Atlas: Grammar-based Procedural Generation of Data Visualizations

Zhicheng Liu* Chen Chen† Francisco Morales‡ Yishan Zhao§

Department of Computer Science
University of Maryland, College Park

ABSTRACT

We present Atlas, a procedural grammar for constructing data visualizations. Unlike most visualization grammars which use declarative specifications to describe visualization components, Atlas exposes the generative process of a visualization through a set of concatenated high-level production rules. Each of these rules describes how an input graphical object is created, transformed, or joined with abstract data to derive an output object. The visualization state can thus be inspected throughout the generative process. We demonstrate Atlas' expressivity through a catalog of visualization designs, and discuss the trade-offs in its design by comparing it to state-of-the-art grammars.

Index Terms: Human-centered computing—Visualization—Visualization systems and tools—Visualization toolkits

1 INTRODUCTION

Visualization grammars [23] enable the construction of a wide range of visualization designs with a relatively small set of building blocks. Most visualization grammar implementations are in the form of declarative specifications [6, 19, 21], where users only need to describe the components of a chart such as marks and encodings, and a compiler translates the textual specification into the desired visualization. Declarative specifications are concise and expressive, and relieve users from low-level details by hiding generative mechanisms and control flows. However, they may pose explainability and extensibility challenges. Debugging and tracing the intermediate results can be difficult, especially for novice users. To those who are used to thinking in terms of graphical objects, the abstract syntax in these specifications may not be intuitive [7, 12]. It is also hard to incorporate user-defined visualization logic if the relevant mark types or structures have not been included in the grammar implementation.

Research on graphical authoring tools [12, 16, 17] addresses some of these challenges. In these tools, users construct data visualizations by directly manipulating graphical objects. By providing immediate and interpretable visual feedback after each operation, these authoring tools allow users to better understand and control visualization generative processes. In many cases, user operations are translated into grammatical expressions in the underlying system. Some tools also offer greater flexibility to compose visual structures without underlying grammatical formalisms [11, 12, 20, 25]. Compared to text-based grammar toolkits, however, graphical interfaces are less ideal for automation (e.g., batch generation of multiple visualizations), and it is challenging to build applications that leverage or augment the capabilities of the tools. A visualization grammar based on the abstractions from these authoring frameworks has the potential to retain the benefits of step-wise graphics-centric authoring while serving as a foundation for automating visualization generation and building future interactive applications.

In this paper, we present Atlas, a visualization grammar that synthesizes and extends ideas from existing graphical authoring tools. Three high-level principles guide the design of Atlas. First, we take a *graphics-centric approach*: instead of representing visualization construction as a data transformation process, we treat graphical objects as first-class citizens, and focus on how graphical objects can be created, modified and joined with data to generate visualizations. Second, the grammar uses a *procedural syntax*, defining a set of production rules, each consisting of an input graphical object, an output graphical object, and an operator describing the nature of transformation. A visualization is created by applying a series of such rules step by step. Finally, we *decouple visualization logic from rendering*, so that users can inspect and debug the graphical objects after every operation, independent of how the objects are rendered. We demonstrate Atlas' expressivity through a catalog of visualization designs, and discuss the trade-offs in its design by comparing to state-of-the-art grammars.

2 THE ATLAS VISUALIZATION GRAMMAR

The Atlas grammar specifies how graphical objects are generated or transformed to create visualizations. Figure 1 shows the Visualization Object Model (VOM) that describes the hierarchical relationships between graphical objects in the Atlas grammar. This object model is derived from the unified terminology used in recent visualization authoring tools [18]. At the lowest level, we have *vertices* and *segments*, which can be used to compose *mark* primitives such as rectangle, circle, line and path. A rectangle, for example, consists of four vertices connected by four line segments. A *glyph* consists of one or more marks. For example, in a box-and-whisker plot, a glyph is composed of multiple marks (the box and the whiskers). A data visualization typically consists of a *collection* of glyphs or marks, where each glyph or mark represents a data case. A collection can be nested, where its children are collections (e.g., small multiples). Finally, a visualization *scene* can contain graphical objects such as stand-alone marks or glyphs, axes, legends and gridlines.

