that start at a light source (`L`), bounce off a diffuse or specular object (`S|D`), then bounce off zero or more specular objects (`S*`), then enter the "eye" or camera (`E`). One or more paths can then be declared that have a given form (Fig. 26). The challenge in generating a diagram from such a specification is that there must be geometry in the scene that supports the given path(s). For instance, for a fixed eye, light, and mirror, there may simply be no path of the form `LSE`. Rather than meticulously constructing a valid scene by hand, we can use a simple Style program to jointly optimize the scene geometry and the light path by specifying constraints such as equality of incoming and outgoing angles

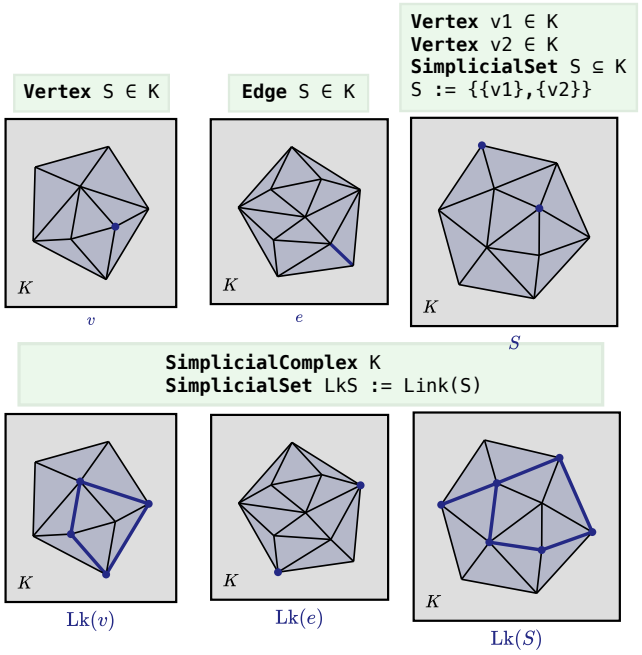


Fig. 25. Domain-specific notation makes it easy to explore an idea by trying out many different examples. Here several subsets of a simplicial complex are specified *(top)* to explore the definition of the "link" *(bottom)*. An external plugin generates random example meshes, further enriching exploration.

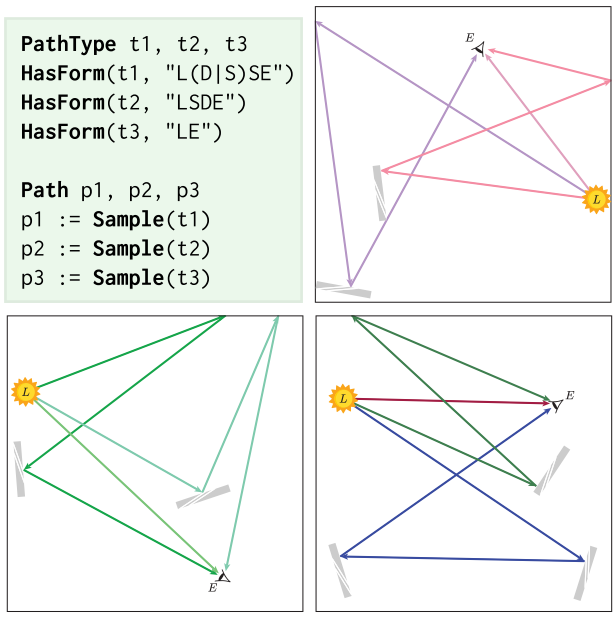


Fig. 26. When drawing ray tracing diagrams by hand, it can be difficult to construct geometry that permits the desired path types. Here we jointly optimize path and scene geometry to match multiple path types simultaneously. Shown are several diagrams generated for the same program.

at a specular bounce. The set of objects in the scene is generated by a simple plugin that expands the regular expression into a set of compatible objects (*e.g.*, a mirror for each specular bounce). This plugin also uses the canvas size to choose appropriate scene and path complexity according to the target output device (Fig. 4). Diagrams for more intricate light transport phenomena could be generated by calling an actual ray tracer (such as *PBRT* [Pharr et al. 2016]) to trace and rejection-sample paths by path type. The added value of generating the final diagrams with Penrose is that initial path and scene geometry generated by the external code can be further optimized to meet other design goals, such as the canvas size. Additionally, the visual style of a large collection of diagrams (*e.g.*, for a set of course notes) can easily be adjusted after the fact.

In our experience, Penrose acts as a *nexus* for diagram generation. It connects disparate components, such as language-based specification and ray tracing, into a diagramming tool that provides the system-level benefits described in Sec. 1.

