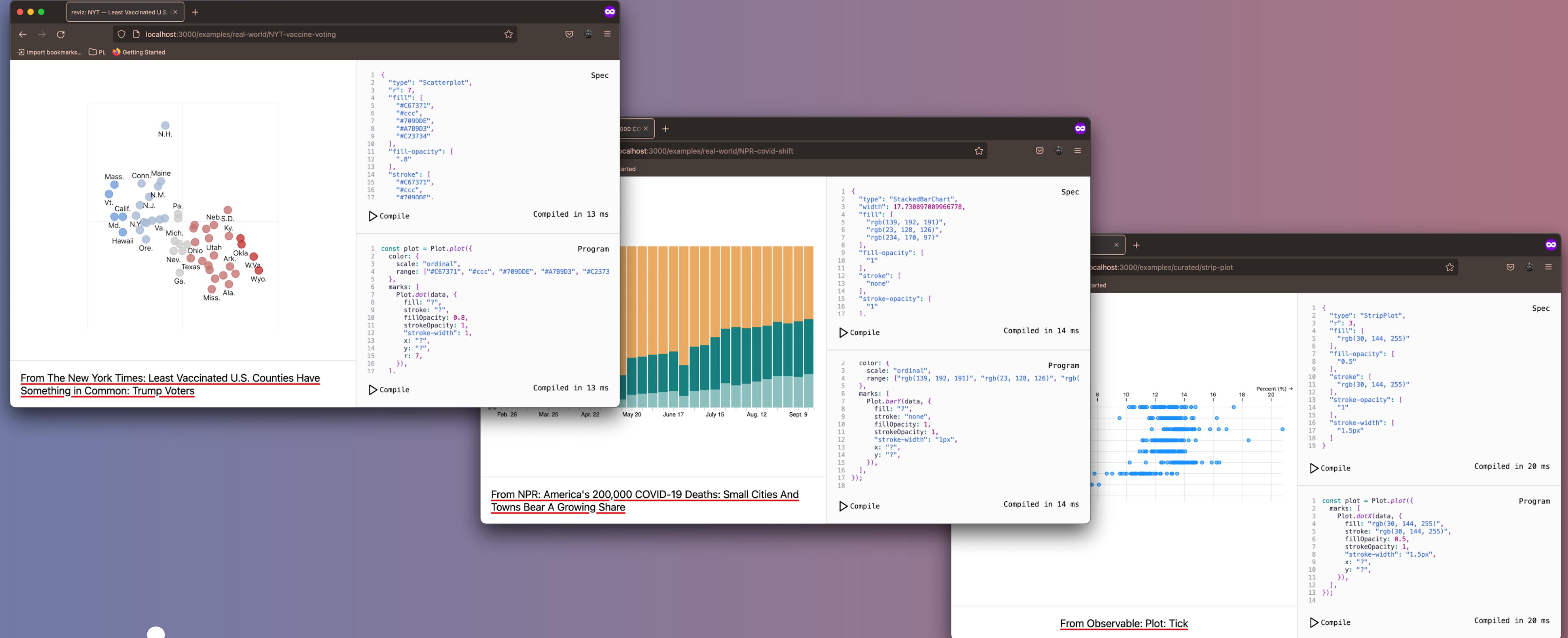


# reviz

A lightweight engine for reverse engineering data visualizations from the DOM



# What is this talk about?

A deep dive into a new  
**programming tool**  
for  
**data visualization authors**

# What is this talk (really) about?

The  
**power of examples**  
to act as  
**intent specification mechanisms**  
for data visualization authors

# What is this talk (really) about?

How can  
**reverse engineering**  
“in-the-wild” examples augment  
**existing techniques**  
authoring data visualizations?

# What is this talk (really) about?

How can **reverse engineering** “in-the-wild” examples augment **existing techniques** for data visualization author?



# *Visualization recommendation*

# Data Visualization

# *Visualization by demonstration*

# Data Visualization

1

# Demonstration

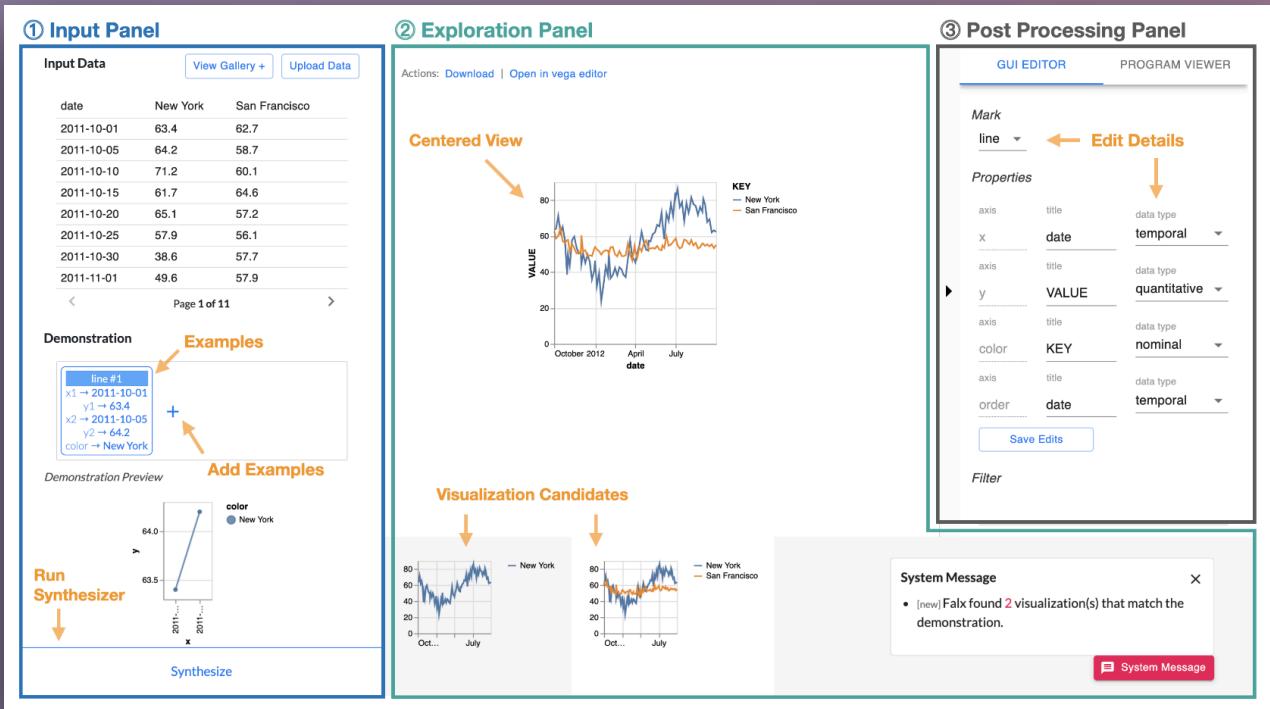
**reviz**

# *Visualization retargeting*

# Data ..... • Visualization

1

## “In-the-wild” Example



# Roadmap

1.

The **promise** of  
(and **problem** with)  
examples in data  
visualization

2.

What is **reviz**? How  
does it address the  
**problem**?

3.

**Questions** and  
**curiosities** for  
future work

# Roadmap

1.

The **promise** of  
(and **problem** with)  
examples in data  
visualization

*“Examples inspire by showing what’s possible; examples demonstrate specific techniques; and **examples are building blocks which help people get started.**”*

**Mike Bostock**

“10 Years of Open Source Visualization”

*“Developers use different strategies in learning APIs ...  
Most developers prefer the strategy of looking at  
examples [12, 21, 28, 33, 36, 38, 40, 41, 45, 51,  
52].”*