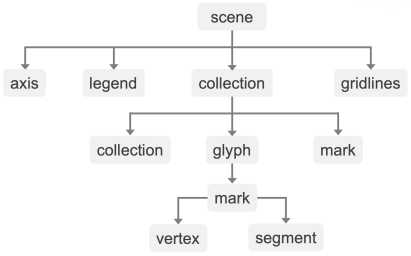


Figure 1: The visualization object model (VOM) in Atlas

Unlike D3 [8] where visualization objects are represented as Document Object Model (DOM) nodes, our VOM is independent of any rendering context. Users can access and inspect VOM objects at any point during the construction process without rendering them.

At the core of the Atlas grammar is a set of rules that specify how these graphical objects are created or manipulated. We group these rules into six high-level categories: initial setup, glyph generation,

*e-mail: leozcliu@umd.edu
†e-mail: cchen24@umd.edu
‡e-mail: morales0@umd.edu
§e-mail: yishanzh@terpmail.umd.edu

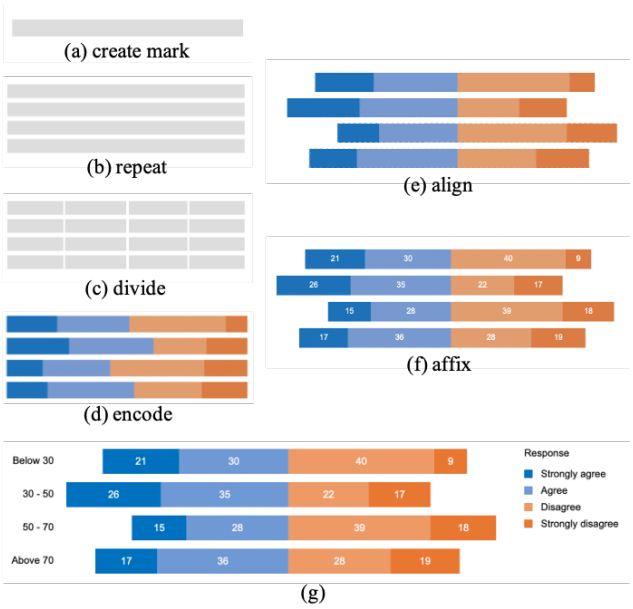


Figure 2: Creating a diverging bar chart using the Atlas grammar

graphics-data join, visual encoding, spatial arrangement, and view transformation. Each rule consists of four components: a descriptive name, an input object, an output object, and parameters. We use the following form to represent a rule, similar rule notations are used by generative grammars in linguistics [9] and shape modeling [14]:

name(parameters) : input object $\rightsquigarrow$ output object

In this section, we describe representative rules in each of these categories, and discuss the design of the corresponding application programming interfaces (APIs) in Atlas.js, a JavaScript implementation of the grammar. We will use the diverging bar chart in Figure 2g as an example to ground our discussion. This chart visualizes a hypothetical dataset reporting people's opinions on a subject matter, broken down by age (below 30, 30 - 50, 50 - 70, above 70) and responses (strongly agree, agree, disagree, strongly disagree). The data table consists of three columns: age, response, and pct (percentage of people with a particular response within an age group). Listing 1 shows the code that generates this chart using Atlas.js.

Listing 1: Using Atlas.js to generate a diverging bar chart

```
1  let scn = atlas.scene(),
2      dt = await atlas.csv('survey_response.csv');
3  let rect = scn.mark('rect', {top:100, left:200, width:700,
       height:30});
4  let coll = scn.repeat(rect, dt, {field:'age'});
5  coll.layout = atlas.layout('grid', {numCols:1, vGap:10});
6  let bars = scn.divide(rect, dt, {field:'response',
       orientation:'horizontal'});
7  rect = bars.firstChild;
8  scn.encode(rect, {field:'pct', channel:'width'});
9  let palette = {'Strongly agree':'#1e71b8', 'Agree':'#7799cf',
       'Disagree':'#e29d6f', 'Strongly disagree':'#da7c43'};
10 scn.encode(rect, {field:'response', channel:'fillColor',
       mapping:palette});
11 let agreeBars = scn.find([{field:'response', value:'agree'}]);
12 scn.align(agreeBars, 'right');
13 let text = scn.mark('text', {fillColor:'white'});
14 scn.repeat(text, dt);
15 scn.encode(text, {field:'pct', channel:'text'});
16 scn.affix(text, rect, 'x');
17 scn.affix(text, rect, 'y');
18 scn.axis('y', 'age', {orientation:'left', x:200,
       ruleVisible:false, tickVisible:false});
19 scn.legend('fillColor', 'response', {x:750, y:100});
20 atlas.renderer('svg').render(scn, 'svgEle');
```

