GoFish: A Grammar of More Graphics!

Josh Pollock and Arvind Satyanarayan

Fig. 1: Graphics built with GoFish. Left to right: A Sankey tree. A flower chart. A polar ribbon. A nested waffle. A scatterpie.

Abstract—Visualization grammars from ggplot2 to Vega-Lite are based on the Grammar of Graphics (GoG), our most comprehensive formal theory of visualization. The GoG helped expand the expressive gamut of visualization by moving beyond fixed chart types and towards a design space of composable operators. Yet, the resultant design space has surprising limitations, inconsistencies, and cliffs — even seemingly simple charts like mosaics, waffles, and ribbons fall out of scope of most GoG implementations. To author such charts, visualization designers must either rely on overburdened grammar developers to implement purpose-built mark types (thus reintroducing the issues of typologies) or drop to lower-level frameworks. In response, we present GoFish: a declarative visualization grammar that formalizes Gestalt principles (e.g., uniform spacing, containment, and connection) that have heretofore been complected in GoG constructs. These *graphical operators* achieve greater expressive power than their predecessors by enabling *recursive composition*: they can be nested and overlapped arbitrarily. Through a diverse example gallery, we demonstrate how graphical operators free users to arrange shapes in many different ways while retaining the benefits of high-level grammars like scale resolution and coordinate transform management. Recursive composition naturally yields an infinite design space that blurs the boundary between an expressive, low-level grammar and a concise, high-level one. In doing so, we point towards an updated theory of visualization, one that is open to an innumerable space of graphic representations instead of limited to a fixed set of "good" designs.

Index Terms—Grammar of Graphics, Graphical operators, Gestalt principles, Relational paradigm

1 Introduction

Leland Wilkinson's Grammar of Graphics (GoG) is our most comprehensive theory of visualization. It underlies most modern visualization software from low-level toolkits [9, 39] to high-level libraries [10, 38, 47] and graphical editors [18, 37, 43, 51]. Its core promise is that, by breaking charts into pieces that can be combined and recombined, we can unlock a more expressive design space than rigid chart typologies. For example, with the GoG, a user can start with a simple bar chart, layer on text marks to label each bar, map data values to the fill and border colors of the bars, and change the coordinate transform from Cartesian to polar to produce a pie chart.

While the GoG's impact on visualization design is undeniable, it nevertheless presents some sharp expressivity cliffs. For example, consider the scatterpie plot (Fig. 1, right) — where pie charts are scattered on a plane, usually in Cartesian space or on a cartographic map. While this type of chart may initially seem unusual, marine biologists frequently use these charts to visualize changes in plankton biodiversity as each pie visualizes a water sample from a particular location [4]. Yet, although scatterpies combine two very common chart types, the GoG cannot express this composition. Fortunately, Guangchuang Yu, a bioinformatics professor, maintains the `geom_scatterpie` mark [53] for ggplot2 users — but this required dropping down to ggproto, ggplot2′s lower-level framework, and reintroduces the rigidity of chart typologies. For instance, if a biologist wished to scatter bar charts instead of pie charts, or to swap bland pie wedges for more visually evocative petals [41], they would be stuck once again. And, it's not just scatterpies — other seemingly simple charts require custom marks, too. For example, waffle charts (which wrap squares in rows), mosaic charts (which encode data on both the width and height of stacked bars), and ribbon charts (which connect stacked bars with areas) all fall outside the GoG's reach. If a custom mark isn't available for their chart type, a designer must switch to a low-level framework like ggproto or D3, giving up the benefits of higher-level grammars like scale resolution and coordinate transform management.

The GoG is also riddled with surprising inconsistencies. For example, major GoG implementations do not agree on the structure of a stacked bar chart. In the original GoG, stacking occurs through a collision modifier, yet ggplot2 accomplishes it via position adjustments, Vega-Lite with an encoding channel, and Observable Plot through a data transform. These inconsistencies and expressivity cliffs suggest there is important structure that the GoG does not capture.

In response, we present GoFish: a declarative visualization grammar that unlocks the compositional potential of the GoG by separating what marks look like from where and how they are arranged. To do so, we replace composite marks like `dot` and `bar` with primitive shapes like `Ellipse` and `Rect`, and to represent spatial arrangements, we introduce *graphical operators*. These operators describe the structure of a graphic using primitives inspired by Gestalt grouping principles [46], which are key to how readers perceptually group elements [44, 54]. Formalizing them as operators both unlocks a larger expressive gamut and also affords a closer mapping between specification and visual output. For instance, with GoFish, a stacked bar chart is expressed by *stacking* rectangles, and a ribbon chart *connects* neighboring bars.

We demonstrate how graphical operators enable smooth, consistent edits between visualizations by transforming a bar chart into a polar ribbon chart step-by-step (Sec. 3). Our approach to graphical operator composition is inspired by the Bluefish library's formalism of Gestalt principles for diagramming [33]. The Bluefish model allows operators to be nested recursively in tree hierarchies. It also lets graphical operators share children with other operators (Sec. 3.3 & Sec. 4.3.1), allowing for complex, overlapping spatial arrangements.

Through a diverse example gallery (Sec. 7), we demonstrate how GoFish blurs the boundary between a concise, high-level grammar and an expressive, low-level framework — several examples in the gallery would previously require purpose-built mark types, domain-specific visualization grammars, or lower-level frameworks to create. We also provide a qualitative comparison to Mascot (Sec. 8). Whereas Mascot draws inspiration from primitives in Adobe Illustrator, GoFish draws inspiration from the GoG. We compare and contrast these approaches.

• *Josh Pollock is with MIT CSAIL. E-mail: jopo@mit.edu.*
• *Arvind Satyanarayan is with MIT CSAIL. E-mail: arvindsatya@mit.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

GoFish's graphical operators are part of a larger paradigm shift in both grammar and graphical perception research that extends the marks-and-channels model with relational theories of visualization. Unlike the flat, combinatorial design space of marks and channels, GoFish's hierarchical and adjacent composition presents a much larger, more structured design space that prompts us to reconsider the role of effectiveness in grammar design (Sec. 9). GoFish is an MIT-licensed open source project available at `gofish.graphics`.

2 Related Work

2.1 Gestalt Principles in Visualization Theory

Gestalt principles are a well-established part of our theories of graphical perception. Larkin & Simon propose a relational model of both charts and diagrams based on Gestalt principles [24]. Pinker argues for the existence of a "visual description" mental model based on marks, channels, and Gestalt principles [32]. Zacks & Tversky argue for the importance of Gestalt principles to convey data relationships in graphics [44, 45, 54], which has been followed up by Boger & Franconeri's and Fygenson & Franconeri's studies of bar chart arrangements [8, 14]. Engelhardt & Richard propose a hierarchical theory of graphics that includes marks, channels, Gestalt principles like arrangement and linking, and coordinate transforms [11–13, 36]. These theories have deeply informed GoFish's design, but whereas they are analytic, GoFish is a formal, generative theory of graphics.

2.2 Grammars of More Graphics

Low-level grammars and toolkits like Vega [39] and D3 [9] are extremely expressive, but in return do not provide much structure or inference. Users typically work directly with scales, coordinate transforms, and layouts that must be carefully sequenced to produce a meaningful specification. Encodable [52] is a library that allows users to create Vega-Lite-style APIs for custom marks written in React or other JavaScript frameworks. It provides a simple interface for specifying encoding channels, sensible defaults, and scale inference. GoFish could adapt Encodable's approach to make it easier for users to author custom marks.

Prior visualization systems that formalize Gestalt principles explicitly are limited to specific classes of visualization. Domain-specific visualization grammars including GoTree for trees [25], Atom for unit visualizations [31], productplots for product plots including nested mosaics [49], and SetCoLa for node-link graphs [20] all provide compositional operators for specific chart types. Many of these operators are inspired by Gestalt principles. For example, SetCoLa provides clustering constraints for spatial proximity and hull constraints for common region. By limiting themselves to specific chart types, these grammars are more tailored to their domains and more concise than GoFish. On the other hand, switching between grammars presents large complexity cliffs, as a user must reorient themselves to a new set of abstractions. GoFish's operators provide a consistent representation across a large class of graphics.

GoFish is most similar to the language underlying Charticulator [34] and to the Mascot (formerly Atlas) grammar underlying Data Illustrator [26–28]. Both systems use shapes and composition abstractions like operators, layouts, and constraints to achieve impressive expressive gamuts. The biggest difference between GoFish and these systems lies in their design goals. Charticulator's representation and Mascot were initially designed to support direct manipulation chart editors whereas GoFish is designed to be authored directly, similar to the GoG. As a result, Charticulator's representation cannot be used outside of the system, and Mascot provides a procedural API with operators inspired by Adobe Illustrator. GoFish, on the other hand, draws inspiration most directly from Bertin and Wilkinson as well as Zacks's and Tversky's work on the role of Gestalt grouping principles in charts [44, 45, 54]. We unpack these implications in Sec. 8.

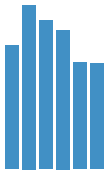
3 GoFish by Example: Building a Polar Ribbon Chart

We demonstrate GoFish by example, incrementally transforming a bar chart into a stacked bar chart, a ribbon chart, and finally a polar ribbon chart. This example touches on the main ideas of GoFish we'll see throughout the rest of the paper. For the first few examples, we will compare to Observable Plot since both GoFish and Observable Plot are embedded in JavaScript. While some details about stacking differ between GoG grammars, the key points generalize to all of them. We encourage you to pick your favorite one and follow along! The dataset we're using, `sf` (short for seafood), is the `counts` of different fish `species` in six `lakes`. The lakes are all connected by the same river. We want to understand how the total population of fish varies among the lakes as well as the variation of each species.