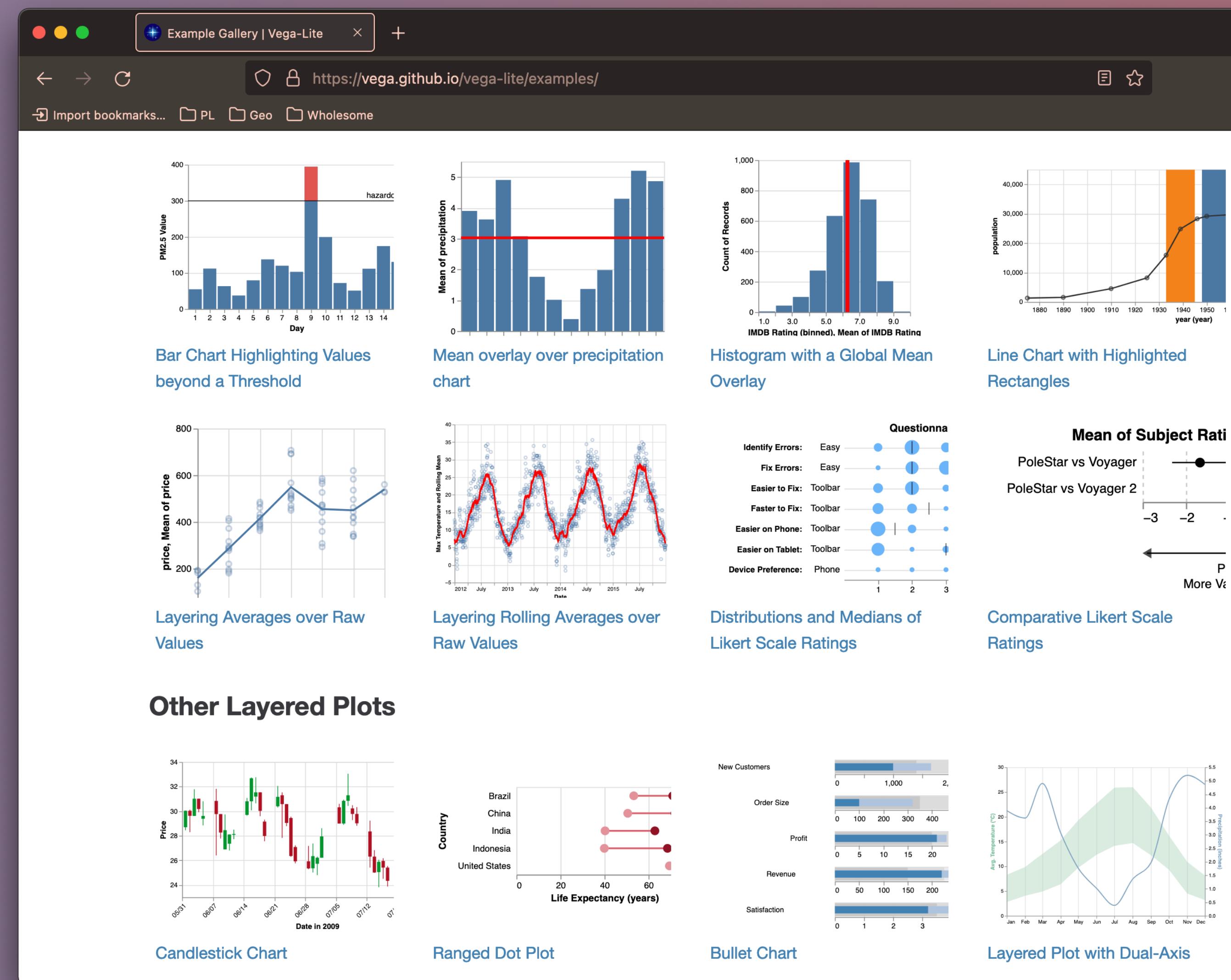
**Kyle Thayer, Sarah E. Chasins, Amy J. Ko**

“A Theory of Robust API Knowledge”

TOCE '21: ACM Transactions on Computing Education

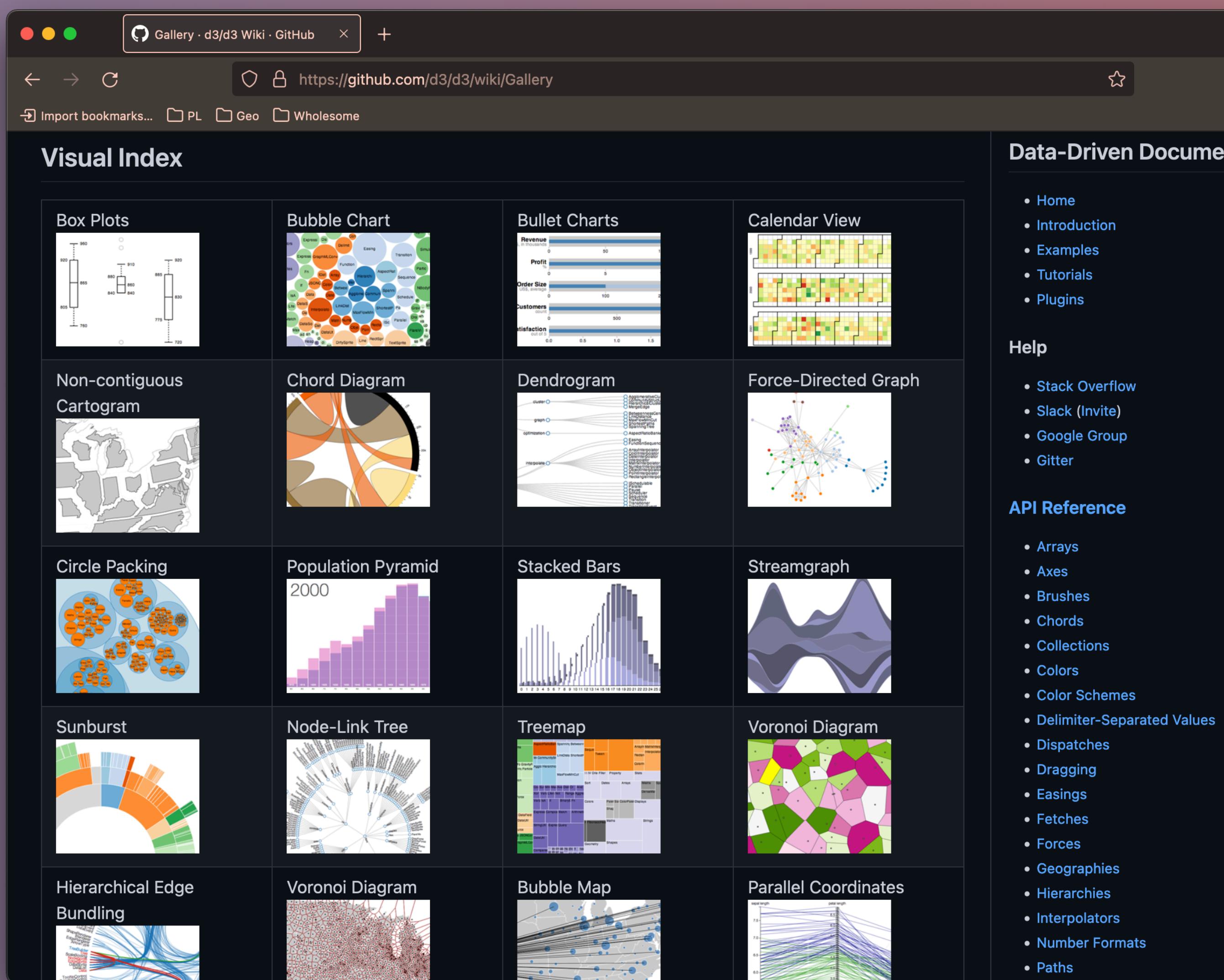
# Examples abound in the data visualization community

