






PiCCL: Data-Driven Composition of Bespoke Pictorial Charts

Haoyan Shi , Yunhai Wang , Junhao Chen , Chenglong Wang , Bongshin Lee 

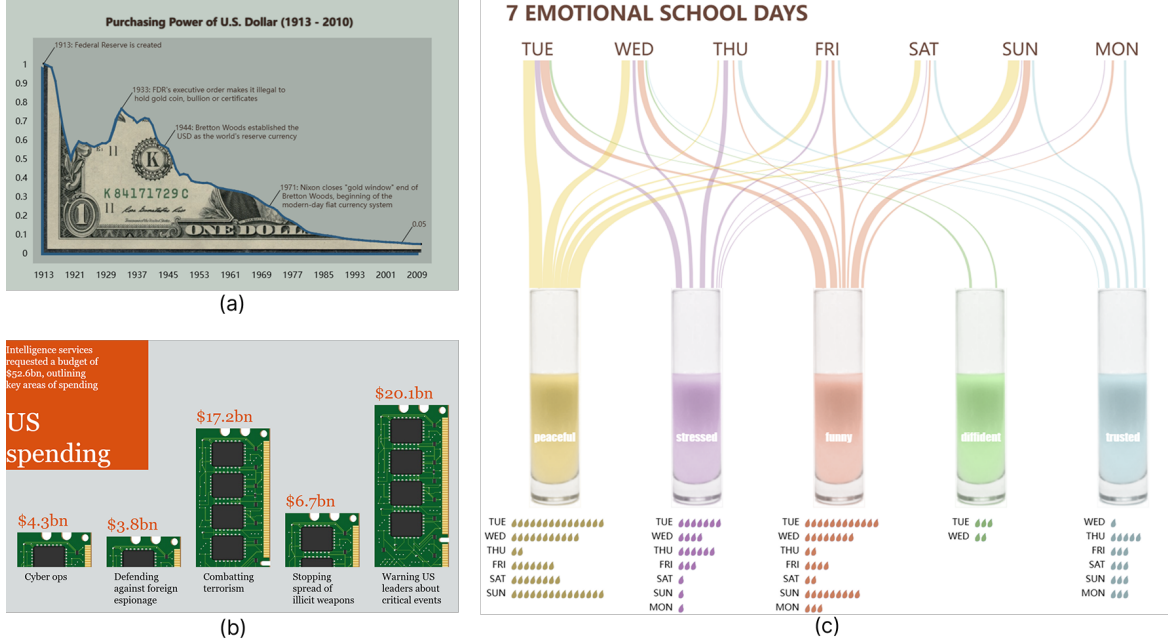


Fig. 1: Reproduction of the three bespoke pictorial charts using our library, piccl.js. (a) A line chart illustrating the trend of the US dollar's purchasing power, integrated with an image of a US dollar bill (<https://tinyurl.com/43w6c22b>); (b) A bar chart visualizing the US budget allocation for various intelligence services, where each bar is represented with a chip image (<https://tinyurl.com/589z4jec>); and (c) A nested chart depicting the relationship between five emotions and each day of the week (<https://tinyurl.com/23eu89tk>).

Abstract—We present PiCCL (Pictorial Chart Composition Language), a new language that enables users to easily create pictorial charts using a set of simple operators. To support systematic construction while addressing the main challenge of expressive pictorial chart authoring—manual composition and fine-tuning of visual properties—PiCCL introduces a parametric representation that integrates data-driven chart generation with graphical composition. It also employs a lazy data-binding mechanism that automatically synthesizes charts. PiCCL is grounded in a comprehensive analysis of real-world pictorial chart examples. We describe PiCCL's design and its implementation as piccl.js, a JavaScript-based library. To evaluate PiCCL, we showcase a gallery that demonstrates its expressiveness and report findings from a user study assessing the usability of piccl.js. We conclude with a discussion of PiCCL's limitations and potential, as well as future research directions.

Index Terms—pictorial charts, data-driven composition, chart composition, parametric representation

1 INTRODUCTION

Pictorial charts, which use symbols or icons to represent data, enhance engagement [31], memorability [4], and impact [1] by transforming abstract numbers into relatable imagery. They serve as a compelling and engaging medium for communicating complex information and have become a popular choice for narrative visualizations such as

infographics, data comics, and data videos, aimed at general audiences. However, constructing effective pictorial charts remains challenging, even for experienced information designers [15]. The main challenge in creating pictorial charts lies in their manual composition. Unlike standard charts, they rely on bespoke glyphs and bind data to custom components beyond traditional visual channels, making their creation cumbersome. Without a high-level abstraction, pictorial charts are typically constructed using two main approaches: (1) iterative authoring with flexible graphics software such as Adobe Illustrator, the most commonly used tool and (2) programming with low-level languages like D3. However, both are time-consuming, labor-intensive, and require significant expertise, posing a major barrier to broader adoption.

Several tools, such as Data-Driven Guides (DDG) [18], extend vector graphics with data binding capabilities, but authoring expressive pictorial charts with them remains labor-intensive. InfoNice [40] allows users to replace standard marks with pictorial objects, and Infoimages [9] supports embedding charts into thematic images. However, both rely on rigid workflows that limit creativity. ChartSpark [44] leverages text-to-image models to generate pictorial charts from data and textual prompts. While promising, it supports only a narrow set

- Haoyan Shi is with Shandong University, Qingdao, Shandong, China. Email: 202315173@mail.sdu.edu.cn.
- Junhao Chen and Yunhai Wang are with Renmin University of China, Beijing, China. Email: {chenjunhao11, wang.yh}@ruc.edu.cn.
- Chenglong Wang is with Microsoft Research, Redmond, Washington, United States. E-mail: chenglong.wang@microsoft.com.
- Bongshin Lee is with Yonsei University, Seoul, Republic of Korea. E-mail: b.lee@yonsei.ac.kr.
- Yunhai Wang is the co-corresponding author.

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.xxx/TVCG.201x.xxxxxx

of chart types and often produces suboptimal results due to source inconsistencies and model limitations.

Libraries like Highcharts [16] offer predefined templates for pictorial charts, but they are restricted to specific types such as Isotype or stacked bar charts. Vega [34] and Vega-Lite [33] allow image embedding but face similar constraints. These limitations stem largely from the Grammar of Graphics [42], which models chart construction as data transformation. However, pictorial chart authoring often requires fine-grained graphical operations (e.g., combining, resizing, and aligning custom elements) that are difficult to express in purely data-driven models. Manipulable Semantic Components (MSC) [19] takes a graphics-centric approach, treating graphical objects as first-class citizens and supporting data-driven manipulation. However, it lacks the high-level abstractions needed to capture the structure and semantics of complex pictorial charts.

In this work, we aim to enable the data-driven composition of custom graphical objects, addressing a key gap in the seamless and efficient creation of diverse, bespoke pictorial charts. Based on an in-depth analysis of a broad collection of examples, we systematically explore the design space of pictorial charts and introduce PiCCL (Pictorial Chart Composition Language), a novel language that extends MSC. PiCCL supports four core classes of operations: (1) parameterizing pictorial objects through visual channels such as position, color, height, and quantity, extending MSC’s encodings; (2) combining charts and pictorial elements at the glyph or chart level using image composition operators tailored to pictorial design; (3) generating data-driven variations of graphical objects to reflect underlying patterns, as supported in MSC; and (4) linking graphical objects to encode relationships, enhancing MSC with more explicit and structured visual connections. Furthermore, PiCCL supports nested pictorial charts, enabling the construction of rich, expressive, and semantically layered visualizations.

Our approach models each chart as a tree, called POT (Pictorial Object Tree), which serves as a parametric chart template that can be instantiated with different datasets. In a POT, leaf nodes represent a graphical object, including standard chart primitives (e.g., points, bars, and lines) as well as pictures. Internal node corresponds to an operator that defines how these objects are organized, categorized as either composition operators or generative operators. The former combines multiple graphical objects into complex structures, while the latter produces variations by generating groups of similar marks based on data attributes. To ensure structural consistency and effective arrangement of the graphical objects, these operations incorporate both constraints, such as alignment, connection, equality, and rotation, and layout mechanisms as parameters.

We implement PiCCL through a JavaScript-based prototype, `piccl.js` (<https://piccl.github.io/>), which provides intuitive APIs that allow users to construct pictorial charts using a set of expressive operators. At the core of our implementation is an instantiated process featuring a lazy data-binding mechanism, which automatically synthesizes charts that can render natively across platforms. To evaluate PiCCL, we conduct a twofold assessment: (1) we demonstrate PiCCL’s descriptive and expressive power, providing a gallery of pictorial charts created with `piccl.js` and (2) we report on a user study conducted to assess the usability of `piccl.js`.