5.7 Large-Scale Diagram Generation

One goal for Penrose is that effort spent on diagramming should be generalizable and reusable (Goal 6). To demonstrate the system's reuse potential, we developed a simple program synthesizer to automatically create any number of diagrams randomly sampled from a domain. Given a Domain program, a Style program, and the number of diagrams (n) as input, the synthesizer analyzes the Domain program to find the mathematical constructs in the domain, randomly creates n Substance programs that contain these constructs, then compiles, optimizes, and renders the results. Fig. 9 demonstrates an example of using the synthesizer to "autocomplete" an underspecified Substance program by automatically enumerating all possible subset relationships, using information from the Domain schema. Automatically generating diagrams at scale can also help users write better Style programs, since synthesizer can "fuzz" the space of possible Substance programs to find corner cases.

To stress-test the system's performance and the reusability of Style and Domain programs, we randomly generated 2000 Substance programs from the sets domain (Sec. 5.1) in the flat disc style. Penrose was able to create diagrams for all samples. Though 1058 of the 2000 programs had conflicting constraints due to randomness, the solver failed gracefully (as in Fig. 2) and reached convergence.

5.8 Performance Evaluation

We hope to support an iterative workflow where the system's performance does not block the user's creative process. One possible goal is to generate most diagrams within ten seconds, since that threshold is deemed a "unit task" in cognitive science [Newell 1994] and is about as long as similar authoring processes take, such as building a LaTeX document or compiling code. Even lower latency (< 500 ms) might enable new applications of Penrose, since this threshold benefits users of data visualization, live programming, and other exploratory creative tools [Liu and Heer 2014].

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

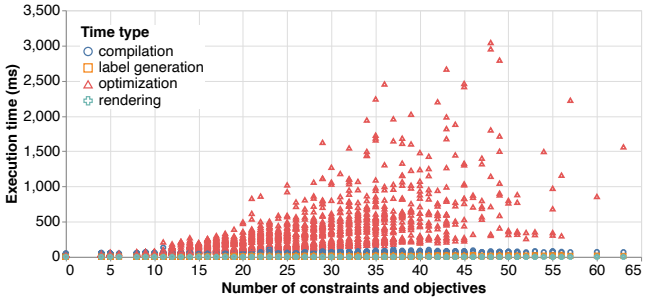


Fig. 27. To stress-test the system and collect timing information, we generated and visualized random Substance programs of different sizes, revealing that optimization dominates the execution time.

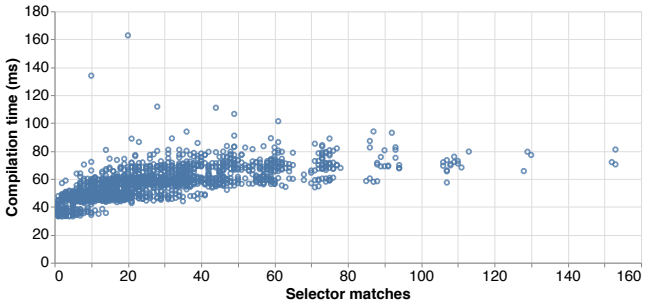


Fig. 28. We evaluated the performance of the Penrose compiler by running it on a large collection of programs, showing that the execution time of the compiler grows slowly as the number of selector matches increases.

(*e.g.* Fig. 1, Fig. 6, and Fig. 21), up to a few minutes. To assess the system's performance, we used diagrams generated in Sec. 5.7 to simulate arbitrary user inputs and measured the time to produce each diagram. To analyze the relative performance of different parts of the system, we separately timed the four stages in the layout engine (Sec. 4): compilation, optimization, label generation, and rendering. Timing was measured on a 2017 MacBook Pro; note that performance in our web-based IDE (Sec. 4.5) is slower due to the cost of editor services and communication with the browser. As shown in Fig. 27, optimization dominates execution time, though the time to convergence grows slowly with the size of the optimization problem. The second slowest part is the compiler, though Fig. 28 shows that compilation time grows linearly with the number of selector matches, suggesting that the compiler scales well.

We are optimistic about our ability to improve the optimization speed, since we are currently using only a simple, generic solver that we implemented ourselves. (See Sec. 4.2 and Sec. 6 for further discussion.) In our experience, optimization often slowed by objectives that involve all pairs of a large collection of objects, especially for label placement, where all labels "repel" all others. Here one could apply standard acceleration strategies for n-body problems, such as the Barnes-Hut algorithm [Barnes and Hut 1986]. Moreover, though it may take time for diagrams to finish, the optimization-based approach provides near-instantaneous *feedback* for most diagrams by displaying the first few steps of the optimization process. These proto-diagrams typically provide enough information about the final layout that the user can halt optimization and continue iterating on the design.