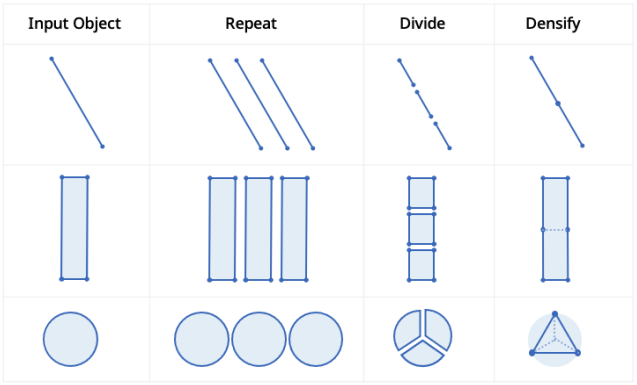


Figure 3: Rules for graphics-data join: repeat, divide, and densify . Here each input mark is joined with three (3) unique field values.

2.1 Initial Setup

In Atlas, the first steps of creating a visualization involve setting up a scene and importing data (:= means "defined as"):

createScene() : $\rightsquigarrow$ scene

importData() : $\rightsquigarrow$ datatable := fields and tuples

In these two rules, no input object is required. Similar to the other visualization grammars, Atlas assumes the imported data is in a tidy tabular format [22], where the rows represent data tuples and columns represent fields or data attributes. A data table can be used across multiple scenes. In atlas.js, these two rules are defined as functions in the atlas namespace (Listing 1, lines 1 and 2).

2.2 Glyph Generation

To create a mark, users specify the mark type and properties, no input object is required (| denotes "or"):

createMark(type, properties) : $\rightsquigarrow$ mark := rect | circle | line | path

One or more marks can be grouped to create a glyph. Charts like the box-plot or the bullet chart use a glyph that consists of multiple primitive marks (+ means one or more):

createGlyph() : mark+ $\rightsquigarrow$ glyph

In Atlas.js, these rules are defined as methods in the Scene class (Listing 1, line 3). The created marks or glyphs (e.g., `rect`) are JavaScript objects, independent of rendering contexts.

2.3 Graphics-Data Join

The rules specifying graphics-data joins are a major difference that sets Atlas apart from the other visualization grammars. Figure 3 illustrates three rules for joining graphics with data:

repeat(field) : glyph | collection $\rightsquigarrow$ collection

divide(field, orientation) : mark $\rightsquigarrow$ collection

densify(field, orientation) : mark $\rightsquigarrow$ polyline | polygon

The repeat rule applies to any glyph and replaces it with a collection of glyphs, each glyph in the collection represents a unique value in the optional field parameter. If unspecified, a glyph is created for every tuple (row) in the data table. All the tuples having the same field value are joined with each corresponding glyph as its data scope [12]. For example, in Listing 1 line 4, we are repeating a rectangle by the age field. Since there are four unique age values in the data (below 30, 30 - 50, 50 - 70, above 70), we get a collection of four rectangles (Figure 2b). The data scope of the first rectangle is all the rows with "below 30" as the age value, and so on and so forth. The repeat rule applies to a collection as well, replacing it with a nested collection. This rule works in the same way as the repeat operator in the Data Illustrator framework [12].

The divide rule splits a mark into a collection of smaller marks. The output object type depends on the mark type and the orientation parameter. For example, applying divide on a circle along the

angular orientation produces a collection of pies (Figure 3), and along the radial orientation produces a collection of rings. The graphics-data joining mechanism is similar to that of the repeat rule: tuples sharing the same field value are assigned as the data scope of each corresponding mark in the output collection.