vega-lite has  
189 examples  
on its website



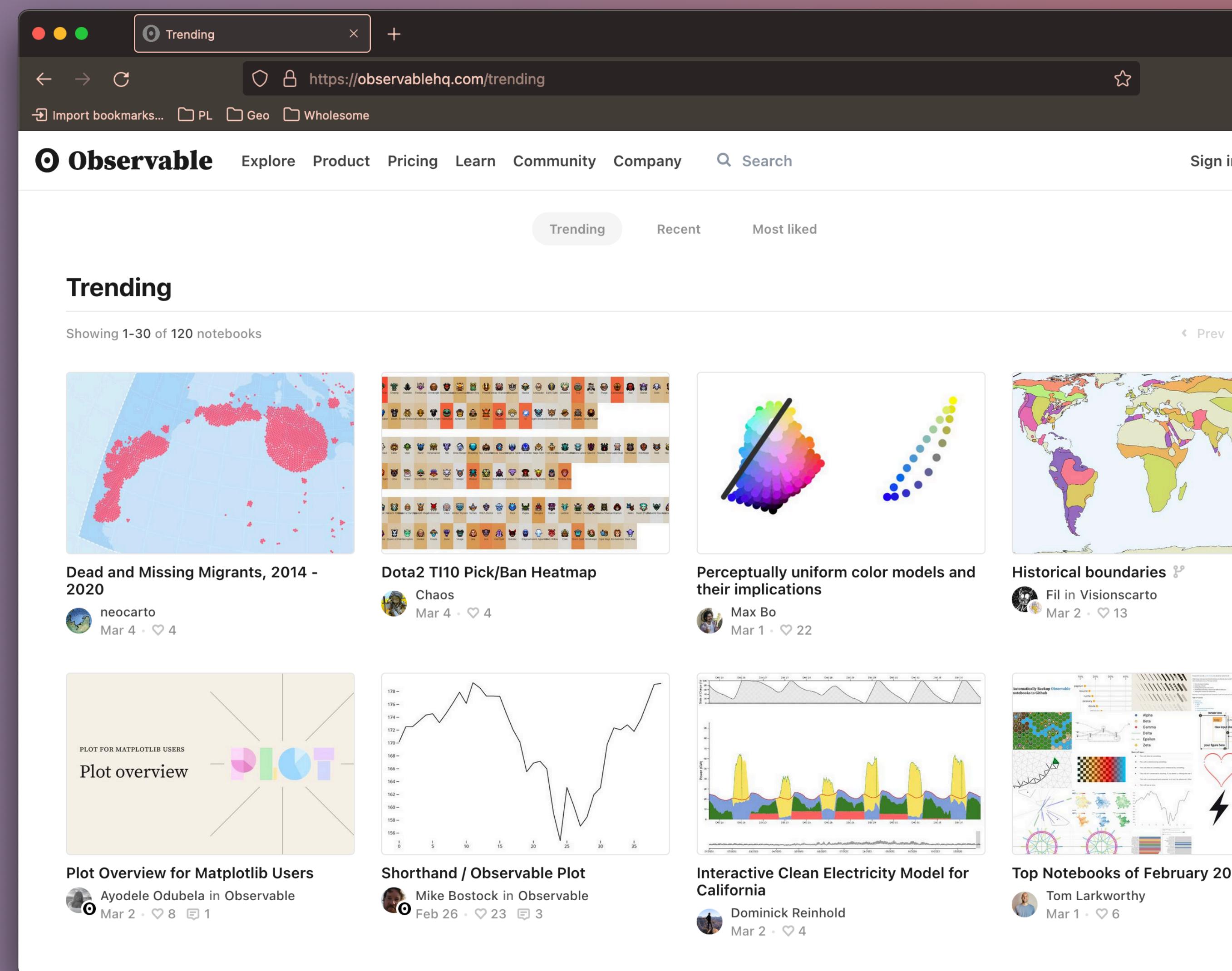
# Examples abound in the data visualization community

D3 has  
**363 examples**  
on its GitHub Wiki

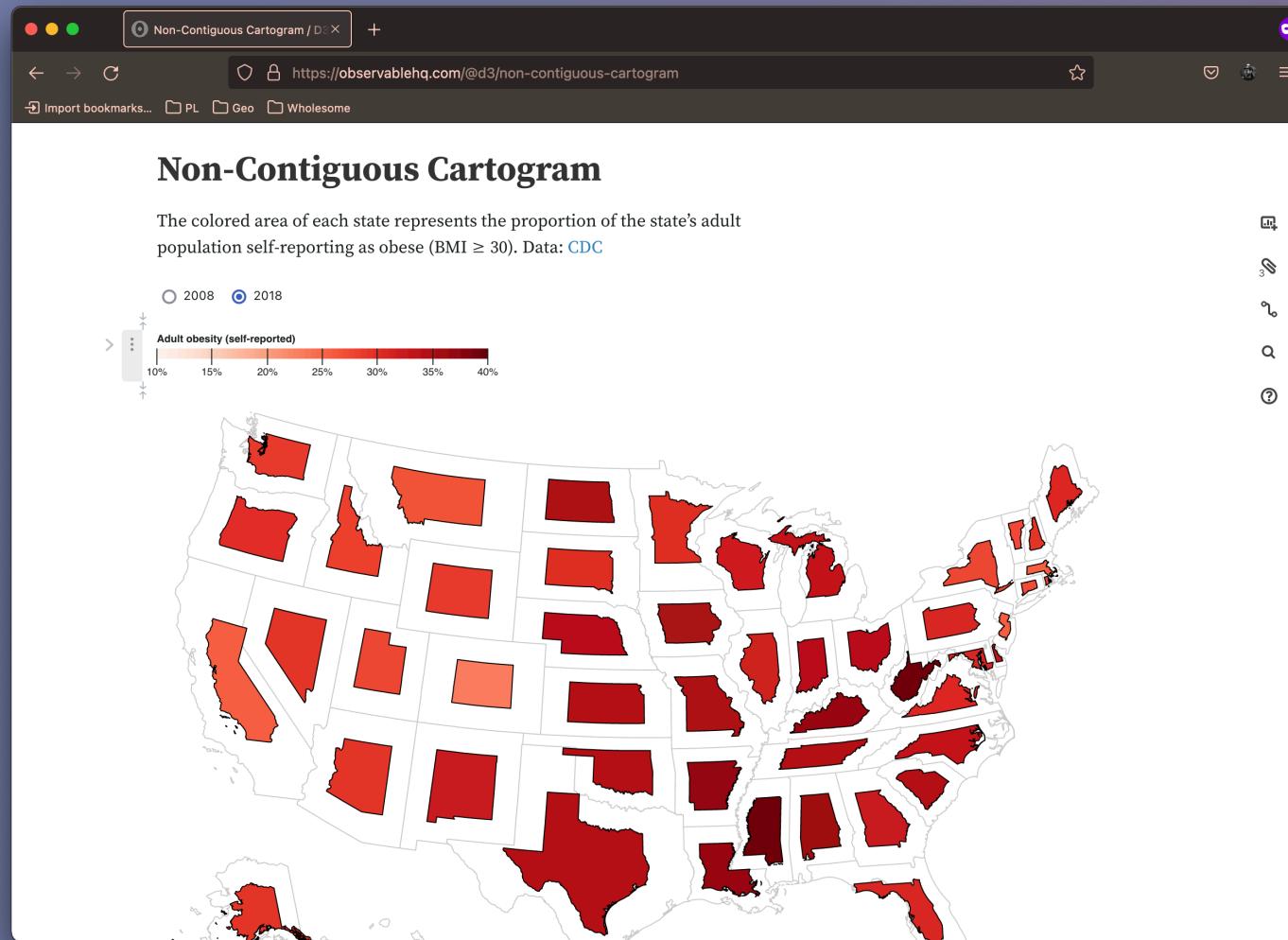


# Examples abound in the data visualization community

Observable has thousands of user-built examples



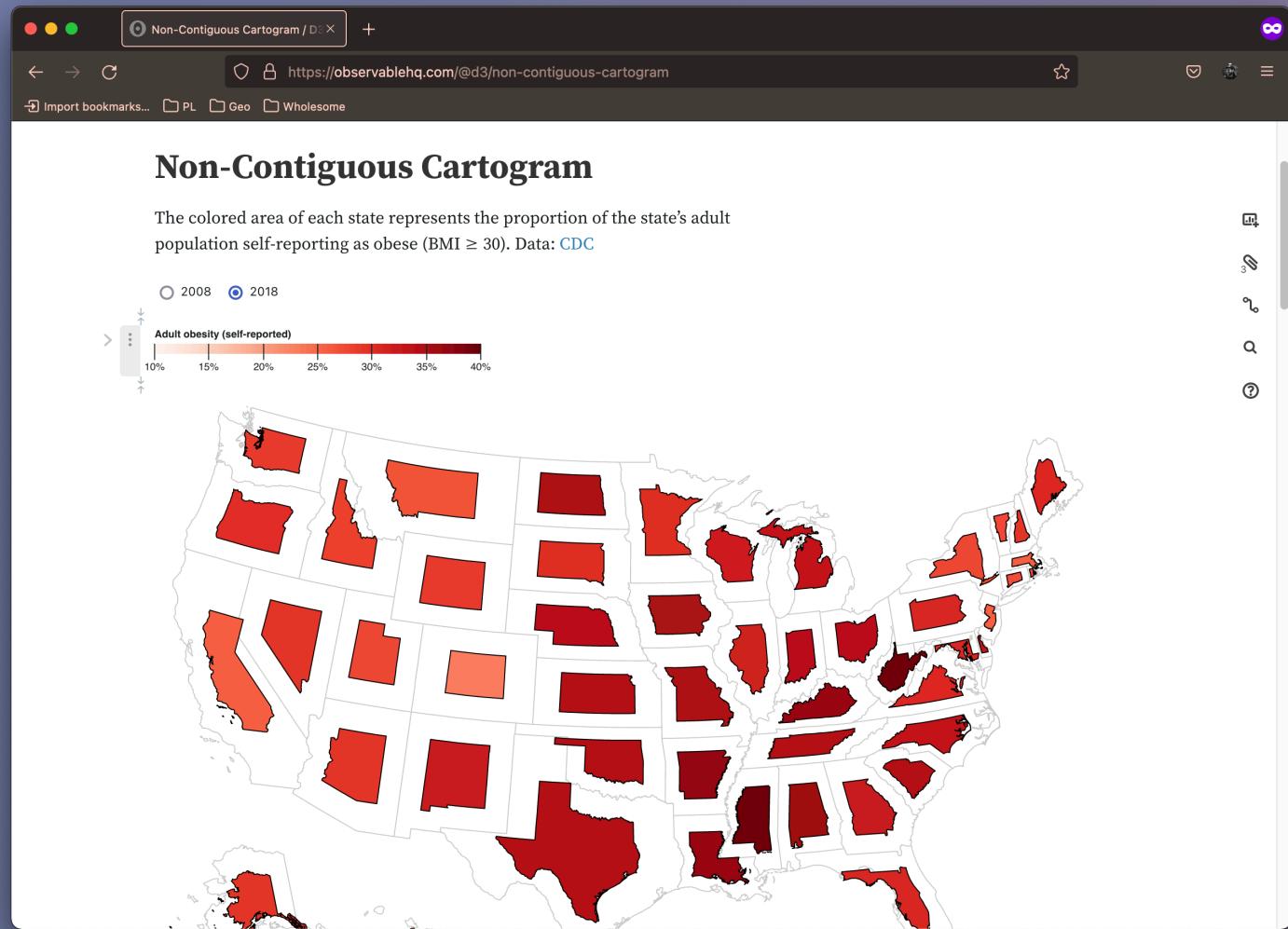
# Why are these examples effective?