In summary, the main contributions of this paper are as follows:

1. An empirical analysis of a large collection of pictorial chart examples we collected.
2. The design of PiCCL, a language that leverages glyph composition to represent pictorial charts; PiCCL takes advantage of the flexibility of graphical composition and data-driven nature of graphical marks from the grammar of graphics to support the construction of expressive pictorial charts.
3. An implementation of PiCCL as a JavaScript library, `piccl.js`, for authoring pictorial charts. We demonstrate PiCCL’s expressiveness power in describing and generating a wide variety of pictorial charts. In addition, a chart reproduction study with 10 participants showed that users can use `piccl.js` to create bespoke pictorial charts.

2 RELATED WORK

Pictorial Charts. The use of pictorial charts for data presentation has a long history. In the mid-1920s, Otto Neurath, Marie Neurath, and Gerd Arntz developed the ISOTYPE system [38], which used the repetition of unit pictographs to represent quantities in social and economic data. Their work laid the foundation for pictorial visualization techniques, which continue to be explored and refined today.

In recent decades, extensive research has examined the effectiveness of pictorial charts on information communication. For example, Borkin et al. [4] found that visualizations incorporating pictorial elements are more memorable than standard statistical charts, particularly when the images are semantically relevant to the data. Similarly, Bateman et al. [3] demonstrated that pictorial charts significantly enhance long-term recall, while maintaining comparable interpretation accuracy to standard charts. Beyond memorability, Haroz et al. [14] found that pictorial bar charts can enhance engagement and interpretation accuracy, but their benefits are most pronounced when representing small values. In contrast, Burns et al. [7] observed that pictographs depicting part-to-whole relationships have little effect on understandability but help users envision the topic and associate it with real-world entities.

These studies highlight the potential of expanding pictorial visualization design to enhance interpretability, emotional connection, and user experience. However, they have primarily focused on visual encoding and stylistic variations. Our work advances research on pictorial chart design and authoring by introducing a four class of composition operators that enable more structured, modular, and reusable pictorial chart designs, along with a library that facilitates their creation.

Interactive Chart Creation. To enable expressive chart design without programming, researchers have developed several interactive authoring tools. Early efforts focused on providing simple interactions for binding data to graphical mark primitives (e.g., Lyra [32] with drop zones and connectors and iVisDesigner [27] with drop-down menus and configuration panels). For more expressive mark creation, Data-Driven Guides (DDG) [18] enables users to draw custom marks in a vector graphics editor, using guides based on length, area, and position to encode data. InfoNice [40] also supports expressive mark design with icons, images, and text, but it is limited to ISOTYPE-style visualizations. To enable more sophisticated layouts, Data Illustrator [20] employs a lazy data binding approach with repeat and partition operators. Chartulator [28] further advances bespoke chart authoring with a constraint-based layout system, enabling the creation of reusable chart layouts. Unlike these tools, StructGraphics [37] takes a data-agnostic approach, allowing designers to freely draw marks and define layout constraints without requiring a specific dataset. Inspired by these tools, our work introduces a unified representation for diverse pictorial charts that is data-agnostic and supports a rich set of operators and layout constraints to compose highly expressive visualizations.

Given the benefits of pictorial charts, researchers have explored efficient methods for generating high-quality pictorial visualizations. Since pictorial objects are central to these charts, several efforts have focused on streamlining their design process. For example, DataQuilt [45] utilizes computer vision techniques to extract and transform real images into pictorial objects, simplifying their creation and integration. To automate the creation process, Qian et al. [25] proposed Retrieve-Then-Adapt, a method that generates proportion-related pictorial charts by first retrieving relevant examples from a library and generating an initial draft, followed by adjusting spatial relationships between visual elements to refine the composition. Chartreuse [11] extends this approach by extracting chart structures from examples and adapting them to new datasets, facilitating the reuse of pictorial designs while maintaining visual consistency and expressiveness. Infomages [9] retrieves an image containing the target visual subject and modifies it using filling, overlaying, replicating, and cutouting techniques to embed data. Vistylist [36] further enhances the pictorial chart design by allowing users to extract and apply styles (e.g., color, font, and icon) from existing examples to new datasets, offering greater flexibility. However, these techniques remain limited to specific data (such as proportional data) and visualization types (bar and pie charts) and often require manual adjustments to ensure accurate style transfer. By leveraging text-to-image generative

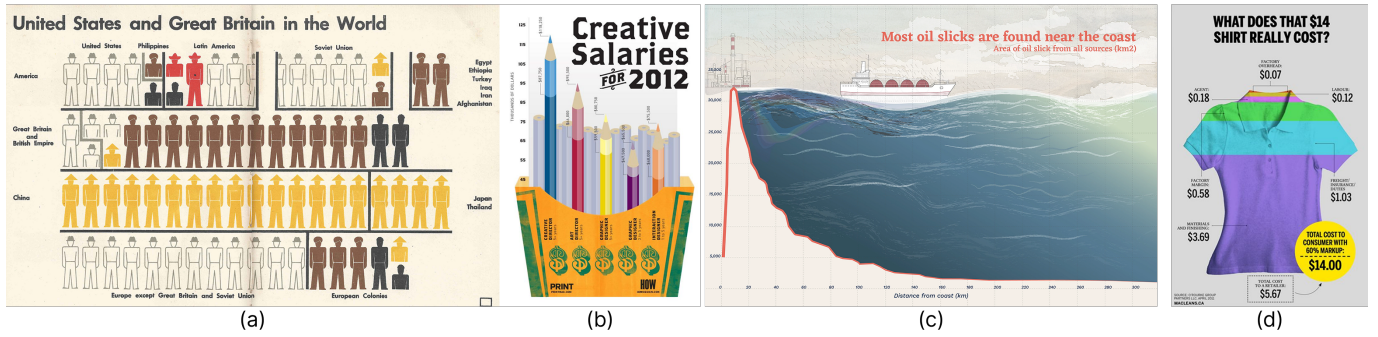


Fig. 2: Examples of pictorial charts from our dataset, showcasing various composition methods used for data encoding.

models to create pictorial charts, ChartSpark [44] provides more flexibility in interpreting text instructions and more expressiveness in visual representation. Yet, the lack of controllability in generative models makes it difficult to ensure faithful data binding and maintain a consistent style across independently generated visual elements. In contrast, our representation PiCCL offers transparency and flexibility, enabling users to customize and reuse marks for expressive chart creation.

Visualization Libraries and Languages. The grammar of graphics [41, 42] laid the foundation for visualization languages and toolkits, providing a structured approach to defining visualizations based on data, marks, and encoding rules. For example, Protovis [5] introduces an extensible toolkit for constructing visualizations using scene graphs, allowing users to define visualizations as a hierarchy of marks with properties derived from data. D3 [6] improves expressiveness and web compatibility by replacing Protovis’ scene graph abstractions with direct manipulation and inspection of the Document Object Model (DOM). To support efficient creation of interactive visualizations, Vega [34] and Vega-Lite [33] introduce a high-level declarative grammar that allows users to specify visualizations using a concise JSON syntax. These systems automatically generate essential components such as axes, legends, and scales. However, because they conceptualize visualization construction primarily as a data transformation process, they offer limited support for creating bespoke charts that require direct manipulation of graphical objects.

Recently, Liu et al. [19] proposed Manipulable Semantic Components (MSC), a framework implemented as the Mascot.js JavaScript library. MSC adopts a graphics-centric approach, representing visualization scene structures through a unified object model based on semantic components, including visual elements, encodings, algorithmic layouts, constraints, and view configurations, along with a set of operations for generating and modifying these structures. Similar to MSC, our PiCCL also follows a graphics-centric paradigm, enabling the manipulation of graphical elements through a range of operators. However, there are two key differences. First, PiCCL introduces a set of image composition operators specifically designed for combining pictorial objects. Second, it enhances the flexibility of certain operators in Mascot.js, such as supporting the linkage of heterogeneous graphical objects. Whereas Mascot.js operates directly on the scene graph, the elements within the scene graph are often fragmented and redundant, which hinders extensibility. To overcome this limitation, we introduce the Pictorial Object Tree (POT), an additional layer of abstraction above the scene graph. Rather than encoding a specific chart instance, the POT serves as an abstract representation of the chart composition process. It structures visualization scenes as hierarchically composed, parameterized visual marks while remaining agnostic to the underlying data.

Composition Tree. A composition tree is a concise and expressive representation widely used in modern CAD systems, exemplified by the Constructive Solid Geometry (CSG) tree. It encodes geometries as hierarchical structures, where boolean operators are recursively applied to primitive shapes [30]. Due to its compactness and flexibility, the CSG tree framework has been extensively adopted in scientific visualization research, offering an effective method for structuring and manipulating complex spatial data. For example, Chen and Tucker [8] extended the

CSG tree to volumetric data, introducing a set of operations on spatial objects to facilitate the exploration of both interior and exterior regions. Woodring and Shen [43] proposed a volume shader tree, allowing users to combine multiple data sources, enhancing the comparison of multivariate, time-varying volume data, and supporting dynamic exploration of complex datasets. Beyond volumetric visualization, Kalkofen et al. [17] employed a CSG-like structure to integrate spatial and contextual information, enabling the seamless fusion of focus and peripheral data in augmented reality environments.