The densify rule adds vertices along the border of a mark, and replaces curve segments with line segments. A line, for example, is replaced by a polyline; and a circle is replaced by a standard polygon (Figure 3). The graphics-data joining mechanism is similar to that of the repeat rule and the divide rule. Each vertex represent a unique value in the field parameter. The divide and densify rules are inspired by the partition operator in Data Illustrator [12] and the densify command in ArcGIS [3]. The partition operation in Data Illustrator produces inconsistent output object types for different marks (e.g., partitioning a line results in a single polyline while partitioning a rectangle results in a collection of rectangles). By splitting the partition operator into two different rules, we enhance the expressivity and clarity of the grammar.

Concatenating these graphics-data join rules will produce nested structures. For example, in Listing 1, we first repeat a rectangle by age (line 4), and divide all the generated rectangles by response (line 6, Figure 2c). In toolkits like D3 [8], to perform the second divide operation, we need first to select all four rectangles in the collection coll generated by repeat. In Atlas.js, since these rules are defined as methods in the Scene class, users only need to pass in any rectangle in coll as an example. Our implementation will find in the scene all the other three peer marks generated by repeat and perform divide on each of them accordingly.

2.4 Visual Encoding

Visual encoding is a central operation in data visualization, mapping abstract data values in a field to visual channel properties. The Atlas grammar requires users to clearly specify which graphical objects (vertices, segments, glyphs, or collections) an encoding rule applies to, as it is possible to map data to the visual channels of multiple types of graphical objects in a visualization design:

object := vertex | segment | glyph | collection

encode(field, channel) : object $\rightsquigarrow$ encoding

In Atlas.js, the encoding rule is also implemented as a method in the Scene class. Users only need to provide a graphical object as an example, and the encoding will be applied to all its peers. For example, we can encode the pct field as the width of rectangles (Listing 1 line 8), and encode the response field as the fill color with self-defined color mappings (line 9 and 10, Figure 2d). Based on the field's data type and the current visual properties of the graphical objects, our implementation automatically creates a scale (i.e., a function that transforms domain data values into a range of visual property values). The encoding rule returns an encoding object, which encapsulates the scale, allowing for further manipulation such as customizing domain and range.

By default, the same scale is used for all the peer graphical objects in the same scene. In visualizations with nested structures, e.g., a trellis plot of bar charts, sometimes it is beneficial for each bar chart to have its own scale. Atlas thus defines a variant of the encoding rule, which returns an encoding object for each group of objects:

encodeInGroup(field, channel) : object $\rightsquigarrow$ [encodings]

2.5 Spatial Arrangement

Mapping data to the position channels (i.e., x and y coordinates) is a common way to arrange graphical objects spatially. In addition to such encodings, Atlas supports algorithmic layouts and graphical constraints since many visualization designs employ these mechanisms to position objects [16]. For example, after repeat, we apply a grid layout to position the rectangles vertically with the vGap parameter set to 10 pixels (Listing 1 line 5):

coll := collection

gridLayout(rows, cols, hGap, vGap) : coll $\rightsquigarrow$ coll.children.position

When a divide is performed, Atlas will apply a default layout to the resulting collection depending on the mark type. For example, dividing a rectangle will also apply a stack layout to the generated smaller rectangles (Listing 1 line 6, Figure 3).

stackLayout(orientation) : coll $\rightsquigarrow$ coll.children.position

In Atlas.js, layout objects are created using functions in the atlas namespace (Listing 1 line 5), the same layout object can be applied to multiple collections.