3.1 Bar Chart: Breaking Marks Into Shapes, Data, and Graphical Operators

❶ Consider Fig. 2. We might start with a bar chart to get a sense of the count in each lake. GoG libraries use marks and channels to map data tables to collections of shapes. In Observable Plot, we might choose the `barY` mark and map the `lake` field to the `x` channel and the `count` field to the `y` channel to set the heights of the bars. By contrast, GoFish splits the `barY` mark into three pieces: `StackX`, `For`, and `Rect`. The `For` function maps over the dataset, calling the `Rect` shape function six times to produce six rectangles. The `StackX` graphical operator evenly spaces these shapes horizontally and aligns them vertically.

❶ Bar Chart



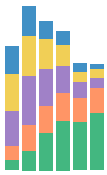
Observable Plot

```
barY(sf, {x: "lake", y: "count"})
```

GoFish

```
StackX({spacing: 8},
  For(groupBy(sf, "lake"), (d) ⇒
    Rect({w: 32, h: v(sumBy(d, "count"))})))
```

❷ Stacked Bar Chart



Observable Plot

```
barY(seafood, {x: "lake", y: "count", fill: "species"})
```

GoFish

```
StackX({spacing: 8},
  For(groupBy(sf, "lake"), (lake) ⇒
    StackY({spacing: 0},
      For(lake, (d) ⇒
        Rect({w: 32, h: v(d.count), fill: v(d.species)})))))
```

Fig. 2: In the GoG, marks like `barY` are primitives and vertical stacks are applied implicitly by color channels. GoFish instead uses explicit `Stack` graphical operators, revealing the spatial structure of the visualizations.

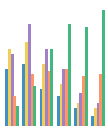
Let's take a closer look at the arguments to these functions. As with all graphical operators in GoFish, `StackX` takes some aesthetic properties and an array of child elements and produces a new element. In this case, we have specified a spacing of 8 pixels between the bars. The `groupBy` data transform in the `For` function creates an array of tuples for each `lake`. Like `StackX`'s spacing, `Rect`'s width is an aesthetic value of 32 pixels. However, its height is a data-driven value representing the sum of the counts of every fish species in the lake. We use GoFish's `v` function to tell the system that the height parameter is a piece of data that must be scaled, not an aesthetic value.

3.2 Stacked Bar Chart: Nesting Graphical Operators

❷ In most GoG libraries, bar stacking happens by default when the user specifies a color channel. By contrast, in GoFish we explicitly create a vertical stack by adding a `StackY` graphical operator inside the `StackX`. We adjust the `h` channel to encode the `count` of a single species and add a `fill` encoding to complete the stacked bar chart. This specification is more verbose than one in the GoG, but its organizational structure explicitly reflects the visual structure of the chart more directly. A stacked bar chart comprises a horizontal stack of vertically stacked rectangles. This one-to-one mapping means that changes to a GoFish specification closely reflect changes to the resulting visualization. Figure 3 shows two such examples.

❸ A grouped bar chart is a stacked bar chart where the bars are stacked horizontally instead of vertically. But the GoG doesn't see it that way. To switch from a stacked to a grouped bar chart in Observable Plot, a user changes the `x` encoding from "lake" to "species" and uses "lake" for the horizontal facet encoding, `fx`. The meaning of the `x` encoding has apparently switched from the position of the bar stack to the position of each individual bar, and the user has to invoke the concept of faceting, which wasn't necessary for vertical stacking. By

❸ Grouped Bar Chart



Observable Plot

```
barY(sf, {x: "species", y: "count", fill: "species", fx: "lake"})
```

GoFish

```
StackX({spacing: 12},
  For(groupBy(sf, "lake"), (lake) ⇒
    StackX({spacing: 1},
      For(lake, (d) ⇒
        Rect({w: 8, h: v(d.count), fill: v(d.species)})))))
```

❹ Waffle Chart

Observable Plot

```
waffleY(sf, {x: "lake", y: "count", fill: "species"})
```

GoFish

```
StackX({spacing: 8},
  For(groupBy(sf, "lake"), (lake) ⇒
    StackY({spacing: 2, alignment: "start"},
      For(lake.flatMap((d) ⇒ Array(d.count).fill(d)).chunk(3),
      (chunk) ⇒
        StackX({spacing: 2},
          For(chunk, (d) ⇒
            Rect({w: 8, h: 8, fill: v(d.species)}))))))))
```

Fig. 3: Variations of a stacked bar chart. While GoFish is more verbose, it's more consistent and composable. Notice `Stack`s can nest arbitrarily.

contrast, in GoFish, the user switches the `StackY` to a `StackX` and adjusts the aesthetic `w` and `spacing` values.

❹ A waffle chart is like a stacked bar chart, except it vertically stacks rows of squares instead of individual `Rect`s. Expressing this chart requires a third level of stacking. Unfortunately, in the GoG we are already out of channels and facets! Observable Plot provides a custom mark, `waffleY`, that is written using D3 and Plot's internal APIs. In contrast, GoFish's graphical operators can nest arbitrarily, so we can add another `Stack` just as before. To turn a stacked bar chart into a waffle chart in GoFish, we add a data transform to produce `species.count` copies of each `species` tuple and chunk them into groups of 3. We then make a `StackX` of square `Rect`s for each chunk.

3.3 Ribbon Chart: Overlapping Graphical Operators

❺ To highlight the changes in species counts between lakes, we can sort the species by count in each lake and turn our stacked bar chart into a ribbon chart. The ribbon chart connects the species rectangles across each lake with intervals. Like waffle charts, ribbon charts can't be decomposed into basic GoG marks and channels, so they are only supported by GoG libraries as custom marks. In GoFish, we can extend our stacked bar chart specification to turn it into a ribbon chart. Figure 4 (left) shows a ribbon chart and its GoFish specification. To connect the bars, we first give each `Rect` shape a `name` (line 8), which will allow us to refer to it later. Next, we create a layer using the `Frame` operator (line 1), which allows us to place multiple shapes in the same space. To place the ribbons, we use the `ConnectX` operator, which horizontally connects its children with paths (line 10). Since there is one ribbon per species, we make one `ConnectX` per species using the `For` function and a data transform on line 9. Notice on line 12 that `ConnectX`'s children are `Ref`s, not `Rect`s. A `Ref` creates a declarative reference that points to an existing element. This allows `ConnectX` to connect the existing `Rect` shapes. `Ref`s are powerful as they let us *overlap* graphical operators. That is, they allow us to use the same shape in two different spatial arrangements without the use of additional concepts like constraints (Sec. 4.3.1).

3.4 Polar Ribbon Chart: Transforming Shapes and Graphical Operators

❻ To turn our ribbon chart into a polar ribbon chart, we add the `Polar` coordinate transform to our `Frame`, adjust some aesthetic values, and switch the `StackX` to `"center"` mode (line 3). Unlike the default spacing mode, a `Stack` in `"center"` mode evenly spaces its children's centers rather than evenly spacing them edge-to-edge. This ensures the bars are evenly distributed along the theta axis.

To summarize, in this section we saw how GoFish trades some of the concision of conventional high-level GoG systems to achieve a significantly more consistent and expressive language. The structure of a GoFish specification closely mirrors the structure of the resulting graphic. Despite its expressiveness, GoFish still handles scale resolution and coordinate transforms declaratively like the original GoG.

4 The GoFish Grammar

GoFish is a visualization grammar embedded in TypeScript comprising a standard library of shapes, data transforms, graphical operators, and coordinate transforms. Additionally, GoFish has a helper for encoding channel specification, `v`, which tags values as data to be scaled, a `Frame` operator for layering elements and applying coordinate transforms, and a `color` object that provides categorical color palettes.

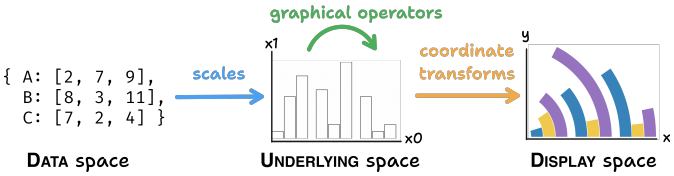
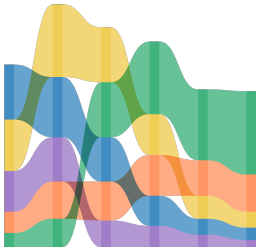


Fig. 5: GoFish is organized around three spaces: Data, Underlying, and Display. These correspond roughly to frame, underlying space, and display space in the original GoG [50, p. 359].

Figure 5 shows the three conceptual spaces that govern the behavior of GoFish: Data, Underlying, and Display space. These spaces were inspired by, but do not directly follow, the GoG's spaces: frame, underlying, and display [50, p. 359]. They help us keep track of values inside the system and which operations can be applied to them. Roughly, GoFish takes Data values assigned to shape channels and scales them to Underlying space, then graphical operators arrange values in Underlying space, and finally coordinate transforms map Underlying values to Display space.