Our work extends composition trees as a unifying framework for structuring graphical objects, visual encodings, and data-binding operations. With the introduction of pictorial objects, our PiCCL provides a parametric representation that enhances the flexibility and expressiveness of pictorial charts in visualization.

3 EMPIRICAL ANALYSIS FOR PICTORIAL CHART COMPOSITION

To analyze the composition of pictorial charts and systematically develop the operations to create them, we collected examples from diverse sources, including the Pictorial Visualization Dataset [39], the Infomages collection [9], and the Anthropographics list [2]. In addition, we retrieved pictorial chart images from design repositories (e.g., Pinterest) and open resources through web searches using keywords—infographic and pictorial chart. To ensure comprehensive coverage, we further expanded our dataset using image search engines to discover additional relevant visualizations. Next, we filtered the collected data based on the criteria established by Coelho and Mueller [9], but without excluding pictorial charts that use icons to convey data information. As a result, we finalized a dataset of 1776 valid pictorial charts.

We reviewed the composition operators used in interactive tools for designing pictorial charts and evaluated their effectiveness in abstractly capturing the construction of our collected examples. We first took MSC’s operator set as a foundation and extended it with operators from tools such as Chartulator [28], Infomage [9], and Chartreuse [11]. Next, two coders independently review and categorize each graphic according to the composition patterns. To evaluate the objectivity and consistency of the labeling process, we calculated inter-coder reliability using Krippendorff’s alpha, where a score of 0.8 or higher is generally considered acceptable in most research contexts [12].

Based on the results, we identified three major issues. First, we identified functional redundancies, for instance, MSC’s densify operator can be replicated by combining replicate (from MSC) with link (from Chartulator). As a result, we merged and refined the operator set, eventually organizing it into four core categories that form the basis of our language. Second, the overlay and cutout compositions in Infomages are overly restrictive for pictorial charts. They operate only at the chart level, while many pictorial charts apply these operations at the mark or glyph level for finer control (e.g., Figure 2b). Additionally, cutout is limited to simple intersections and cannot express more general operations, such as subtracting a chart from a background (e.g., Figure 2c). These constraints limit the expressiveness of pictorial chart composition. Last, in Chartreuse [11], six operations—morph, move, fix, recolor, repeat, and partition—primarily function at the mark level of the proportional charts and lack support for nested composition, limiting their ability to construct complex pictorial charts.

To address these issues, we developed a language with four classes of operations. We refined existing patterns, unified overlay and cutout into a flexible combination operator that supports both chart- and glyph-level composition, and introduced a linking operator to capture semantic and structural relationships. To the best of our knowledge, no existing visualization libraries or languages fully support all of these operations. Our language offers a unified approach that combines data binding with graphical composition, enabling a compact yet expressive library that captures a wide range of pictorial chart constructions in a consistent and reusable manner.

4 PICCL: A LANGUAGE OF PICTORIAL CHART COMPOSITION

Based on our study, we developed PiCCL, a new language for representing pictorial charts through compositional structure. As illustrated in Figure 3, a chart in PiCCL is constructed by binding data to visual objects, which are in turn built using geometry-aware composition operators. These visual objects are defined through three core concepts: a *mark*, a *glyph*, and a *collection*. A *mark* refers to a basic visual element, such as Picture, Rect, Line, Circle, Point, Text, Axis, or Legend. A *glyph* is a composite object formed by combining marks, other glyphs, or collections. A *collection* represents a group of homogeneous elements, each bound to a different data item.

Starting from basic marks, visual objects are constructed iteratively using four classes of operators: **mark encoding**, which binds data fields to marks; **mark combination**, which composes multiple visual objects; **mark generation**, which creates data-driven collections; and **object linking**, which connects objects into linked structures. These operators define a chart as a POT (Pictorial Object Tree), capturing its hierarchical and compositional structure.

```

chart ← bind(object*, data)
object ← mark | glyph | encodedObject | collection | linkObject
mark ← Picture | Rect | Circle | Line | Point | Text | Axis | Legend
glyph ← combine(object, object, constraint*)
encodedObject ← mapValue(object, channel, field, params)
               | repeat(object, n, layout*)
collection ← replicate(object, field, layout*)
             | divide(mark, field, layout*)
linkObject ← link(object, params)
combine ← Over | In | Out | Xor | Atop
constraint ← PointSnap | LineSnap | OrientMatch | LengthMatch
layout ← StackLayout | GridLayout | CirclePackingLayout | ...

```

Fig. 3: The syntax of PiCCL. Main building blocks of the chart, objects, are built of glyphs, encoded objects, collections, and links through four classes of composition operators. We use *item** to represent list.

4.1 Mark Encoding Operators

Each pictorial object is treated as a visual element, with geometric properties such as position, length, and color parameterized as visual channels to encode data fields. Specifically, we categorize mark encoding operations into two types: *value mapping* and *repeat*. The *value mapping* operator adjusts properties like position, size (length or area), and color or texture of pictorial marks to represent different data values. This is consistent with the visual encoding operator defined in MSC, but we introduce *part-aware resizing* and *structure-aware recoloring* to support length- and color-based visual encodings tailored specifically for pictorial objects. Unlike value mapping, the *repeat* operator duplicates pictorial marks to establish a one-to-many mapping from a single data value to multiple marks. This is commonly used in unit-based pictograms and proportional visualizations (e.g., Figure 2a), and is unique to our language.

The *repeat* operator is defined as a tuple:

$\text{repeat} := \langle \text{object}, n, \text{layout} \rangle$,

where *object* is the visual object to repeat, *n* is the data value specifying the number of repetitions, and *layout* defines the strategy for arranging the repeated elements. This operator generates a set of identical marks, which are treated as a collection in subsequent operations.

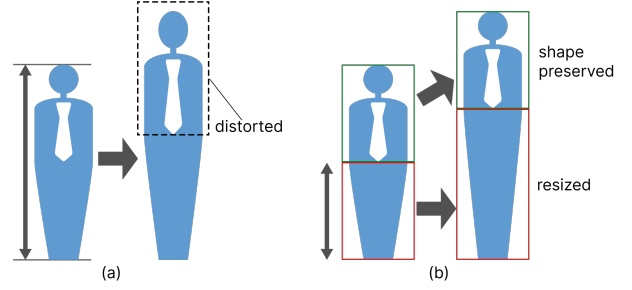


Fig. 4: Part-aware resizing of pictorial objects. (a) Resizing the entire chart distorts the top part highlighted by a dashed rectangle; (b) Resizing only the bottom section preserves the shapes of the other regions.

The *value mapping* operator uses a pictorial mark to encode data with various mark channels, including position, color, and size. It can be defined by a visual object *object*, a visual channel *channel*, a data attribute *field*, and optional parameters *params*:

$\text{mapValue} := \langle \text{object}, \text{channel}, \text{field}, \text{params} \rangle$,

where the scale is optionally included by *params*; if the user does not specify it, the system will automatically generate one. The content of *params* depends on the type of visual channel. In the following, we detail two commonly used channels: height and color.

Part-aware Resizing. Resizing a pictorial object composed of multiple distinct parts can lead to visual distortion. For example, the human icon shown in Figure 4 consists of two components: the upper body and the lower body, where each with a unique shape. Directly resizing the entire object may cause undesired deformations, such as the head becoming elliptical and the tie becoming over-stretched (Figure 4a). Inspired by dataQuilt [45], we introduce *part-aware resizing*, a technique that allows users to define a rectangular resizable region \mathbb{R} within an image to maintain visual consistency of key features during stretching. Users can manually specify this region to indicate the area that should deform smoothly, while the surrounding areas remain unchanged. This method preserves the visual integrity of pictorial marks while enabling flexible and controlled resizing. As illustrated in Figure 4b, resizing only the lower body ensures that the shapes of the top piston and bottom joint remain intact. The region \mathbb{R} is specified via the parameter *params*, and defaults to the full extent of *m* if not explicitly set by the user. Similar effects can sometimes be achieved by repeating a short or narrow pictorial object. However, this approach relies on the object being divisible into repeatable segments, which is not possible for objects with unequal widths, such as the bottom part in Figure 4.

Structure-aware Recoloring. When recoloring a mark, the parameters *params* specify the color scale used for encoding. For categorical data, each distinct value is mapped to a unique color from a predefined palette. For numerical data, colors are assigned by interpolating along a continuous colormap based on their magnitude.

Consider a pictorial object, such as the cup in Figure 1c, which uses color to encode emotion type. To apply the labeling color c_t to the object, we compute the final color C_f by compositing c_t with each pixel color C_s in the source object:

$$C_f = (H_t, S_t, L_t + (1 - L_t) \cdot L_s), \quad (1)$$

where, (H_t, S_t, L_t) is the HSL representation of the target labeling color c_t , and L_s is the source pixel's lightness. This formulation retains the object's structural shading while applying the desired hue and saturation.