6 DISCUSSION AND FUTURE WORK

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

The system has several limitations that make interesting topics for future work. For example, the Domain, Substance, and Style languages are limited in what they can express. Thus, Substance might be extended with more constructs from mathematical language, such as anonymous expressions, and Style might be extended to provide greater flexibility, *e.g.*, via user-defined priorities on objectives. Additionally, the system currently supports only a fixed set of objectives, constraints, functions, graphical primitives, and renderers, as detailed in Sec. 4.6. However, our software architecture does not present major roadblocks to greater extensibility, such as enabling programmers to define constraints inline or emit output for other targets such as 3D or interactive platforms. The system also presents opportunities to study questions of usability, learnability, and debugging, such as the natural way that Style users might want to express spatial layout constraints and the learning curve for different classes of users.

The cost of optimization is the biggest bottleneck in our pipeline, as seen in Sec. 5.8, which is not surprising given that we currently use a "catch-all" solver. A nice feature of our design is that the program semantics provide rich information about the structure of the optimization problem. This structure should make it possible to adopt highly effective problem-specific optimization strategies of the kind described in Sec. 2.2 and Sec. 5.8, *e.g.*, analyzing the computation graph to break down the diagram into smaller constituent pieces. Beyond finding local minima, we are excited about different design modalities enabled by exploring the constraint space defined by the Style program, such as sampling a diverse set of examples or interpolating between different layouts to create animations.

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

ACKNOWLEDGMENTS

We thank Lily Shellhammer and Yumeng Du for their help. The first author was supported by a Microsoft Research PhD Fellowship and ARCS Foundation Fellowship while completing this work; the

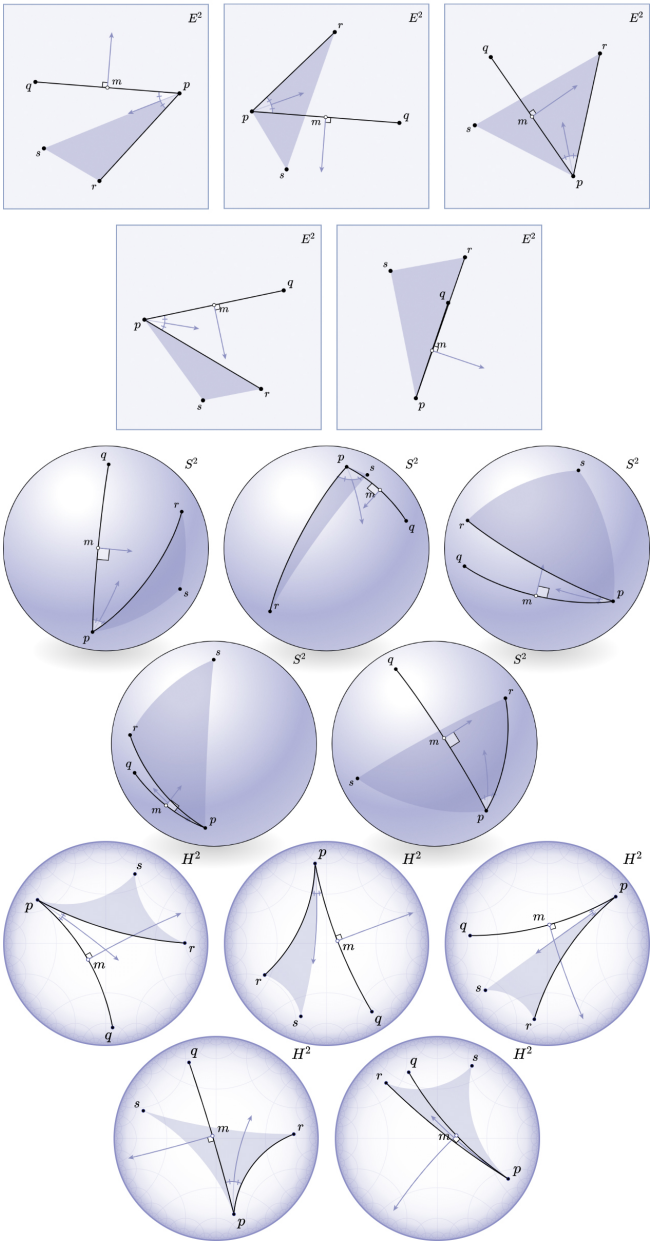


Fig. 29. Though the energy in the objective function can help guide a sampling strategy, choosing which diagram looks "best" to a given user is necessarily a subjective decision. The ability to automatically generate many alternatives makes it easier to find a satisfactory diagram. Here we show a gallery of automatically-generated variants of Fig. 1.

last author was supported by a Packard Fellowship. This material is based upon work supported by the National Science Foundation under grants DMS-1439786 and CCF-1910264, the AFRL and DARPA under agreement FA8750-16-2-0042, and the Alfred P. Sloan Foundation under award G-2019-11406.