❺ Ribbon Chart



```
1  Frame([
2    StackX({spacing: 64},
3      For(groupBy(sf, "lake"), (lake) ⇒
4        StackY({spacing: 0},
5          For(sortBy(lake, "count"), (d) ⇒
6            Rect({w: 16, h: v(d.count),
7                  fill: v(d.species)})
8            .name(`${d.lake}-${d.species}`))))),
9    For(groupBy(sf, "species"), (species) ⇒
10     ConnectX({opacity: 0.8},
11       For(species, (d) ⇒
12         Ref(`${d.lake}-${d.species}`))))])
```

❻ Polar Ribbon Chart

```
1  Frame({coord: Polar()}, [
2    StackX({x: -Math.PI / 2, y: 50,
3      spacing: 2 * Math.PI / 6, mode: "center"},
4      For(groupBy(sf, "lake"), (lake) ⇒
5        StackY({spacing: 0},
6          For(sortBy(lake, "count"), (d) ⇒
7            Rect({w: 0.1, h: v(d.count),
8                  fill: v(d.species)})
9            .name(`${d.lake}-${d.species}`))))
10   ),
11   For(groupBy(sf, "species"), (species) ⇒
12     ConnectX({opacity: 0.8},
13       For(species, (d) ⇒
14         Ref(`${d.lake}-${d.species}`))))])
```

Fig. 4: GoFish ribbon and polar ribbon charts. We add the `ConnectX` operator to connect the rectangles. Its children are `Ref`s that reference the `Rect` shapes we created earlier. `Frame` allows us to layer shapes and operators. We add a `Polar` coordinate transform to the `Frame` and adjust `StackX`'s and `Rect`'s parameters to turn our ribbon chart into a polar ribbon chart.

shapes

```
shape(channels: Channels) ⇒ Elem
```

- `Rect()`: a rectangle shape
- `Ellipse()`: an SVG ellipse
- `Ref(select: string)`: a declarative reference to another shape

data transforms and operators

```
groupBy(data: T[], key: string) ⇒ Record<string, T>
sortBy(data: T[], key: string) ⇒ T[]
sumBy(data: T[], key: string) ⇒ number

For(data: Coll<T>, cb: (d: T, i: Key) ⇒ Elem) ⇒ Elem[]
v(datum: any) ⇒ DataValue
```

graphical operators

```
graphicalOp(aes: Aesthetics, children: Elem[]) ⇒ Elem
```

- `Stack()`: stack elements horizontally or vertically
- `Connect()`: connect elements with an interval
- `Enclose()`: enclose elements in a rectangle

```
Frame({ coord: Coord }, children: Elem[]) ⇒ Elem,
```

coordinate transforms

```
coord() ⇒ Coord
```

- `Linear()`: $(x, y) \mapsto (x, y)$
- `Polar()`: $(\theta, r) \mapsto (r\cos\theta, r\sin\theta)$
- `Wavy()`: $(x, y) \mapsto \left(x + 5 \cdot \sin\left(\frac{y}{10}\right), y + 5 \cdot \sin\left(\frac{x}{10}\right)\right)$

type definitions

```
type AestheticLit := any
type ChannelValue := AestheticLit | DataValue
type Aesthetics := { [key: string]: AestheticLit }
type Channels := { [key: string]: ChannelValue }
type Coll<T> := T[] | Record<string, T>
```

Listing 1: GoFish comprises four main concepts: shapes, data transforms and operators, graphical operators, and coordinate transforms. `Frame` is used to layer elements and apply coordinate transforms. `v` distinguishes data from aesthetic literals. `Stack` and `Connect` have `X` and `Y` variants that specialize the original operators to a specific direction.

4.1 Shapes

A GoFish shape is a basic graphical element like a rectangle or ellipse. Shapes have spatial channels: `x`, `y`, `cx`, `cy`, `x2`, `y2`, `w`, `h`. They also have properties corresponding to SVG attributes such as `fill`, `stroke`, and `opacity`. In addition to basic elements, a shape can be a `Ref`, which is a declarative reference to another basic graphic element (Sec. 4.3.1).

4.2 Data Transforms

We re-export Lodash's `groupBy`, `sortBy`, and `sumBy` transforms so that most charts can be made without importing a separate library. However, GoFish users can use any JS data transform library.

The `For` operator maps over an object, array, or Lodash collection, calling the `cb` function on each entry. This operator is inspired by SolidJS's `For` component. Although `For` subsumes `Array.map`, we use it only when returning GoFish `Elems`. We have found that using `For` instead of `.map` in these cases makes GoFish specifications clearer, because it separates data transforms from element creation code.

The `v` function. Inspired by Vega-Lite and SwiftCharts, GoFish uses the `v` function to distinguish between data values, which must be scaled, and aesthetic literals. Similar to SwiftCharts, values must be explicitly tagged because GoFish specs are made of individual shapes rather than mark functions that generate shapes using field accesses like ggplot2's and Observable Plot's primitives.

4.3 Graphical Operators

GoFish's graphical operators are heavily inspired by Bluefish's relations, which operationalize Gestalt principles for diagramming [33]. A graphical operator takes some aesthetic properties and child elements as input, and produces a new element as its output. A child element of a graphical operator may be a primitive shape or an element produced by another graphical operator. The resulting graphical element is an arrangement of the operator's child elements as well as any new shapes the operator may have drawn. GoFish provides three main graphical operators — `Stack`, `Connect`, and `Enclose` — and a supporting operator, `Frame`, that groups elements and applies coordinate transforms.

Graphical operators are very general. They can run arbitrary layout code as long as they are immutable (Sec. 4.3.2) and depend only on their input parameters, some limited parent context information, and their children's bounding boxes. To decide which operators to implement, we first aimed to support common chart types like scatter plots, line charts, and bar charts. To design operators that could describe the perceptual structure of these charts, we drew inspiration from existing research in cognitive and perceptual psychology [32, 44, 45, 54] as well as descriptive theories of graphic structure [13, 24, 33, 35]. This literature emphasizes the structural role Gestalt grouping principles play in graphics. We ultimately settled on the Gestalt principles of uniform spacing (`Stack`), element connectedness (`Connect`), and common region (`Enclose`) as GoFish's initial set of graphical operators. By keeping this set small, we demonstrate that GoFish's expressive power is due to its novel compositional abilities rather than bespoke algorithmic layouts (Sec. 7).

The `Stack` operator evenly spaces its children on its primary axis and aligns them on the other unless they have already set their own positions with encoding channels. `Stack` has two modes: `"center"` and `"edge"`. `"center"` stacking is similar to D3's ordinal `pointScale`, which equally spaces points. `"center"` equally spaces its children's centers. `"edge"` stacking is similar to D3's ordinal `bandScale`, which creates equally sized bands with gaps between them. However, unlike `bandScale`, `Stacks` children do not need to be equally sized. `Stack` defaults to `"edge"` stacking.

The `Connect` operator connects elements with a `path` element. This operator replaces the typical role of a `line` or `area` mark in a GoG system, instead creating an explicit connection between existing shapes. Similar to `Stack`, the `Connect` operator has `"edge"` and `"center"` modes where it connects the edges or the centers of its children, respectively. `Connect` also defaults to `"edge"` mode.

The `Enclose` operator contains its children in a common region using a rectangle. `Enclose` takes `padding` and `rx` and `ry` parameters to adjust its appearance.

The `Frame` operator overlays its children and optionally applies a coordinate transform. When used in conjunction with the `Ref` shape, `Frame` allows graphical operators to share children with other graphical operators (Sec. 4.3.1). Notice that GoFish doesn't have a `facet` operator. This is because a `Stack` operator whose children are `Frames` acts as a faceting operator.

Notice graphical operators are separate from data transforms. While spatial composition is sometimes combined with data transforms [31], we chose to keep them separate in GoFish to allow users to write data transforms in the host language. This makes it easier to write more bespoke data transforms, such as the one for waffle charts in Fig. 2.

4.3.1 Overlapping Graphical Operators with `Ref`

As we saw in Sec. 3.3, graphical operators can share children (i.e., overlap) using the `Ref` shape. Overlapping graphical operators is especially useful when `Connecting` shapes that have already been placed by `Stacks`. None, some, or all of an operator's children may be `Refs`. A `Ref` works like a declarative query selector. A user can reference a mark by its name, which is a globally defined string like an HTML ID. Future work may explore more complex query selectors, perhaps inspired by Cicero's specifiers [22] or Atlas's `find` [27]. We considered using JavaScript's own variable bindings instead of a separate `Ref` shape, since variable definitions allow values to be reused in code. But this makes specification authoring more viscous, because a user must hoist a node out of their specification in order to reuse it. It also obscures the correspondence between the hierarchical structure of a GoFish specification and the hierarchical structure of the resulting graphic.

`Ref` reduces the number of concepts required to express common spatial arrangements. Systems like Charticulator [34] and Mascot [26] introduce constraints, links, and layouts as separate concepts. Constraints can refer to elements across a specification, but are limited to simple relationships and cannot render shapes. Links can also connect elements across a spec, but can only draw shapes, not position children. Layouts cannot be overlapped, but can express complex relationships and draw shapes. Graphical operators unify these ideas with `Ref`. Because an operator cannot tell if its children are normal graphical elements or `Refs`, a graphical operator can be written like a layout

— using arbitrary code and drawing shapes — while also overlapping with other operators like a constraint or link.

4.3.2 Graphical Operators Are Immutable

Graphical operators cannot mutate the size or position of an element if it has already been set. This ensures the declarative nature of a GoFish specification. If a `Stack` exists in the spec, its children are stacked. If there's a `Connect`, then its children are connected. This property arises naturally in grammars with recursive composition but no overlaps, such as Atom or productplots, because child sizes and positions depend only on their arguments, their own children, and their immediate parent. However, with overlapping operators we need to take special care that other operators don't accidentally mutate an existing value. To that end, we adopt Bluefish's dimension ownership model [33]. When a shape or operator writes to a size or position field, that field is *owned* by the writer and can't be modified by anything else.

4.4 Coordinate Transforms

The coordinate transforms in GoFish's standard library are `Linear`, `Polar`, and `Wavy`. See Lst. 1 for their formal definitions. In addition to these built-in transforms, GoFish supports user-defined ones. Coordinate transforms are similar to graphical operators in that they both manipulate arrangements of graphical elements. But they behave slightly differently. Table 1 summarizes those differences.