4.2 Object Combination Operators

A glyph is constructed by combining standard chart elements, visual marks, or pictorial objects with additional pictorial components using a *combine* operation, which is unique to PiCCL. A pictorial chart can be seen as the final form of such a composed glyph. This compositional strategy enriches the semantic meaning of visualizations, enabling

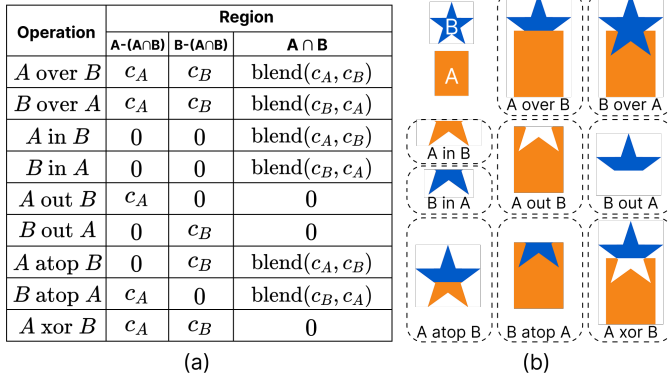


Fig. 5: (a) Different operations combine two marks, A and B, with corresponding colors c_A and c_B , to yield a resulting color for each region. (b) This example illustrates each operation using marks A and B, as shown in the top left.

expressive and visually engaging pictorial charts while preserving data accuracy. This operation can be applied both at the glyph level, where individual glyphs are modified, and at the chart level, where the entire chart is transformed. As a result, PiCCL supports flexible and nested composition at multiple levels of the chart.

Composition Operations. Composition operators are used to combine multiple objects into a single glyph. We consider object composition from two aspects: i) region calculation determines which areas of overlapping objects should be discarded and which should be retained; and ii) color blending defines how the colors of overlapping regions should be set. Following the composition patterns summarized by Porter and Duff [24], we provide nine types of operations for combining two marks A and B, where AB denotes the intersection region. As shown in Figure 5a, the *over* operation merges multiple marks into a single entity, including all regions covered by any of the shapes; the *in* operation retains only the overlapping region between two shapes; the *out* operation subtracts the overlapping area between two shapes, keeping only the part that belongs exclusively to one of them; the *atop* operation places one shape on top of another, but only retains the portion of the top shape that overlaps with the bottom shape; and the *xor* operation retains the regions that belong exclusively to one shape or the other, but discards any overlapping areas.

Blending Modes. For the intersection region AB, the final color is determined through color blending. By default, the blended color inherits the color of the overlaid pictorial object. Since Porter-Duff operators are limited to blending methods based solely on alpha values, we provide alternative blending modes commonly used in image editing software [10] such as darken, multiply, and color burn. These modes offer greater flexibility in visual representation and allow for enhanced customization of the chart’s appearance.

Equality Constraints. When performing composition operations, multiple marks are combined into a single glyph. A crucial aspect of this process is maintaining the correct relative positioning and size relationships between two source and target objects, *src* and *tgt*. To achieve this, we introduce four types of equality constraints as illustrated in Figure 6.

- **Point Snapping:** Each object selects a reference point *ref* in its local coordinate system, and then the two reference points are aligned in the global coordinate system with an optional offset. This method, shown in Figure 6a, ensures precise positioning of elements relative to each other:

$$tgt.ref = src.ref + \Delta, \quad (2)$$

where *src.ref* and *tgt.ref* represent the coordinates of two anchor points in the global coordinate systems, while Δ specifies the alignment offset.

- **Line Snapping:** Each object selects a reference line *refL*, and the two lines are constrained to be parallel in the global coordinate system

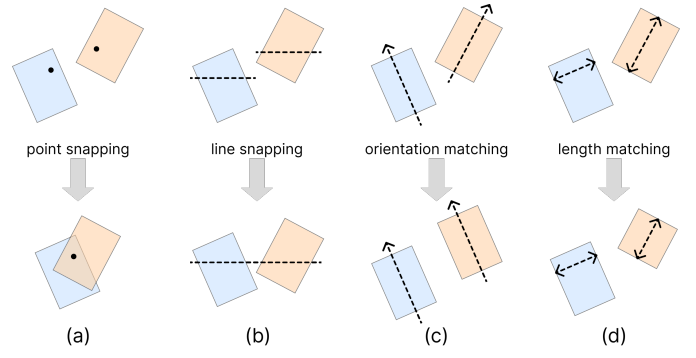


Fig. 6: Illustration of four types of constraints.

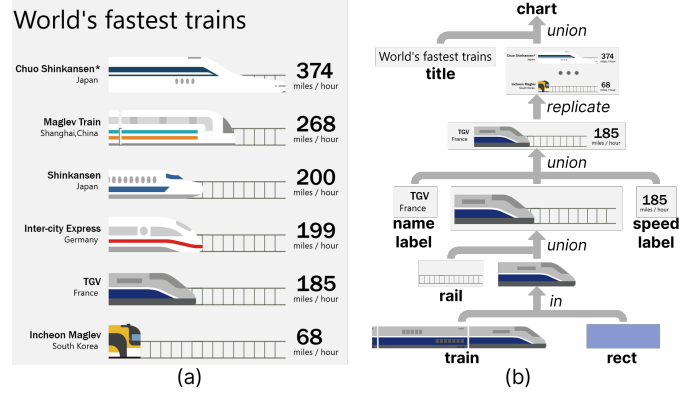


Fig. 7: An example of a pictorial chart (a), defined through a hierarchy of nested compositions (b).

with an optional separation distance. This constraint is defined as the one in Equation 2 but for lines (Figure 6b).

- **Orientation Matching:** To maintain a consistent angular relationship, reference lines selected from both objects are constrained to a fixed orientation difference. This ensures rotational alignment, where the lines remain parallel if the difference is zero; otherwise, they maintain a predefined angular displacement.
- **Length Matching:** This constraint, shown in Figure 6d, ensures that a specific length parameter of two elements remains equal.

With these constraints, the composition operations listed in Figure 5a can reliably preserve the intended spatial relationships, even when varying data parameters produce different geometries. Figure 7b shows an example where such constraints are applied. During the *in* operation between the train and the rectangle, two constraints are used: a length matching constraint ensures the train’s height matches that of the rectangle to avoid incomplete cropping, and a point snapping constraint aligns their bottom right corners to preserve the train’s front.

By default, the visual properties of a target object are not data-driven (e.g., fixed bar widths in bar charts). However, these properties may be influenced by multiple constraints and layouts, potentially causing conflicts. Similarly, for data-dependent properties, conflicts can arise between data encoding and other operations. To resolve them, we define a priority hierarchy among the three types of operators: encoding has the highest priority, followed by constraint-based composition operators, and then layout operators. For each property, only the highest-priority operator is applied to change this property, ensuring a consistent and predictable visual representation. If conflicts arise from multiple constraints, only the constraint defined by the earliest-specified operation is applied, providing a clear resolution mechanism.

Note that both point snapping and line snapping are supported by Charticulator [28], StructGraphics [37], and MSC [19], while length matching is unique to PiCCL. Orientation matching is only partially supported in StructGraphics, which allows rotating an element to a

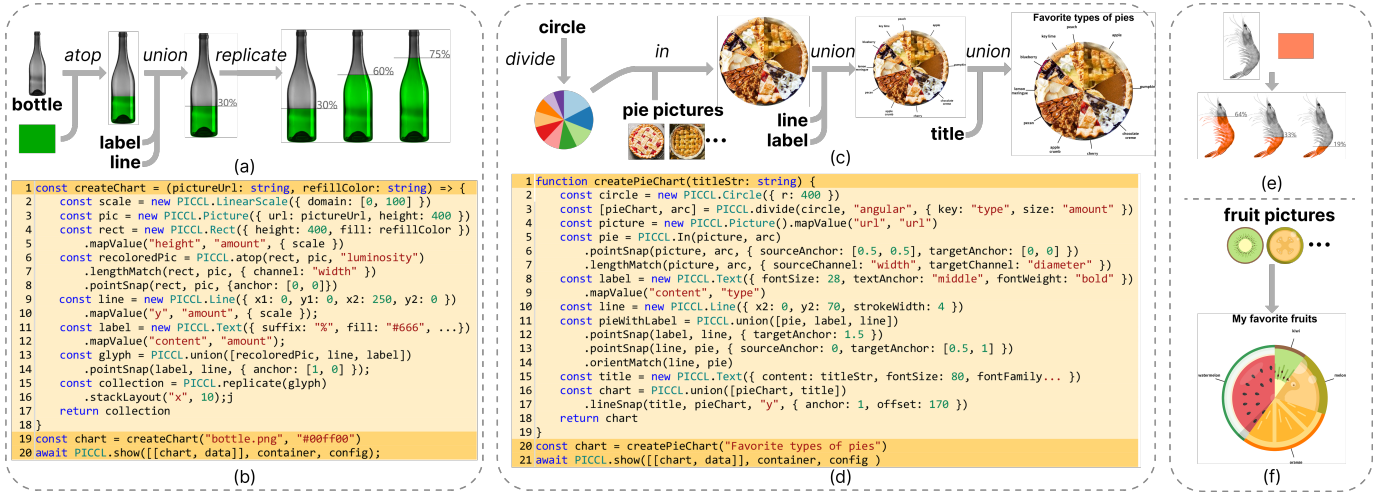


Fig. 8: Illustration of single-level combination for authoring a wine-themed pictorial bar chart (a) and a food-themed pictorial pie chart (c) using a sequence of operators, with corresponding piccl.js code snippets shown in (b) and (d). Replacing the pictorial object images in (a) and (c) results in the new theme-related bar chart (e) and pie chart (f).

fixed angle but does not support matching an angular displacement. Nonetheless, both length matching and orientation matching are essential for achieving structural alignment in pictorial charts, see the examples in Figure 7b and Figure 8c.