A screenshot of the ObservableHQ code editor displaying the D3.js code for the cartogram. The code uses the topojson library to create an SVG map of the US states, applying a non-contiguous stroke effect to each state's boundary. The code also includes logic to update the map for two different years (2008 and 2018) and to add a title with state names and year-specific data.

“They’re very lightweight, they’re informal, they’re something you can do in 15 minutes. They don’t have the ceremony or the overhead of doing published graphics.”

**Mike Bostock**  
“For Example”, EYEO Festival 2013

# Why are these examples effective?

A screenshot of an Observable notebook cell showing the source code for the cartogram. The code uses D3.js to create an SVG, append a path for the US state boundaries, and then apply a color scale to each state based on its obesity rate. It also includes logic for updating the visualization over time (2008 vs 2018).

- **Colocation** of source code with the visualization
- **Immediate** interaction between source code and visualization
- Ability to see a visualization applied to your own data

# Why are these examples effective?

Ability to see a visualization **applied to your own data**

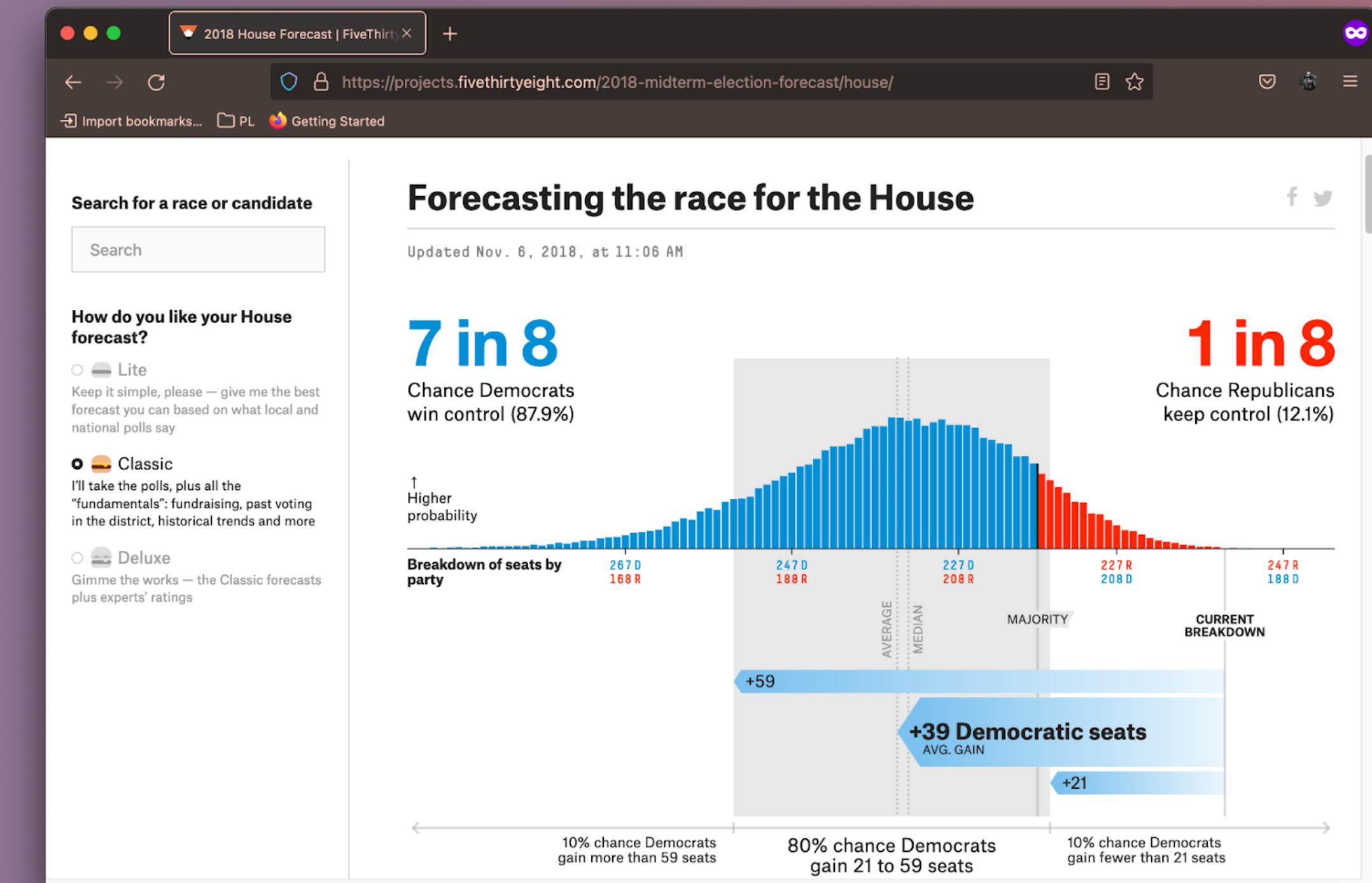
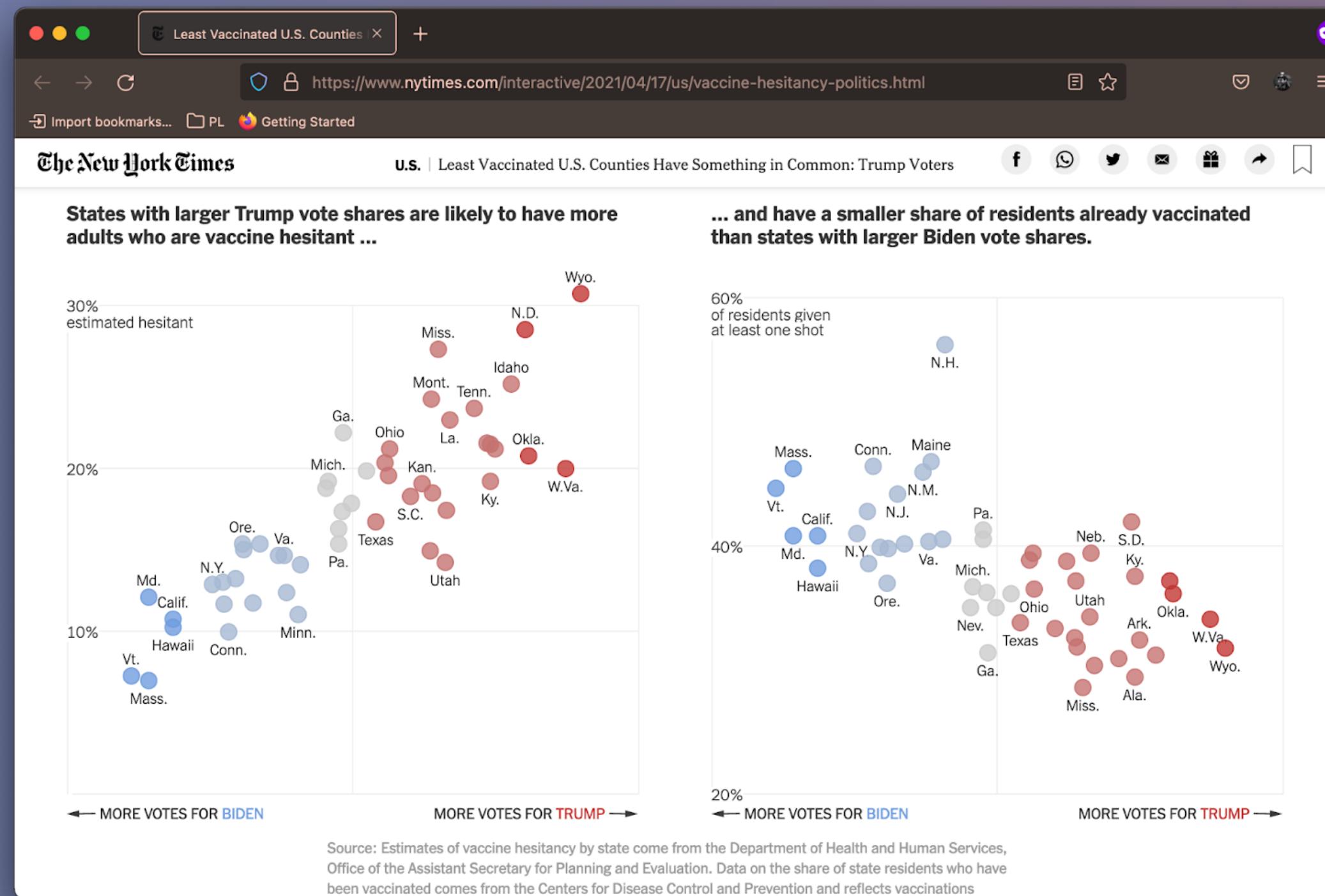
“Does this visualization encoding show me something **unexpected** about my data?”