Table 1: Graphical operators vs. coordinate transforms

Characteristic	Graphical Operators	Coordinate Transforms
arrangement kind	discrete arrangements	continuous arrangements
layout scope	affects direct children	affects all descendants
order sensitivity	depends on child order	independent of child order

Intuitively, a coordinate transform provides the "substrate" for shapes, while graphical operators adjust the sizes and positions of shapes within this substrate. As a result, graphical operators are well-suited to discrete arrangements while coordinate transforms are better for continuous ones. Graphical operators arrange only their direct children (which may be primitive shapes or composite elements) whereas coordinate transforms ignore element boundaries and affect all descendant primitive shapes.[1] Child ordering is important to a graphical operator — for example `Stack`'s arrangement reflects its child ordering — but coordinate transforms apply to all their descendants independently.

5 Untangling GoG Marks from Operators and Coords

The central observation driving the design of GoFish is that GoG marks entangle shapes, spatial arrangements, and coordinate transforms. Though the GoG moved away from chart typologies towards more composable pieces, we identify two entanglements that create mark typologies. First, marks are complected with spatial arrangements, which results in large taxonomies of marks — one per arrangement. Second, marks are complected with their *dimension* (0D, 1D, or 2D), which creates yet another set of taxonomies. GoFish disentangles marks from spatial arrangements by introducing graphical operators for arrangements and reducing marks to primitive shapes (Sec. 5.1). GoFish disentangles shapes from dimensions by adapting Bertin's notion of *shape embedding* [5, 6], inferring a shape's dimension from its encoding channels and the graphical operators that arrange it (Sec. 5.2). Untangling these concepts is what makes GoFish more expressive than the GoG.

5.1 GoG Marks Are Spatial Arrangements of Shapes

GoG systems implicitly define marks as collections of individual shapes laid out in space. For example, a `dot` mark is collection of circles positioned with data-driven x and y coordinates. Similarly, a `bar` mark is a collection of rectangles with data-driven heights that are evenly spaced on the x-axis and vertically aligned on the y-axis. This tightly couples spatial arrangements to mark types. As a result, adding a new organizational strategy, like arranging marks in waffles or trees, requires adding a new mark. To make matters worse, since GoG marks are typically named after the low-level shapes they use, not their spatial arrangements, GoG systems accumulate similar-sounding names for marks that use the same shapes but vary their spatial arrangements. For example, Vega-Lite uses `rect` for heatmaps, `bar` for bar charts, `square` for scatter plots, and `arc` for polar-mapped rectangles. It can be hard to know which mark type to use in which situation, because these mark names lack a clear organizational structure.

In contrast to the GoG, GoFish shapes formalize the notion of mark developed by Bertin [5, 6]. Though he never defines it precisely, we can infer from context that Bertin uses "mark" to mean a basic graphical element like a point or line segment, similar to its use in the visual arts and drafting, rather than a collection of elements. By separating shapes from their arrangements, we allow users to compose them more flexibly and reuse the same arrangement with a different shape. For example, a GoFish user can easily turn a spec for a bar chart into a spec for a bar chart of ellipses by keeping the `Stack` fixed and changing the `Rect` shape to an `Ellipse`. This separation also enables recursive composition. A `bar` mark cannot be recursively composed with another `bar`, but `Stack`s can be easily nested to create a grouped bar chart.

Although Wilkinson drew heavily from Bertin, he didn't formalize Bertin's notion of mark directly. Wilkinson instead introduced "geometric graphs" [50, p. 155], functions that map data to collections of basic graphical elements. Though he took care to explain that these graphs are distinct from Bertin's marks, they are still named after the shapes they produce, like `point` and `line`. This ambiguity has grown with each new implementation of the GoG. In ggplot2, "geometric graph" became "geom" [47], a "geometrical object that a plot uses to represent data" [48], which could equally describe a low-level shape or a geometric graph. Vega-Lite uses "mark type" to refer both to kinds of low-level shape and to the geometric graphs used to produce them [38]. Observable Plot merges "mark" and "mark type" completely, saying both that marks "are geometric shapes" and that, e.g., "the dot mark draws circles" [10].

GoFish resolves the conceptual ambiguity between Bertin's and Wilkinson's definitions of "mark" by breaking up Wilkinson's concept into its constituent pieces. Wilkinson marks are compositions of graphical operators, data transforms, and Bertin marks (i.e., GoFish shapes).

5.2 Shapes Can Be *Embedded* in Underlying Space

In conventional GoG systems, each mark has a fixed *dimension*: a `dot` is 0D; a `path` is 1D; a `polygon` is 2D [47]. Dimensions control how marks can be arranged by collision modifiers and manipulated by coordinate transforms. For example, a 0D mark is only translated, not scaled or warped, by a coordinate transform. This seems reasonable at first, but tightly coupling marks and their dimensions is limiting in practice. For example, in ggplot2, `geom_text` is 0D. As a result, it can't be rotated and stretched by a coordinate transform, making it difficult to use for labels in polar space.

Similar to different spatial arrangements, GoG systems provide different marks for zero-, one-, and two-dimensional versions of the same shape. These marks have confusing names, too. In ggplot2, for example, `geom_tile`, `geom_segment`, and `geom_rect` represent 0D, 1D, and 2D rectangles, respectively. Since these mark names and behaviors overlap, especially in Cartesian space, picking the right mark type is subtle and error-prone.

Rather than a shape like `Rect` having a fixed dimension, in GoFish we determine whether a shape's width and/or height are embedded in Underlying space — and thus warped by a coordinate transform — or mapped directly to Display space. A shape is 0-, 1-, or 2-dimensional depending on whether zero, one, or both of its width and height are embedded. Shape embedding is important, because embedded widths and heights are warped by coordinate transforms. For example, consider Fig. 6 (left). When a `Rect` is used in a polar scatter plot, it needs to be zero-dimensional so that the coordinate transform only translates the shape and doesn't warp it. But in a polar bar chart, the `Rect`s must be one-dimensional so that they are rotated to the right positions and their widths remain fixed. Finally, in a pie chart, the `Rect` must be treated as two-dimensional to be warped into a wedge shape.

In GoFish, a shape's dimension is determined by its width and height encodings as well as any graphical operators that may read or write the width or height of the shape. Figure 6 (right) shows the embedding rules with some abstract examples. First, if the width (resp. height) of a shape is a data-driven value, then GoFish considers the width

[1] We also transform shapes *produced* by operators like `Connect` and `Enclose`.

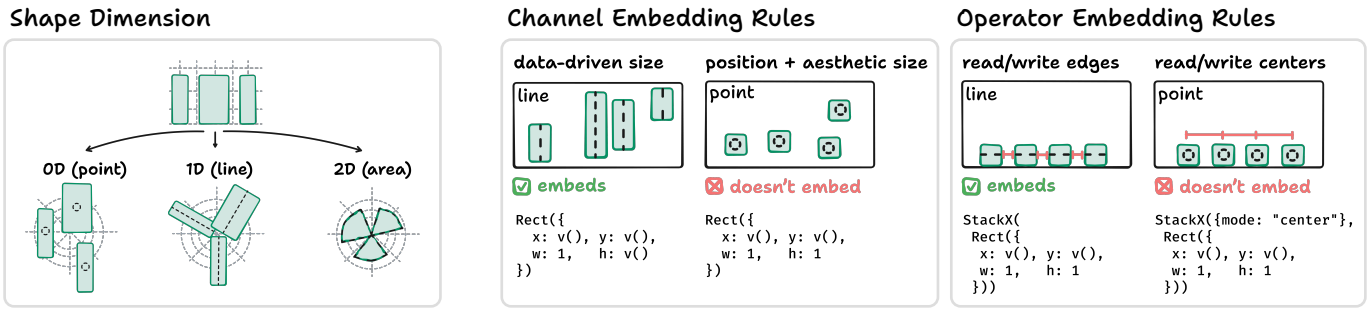


Fig. 6: (left) 0D, 1D, and 2D embeddings of a rectangle and how they are transformed by a polar space. (right) Embedding rules for shape channels and graphical operators. We use these rules to automatically infer the dimension of each shape.

(resp. height) of the shape to be embedded in the UNDERLYING space. This means it is subject to manipulation by the coordinate transform as shown in Fig. 6 (left). For example, shapes in a scatter plot do not have data-driven sizes, so they are not embedded and thus not warped by a coordinate transform. But `Rects` in a bar chart have data-driven heights, so they do get warped.

To maintain the declarative nature of the specification, a graphical operator cannot "de-embed" a shape's dimension if it has already been embedded. However, a graphical operator may further embed a shape. For example, the `StackX` operator reads and writes the horizontal edges of its children, and in doing so it embeds the widths of its children in UNDERLYING space. On the other hand, if `StackX` is used in `"center"` mode, then it only reads and writes the horizontal centers of each child, so it does not embed their widths.

Graphical operators participate in shape embedding so that their spatial arrangements will be preserved by coordinate transformations. For example, consider the `StackXs` in Fig. 6 (right). If `Rects` are spaced horizontally edge-to-edge, then we'd expect this spacing to be preserved by a polar transformation. To do so, we have to embed the width of each `Rect`. If their widths were not embedded, the `Rects` would appear as points to the UNDERLYING space. The spacing between these points is larger than the edge-to-edge spacing between the `Rects`, and so the polar-transformed `Rects` would appear farther apart from each other than they should be.

While this approach to shape embedding covers many common cases, there are some it does not currently support. For example, although a bubble chart is a scatter plot with data-driven radii, the widths and heights of the circles should not be embedded, since they are typically a different kind of data than the position data. Similarly, in a dual axis chart, the UNDERLYING space has a more complicated structure that is just beyond our current approach to shape embedding. We suspect that a more strongly typed encoding model incorporating units of measure could support these more challenging cases.

6 IMPLEMENTATION

GoFish is based heavily on Bluefish, a diagramming library [33]. We briefly describe graphical operator layout (Sec. 6.2), but it mostly follows the implementation in Bluefish. However, Bluefish only deals with UNDERLYING space. To adapt it to statistical graphics, we had to add recursive scale resolution (Sec. 6.1) and coordinate transforms (Sec. 6.3).