4.3 Object Generation Operators

Generative Operators create multiple similar marks or glyphs that vary in parameters to represent different data values. The two primary operators are *replicate* and *divide*, which correspond to those introduced in Data Illustrator [20] and MSC [19]. We strive to minimize the number of supported operations, ensuring a simplified learning process for users. As a result, additional generative operators introduced by MSC are not included, as some of them can be effectively replicated using our operations. For example, the *densify* operation can be achieved by using repeat to generate a point collection and then inserting a visual link mark between consecutive points to connect them. This approach allows for the construction of lines or area charts without requiring additional operators.

Operations. The *replicate* operation creates a collection of multiple pictorial objects, each mapped to a separate data item and encoded through visual channels such as size, color, or orientation. Unlike the *repeat* operation described in subsection 4.1, which produces identical duplicates, *replicate* binds each instance to data. For example, while each row in Figure 2a is generated using *repeat*, the overall set of rows is created through *replicate*. The resulting collection is taken as an object for further application of other operations. Figure 7b shows an example, where the collection of glyphs is combined with the title.

The *divide* operation segments a single pictorial object into parts, each representing a data subset (see Figure 2d). This is particularly effective for visualizing proportions within a single unit, transforming an image into a multi-part data container. For the marks generated by *divide*, users can apply any operation to each generated mark or the whole collection; see the example in Figure 8a.

Layouts. The layout method is introduced as a parameter within generative operators to automatically determine the positions of generated marks. Similar to MSC [19], we provide two types of layout algorithms: constraint-based layouts and algorithmic layouts. The former positions elements automatically by applying a set of predefined constraints, with common types including stack, grid, circle, and align layouts. If conflicts arise such as applying both align and stack layouts along the same X-axis, the system prioritizes the first constraint in the sequence to ensure consistency. In contrast, algorithmic layouts employ specialized algorithms such as circle packing [23] and force-directed layouts [21] to determine mark placement, providing greater flexibility for complex spatial arrangements that constraint-based methods struggle to handle.

4.4 Object Linking Operator

Beyond encoding data values through mark channels and mark quantity, we introduce a new type of operator, *link*, for representing networks (see Figure 1c), hierarchies, and cause-and-effect relationships. It establishes connections between pictorial marks/glyphs or between pictorial and standard marks/glyphs, defined as:

$$\text{link} := \langle m, (e_{\text{src}}, e_{\text{dst}}) \rangle,$$

where m is a pictorial object used to render the link, and $(e_{\text{src}}, e_{\text{dst}})$ denotes the source and target marks to be connected. This operation returns a link object that can be visualized as a straight line, a curve, or a band, depending on the nature of the relationship and the desired emphasis in the visualization. Its visual properties (e.g., color, thickness, and width) can be further utilized to encode relevant data attributes.

Since the source and target marks can be defined explicitly or implicitly, our representation supports both sequential links and data-driven links, similar to Chartulator [28], while extending to various types of visual marks for greater expressiveness. In contrast, MSC does not support this operation directly; instead, it treats links as additional marks and uses repeat operators to construct connections between elements. This approach restricts linking to objects within the same collection, failing to support cases like the one shown in Figure 1c.

4.5 Pictorial Object Tree

Based on these operators, we represent each chart as a parameterized tree structure, termed the Pictorial Object Tree (POT). In this tree, each leaf node corresponds to one or more graphical objects, referred to as semantic components [19]. The edges denote operators that define the relationships and transformations applied to the components, while the internal nodes represent intermediate results produced by applying these operations.

For example, Figure 7b depicts the POT of the corresponding pictorial chart, Figure 7a. Serving as a data-independent abstraction, the POT allows users to construct the visualization structure before binding data for instantiation. Once data is specified, a scene graph is generated to encode the spatial relationships among graphical objects. Furthermore, Figure 7b represents a hierarchy of nested operations to create Figure 7a, further demonstrating PicCL's flexibility and expressiveness.

5 PICCL.JS: A LIBRARY FOR PICTORIAL CHART CREATION

While our language PicCL is inspired by Mascot.js [19], we chose not to extend it directly for pictorial chart authoring. This is primarily because Mascot.js lacks an abstraction layer like our Pictorial Object Tree (POT), instead embedding the object model directly into the scene graph. While this design is effective for reverse engineering or editing

existing charts, it poses significant limitations for creating new ones. In particular, the tight coupling between the scene graph and specific data makes chart reuse and composition difficult.

To address these challenges, we developed a JavaScript prototype library called `piccl.js`, built around POT as an independent abstraction layer. During the construction phase, POT remains decoupled from data, resulting in a compact and low-redundancy representation. Only during the instantiation phase is the POT tree bound to the dataset, at which point the full scene graph is generated in a single pass. This approach eliminates the need for repeated modifications to the scene graph and significantly improves system flexibility and maintainability. All graphical objects, including pictures, are defined as JavaScript objects, with operations implemented as global functions. Encoding methods are specified as member functions in their corresponding graphical objects, while constraint methods are applied to the objects returned by composition functions, and layout methods are executed by repeat and data-driven generation functions. When multiple methods are invoked by an object, they are sequentially chained.

Scale Inference. Most existing tools like Chartulator [28] rely on a fixed viewport size and determine the scale in a top-down manner, where the constraints and scale are solved simultaneously. However, the generated scale depends solely on the data and the viewport, without accounting for the aspect ratio of the pictorial objects, often leading to visual distortion. To address this issue, we propose a bottom-up scale inference mechanism that aims to minimize distortion by inferring a suitable scale range based on the dimensions of the pictorial objects. In this process, the actual size and position at each level are determined by the transformed results of lower-level components under the current scale. Additionally, the viewport size dynamically adjusts based on the number of data items, preserving both visual faithfulness and flexibility across different data distributions. For example, in size encoding, we determine the mark size by mapping one aggregate data value proportionally to the original glyph size, with options such as the maximum or average value. When using position to represent categorical data, elements are arranged at equal distances while maintaining their original size, rather than allocating space for each category based on a fixed viewport size.

Instantiation Process. Our library instantiates chart specifications through a three-stage pipeline: Generate – Solve – Render.

In the Generate phase, the system transforms the PiCCL to generate a scene graph based on the provided data. This process involves two key steps. First, it executes generative operators in a top-down manner to dynamically create scene graph elements and assign data to child nodes. Then, it maps data values to visual properties using encoding operators, where data values are translated into the visual attributes of scene graph elements. This phase bridges the gap between the abstract tree structure and concrete visual instances, preparing the scene for constraint solving, layout computation, and final rendering.

In the Solve phase, the library determines the exact attribute values of elements within the scene graph. Layout rules are automatically converted into constraints; for instance, a stack layout is translated into a series of line-snapping constraints, ensuring that the end of each element aligns with the start of the next. Once all constraints are established, the system solves them to compute the spatial parameters of each element. For algorithmic layouts such as circle packing, which cannot be fully expressed through constraints, dedicated layout algorithms are applied separately to compute the final spatial arrangement.

In the Render phase, the library compiles the scene graph into the target output format to generate the final image. It currently supports PNG and SVG outputs. For SVG, basic graphical elements are mapped to corresponding SVG geometric primitives, while various operators are translated into SVG groups (`<g>`), masks (`<mask>`), and filters (`<filter>`). This approach enriches SVG with semantic components, similar to MSC [19], enabling visualization deconstruction and reuse.

Custom Composition. The `<g>` tag provides basic stacking functionality, which corresponds to the *over* operator in PiCCL, but it cannot represent more advanced compositions. Although SVG’s `<feComposite>` filter supports Porter-Duff operations, it does not allow for custom

blend modes. To enable flexible, operator-specific composition, we use additional SVG features such as `<mask>` and `<feBlend>`.

For each element, we identify overlapping and non-overlapping regions. The overlapping area is masked using the alpha channel of the other element, while the non-overlapping part is extracted using `<feColorMatrix>`. Depending on the operator, we retain or discard specific regions. For example, *in* keeps only the overlap, while *out* discards it entirely. If the operator retains the overlap, we apply the blend mode using `<feBlend>`. Otherwise, simple grouping with `<g>` is used.