*Exploratory*

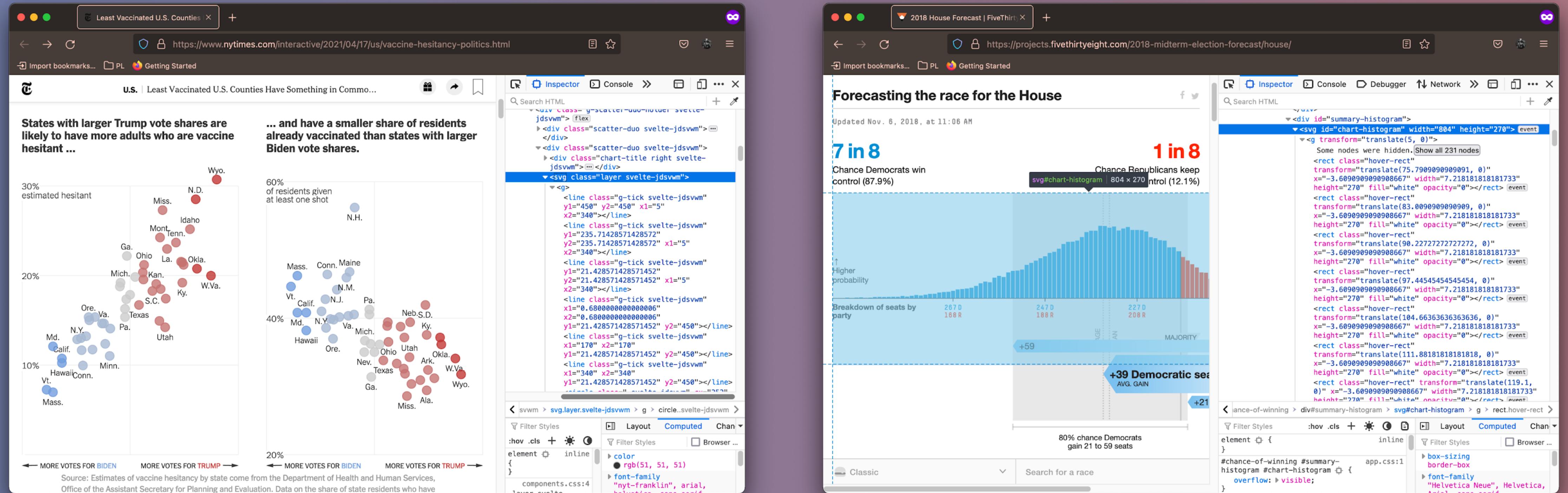
“Would an alternative encoding **better represent** what I want to show about my data?”

*Explanatory*

# But what if the **code** powering an example **isn't visible**?

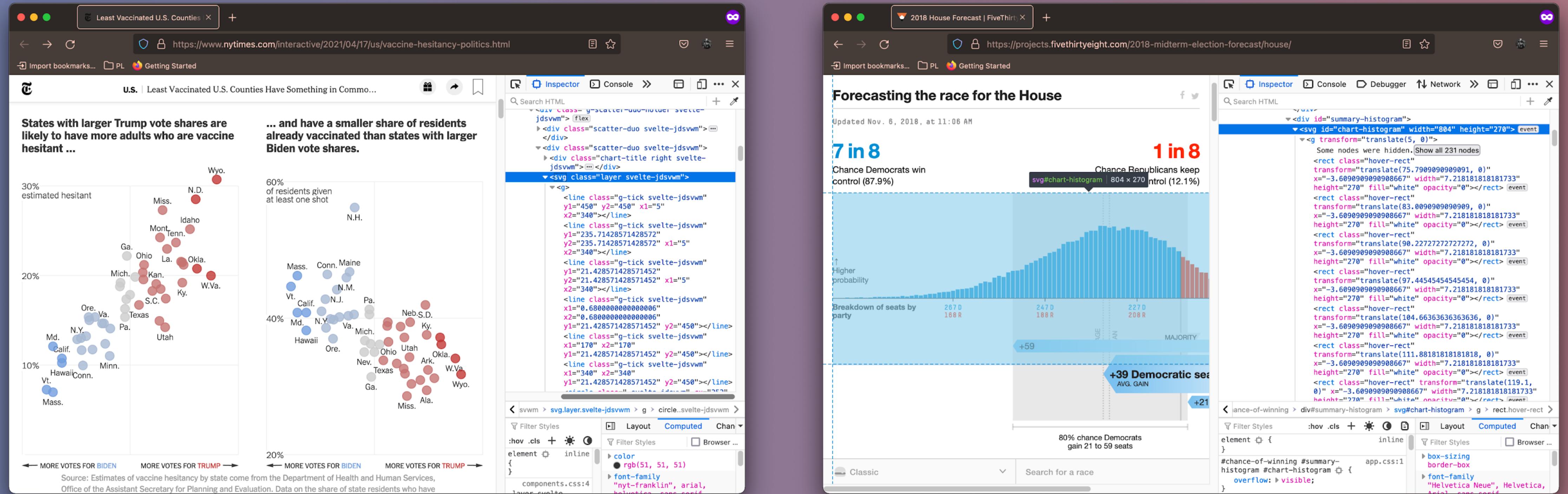


# But what if the code powering examples isn't visible?



We *could* manually reverse engineer them.

# But what if the code powering examples isn't visible?



*Could we automatically  
reverse engineer them?  
We could manually reverse engineer them?*

# Roadmap

1.

The **promise** of  
(and **problem** with)  
examples in data  
visualization

# Roadmap

1.

The **promise** of  
(and **problem** with)  
examples in data  
visualization

2.

What is **reviz**? How  
does it address the  
**problem**?