In addition to algorithmic layouts, Atlas supports graphical constraints that specify object positions. The alignment constraint makes sure the anchors of objects are arranged in a straight line:

objects := glyphs | collections

align(anchor) : objects $\rightsquigarrow$ objects.position

In Figure 2g, we have a vertical baseline separating the blue bars from the orange bars, representing the divergence of opinions. To achieve this spatial arrangement, we can either align all the light blue bars to the right, or all the light orange bars to the left. The Scene class in Atlas.js provides a method to find graphical objects matching a list of filtering predicates. In this case, we find all the light blue bars by their shared field value (Listing 1 line 11). Alternatively, we can also find the same set of bars by their shared fill color. With the bars identified, line 12 applies an alignment constraint (Figure 2e). Due to the graphics-centric and procedural nature of Atlas, it is easy and flexible to specify such spatial arrangements. In other approaches, similar designs often require data transformations [4].

Another type of constraint is the affixation constraint, which specifies the position of one object relative to a reference object:

object := glyph | collection

affix(channel, anchor, offset) : (object, refObject) $\rightsquigarrow$ object.position

In Figure 2g, we have a set of white labels affixed to the center of each bar. This is achieved by creating a collection of text marks, encoding the field pct as the text content, and applying affix constraints to position the text marks at the center of the rectangles (Listing 1 lines 13-17, Figure 2f). Here, the optional anchor and offset parameters are not provided and default to center and 0.

2.6 View Transformation

The final rule category is view transformation, where geometric transformation (e.g., rotate, zoom, skew) is applied to an entire scene. The Polar transformation rule, for example, converts Cartesian coordinates into polar coordinates.

polar(center) : scene $\rightsquigarrow$ vertices.coordinates

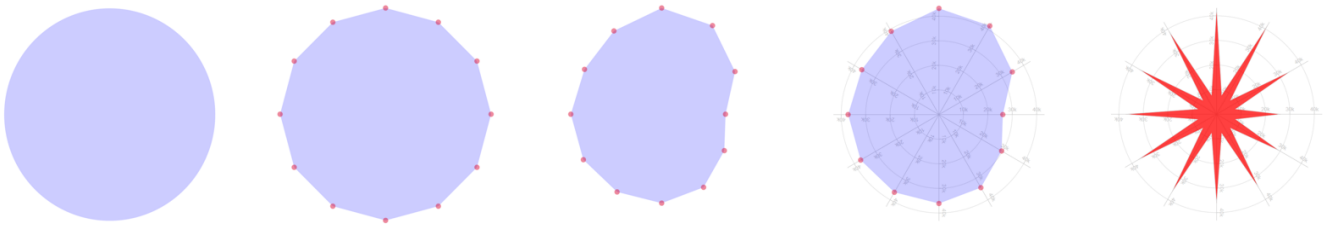
The Scene class in Atlas.js also provides methods to create axes and legends. Users only need to specify the channel, field and preferred properties such as position and visibility (lines 18-19). All the objects discussed so far are JavaScript objects in Atlas.js, which can be printed to the console. To display these objects, we need to create renderers. In Listing 1 line 20, we create an SVG renderer and use it to render the scene in an SVG element with the id "svgEle". Atlas supports rendering using WebGL as well.

A catalog of visualization examples created using Atlas.js is available at `https://atlas-vis.github.io/gallery.html`.

3 Discussion

Any visualization grammar needs to consider and make trade-offs between expressiveness, conciseness, extensibility, explainability and learnability. We discuss how Atlas fares along these dimensions by comparing it to three state-of-the-art grammars and toolkits: Vega-Lite [19], Vega [6] and D3 [8].

In terms of static visualizations, the expressiveness of Atlas in its current form is comparable to that of Vega-Lite. Atlas supports most examples in Vega-Lite's gallery except maps, horizon graph, and visualizations with variable stroke widths. Meanwhile, the violin



(a) create mark (b) densify (c) encode (d) Radar Chart (e) Stellar Chart

Figure 4: The process of creating Radar Chart and Stellar Chart both showing adjusted monthly gasoline sales in the U.S. in 2020.

plot, radar plot, stellar chart and ridgeline plot are supported by Atlas but are missing from Vega-Lite's gallery. Atlas' graphics-centric approach also offers flexibility in creating nested visualizations with custom layouts (e.g., multiple bar charts positioned by data [2]) and infographics (e.g., [1]). Further evaluation is needed to confirm if such designs can be created using Vega-Lite. Atlas does not yet have built-in support for statistical abstractions, however it is possible to perform such calculations using JavaScript and visualize the results using Atlas. Vega-Lite goes beyond static charts and supports interactive behavior, which is currently absent in Atlas.