Like Bluefish, GoFish's primitives produce a tree-structured scenegraph consisting of standard nodes and ref nodes that act as proxies for the nodes they reference. GoFish's runtime extends the UI layout architecture approach of local constraint propagation. After resolving `Ref` selections, we perform three passes over the scenegraph: resolve size scales (Sec. 6.1); layout shapes in UNDERLYING space using graphical operators (Sec. 6.2); and apply coordinate transforms and aesthetic properties to map shapes to DISPLAY space (Sec. 6.3).

6.1 Recursive Size Scale Resolution

Scale resolution is an important part of GoG systems, because it means that designers don't have to specify scales manually most of the time. Scales are often merged across subplots in a facet or among marks overlaid on a shared frame. Custom layouts like `d3-tree` and `d3-sankey` also perform scale resolution internally. However, existing approaches to scale resolution assume that facets or stacks can nest only finitely many times. For example, facets in Vega-Lite and stack data transforms in Observable Plot are not nestable. In GoFish, we place no such restrictions on the `Stack` operator. But this makes scale resolution much harder.

While some scales like colors and positions are easy to resolve with arbitrarily nested operators, size scales are particularly challenging, because they often depend on the output canvas size, data values, and aesthetic values. For example, suppose we have a collection of data, d_{ij}, that we want to visualize in a stacked bar chart. To compute the height scale factor, `sf`, that we will apply to each d_{ij}, we need to solve the following equation. Given a frame height, `height`, and an inter-bar spacing of `spacing` pixels,

$$\text{height} = \max_i \left(\sum_j (\text{sf} \cdot d_{ij} + \text{spacing}) - \text{spacing} \right)$$

Simple compositions like these can be solved in closed form, but this is not true in general. To resolve scales for arbitrary compositions of operators, we first observe that the formulas that appear on the right hand side are always monotonically increasing in `sf`. That is, when `sf` increases, the height increases, too. As a result, we can binary search on `sf` to find a value that produces the desired output `height`. This approach is linear in the size of the scenegraph, because we do $O(1)$ work at each node to compute the height for a given scale factor. The running time thus depends on finding good candidate bounds on the search to limit the number of iterations. In practice we can provide reasonable estimates using information about the size of the visualization and the data collection.

6.2 Graphical Operator Layout

In contrast to other systems like Charticulator [34] that use linear programming, GoFish relies on parent-child local propagation layout like Bluefish [33]. Because of its simplicity, this architecture is very expressive, used by all major UI layout engines (including web, desktop, and mobile), and performs only constant work for each node in the scenegraph. In parent-child local propagation, a parent component sends each of its children a width and height they want the child to be. The child then performs its own layout before reporting its width and height (as well as its position if it was set during the child's layout). In addition to this conventional information, we also pass computed scale factors to children, which they may use to compute their size.

6.3 Coordinate Transform Layout and Rendering

During layout, a coordinate transform flattens its subtree hierarchy completely, accumulating nested transforms, erasing graphical operators, and producing a flat list of child shapes to be rendered. During rendering, each shape uses its coordinate transform context, and information about which dimensions are embedded, to determine how to render to the screen. We use SVG as our rendering target due to its simplicity and ease of debugging, but since all layout and DOM output is controlled by GoFish, we could in principle target other backends.

7 EXAMPLE GALLERY

GoFish provides a very expressive, yet structured design space. To test the limits of GoFish's composition, we constructed an example gallery of 30 graphics, which can be found at `gofish.graphics`. We highlight two clusters of examples we found particularly interesting. First, we replicated several highly nested charts that previously required either

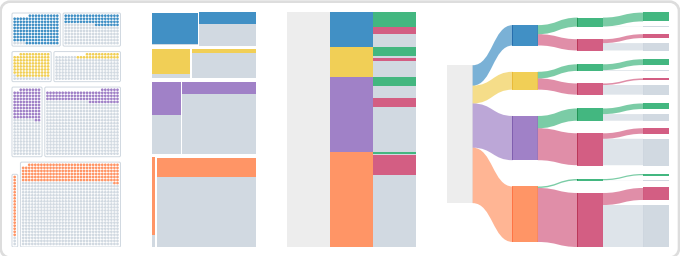


Fig. 7: Different visualizations of the Titanic dataset. A nested waffle chart using `Enclose` for borders, similar to unit visualizations produced by the Atom grammar [31]. A nested mosaic based on Hofmann's [21] and those produced by productplots [49]. An icicle chart. A Sankey tree.

low-level toolkits or specialized grammars to construct: nested waffles, nested mosaics, icicles, and Sankey trees (Sec. 7.1). By separating shapes and graphical operators, GoFish allows users to combine shapes and spatial arrangements independently. Second, we demonstrate how GoFish's expressiveness gives users more ability to express their personality. By changing shapes, we can switch the style of a visualization from scientific, to artistic, to whimsical (Sec. 7.2). Finally, we summarize the limitations of GoFish we discovered through building our examples (Sec. 7.3).

7.1 Waffles to Nested Mosaics, Icicles to Node-Link Trees

To push the limits of GoFish's compositional abilities, we recreate several highly nested charts to visualize the dataset of passengers who perished in the Titanic disaster (Fig. 7). Nested waffle and nested mosaic plots are very similar, but currently require different specialized grammars to construct. For example, the Atom grammar supports nested waffle charts [31] and productplots supports nested mosaics, but neither grammar supports the other kind of chart. In GoFish, similar to our switch from a stacked bar chart to a waffle chart (Sec. 3.2), the specifications of these two charts are very similar. We create nested `Stacks` to represent successive groupings by "class" and "sex." The nested waffle bottoms out as a `StackY` of a collection of `StackXs` of `Ellipses` surrounded by an `Enclose`. The nested mosaic bottoms out as a `StackY` of `Rects`. The widths and heights of the nested mosaic chart are determined by normalizing survival counts as detailed by Hofmann [21]. The waffle chart uses a normalized survival count to determine the number of `Ellipses` per `StackX`.

Tree visualizations are supported by specialized grammars like GoTree [25]. We recreate some trees in GoFish. The icicle chart is created similarly to the nested mosaic chart by nesting `StackXs` and `StackYs`. However, in this chart, each `StackX` consists of a `Rect` representing the sum of a particular group and the remaining subvisualization to the right of that `Rect`. To turn the icicle chart into a Sankey tree, we add horizontal and vertical spacing to the `Stacks`, split each `Rect` into a `StackY` of `Rects`, and then connect neighboring `Rects` with `ConnectX` operators. A Sankey tree is not representable in GoTree, because it provides only line-like connectors, not intervals like `ConnectX`. With explicit graphical operators, GoFish makes connections between these related chart types more apparent. However, the tradeoff is that GoFish specifications are more verbose than restricted, domain-specific grammars.

7.2 Scatter Pies and Flowers and Balloons… Oh My!

Scatterpies are often considered ineffective, a balloon chart may be cast aside as unserious, yet the flower chart is heralded as an impactful data art piece [41]. Despite the differences in our attitudes towards these charts, their specifications are very similar. Conventional grammars often aim to avoid charts the community deems ineffective, like scatterpies, but at the same time they hope to support impactful charts like the flowers. Not only is effectiveness context-specific, as evidenced by the popularity of scatterpies among marine biologists, but we argue that it is not possible to build a compositional grammar like the GoG or GoFish that supports one while rejecting the other.

Figure 8 provides images and GoFish specifications for a scatterpie, a flower chart, and a balloon chart. To create a scatterpie, we create an outer `Frame` that contains one inner `Frame` per `sample` in our dataset. This inner `Frame` `Polar` maps a horizontal stack of rectangles to create

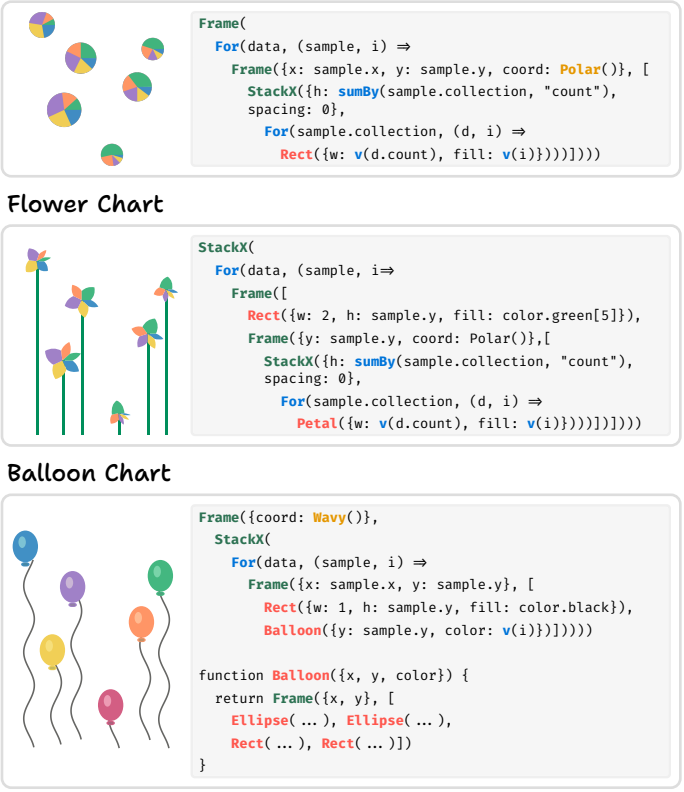


Fig. 8: A scatterpie, flower chart, and balloon chart. Their specifications are closely related. The flower chart replaces the scatterpie's `Rect` with a custom `Petal` and adds a stem. The balloon chart replaces the flower with a custom `Balloon` element and adds a `Wavy` coordinate transform.

pie chart. By replacing the `Rect` shape with a custom `Petal` shape and adding green `Rects` for stems, we can create a flower chart reminiscent of the OECD Better Life Index visualization [41]. Our petal is inspired by Jeremi Stucki's reimplementation of the flowers[2]. While this specification is more composable than a typical GoG specification, it still require GoFish's low-level API. We discuss this further in Sec. 7.3. By replacing the inner `Frame` with a custom `Balloon` element (this time created as a composition purely of GoFish `Rect` and `Ellipse` shapes), changing the color of the stem, and applying a `Wavy` transform to the outer `Frame`, we can create a whimsical balloon chart.