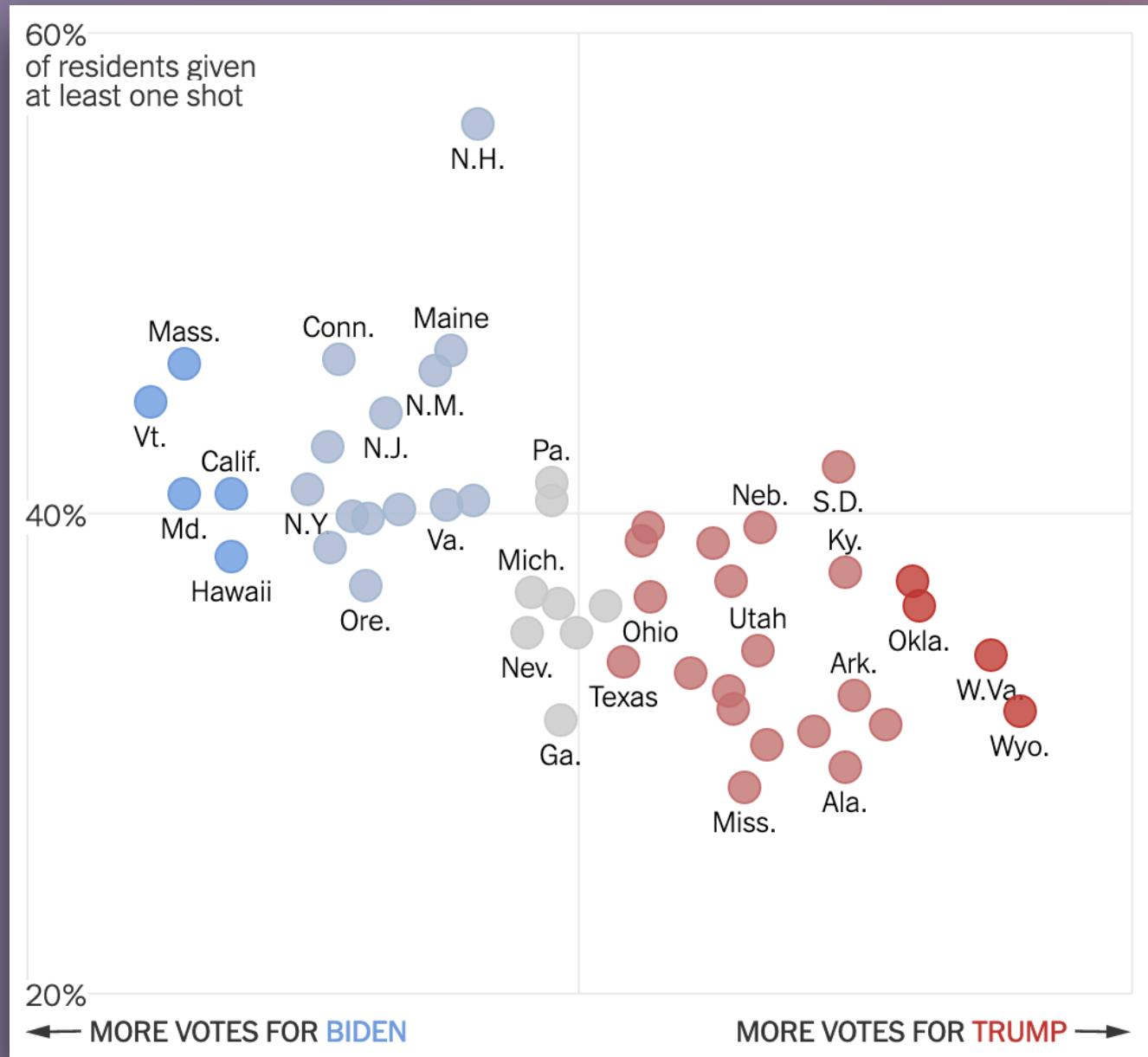
A “compiler” from **SVG subtrees** to **data visualizations**



A “compiler” from **SVG subtrees** to **data visualizations**  
**partial programs in Observable Plot**

# How does it work?

A user **identifies a visualization** they want to use for retargeting.

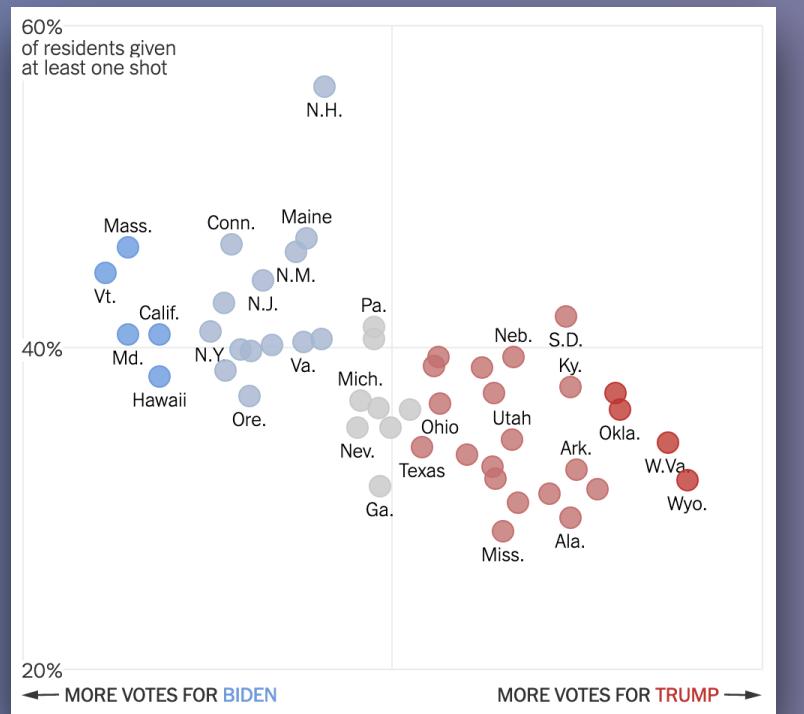


“Least Vaccinated U.S. Counties Have Something in Common: Trump Voters” from *The New York Times*  
<https://www.nytimes.com/interactive/2021/04/17/us/vaccine-hesitancy-politics.html>

# How does it work?

reviz  
v l v l v l v l

1.



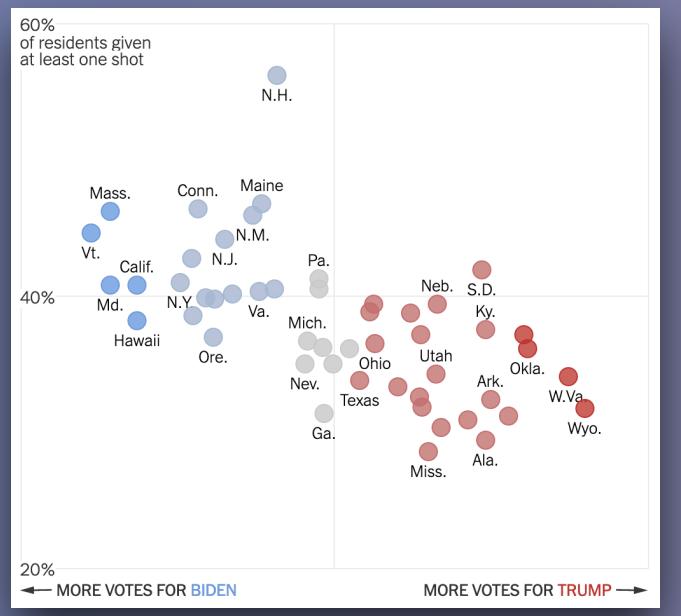
They pass its `svg`  
`subtree` to `reviz`.

reviz  
v l v l v l v l



# How does it work?

1.



reviz generates a *partial JavaScript program* using Observable's Plot library.

2.

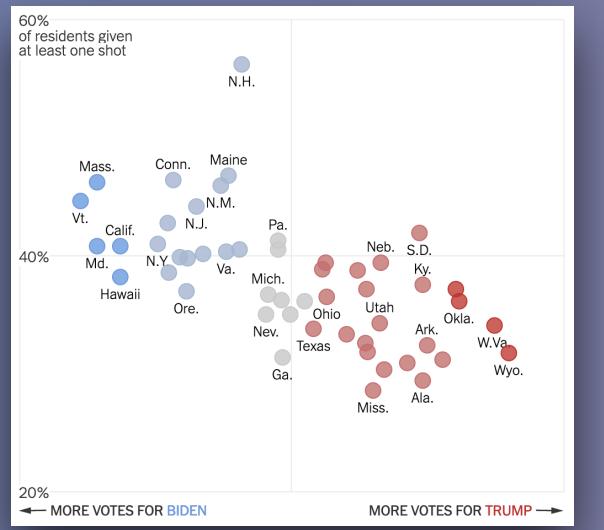


Program

```
1 const plot = Plot.plot({
2   color: {
3     type: "categorical",
4     range: ["#C67371", "#ccc", "#709DDE", "#A7B9D3", "#C23734"],
5   },
6   marks: [
7     Plot.dot(data, {
8       x: "??",
9       y: "?",
10      r: 7,
11      fill: "?",
12      fillOpacity: 0.8,
13      stroke: "?",
14      strokeOpacity: 1,
15      strokeWidth: 1,
16    }),
17  ],
18 });
19
```

# How does it work?

1.



The user **fills in the “holes”**  
in the partial program to get  
a retargeted visualization.

2.