All the designs discussed so far can be created using D3 and Vega. In addition, these two libraries support more diverse examples such as tree and network visualizations. Atlas is thus less expressive than D3 and Vega, but it is more concise. Most of Atlas' examples require less than 30 lines of code, and 43 out of 50 examples in the current gallery are realized in no more than 20 lines of code.

Atlas addresses the extensibility issue through a graphics-centric approach, where users directly modify graphical objects in the scene graph. For example, a radar chart is composed by (1) creating a circle mark, (2) performing densify operator to turn the circle into a polygon, and (3) binding the radial distance channel with data (Figure 4). To create a stellar chart, which is arguably better than a radar chart [13], there is no need to introduce a new star mark. We can simply insert new vertices between each pair of vertices in the radar chart with a fixed radial distance (Figure 4). Similarly, to create a violin plot, no definition of a new violin mark is necessary. Users can densify a rectangle to get an area mark and combine it with a box plot glyph. Atlas thus enables us to compose complex glyphs from primitive marks.

Atlas defines its own visualization object model (VOM) based on the unified terminology from recent authoring tools [18], while D3 relies on the document object model (DOM). Whether the custom VOM or the DOM is more advantageous depends on context [8]. For proficient web developers, leveraging the DOM as the scene graph abstraction allows them to use their prior knowledge; but for novices, learning the VOM may be easier than mastering the DOM and the associated technology stack including SVG and CSS. The custom VOM enables flexible rendering to either SVG or WebGL with the same code that constructs a visualization; while it is possible to use D3 to render visualizations to an HTML canvas, significant efforts are required from users.

Finally, the step-by-step authoring process in Atlas potentially makes learning and debugging easier since the user could review intermediate results and locate the problems. We acknowledge that this claim is speculative and needs to be validated through empirical studies.

4 Related Work

Grammar-based Procedural Modeling. In computer graphics, procedural grammars are often used to create complex objects and scenes such as buildings, cities, plants and terrains. In architectural modeling, the generative process starts with an initial shape, and a shape grammar defining production rules that replace a shape by a new set of shapes [14, 15, 24]. For example, the *repeat* rule takes a shape as input and produces a tiling of shapes along the specified x or y axis; the *split* rule divides a facade into a set of smaller components [14]. Applying these rules iteratively can quickly generate complex models with rich details. While such shape grammars are predominantly used in architectural domains, Karnick et al. proposed a shape grammar for glyph-based geographic visualizations [10]. We propose a procedural grammar for a broader range of glyph-based data visualizations.

Visualization Grammars. Atlas adopts a procedural and graphics-centric approach in grammar design. Unlike existing visualization grammars [5, 6, 19, 21] which use textual specifications, Atlas constructs visualizations through stepwise operations. Developers are thus able to inspect and debug intermediate visualization states. To enable such procedural construction processes, Atlas takes a graphics-centric approach by letting developers apply production rules on graphical objects including geometric primitives, marks, groups and collections. This approach entails that we need to design data-graphics join and layout operations differently compared to existing visualization grammars. Sections 2.3 and 2.5 elaborate on these differences.

Protovis and D3. Protovis argues that to improve accessibility, designers should be able to think in terms of graphical marks, not abstract specifications [7]. D3 allows users to directly manipulate graphical marks by leveraging the document object model (DOM) [8]. Similarly, Atlas adopts a graphics-centric approach. It differs from Protovis by explicitly formulating a set of grammatical rules with clearly defined graphical input and output. Unlike D3, Atlas defines its own visualization object model instead of using the DOM, hence decoupling visualization logic from rendering technologies.

Visualization Authoring Frameworks. The grammatical rules in Atlas are directly informed by recent visualization authoring frameworks and tools [18]. Specifically, the graphics-centric approach in Atlas is consistent with the lazy data binding philosophy in Data Illustrator [12], and the glyph generation rules in Atlas are derived from the repeat and partition operators in Data Illustrator [12]; Charticulator's constraint-based layout mechanism [16] inspired the spatial arrangement rules in Atlas.