6 EVALUATION

In this section, we present two forms of evaluation. After describing PiCCL’s descriptive and expressive power, we report a user study conducted to assess the usability of `piccl.js`.

6.1 Evaluation of PiCCL

Descriptive Power. To evaluate its descriptive power, we conducted a comprehensive analysis of 1,776 collected pictorial charts, mapping them to four classes of operators: mark parameterization, object combination, object generation, and object linking. While all charts utilize mark channels for data encoding, our labeling specifically focuses on parameterization applied to pictorial marks, excluding standard marks. The detailed mappings of all charts are provided on the website (<https://piccl.github.io/dataset>). The results reveals that all pictorial charts involve generative operators to generate data-driven mark variations, 85% use parameterization, 48% involve combination, and 4% require linking. Although this distribution analysis does not directly quantify expressive power, it illustrates the diversity and frequency of operators needed for real-world pictorial chart designs, thereby reflecting the expressive breadth of `piccl.js`.

Expressive Power. To demonstrate the expressive power of PiCCL, we produced a variety of bespoke pictorial charts. Figure 1 shows three examples from our gallery, and more examples are available at <https://piccl.github.io/gallery>. PiCCL constructs pictorial charts using mark encoding, object combination, object generation operators, and optional object linking operators. These elements are organized as a POT, enabling the creation of diverse pictorial charts through the combination, reuse, and substitution of its components. The combination operators can be applied to marks, glyphs, or collections, and can operate at both single and nested levels. Below, we demonstrate how new pictorial charts can be created by applying different combinations of these components.

Single-level Combination. The input elements for the combination operators can be any type of visual mark, such as pictorial marks, rectangles, labels, glyphs, or collections. Figure 8a shows a pictorial chart created with `piccl.js`, as illustrated in Figure 8b. A wine bottle image is combined with a rectangle using the *atop* operator (lines 6-8), and then further composed with labels and lines via the *union* operator to form a glyph (lines 13-14). This glyph is replicated in a data-driven manner to generate a series of bars.

In another example to produce the food-related pie chart (Figure 8c) with `piccl.js` specifications (Figure 8d) a circle is first divided into a collection of sectors via *divide* operation (line 3). Each sector is then combined with a pie image using the *in* operator (line 5), with the constraints ensuring that their centers are aligned (line 6) and image’s width and height match the circle’s diameter (line 7). Then, the resulting glyph is then further enhanced with a label using the *union* operator at the mark level (line 11-14). Finally, these composite glyphs are assembled into a chart through another *union* operation (line 16).

Given an example pictorial chart, the corresponding POT can be reused to create new charts by simply substituting theme-related images and data, enabling rapid generation of thematically consistent visualizations. Figure 8e and Figure 8f show two examples generated by replacing the original pictorial objects with alternative ones.

Nested Combination. Each pictorial chart is treated as a graphical object, enabling it to be further combined with other objects through compositional operations. As shown in Figure 9, this process builds

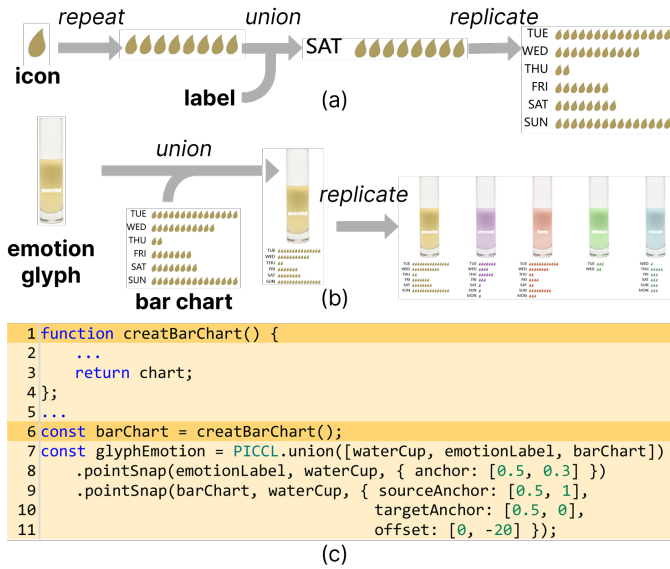


Fig. 9: Illustration of nested combination for creating new pictorial charts. (a) An ISOTYPE bar chart is constructed using a sequence of operations. (b) The resulting chart is then combined with an emotion glyph, and the composite glyph is replicated to represent different emotions. (c) A code snippet demonstrating how to combine a pictorial chart with other objects using piccl.js.

upon the chart in Figure 1c, with Figure 9c presenting the core piccl.js specification. In Figure 9a, an ISOTYPE bar chart is first constructed by applying a sequence of operations including repeat, union, and replicate to duplicate and arrange icons that encode data values (lines 1-4). Once completed, the entire bar chart is treated as a single composite object. This object is then combined with an emotion glyph using the union operator (lines 7-11), forming a new expressive visual unit. By replicating this combined glyph, the system generates multiple instances, as shown in Figure 9b, each representing a different emotion and enabling visual comparison across five emotional categories. Following the same approach, the resulting chart can be further enhanced by linking it with a set of text labels, producing the final visualization shown in Figure 1c.

The created chart object can also be reused to replace glyphs in an existing chart, enabling further combinations. For example, in Figure 10a, a pictorial chart is constructed using a sequence of operations (mapValue, union, link, replicate) to combine four components: a flower shape encoding GDP (area and color), a root encoding 2017 DESI (x-position), a center encoding 2022 DESI (x) and digital economy value (y), and a text label for the country name. Suppose the user now wishes to explore GDP distribution across sub-sectors of the tertiary sector (e.g., transportation, finance, education). After browsing previously created examples, she discovers a chart (shown in Figure 10b) that uses a radial layout with petals of varying lengths to represent sector-specific values. Finding this design compelling, the user decides to incorporate the flower glyph structure into her existing chart. By replacing the original flower specification with the one used in Figure 10b, she quickly generates a new pictorial chart that provides a detailed breakdown of GDP across sectors, as shown in Figure 10c.

6.2 User Study: Usability of piccl.js

To assess the usability of piccl.js, we conducted a user study with 10 participants (6 males, 4 females) from a local university. Following a commonly used methodology for evaluating chart authoring tools [20, 26, 28, 29, 45], participants were asked to reproduce bespoke charts. All had prior experience with JavaScript and were familiar with visualization libraries like D3.js and Vega-Lite.

Procedure. The study began with a training session introducing PiCCL and piccl.js. Participants were guided through three hands-on exercises covering key concepts and API usage, including mark definition, en-

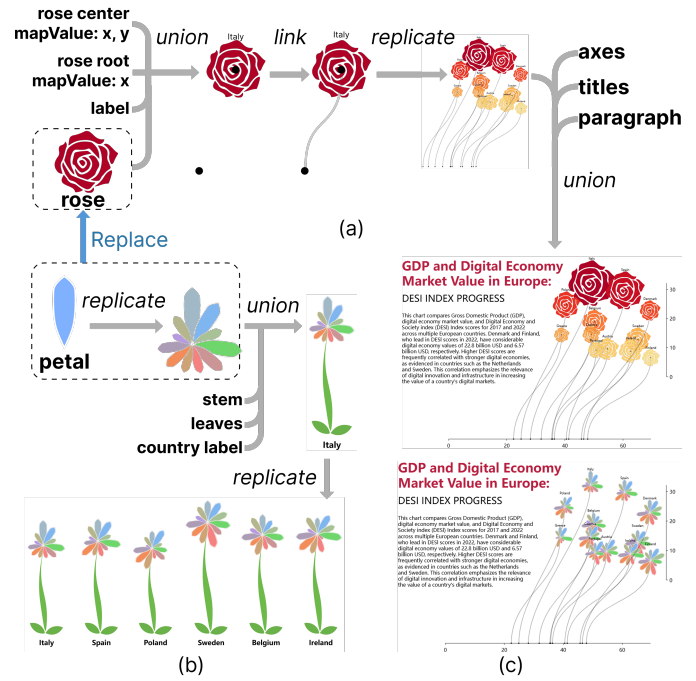


Fig. 10: Creating a new pictorial chart by reusing components from other examples. (a) A pictorial chart where flower size, center's x, y position, and root's x position are used to encode four different variables. (b) A flower chart with petal lengths representing multiple variables. (c) A new chart created by replacing the flower glyph in (a) with the petal-based glyph from (b), allowing each flower to encode GDP values across various sectors.

coding, composition operators, as well as the use of constraints and layouts. Each exercise was accompanied by a solution. Participants were encouraged to ask questions during the training phase, which lasted approximately 1.5 hours.