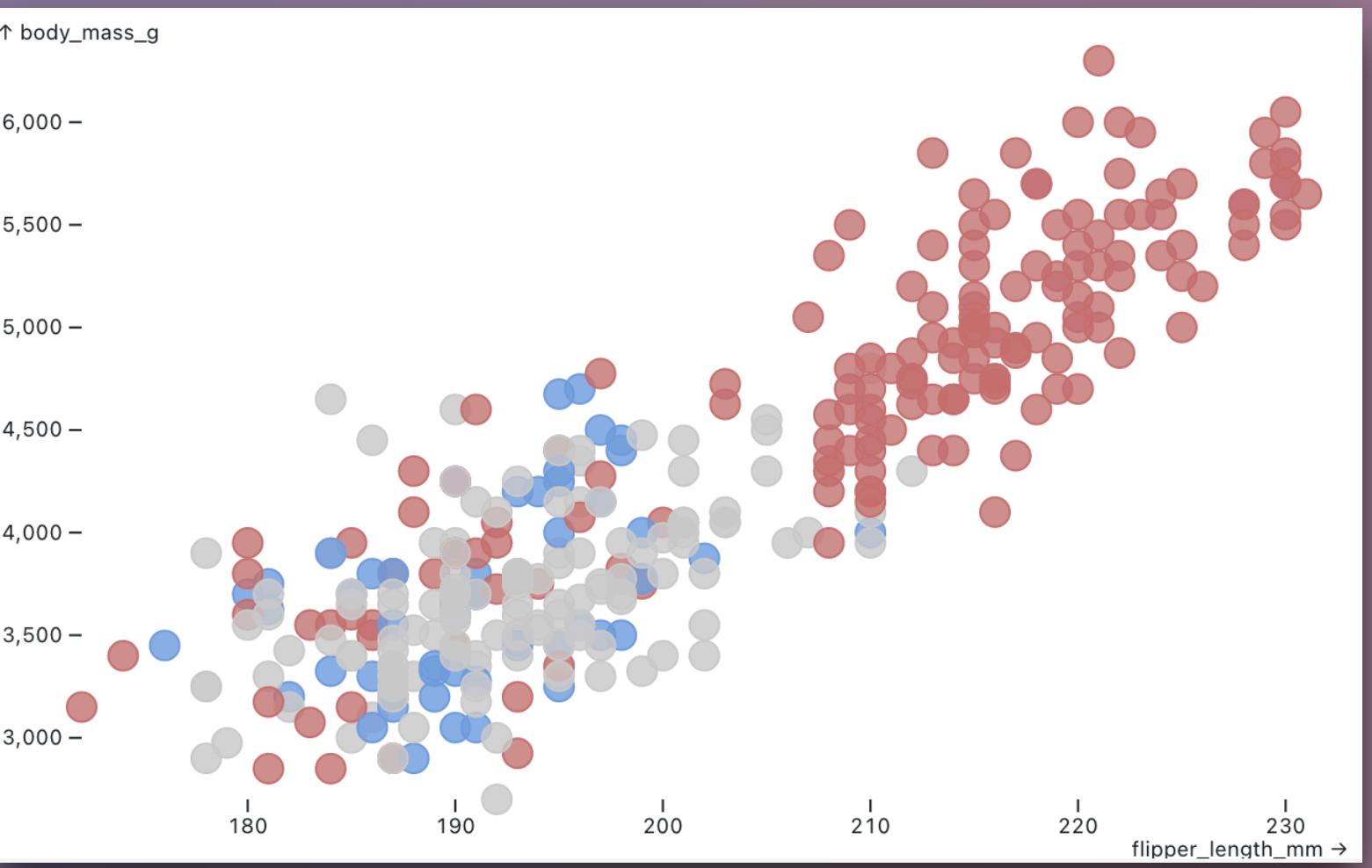


3.

```

1 const plot = Plot.plot({
2   color: {
3     type: "categorical",
4     range: ["#C67371", "#ccc", "#709DDE", "#A7B9D3", "#C23734"],
5   },
6   marks: [
7     Plot.dot(data, {
8       x: "?",
9       y: "?",
10      r: 7,
11      fill: "?",
12      fillOpacity: 0.8,
13      stroke: "?",
14      strokeOpacity: 1,
15      strokeWidth: 1,
16    }),
17    ],
18  });
19

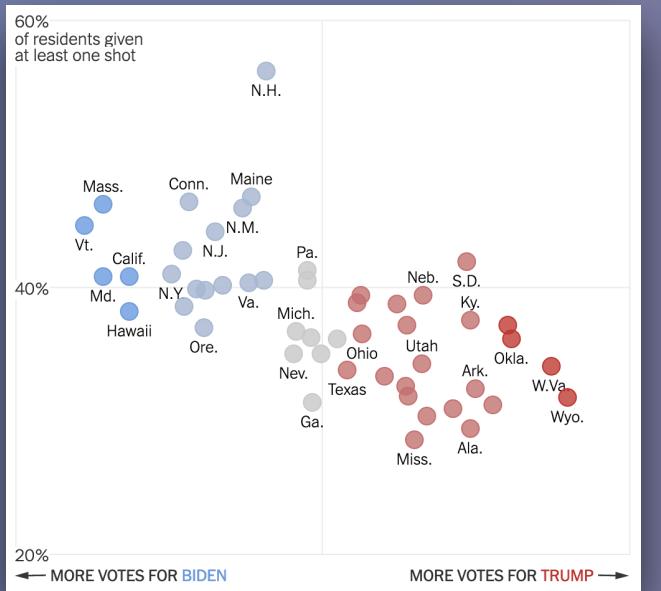
```



# How does it work?



1.



2.

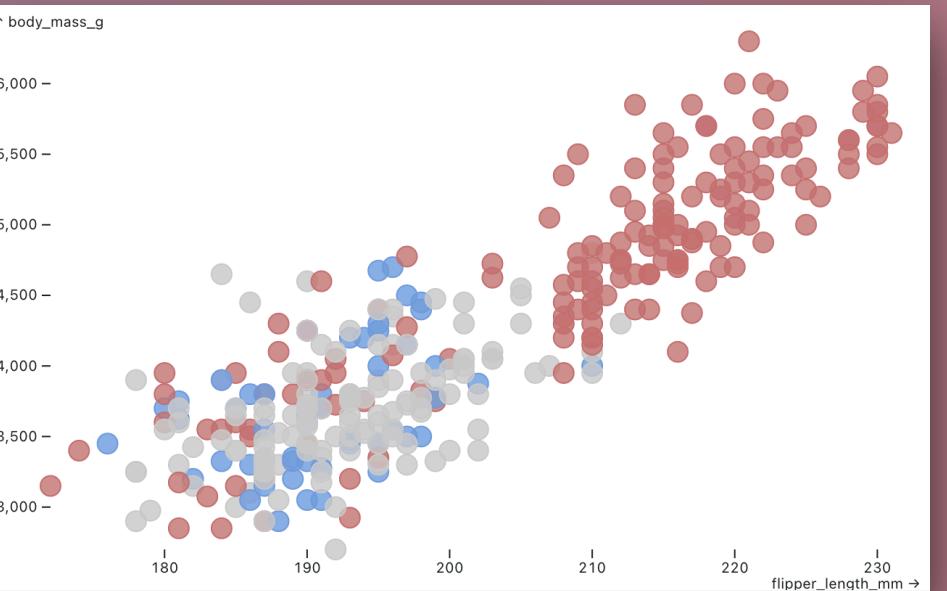


3.

```
1 const plot = Plot.plot({
2   color: {
3     type: "categorical",
4     range: ["#C67371", "#ccc", "#709DDE", "#A7B9D3", "#C23734"],
5   },
6   marks: [
7     Plot.dot(data, {
8       x: "?",
9       y: "?",
10      r: 7,
11      fill: "?",
12      fillOpacity: 0.8,
13      stroke: "?",
14      strokeOpacity: 1,
15      strokeWidth: 1,
16    }),
17  ],
18 });
19
```

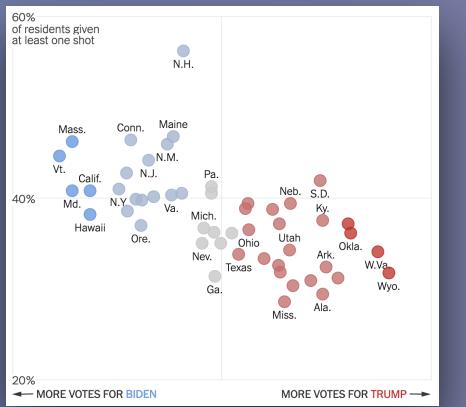
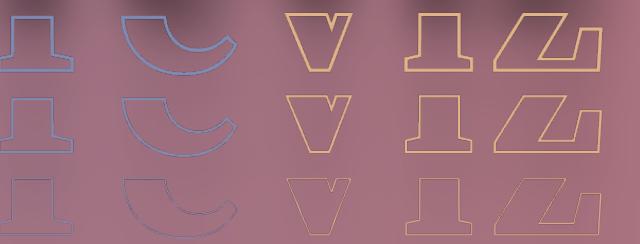
Program

4.