5 Conclusion and Future Work

Atlas offers a different perspective on the design of visualization grammars that goes beyond text specification. Taking a graphics-centric approach, Atlas synthesizes concepts and operators from authoring tools, and integrates rule-based grammars into its design.

Several unsolved questions need to be explored as future directions. The current implementation of Atlas.js imposes certain limitations on the order in which the grammatical rules are applied. For example, the divide and densify rules must be applied after repeat; constraints such as affix and align need to be specified after layouts. Such restrictions should be removed in future implementations. Secondly, Atlas in its current form does not support visualizations of trees, networks or geographic data. How the procedural and graphics-centric approach can be extended to these visualization types is an open question. Finally, we need to conduct studies to validate the benefits of Atlas in terms of learnability and explainability.

REFERENCES

[1] Atlas: Infographics with Dumbbell Chart. https://atlas-vis.github.io/gallery.html#DumbbellChart.

[2] Atlas: Small Multiples Bar Charts. https://atlas-vis.github.io/gallery.html#MultipleBarCharts.

[3] Densify (Editing)—ArcGIS Pro | Documentation. https://pro.arcgis.com/en/pro-app/latest/tool-reference/editing/densify.htm.

[4] Divergent Bars in ggplot2. https://onceupondata.com/2019/01/25/ggplot2-divergent-bars/.

[5] Observable Plot, May 2021. https://observablehq.com/@observablehq/plot.

[6] Vega: a visualization grammar, Apr. 2021. https://github.com/vega/vega.

[7] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics*, 15(6):1121–1128, 2009. Publisher: IEEE.

[8] M. Bostock, V. Ogievetsky, and J. Heer. D3: data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011. Publisher: IEEE.

[9] N. Chomsky. *Syntactic Structures*. Walter de Gruyter, 2002.

[10] P. Karnick, S. Jeschke, D. Cline, A. Razdan, E. Wentz, and P. Wonka. A Shape Grammar for Developing Glyph-based Visualizations. *Computer Graphics Forum*, 28(8):2176–2188, 2009. doi: 10.1111/j.1467-8659.2009.01428.x

[11] 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. Publisher: IEEE.

[12] 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 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2018.

[13] A. Morin-Chassé. The Stellar Chart: An Elegant Alternative to Radar Charts, Jan. 2021. https://medium.com/nightingale/the-stellar-chart-an-elegant-alternative-to-radar-charts-ae6a6931a28e.

[14] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pp. 614–623. Association for Computing Machinery, Boston, Massachusetts, July 2006. doi: 10.1145/1179352.1141931

[15] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pp. 301–308. Association for Computing Machinery, New York, NY, USA, Aug. 2001. doi: 10.1145/383259.383292

[16] 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. Publisher: IEEE.

[17] A. Satyanarayan and J. Heer. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum*, 33(3):351–360, 2014. doi: 10.1111/cgf.12391

[18] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. Thompson, M. Brehmer, and Z. Liu. Critical reflections on visualization authoring systems. *IEEE transactions on visualization and computer graphics*, 26(1):461–471, 2019. Publisher: IEEE.

[19] 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. Publisher: IEEE.

[20] R. Vuillemot and J. Boy. Structuring Visualization Mock-Ups at the Graphical Level by Dividing the Display Space. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):424–434, Jan. 2018. doi: 10.1109/TVCG.2017.2743998

[21] H. Wickham. *ggplot2: Elegant graphics for data analysis*. Springer, 2009.

[22] H. Wickham. Tidy data. *Journal of statistical software*, 59(10):1–23, 2014.

[23] L. Wilkinson. The grammar of graphics. In *Handbook of computational statistics*, pp. 375–414. Springer, 2012.

[24] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, July 2003. doi: 10.1145/882262.882324

[25] H. Xia, N. Henry Riche, F. Chevalier, B. De Araujo, and D. Wigdor. DataInk: Direct and Creative Data-Oriented Drawing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pp. 1–13. Association for Computing Machinery, Montreal QC, Canada, Apr. 2018. doi: 10.1145/3173574.3173797