Participants were then asked to complete four tasks, and encouraged to think aloud about their reasoning and the challenges they faced. For each task, we observed their workflows, collecting verbal feedback, and recorded task completion time. Tasks involved transforming a basic chart (bar chart or pie chart) into a pictorial chart, and were designed with increasing complexity; Task 1 (Figure 11a) involved simple chart-level combine operations and Task 4 (Figure 11d) required the use of nested repeat and union operators. Participants were provided with the initial chart code, data, and image assets, and used a pre-configured workspace. A POT diagram for each task was offered as a hint if participants could not make progress in five minutes.

Results. Seven out of 10 participants successfully completed all four tasks with minimal API-related guidance, while the other three completed all tasks independently (<https://osf.io/5eqb7>). As summarized in Figure 11, the average completion time across all tasks and participants was under 10 minutes, ranging from 4.7 to 12.9 minutes per individual. Participants were not instructed to prioritize speed. Participants evaluated piccl.js on five standard usability dimensions using a 7-point Likert scale (1 = "Strongly Disagree," 7 = "Strongly Agree"): learnability (6.1), efficiency (6.4), memorability (5.8), error recovery (4.9), and satisfaction (6.4).

Participants consistently appreciated piccl.js' expressiveness and flexibility, noting that it enabled tasks that would be difficult or nearly impossible with other tools. Many were especially impressed with the Boolean composition operators, particularly intersect. One participant remarked, "With D3, achieving this would've required a ton of complex code—now, a single operator does it." Two-thirds of the participants highlighted the value of the constraint system, which helped them control layout and alignment, reducing manual adjustments. Several participants commented that the overall design model of piccl.js aligned closely with their way of thinking. One participant said, "I

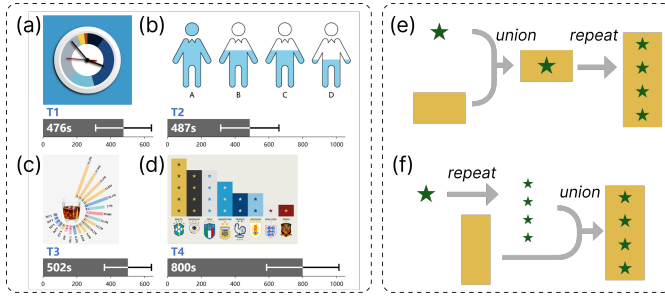


Fig. 11: (a-d) Top: the target pictorial charts used in the user study as tasks (these charts were created using `piccl.js`); bottom: the task completion times in seconds, with error bars indicating standard deviations. (a) Task 1 – Clock-style pie chart composition; (b) Task 2 – intersect each icon with a blue bar; (c) Task 3 – multi-level mark and chart composition with layout adjustment; (d) Task 4 – nested repeat and glyph composition for decorated bar charts. (e, f) Two approaches to constructing a bar overlaid with multiple star icons.

can assemble the shapes I want in the way I imagine, without having to think about how each element’s attributes are calculated.” Another added, “I can imagine a tree-like structure of my chart, which makes the logic behind the construction much clearer.” Participants also found the code easy to understand, making modification straightforward.

An interesting observation emerged in Task 4, the pictorial bar chart shown at the top of Figure 11d. In this chart, the height of each bar represents the number of championships won by a team, and star-shaped decorations placed above the bar encode the same value. There are two valid approaches to constructing a single decorated bar. The first approach (Figure 11e) involves combining a short rectangle and a star icon using the `union` operator to form a basic unit, which is then repeated vertically based on the data value. The second approach (Figure 11f) first repeats the star icon according to the data value and then overlays the repeated stars with a rectangle whose height also encodes the same value. In our original design, the method shown in Figure 11e was intended as the expected solution. However, most participants intuitively opted for the method in Figure 11f, even though it required fine-tuning the spacing between stars to ensure they fit within the bar’s bounds. We hypothesize that this choice is influenced by users’ starting points: those constructing the chart from scratch are more likely to build from compositional units (favoring Figure 11e), while those editing an existing bar focus on augmenting it with visual elements (favoring Figure 11f). This finding highlights the flexibility of our compositional framework, allowing multiple valid construction strategies for the same visual outcome and supporting a range of user workflows and mental models.

We observed a recurring issue during the use of the `intersect` operator: after applying the operation, only the overlapping region remains, which can make it difficult to identify errors if constraints are incorrectly configured. To address this, we plan to incorporate a debug mode in `piccl.js`. In this mode, the original shapes involved in `intersect` or `difference` operations will be preserved and displayed semi-transparently, enabling users to better understand and debug spatial relationships among elements.

7 DISCUSSION AND FUTURE WORK

Expressiveness. Although PiCCL is specifically designed for constructing pictorial charts, it is also capable of modeling most standard chart types created using existing visualization libraries and languages such as D3 and Vega. This versatility stems from its compositional design and support for essential visualization constructs, including mark encoding, layout, glyph composition, and data binding. Unlike traditional systems that follow the grammar of graphics paradigm, PiCCL introduces a modular, hierarchical, and part-based representation that enhances reusability, extensibility, and composability in visualization authoring, as demonstrated in Section 6.1. Compared to MSC [19], POT provides a more compact structure. Its use of fewer operators and lazy data binding simplifies programmatic manipulation and facilitates

seamless design extension.

However, PiCCL is not able to represent all examples in our set of collected pictorial charts. Specifically, it cannot effectively model charts with overly flexible layouts, highly diverse or irregular elements, or those that rely on 3D visual effects. In addition, PiCCL currently lacks built-in support for common data transformation operations such as aggregation, filtering, and pivoting. As a result, these tasks must be performed externally prior to visualization. In future work, we plan to incorporate data transformation capabilities directly into the POT, enabling a fully integrated authoring workflow—from raw data to expressive visual forms.

Usability. Our user study demonstrates that PiCCL and its implementation in `piccl.js` make it relatively easy to create pictorial charts. However, most participants had prior experience with D3 or JavaScript. This level of technical proficiency may not be common among many pictorial chart designers, especially those without a programming background. This limitation highlights the challenge of programmatically defining and managing visual variability within a code-based framework.

To address these challenges, we plan to develop an interactive authoring tool based on PiCCL. To enable the construction of data-driven pictorial charts without programming, it will leverage direct manipulation of graphical elements, including support for freeform sketching [22]. Inspired by systems such as CAST [13] and CAST+ [35], our tool will incorporate auto-completion techniques to guide users through the chart construction process, suggesting appropriate operations and visual encodings as needed while still allowing for creative flexibility.

Interaction. Our `piccl.js` currently focuses on static bespoke chart. An important direction for future work is extending the prototype to support interactive pictorial charts, where pictorial elements can reflect dynamic states such as hover, selection, or filtering. This would enable richer user experiences, allowing charts to respond to user input through visual feedback like highlighting. Integrating interaction also raises new challenges in maintaining semantic and spatial consistency across dynamic states, especially for complex composed glyphs. We plan to explore interaction models that preserve the compositional logic of the POT representation while enabling flexible interaction design.

Generative AI. Currently, users are required to manually source theme-relevant pictorial objects, often spending significant time browsing databases or searching online. To alleviate this burden, we plan to integrate generative models (e.g., controllable text-to-image diffusion [46]) to suggest or generate suitable pictorial assets automatically. This integration will streamline the creative workflow, reduce the effort needed for asset collection, and make pictorial chart authoring more accessible, particularly for users with limited design or programming experience.

8 CONCLUSION

In this paper, we introduced a composition model and PiCCL, a new language that uses glyph composition to support the creation of expressive and semantically meaningful pictorial charts. We implemented PiCCL as a JavaScript library, `piccl.js`, which provides a set of simple composition operations for authoring diverse pictorial charts. We demonstrated that `piccl.js` is expressive enough to reproduce a variety of examples and is easy to learn and use, as shown through a user study with 10 participants. Finally, we discussed directions for future work, including extending PiCCL’s expressiveness and developing interactive authoring tools.

ACKNOWLEDGMENTS

This work was supported by the grants of the National Key R&D Program of China under Grant 2022ZD0160805, NSFC (No.62132017 and No.U2436209), the Shandong Provincial Natural Science Foundation (No.ZQ2022JQ32), the Beijing Natural Science Foundation (L247027), the Fundamental Research Funds for the Central Universities, the Research Funds of Renmin University of China. This work was also supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korean Government (MSIT), Artificial Intelligence Graduate School Program, Yonsei University, under Grant RS-2020-II201361.