7.3 Limitations

Embedding aesthetic dimensions. While we embed aesthetic dimensions when they are used by graphical operators, this can lead to confusing behavior. Embedded aesthetic values refer to UNDERLYING space while non-embedded values refer to DISPLAY space. Since a graphical operator can change the embedding mode automatically, this creates a hidden dependency between graphical operators and shape embedding that can be surprising. See Fig. 9. While we believe our embedding properties are correct, this spooky action at a distance suggests that our current approach to embedding inference may be too aggressive. We wonder if lightweight user annotations for embedding aesthetic values could alleviate some of this confusion.

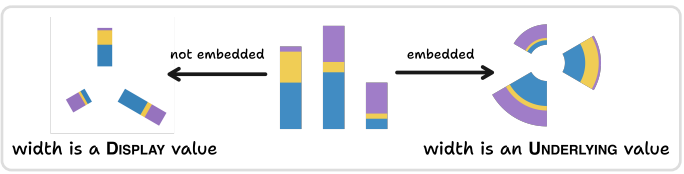


Fig. 9: Shape embedding can implicitly change the meaning of aesthetic channels as they switch from DISPLAY to UNDERLYING space.

[2] http://bl.ocks.org/herrstucki/6199768

Scale-aware operators. Our approach to recursive size scale resolution currently only works for data-driven shape sizes. However, the nested mosaic chart required setting data-driven sizes for intermediate `Stacks`. Similarly, unlike specialized `waffle` mark implementations, our implementation of waffle charts cannot compute the number of items per row automatically. The Atom grammar solved similar issues using scale resolution that terminates at intermediate nodes and by letting operator layout depend on scales. We could implement similar functionality by surfacing scale constraints during the layout process.

Axes and legends. Our current implementation supports one global discrete color scale, one x-axis, and one y-axis. While building our example gallery, we sought to match existing GoG library axis and legend designs. However, we noticed that charts using center-alignment, like streamgraphs, or stacks with non-zero spacing do not typically have axes even though they use data-driven encodings. These charts are easy to specify in GoFish and generate complex internal scales (Sec. 6.1). We intend to refine our theory of scales to create a more robust and expressive axis and legend model in future iterations of GoFish.

Algorithmic layout. To emphasize GoFish's compositional power, we did not explore custom layouts like tidy-tree or Sankey. However, these are natural next steps to investigate. As the GoFish layout architecture isn't based on a constraint solver, these layouts could be added as custom graphical operators. Based on a brief investigation of D3′s implementations, we suspect there is additional structure in these layouts that could be integrated more deeply within GoFish, including scale resolution calculations and additional graphical operators (e.g., alignment and distribution) that many of these layouts encapsulate.

Extensible elements and operators. While simple function abstractions over collections of shapes and operators is possible (Fig. 8), we noticed limits to this extensibility when dealing with more complex compositions. For example, we used the waffle stacking pattern twice, once in the walkthrough and again in the example gallery. It would be natural to try to create a custom `waffle` operator by composing GoFish primitives. However, in doing so, we find ourselves wanting some of the affordances of the GoG, like the pattern of a mark taking a dataset and field encodings rather than the GoFish pattern of raw data values. We suspect a framework like Encodable [52], which wraps low-level shapes to create GoG-style ones, could be adapted to help GoFish users create their own custom chart templates by composing GoFish's primitive shapes and operators.

Editing viscosity. Through building examples and gathering informal feedback, we found that the highly nested structure of GoFish specifications, while expressive, can make editing large specifications difficult as it is easy to get lost in scopes and parentheses. Moreover, most of our specifications have linear structures that don't benefit from the branching afforded by nesting. We also noticed that graphical operators are often combined with the same data operators. For example, `Stack` and `groupBy` almost always appear together. Based on comparisons to GoG implementations and early feedback on GoFish's design, we have started prototyping a lighter-weight alternative syntax that removes the need for callbacks and nesting in most cases, and that may be more familiar to GoG users. For example, the ribbon chart in Fig. 4 could be written like so:

```
rect(sf, {w: 16, h: "count", fill: "species"})
  .stackY("species", {spacing: 0, sortBy: "count"})
  .stackX("lake", {spacing: 64})
  .connectX("species", {opacity: 0.8})
```

8 Comparison to Mascot

Mascot [26] is a recent visualization grammar that, like GoFish, separates marks into shapes and operators. However, whereas GoFish draws its inspiration from the ideas of Wilkinson, Bertin, and Tversky — and thus more naturally integrates and extends the GoG — Mascot offers a more notable departure from this tradition by drawing inspiration from direct manipulation graphics systems such as Adobe Illustrator as well as interactive systems like ArcGIS. These different lineages result in contrasting language designs with several important consequences for how charts are constructed with each system.

```
let scn = msc.scene()
let rect = scn.mark("rect", {top:0, left:0, width:32, height:300,
fillColor:"#fff"})
let sf = ...
let lakes = scn.repeat(rect, sf, {attribute: "lake"})
lakes.layout = msc.layout("grid", {numRows: 1, colGap: 8})
let {newMark} = scn.divide(rect, sf, {orientation: "vertical",
attribute: "species"})
scn.encode(newMark, {attribute: "count", channel: "height"})
scn.encode(newMark, {attribute: "species", channel: "fillColor"})
```

Listing 2: A Mascot specification of the stacked bar chart from Sec. 3.2.

8.1 Mutable vs. Declarative

Mascot grew out of Data Illustrator's intermediate representation [28] and, as a result, its primitives are designed to reflect the procedural steps a user would perform in Data Illustrator's graphical interface. For instance, as the Mascot specification in Lst. 2 shows, to produce a stacked bar chart, a user first creates a single `rect` mark with a given position, size, and color; `repeats` the rectangle for every unique `lake` in the dataset, `sf`; and lays the resultant rectangles in a row. To stack the rectangles, the user `divides` each rectangle vertically, one for each `species`; and then they encode `count` to each divided rectangle's height and `species` to color. In contrast, GoFish specifications are designed to reflect a chart's ultimate visual structure. For instance, to produce the same stacked bar chart, a GoFish user would author the specification presented in Sec. 3.2: a user nests rectangles (`Rect`) in vertical stacks (`StackY`) within horizontal stacks (`StackX`).

As the Mascot authors argue, one of the key advantages of its stepwise procedural approach is the ability for visualization authors to "inspect and debug intermediate visualization states" [27]. However, as we demonstrate in Sec. 3, incremental stepwise authoring is also possible with GoFish's declarative approach. For instance, with our running example, a GoFish user can start with a single rectangle: `Rect({x: 0, y: 0, w: 32, h: 300, fill: "#fff"})`. Then successively bind it to data, and introduce each level of stacking one at a time (i.e., `Rect` → `For(lake, ...)` → `StackY` → `For(groupBy(...))` → `StackX`) where each step would produce a valid, renderable output.

A bigger difference, however, lies in how the (im)mutability of each system's primitives impacts a user's ability to reason about a chart specification after it is written. Mascot's primitives are mutable: each successive statement may overwrite previous statements. In Lst. 2, we can see that although the `rects` are initialized with a given size and position, these are overwritten by subsequent `layout`, `divide`, and `encode` operators. In contrast, and as we describe in Sec. 4.3.2, GoFish's graphical operators are immutable: they cannot further modify any properties of a shape that have been set. As a result, Mascot favors local modification while GoFish favors local reasoning. That is, although GoFish can support incremental authoring, certain steps may require changing the specification in multiple places (e.g., moving the height encoding of `300` from the `Rect` shape to the overall chart). Mascot's procedural steps enable more atomic and local edits, but pose challenges when an author needs to debug an unexpected output state. For instance, although the the initial `rect` mark in Lst. 2 defines a white fill color, a width, and a height, a user must scan the full specification to understand that the resultant values of these properties are determined in three different ways: the white fill does not affect the result, because it is overridden by a data-driven fill encoding; the defined width indeed reflects the width of the rendered bars; but the height corresponds to the height of the overall chart raher than the individual `rects`. In contrast, a user is able to reason about a GoFish specification in a more localized fashion: looking at the GoFish `Rect` call, they can see immediately that its width is constant while its height and fill are data-driven.

8.2 Operators

Mascot and GoFish share a similar collection of operators, but these operators differ significantly because of the systems' different influences and motivations.

Repeat vs. For. Mascot uses an editing style reminiscient of GARNET's prototype-instance model [30]. The `repeat` operator takes the `rect` "prototype" and creates copies with the same attributes but different, implicit data scopes. GoFish's `For`, on the other hand, uses function composition and a callback whose argument serves as an explicit data scope. By separating encodings from the initial mark definition, Mascot avoids the use of anonymous functions, a construct that can be confusing for new users. The tradeoff, however, is that data scopes become a hidden dependency. Mascot's `encode` function requires a user to search for previous operations in the spec that created the current data scope, making it harder to customize that scope.