REFERENCES

J. Barnes and P. Hut. 1986. A hierarchical O (N log N) force-calculation algorithm. *nature* 324, 6096 (1986), 446–449.

B. Beeton and R. Palais. 2016. Comm. of Math. with TEX. *Visible Language* 50, 2 (2016).

T. Berners-Lee and D. Connolly. 1995. Hypertext markup language-2.0.

Y. Bertot and P. Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

E. D. Bloch. 1997. *A first course in geometric topology and differential geometry.* Springer Science & Business Media.

S. Boyd and L. Vandenberghe. 2004. *Convex Optimization.* Cambridge University Press.

M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. 1995. Scaling up visual programming languages. 28, 3 (1995), 45–54.

O. Byrne. 1847. *The first six books of the Elements of Euclid: in which coloured diagrams and symbols are used instead of letters for the greater ease of learners.*

N. Carter and R. Ellis. 2019. *Group Explorer version 3.0.* Waltham, MA. https://github.com/nathancarter/group-explorer

D. Cervone. 2012. MathJax: a platform for mathematics on the Web. *Notices of the AMS* 59, 2 (2012), 312–316.

D. Cole. 2014. The Chinese Room Argument. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

Wikimedia Commons. 2006. Illustration to Euclid's Proof of the Pythagorean Theorem.

K. Crane, F. de Goes, M. Desbrun, and P. Schröder. 2013. Digital Geometry Processing with Discrete Exterior Calculus. In *ACM SIGGRAPH 2013 courses.* ACM.

L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. 2015. The Lean theorem prover. In *International Conf. on Automated Deduction.* Springer, 378–388.

C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen. 2007. National instruments LabVIEW: a programming environment for laboratory automation and measurement. *JALA: J. of Assoc. for Lab. Auto.* 12, 1 (2007), 17–24.

J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. 2001. Graphviz—open source graph drawing tools. In *International Sym. on Graph Drawing.* Springer, 483–484.

J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. 2004. Graphviz and dynagraph–static and dynamic graph drawing tools. Springer, 127–148.

Facebook. 2020. React: A JavaScript library for building user interfaces. https://reactjs.org/

M. Ganesalingam. 2013. The Language of Mathematics.

M. Hohenwarter and K. Fuchs. 2004. Combination of dynamic geometry, algebra and calculus in the software system GeoGebra. In *Computer algebra systems and dynamic geometry systems in mathematics teaching conference.* 3810–193.

P. Janičić. 2006. GCLC-a tool for constructive euclidean geometry and more than that. In *Int. Con. on Math. Soft.* Springer, 58–73.

E. Kmett. 2015. ad: automatic differentiation. https://hackage.haskell.org/package/ad

T. Kosar, M. Mernik, and J. C. Carver. 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering* 17, 3 (2012), 276–304.

A. S. Lewis and M. L. Overton. 2009. Nonsmooth optimization via BFGS. *SIAM J. Optimiz* (2009), 1–35.

H. W. Lie, B. Bos, C. Lilley, and I. Jacobs. 2005. *Cascading style sheets.* Pearson India.

Z. Liu and J. Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2014).

D. Ma'ayan, W. Ni, K. Ye, C. Kulkarni, and J. Sunshine. 2020. How Domain Experts Create Conceptual Diagrams and Implications for Tool Design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* ACM.

M. Mashaal. 2006. *Bourbaki.* American Mathematical Soc.

R. McNeel et al. 2010. Grasshopper: generative modeling software for Rhino. (2010).

R. Milner, M. Tofte, R. Harper, and D. MacQueen. 1997. *The definition of standard ML.*

R. Miner. 2005. The importance of MathML to mathematics communication. *Notices of the AMS* 52, 5 (2005), 532–538.

A. Newell. 1994. *Unified theories of cognition.* Harvard University Press.

M. Pharr, W. Jakob, and G. Humphreys. 2016. *Physically based rendering: From theory to implementation.* Morgan Kaufmann.

P. Prusinkiewicz and A. Lindenmayer. 2012. *The algorithmic beauty of plants.* Springer Science & Business Media.

R. Sawhney and K. Crane. 2017. geometry-processing.js: A fast and flexible geometry processing library. http://geometry.cs.cmu.edu/js.

I. E. Sutherland. 1964. Sketchpad a man-machine graphical communication system. *Simulation* 2, 5 (1964), R–3.

T. Tantau. [n. d.]. *The TikZ and PGF Packages.*

W. P. Thurston. 1998. On Proof and Progress in Mathematics. *New directions in the philosophy of mathematics* (1998), 337–355.

A. Van Deursen, P. Klint, and J.> Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35, 6 (2000), 26–36.

H. Yamashita and T. Tanabe. 2010. A primal-dual exterior point method for nonlinear optimization. *SIAM J. Optimiz.* 20, 6 (2010), 3335–3363.