REFERENCES

- [1] N. S. Alrwele. Effects of infographics on student achievement and students' perceptions of the impacts of infographics. *Journal of Education and Human Development*, 6(3):104–117, 2017. doi: 10.15640/jehd.v6n3a12 1
- [2] List of anthropographics. <https://luizaugustomm.github.io/anthropographics/>. Accessed: 2025-03-04. 3
- [3] S. Bateman, R. L. Mandryk, C. Gutwin, A. Genest, D. McDine, and C. Brooks. Useful junk? the effects of visual embellishment on comprehension and memorability of charts. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 2573–2582. ACM, 2010. doi: 10.1145/1753326.1753716 2
- [4] M. A. Borkin, A. A. Vo, Z. Bylinskii, P. Isola, S. Sunkavalli, A. Oliva, and H. Pfister. What makes a visualization memorable? *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2306–2315, 2013. doi: 10.1109/TVCG.2013.234 1, 2
- [5] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009. doi: 10.1109/TVCG.2009.174 3
- [6] M. Bostock, V. Ogievetsky, and J. Heer. D³ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185 3
- [7] A. Burns, C. Xiong, S. Franconeri, A. Cairo, and N. Mahyar. Designing with pictographs: Envision topics without sacrificing understanding. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):4515–4530, 2021. doi: 10.1109/TVCG.2021.3092680 2
- [8] M. Chen and J. V. Tucker. Constructive volume geometry. *Computer Graphics Forum*, 19(4):281–293, 2000. doi: 10.1111/1467-8659.00464 3
- [9] D. Coelho and K. Mueller. Infomages: Embedding data into thematic images. In *Computer Graphics Forum*, vol. 39, pp. 593–606. Wiley Online Library, 2020. doi: 10.1111/cgf.14004 1, 2, 3
- [10] blend-mode. <https://docs.gimp.org/2.8/en/gimp-concepts-layer-modes.html>. Accessed: 2025-06-04. 5
- [11] W. Cui, J. Wang, H. Huang, Y. Wang, C.-Y. Lin, H. Zhang, and D. Zhang. A mixed-initiative approach to reusing infographic charts. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):173–183, 2021. doi: 10.1109/TVCG.2021.3114856 2, 3
- [12] J. M. Ford. Content analysis: An introduction to its methodology. *Personnel Psychology*, 57(4):1110, 2004. 3
- [13] T. Ge, B. Lee, and Y. Wang. Cast: Authoring data-driven chart animations. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–15. ACM, 2021. doi: 10.1145/3411764.3445452 9
- [14] S. Haroz, R. Kosara, and S. L. Franconeri. Isotype visualization: Working memory, performance, and engagement with pictographs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1191–1200. ACM, 2015. doi: 10.1145/2702123.2702275 2
- [15] S. R. Herring, C.-C. Chang, J. Krantzler, and B. P. Bailey. Getting inspired! understanding how and why examples are used in creative design practice. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 87–96. ACM, 2009. doi: 10.1145/1518701.1518717 1
- [16] highcharts. <https://www.highcharts.com/demo/highcharts/pictorial-stackshadow>. Accessed: 2025-03-04. 2
- [17] D. Kalkofen, E. Mendez, and D. Schmalstieg. Interactive focus and context visualization for augmented reality. In *Proceedings of the IEEE and ACM International Symposium on Mixed and Augmented Reality*, pp. 191–201. IEEE, 2007. doi: 10.1109/ISMAR.2007.4538846 3
- [18] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister. Data-driven guides: Supporting expressive design for information graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):491–500, 2016. doi: 10.1109/TVCG.2016.2598620 1, 2
- [19] Z. Liu, C. Chen, and J. Hooker. Manipulable semantic components: A computational representation of data visualization scenes. *IEEE Transactions on Visualization and Computer Graphics*, 2024. doi: 10.1109/TVCG.2024.3456296 2, 3, 5, 6, 7, 9
- [20] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. Stasko. Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–13. ACM, 2018. doi: 10.1145/3173574.3173697 2, 6, 8
- [21] C. Nobre, M. Meyer, M. Streit, and A. Lex. The state of the art in visualizing multivariate networks. *Computer Graphics Forum*, 38(3):807–832, 2019. doi: 10.1111/cgf.13728 6
- [22] A. Offenwanger, T. Tsandilas, and F. Chevalier. Datagarden: Formalizing personal sketches into structured visualization templates. *IEEE Transactions on Visualization and Computer Graphics*, 2024. doi: 10.1109/TVCG.2024.3456336 9
- [23] D. Park, S. M. Drucker, R. Fernandez, and N. Elmqvist. Atom: A grammar for unit visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 24(12):3032–3043, 2017. doi: 10.1109/TVCG.2017.2785807 6
- [24] T. Porter and T. Duff. Compositing digital images. In *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques*, pp. 253–259. ACM, 1984. doi: 10.1145/964965.808606 5
- [25] C. Qian, S. Sun, W. Cui, J.-G. Lou, H. Zhang, and D. Zhang. Retrieve-then-adapt: Example-based automatic generation for proportion-related infographics. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):443–452, 2020. doi: 10.1109/TVCG.2020.3030448 2
- [26] D. Ren, M. Brehmer, B. Lee, T. Höllerer, and E. K. Choe. ChartAccent: Annotation for data-driven storytelling. In *Proc. IEEE Pacific Visualization Symposium*, pp. 230–239. IEEE, 2017. doi: 10.1109/PACIFICVIS.2017.8031599 8
- [27] D. Ren, T. Höllerer, and X. Yuan. ivisdesigner: Expressive interactive design of information visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2092–2101, 2014. doi: 10.1109/TVCG.2014.2346291 2
- [28] D. Ren, B. Lee, and M. Brehmer. Charticulator: Interactive construction of bespoke chart layouts. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):789–799, 2018. doi: 10.1109/TVCG.2018.2865158 2, 3, 5, 6, 7, 8
- [29] D. Ren, B. Lee, M. Brehmer, and N. Henry Riche. Reflecting on the evaluation of visualization authoring systems: Position paper. In *IEEE Evaluation and Beyond-Methodological Approaches for Visualization*, pp. 86–92. IEEE, 2018. doi: 10.1109/BELIV.2018.8634297 8
- [30] A. A. Requicha and J. R. Rossignac. Solid modeling and beyond. *IEEE Computer Graphics and Applications*, 12(5):31–44, 1992. doi: 10.1109/38.156011 3
- [31] H. Romat, N. Henry Riche, C. Hurter, S. Drucker, F. Amini, and K. Hinckley. Dear pictograph: Investigating the role of personalization and immersion for consuming and enjoying visualizations. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–13. ACM, 2020. doi: 10.1145/3313831.3376348 1
- [32] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, vol. 33, pp. 351–360. Wiley Online Library, 2014. doi: 10.1111/cgf.12391 2
- [33] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2016. doi: 10.1109/TVCG.2016.2599030 2, 3
- [34] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2015. doi: 10.1109/TVCG.2015.2467091 2, 3
- [35] Y. Shen, Y. Zhao, Y. Wang, T. Ge, H. Shi, and B. Lee. Authoring data-driven chart animations through direct manipulation. *IEEE Transactions on Visualization and Computer Graphics*, 31:1613–1630, 2025. doi: 10.1109/TVCG.2024.3491504 9
- [36] Y. Shi, P. Liu, S. Chen, M. Sun, and N. Cao. Supporting expressive and faithful pictorial visualization design with visual style transfer. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):236–246, 2022. doi: 10.1109/TVCG.2022.3209486 2
- [37] T. Tsandilas. Structgraphics: Flexible visualization design through data-agnostic and reusable graphical structures. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):315–325, 2020. doi: 10.1109/TVCG.2020.3030476 2, 5
- [38] M. Twyman. The significance of isotype. *Graphic Communication Through Isotype*, 7, 1975. 2
- [39] Pictorial visualization dataset. <https://idxlab.com/vistylist/>. Accessed: 2025-03-04. 3
- [40] Y. Wang, H. Zhang, H. Huang, X. Chen, Q. Yin, Z. Hou, D. Zhang, Q. Luo, and H. Qu. Infonce: Easy creation of information graphics. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–12. ACM, 2018. doi: 10.1145/3173574.3173909 1, 2
- [41] H. Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010. doi: 10.1198/jcgs.2009.07098 3
- [42] L. Wilkinson. The grammar of graphics. In *Handbook of Computational*

Statistics: Concepts and Methods, pp. 375–414. Springer, 2011. 2, 3

- [43] J. Woodring and H.-W. Shen. Multi-variate, time varying, and comparative visualization with contextual cues. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):909–916, 2006. doi: [10.1109/TVCG.2006.164](https://doi.org/10.1109/TVCG.2006.164) 3
- [44] S. Xiao, S. Huang, Y. Lin, Y. Ye, and W. Zeng. Let the chart spark: Embedding semantic context into chart with text-to-image generative model. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):284–294, 2023. doi: [10.1109/TVCG.2023.3326913](https://doi.org/10.1109/TVCG.2023.3326913) 1, 3
- [45] J. E. Zhang, N. Sultanum, A. Bezerianos, and F. Chevalier. Dataquilt: Extracting visual elements from images to craft pictorial visualizations. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–13. ACM, 2020. doi: [10.1145/3313831.3376172](https://doi.org/10.1145/3313831.3376172) 2, 4, 8
- [46] L. Zhang, A. Rao, and M. Agrawala. Adding conditional control to text-to-image diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3836–3847. IEEE, 2023. doi: [10.1109/ICCV51070.2023.00355](https://doi.org/10.1109/ICCV51070.2023.00355) 9