Repeat and Divide vs. Stack. Notice that Mascot uses two operators, `repeat` and `divide`, for the bar chart while GoFish only uses one. The advantage of distinguishing between `repeat` and `divide` is that it better represents the structure of the graphic. The `divide` operator emphasizes that the vertical stacking in a stacked bar chart creates a part-to-whole relationship (where a whole rectangle has been divided into parts), whereas `repeat` emphasizes that the horizontal stacking places marks in a 1D grid of evenly sized regions. This may encourage users to think more explicitly about what spatial relationships they are encoding in their graphic. By opting to use a single `Stack` operator for both structures, GoFish provides a more consistent design with fewer edge cases (as the `divide` operator only applies to a few mark types) and only one stack-like operator to choose from. But it does not explicitly capture this part-to-whole structure. We are eager to explore whether our "edge" and "center" modes could be adapted to serve the roles of `divide` and 1D grid layout, respectively, while retaining `Stack`'s composability.

Divide vs. Coordinate Transforms. Mascot supports pie and donut charts using the same `divide` operator as for vertically stacking bars. In this more general setting, `divide` acts as a data-driven version of the Scissors and Knife tools in Adobe Illustrator [1, 28], which cut objects into pieces. For example, a Mascot user can `divide` a circle into data-driven pie wedges or nested rings. These tools provide a powerful physical metaphor for chart authoring, and they tie Mascot's primitives closely to Illustrator's. However, what they gain in closely mapping to Illustrator, they lose in generality. Mascot division is tied closely to the shape being divided, so it is not possible to create arbitrary coordinate systems in Mascot or take an existing chart and transform it to polar space. On the other hand, Mascot's APIs use more familiar terms. In GoFish, users must mentally translate Cartesian encoding and operator names like `h` and `StackX` to polar coordinates. And coordinate transformation in general may be less familiar to Adobe Illustrator users. One interesting path forward that blends the Mascot and GoFish approaches would be to let paths and polygons induce coordinate systems as can be done already in Inkscape and Illustrator.

Densify vs. Connect. Mascot constructs area and line charts using the `densify` operator whereas GoFish constructs them using `Connect`. Because `densify` modifies lines and areas directly, Mascot cannot easily connect shapes to each other. In a connected scatter plot, for example, a Mascot user must specify the scattered circles with parameters on the line mark:

```
scn.mark("line", {x1: 100, y1: 100, x2: 600, y2:500, strokeWidth:
2.5, strokeColor: "black", vxShape: "circle", vxRadius: 3.5,
vxFillColor: "white", vxStrokeColor: "black", vxStrokeWidth: 1});
let polyline = scn.densify(line, drivingShifts);
polyline.curveMode = "natural";
scn.encode(polyline.firstVertex, {attribute: "miles", channel: "x"});
scn.encode(polyline.firstVertex, {attribute: "gas", channel: "y"});
```

The `vxShape` parameter is restricted only to rectangle and circle marks and the other `vx` parameters are passed to the vertex mark. In GoFish a connected scatter plot uses `Connect` and a shape like `Ellipse`:

```
Frame([
  For(drivingShifts, (d) ⇒
    Ellipse({x: v(d.miles), y: v(d.gas), ...}).name(`${d.year}`)),
  ConnectX({ ... }, For(drivingShifts, (d) ⇒ Ref(`${d.year}`)))])
```

However, unlike Mascot's `line`, any primitive or composite element can be used to mark the points.

9 Towards an Open Formal Theory of Visualization

In this paper we presented GoFish, an expressive visualization grammar that formalizes Gestalt principles for creating a wide range of statistical graphics. By decomposing GoG marks into Bertin marks, data transforms, and graphical operators, GoFish formalizes more of the "deep grammatical structure" [50, p. x] of graphics, yielding a more expressive and composable language.

GoFish is part of a bigger shift in visualization: the *relational paradigm*. Researchers in both grammar design and graphical perception have begun to question the ability of marks and channels alone to satisfactorily build or analyze charts. McNutt identified an explosion of domain-specific grammars, many of which present domain-specific compositional operators inspired by Gestalt principles [29]. For example, GoTree enumerates Gestalt principles shared by many tree visualizations including rough analogs to `Stack`, `Connect`, and `Enclose`. As for graphical perception, Bertini et al. note that if we take channel effectiveness seriously, then "any chart that is not a dot plot or scatter plot is deficient and should be avoided," because position along a common axis is the most effective channel [7]. As Ziemkiewicz and Kosara argue, "we lack a model of visualization that can satisfyingly explain" how people derive meaning "purely from differences in shape and arrangement, rather than from real differences in data values" [55]. Recent work has explored how varying Gestalt principles and relative label placements affect the afforded messages of various charts [8, 14, 42]. GoFish contributes a formal model of Gestalt principles in visualization that could be used to more systematically explore new domain-specific grammar designs or conduct effectiveness studies.

The relational paradigm changes the kinds of research questions we can ask. For example, it prompts us to look for, and potentially formalize, Gestalt principles in interaction and animation. As identified by Pollock et al. [33], by re-examining existing animation grammars we can find many temporal Gestalt principles. The Gemini grammar's [23] `concat` and `sync` operators act as temporal spacing and alignment, respectively, which are the constituent Gestalt principles of `Stack`. In the CAST animation system [15, 16, 40], animations may be staged or nested, conveying information similar to a temporal `Enclose`. We wonder whether future animation grammars could benefit from a temporal `Connect` that animates an element along a path between two other elements. We can find potential analogs in interaction, too. Gestalt principles seem to manifest in interactions as user-driven groupings. For example, brushing can be thought of as user-driven enclosure, and generalized selections [19] allow users to select marks based on shared attributes like color and shape.

But motivated by the parallel shift in graphical perception research, perhaps the most important question we can ask about grammar design is "what role does effectiveness play in a world beyond marks and channels?" In the GoG, a fixed collection of marks and channels creates a finite, combinatorial design space. In an effort to help users make useful charts, many GoG libraries aim to make every point in this space "good." To do so, they make classically ineffective charts, like pie charts, inexpressible. For example, the pie chart issue in Vega-Lite remained open for four and a half years with the explanation that "[w]e have not spent much time on [this issue] because pie chart[s]… generally [lead] to perceptual issues" [3]. Similarly, Observable Plot's pie chart issue has been open for five years [2].

In contrast to the GoG, a finite set of shapes and *graphical operators* yield an infinite, yet structured, design space. We can't evaluate every point in this space, one-by-one, and determine if it's good or bad. It's not even clear that we should. Research has found that pie charts, for instance, can sometimes be more effective than other representations [17], and as we saw in Sec. 7.2, support for pie charts is a prerequisite for both scatterpies and flower charts. To move forward and continue to support users' needs, we need to return to Wilkinson. The GoG "is capable of producing some hideous graphics. There is nothing in its design to prevent its misuse" [50, p. 15]. Yet the structure of the grammar ensures that it "cannot produce a meaningless graphic," [50, p. 15]. That is, by ensuring that scales, geometry, and coordinate transforms are applied in the correct order, the GoG ensures a chart is meaningful. By managing scales, shape embeddings, and coordinate transforms for a much larger design space, we continue Wilkinson's aim of helping people create meaningful visualizations. A scatterpie helps a scientist understand plankton biodiversity, a flower chart invites engagement, a balloon chart cheers up a friend on a bad day.

Acknowledgments

Thanks to Yuri Vishnevsky for feedback on GoFish's syntax. Thanks to our anonymous reviewers for their thoughtful feedback on our grammar. This work was supported by the National Science Foundation under award #2341748 as well as an Alfred P. Sloan Fellowship.

References

[1] "Learn how to cut, divide, and trim objects into geometric and freehand shapes," 2025. https://helpx.adobe.com/illustrator/using/cutting-dividing-objects.html.

[2] "Arc mark (for donuts and pies). Issue #80," 2020. https://github.com/observablehq/plot/issues/80.

[3] "Arc Mark / Pie Chart. Issue #408," 2016. https://github.com/vega/vega-lite/issues/408.

[4] C. J. Ashjian, R. G. Campbell, C. Gelfman, P. Alatalo, and S. M. Elliott, "Mesozooplankton abundance and distribution in association with hydrography on Hanna Shoal, NE Chukchi Sea, during August 2012 and 2013," *Deep Sea Res. Part II*, vol. 144, pp. 21–36, 2017, doi: 10.1016/j.dsr2.2017.08.012.

[5] J. Bertin, *Semiology of Graphics*. University of Wisconsin press, 1983.

[6] J. Bertin, "La Graphique et le Traitement Graphique de l'Information," 1977.

[7] E. Bertini, M. Correll, and S. Franconeri, "Why Shouldn't All Charts Be Scatter Plots? Beyond Precision-Driven Visualizations," in *IEEE Trans. Visual Comput. Graphics*, 2020, pp. 206–210, doi: 10.1109/VIS47514.2020.00048.

[8] T. Boger and S. Franconeri, "Reading a graph is like reading a paragraph.," *J. Exp. Psychol. Gen.*, vol. 153, no. 7, p. 1699, 2024, doi: 10.1037/xge0001604.

[9] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-Driven Documents," *IEEE Trans. Visual Comput. Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011, doi: 10.1109/TVCG.2011.185.

[10] M. Bostock and P. Rivière, "Observable Plot." Observable, Inc., 2020.

[11] Y. Engelhardt and C. Richards, "A Building-Block Approach to the Diversity of Visualization Types–Each Type Expressed Visually, and as a Systematically Generated Sentence," in *International Conference on Theory and Application of Diagrams*, 2024, pp. 28–43, doi: 10.1007/978-3-031-71291-3_3.

[12] Y. Engelhardt and C. Richards, "A Universal Grammar for Specifying Visualization Types," in *Diagrammatic Representation and Inference: 12th International Conference, Diagrams 2021, Virtual, September 28–30, 2021, Proceedings 12*, 2021, pp. 395–403, doi: 10.1007/978-3-030-86062-2_40.

[13] J. von Engelhardt, *The Language of Graphics: A Framework for the Analysis of Syntax and Meaning in Maps, Charts and Diagrams*. Yuri Engelhardt, 2002.

[14] R. Fygenson, S. Franconeri, and E. Bertini, "The Arrangement of Marks Impacts Afforded Messages: Ordering, Partitioning, Spacing, and Coloring in Bar Charts," *IEEE Trans. Visual Comput. Graphics*, vol. 30, no. 1, pp. 1008–1018, 2023, doi: 10.1109/TVCG.2023.3326590.

[15] T. Ge, B. Lee, and Y. Wang, "CAST: Authoring Data-Driven Chart Animations," in *Proc. CHI*, 2021, pp. 1–15, doi: 10.1145/3411764.3445452.

[16] T. Ge, Y. Zhao, B. Lee, D. Ren, B. Chen, and Y. Wang, "Canis: A High-Level Language for Data-Driven Chart Animations," in *Comput. Graphics Forum*, 2020, vol. 39, no. 3, pp. 607–617, doi: 10.1111/cgf.14005.

[17] A. Hakone *et al.*, "PROACT: Iterative Design of a Patient-Centered Visualization for Effective Prostate Cancer Health Risk Communication," *IEEE Trans. Visual Comput. Graphics*, vol. 23, no. 1, pp. 601–610, 2016, doi: 10.1109/TVCG.2016.2598588.

[18] P. Hanrahan, "VizQL: A Language for Query, Analysis and Visualization," in *Proc. SIGMOD*, 2006, p. 721, doi: 10.1145/1142473.1142560.

[19] J. Heer, M. Agrawala, and W. Willett, "Generalized Selection via Interactive Query Relaxation," in *Proc. CHI*, 2008, pp. 959–968, doi: 10.1145/1357054.1357203.

[20] J. Hoffswell, A. Borning, and J. Heer, "SetCoLa: High-Level Constraints for Graph Layout," in *Comput. Graphics Forum*, 2018, vol. 37, no. 3, pp. 537–548, doi: 10.1111/cgf.13440.

[21] H. Hofmann, "Constructing and Reading Mosaicplots," *Comput. Stat. Data Anal.*, vol. 43, no. 4, pp. 565–580, 2003, doi: 10.1016/S0167-9473(02)00293-1.

[22] H. Kim *et al.*, "Cicero: A Declarative Grammar for Responsive Visualization," in *Proc. CHI*, 2022, pp. 1–15, doi: 10.1145/3491102.3517455.

[23] Y. Kim and J. Heer, "Gemini: A Grammar and Recommender System for Animated Transitions in Statistical Graphics," *IEEE Trans. Visual Comput. Graphics*, vol. 27, no. 2, pp. 485–494, 2020, doi: 10.1109/TVCG.2020.3030360.

[24] J. H. Larkin and H. A. Simon, "Why a Diagram is (Sometimes) Worth Ten Thousand Words," *Cognit. Sci.*, vol. 11, no. 1, pp. 65–100, 1987, doi: 10.1111/j.1551-6708.1987.tb00863.x.

[25] G. Li, M. Tian, Q. Xu, M. J. McGuffin, and X. Yuan, "Gotree: A Grammar of Tree Visualizations," in *Proc. CHI*, 2020, pp. 1–13, doi: 10.1145/3313831.3376297.

[26] Z. Liu, C. Chen, and J. Hooker, "Manipulable Semantic Components: a Computational Representation of Data Visualization Scenes," *IEEE Trans. Visual Comput. Graphics*, 2024, doi: 10.1109/TVCG.2024.3456296.

[27] Z. Liu, C. Chen, F. Morales, and Y. Zhao, "Atlas: Grammar-Based Procedural Generation of Data Visualizations," in *IEEE Trans. Visual Comput. Graphics*, 2021, pp. 171–175, doi: 10.1109/VIS49827.2021.9623315.

[28] Z. Liu *et al.*, "Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring," in *Proc. CHI*, 2018, pp. 1–13, doi: 10.1145/3173574.3173697.

[29] A. M. McNutt, "No Grammar to Rule Them All: A Survey of JSON-Style DSLs for Visualization," *IEEE Trans. Visual Comput. Graphics*, vol. 29, no. 1, pp. 160–170, 2022, doi: 10.1109/TVCG.2022.3209460.

[30] B. A. Myers *et al.*, "GARNET: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *Readings Hum. Comput. Interact*. Elsevier, pp. 357–371, 1995, doi: 10.1109/2.60882.

[31] D. Park, S. M. Drucker, R. Fernandez, and N. Elmqvist, "ATOM: A Grammar for Unit Visualizations," *IEEE Trans. Visual Comput. Graphics*, vol. 24, no. 12, pp. 3032–3043, 2017, doi: 10.1109/TVCG.2017.2785807.

[32] S. Pinker, "A Theory of Graph Comprehension," *Artificial intelligence and the future of testing*, vol. 73, p. 126, 1990, doi: 10.4324/9781315808178.

[33] J. Pollock, C. Mei, G. Huang, E. Evans, D. Jackson, and A. Satyanarayan, "Bluefish: Composing Diagrams with Declarative Relations," in *Proc. UIST*, 2024, pp. 1–21, doi: 10.1145/3654777.3676465.

[34] D. Ren, B. Lee, and M. Brehmer, "Charticulator: Interactive Construction of Bespoke Chart Layouts," *IEEE Trans. Visual Comput. Graphics*, vol. 25, no. 1, pp. 789–799, 2018, doi: 10.1109/TVCG.2018.2865158.

[35] C. Richards, "The Fundamental Design Variables of Diagramming," *Diagrammatic representation and reasoning*, pp. 85–102, 2002, doi: 10.1007/978-1-4471-0109-3.

[36] C. Richards, "Diagrammatics," 1984.

[37] A. Satyanarayan and J. Heer, "Lyra: An Interactive Visualization Design Environment," in *Comput. Graphics Forum*, 2014, vol. 33, no. 3, pp. 351–360, doi: 10.1111/cgf.12391.

[38] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-Lite: A Grammar of Interactive Graphics," *IEEE Trans. Visual Comput. Graphics*, vol. 23, no. 1, pp. 341–350, 2016, doi: 10.1109/TVCG.2016.2599030.

[39] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer, "Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization," *IEEE Trans. Visual Comput. Graphics*, vol. 22, no. 1, pp. 659–668, 2015, doi: 10.1109/TVCG.2015.2467091.

[40] Y. Shen, Y. Zhao, Y. Wang, T. Ge, H. Shi, and B. Lee, "Authoring Data-Driven Chart Animations," *IEEE Trans. Visual Comput. Graphics*, 2024, doi: 10.1109/TVCG.2024.3491504.

[41] M. Stefaner, D. Baur, and Raureif, "Truth & Beauty - OECD Better Life Index," 2013. https://truth-and-beauty.net/projects/oecd-better-life-index.

[42] C. Stokes, V. Setlur, B. Cogley, A. Satyanarayan, and M. A. Hearst, "Striking a Balance: Reader Takeaways and Preferences When Integrating Text and Charts," *IEEE Trans. Visual Comput. Graphics*, vol. 29, no. 1, pp. 1233–1243, 2022, doi: 10.1109/TVCG.2022.3209383.

[43] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A System for Query, Analysis, and Visualization of Multidimensional Databases," *Commun. ACM*, vol. 51, no. 11, pp. 75–84, 2008, doi: 10.1145/1400214.1400234.

[44] B. Tversky, "Spatial Schemas in Depictions," in *Spatial schemas and abstract thought*, 2001, vol. 79, p. 111, doi: 10.7551/mitpress/6392.003.0012.

[45] B. Tversky, J. Zacks, P. Lee, and J. Heiser, "Lines, Blobs, Crosses and Arrows: Diagrammatic Communication with Schematic Figures," in *Theory Appl. Diagrams*, 2000, pp. 221–230, doi: 10.1007/3-540-44590-0_21.

[46] J. Wagemans *et al.*, "A Century of Gestalt Psychology in Visual Perception: I. Perceptual Grouping and Figure-Ground Organization," *Psychol Bull*, vol. 138, no. 6, pp. 1172–1217, Nov. 2012, doi: 10.1037/a0029333.

[47] H. Wickham, "A Layered Grammar of Graphics," *J. Comput. Graphical Stat.*, vol. 19, no. 1, pp. 3–28, 2010, doi: 10.1198/jcgs.2009.07098.

[48] H. Wickham, G. Grolemund, and others, *R for Data Science*, vol. 2. O'Reilly Sebastopol, CA, 2017.

[49] H. Wickham and H. Hofmann, "Product Plots," *IEEE Trans. Visual Comput. Graphics*, vol. 17, no. 12, pp. 2223–2230, 2011, doi: 10.1109/TVCG.2011.227.

[50] L. Wilkinson, "The Grammar of Graphics," *Handb. Comput. Stat.: Concepts and Methods*. Springer, pp. 375–414, 2011, doi: 10.1007/978-3-642-21551-3_13.

[51] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations," *IEEE Trans. Visual Comput. Graphics*, vol. 22, no. 1, pp. 649–658, 2015, doi: 10.1109/TVCG.2015.2467191.

[52] K. Wongsuphasawat, "Encodable: Configurable Grammar for Visualization Components," in *IEEE Trans. Visual Comput. Graphics*, 2020, pp. 131–135, doi: 10.1109/VIS47514.2020.00033.

[53] G. Yu and S. Xu, "scatterpie R package," 2018. https://cran.r-project.org/web/packages/scatterpie/vignettes/scatterpie.html.

[54] J. Zacks and B. Tversky, "Bars and Lines: A Study of Graphic Communication," *Mem. Cognit.*, vol. 27, pp. 1073–1079, 1999, doi: 10.3758/BF03201236.

[55] C. Ziemkiewicz and R. Kosara, "Beyond Bertin: Seeing the Forest Despite the Trees," *IEEE Comput. Graphics Appl.*, vol. 30, no. 5, pp. 7–11, 2010, doi: 10.1109/MCG.2010.83.