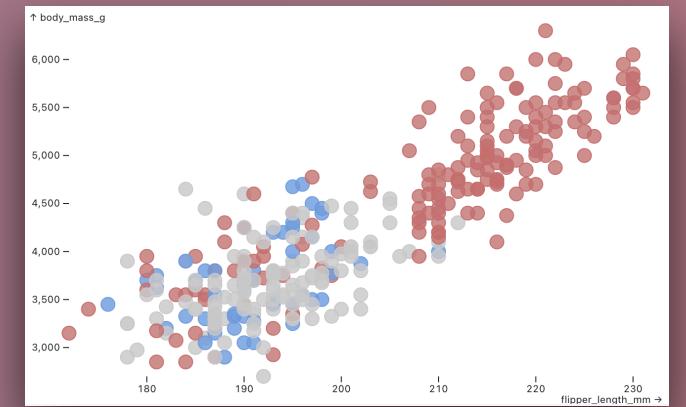
# How does it work?

reviz



Program

```
1 const plot = Plot.plot({
2   color: {
3     type: "categorical",
4     range: ["#C67371", "#ccc", "#7090DE", "#A7B9D3", "#C23734"],
5   },
6   marks: [
7     Plot.dot(data, {
8       x: "?",
9       y: "?",
10      r: 7,
11      fill: "?",
12      fillOpacity: 0.8,
13      stroke: "?",
14      strokeOpacity: 1,
15      strokeWidth: 1,
16    }),
17  ],
18 });
19 
```



reviz

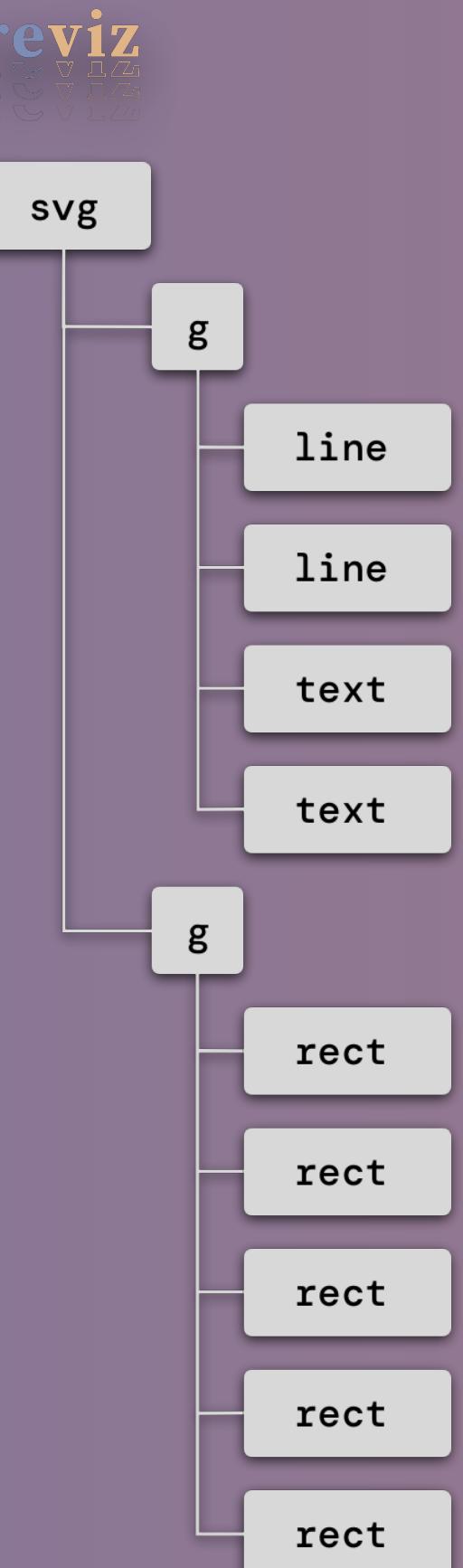


What **actually happens** in this step?

# What actually happens in this step?



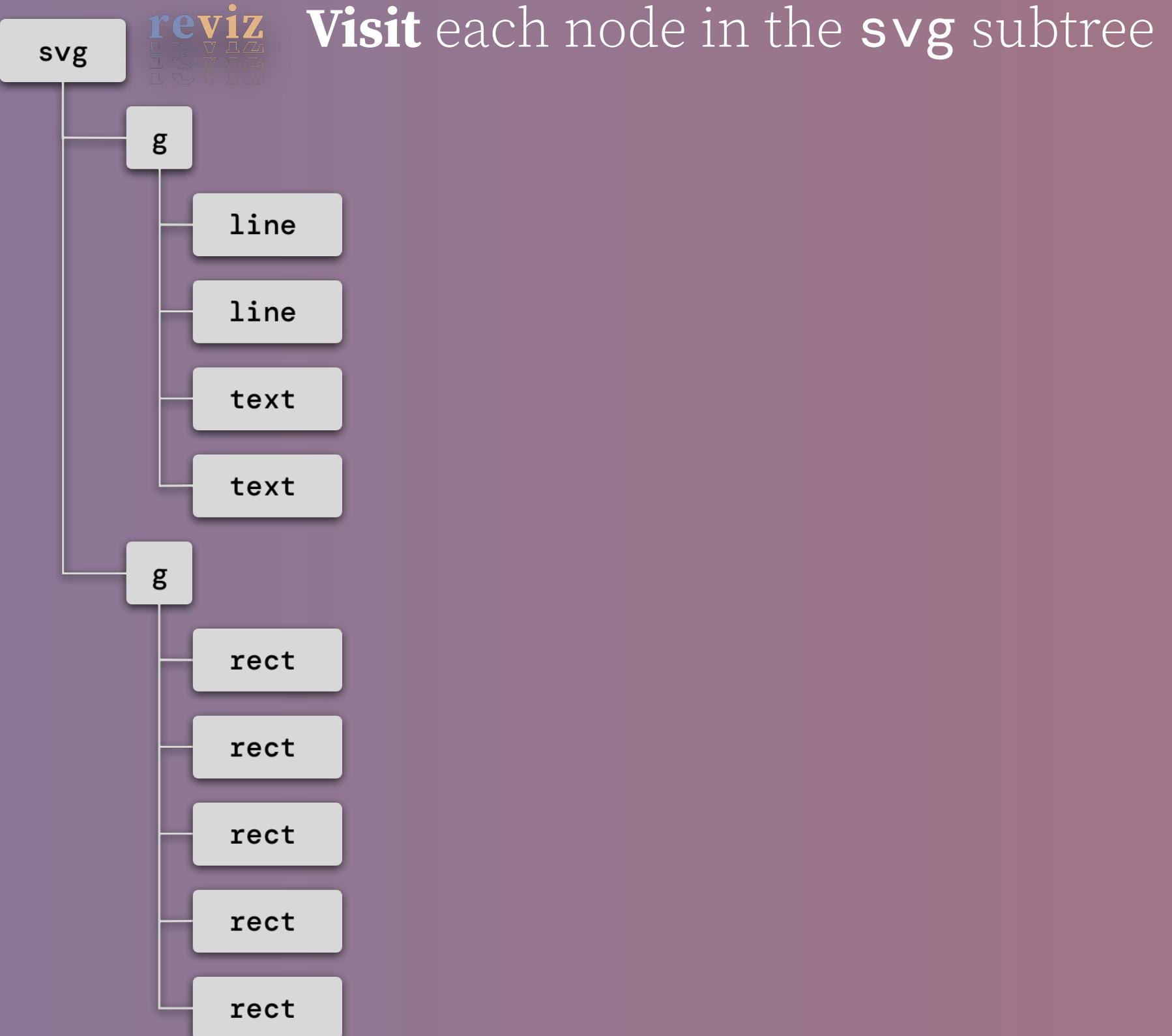
It all starts with  
a **walk**.



# What actually happens in this step?



It all starts with  
a **walk**.



# What actually happens in this step?



It all starts with  
a **walk**.

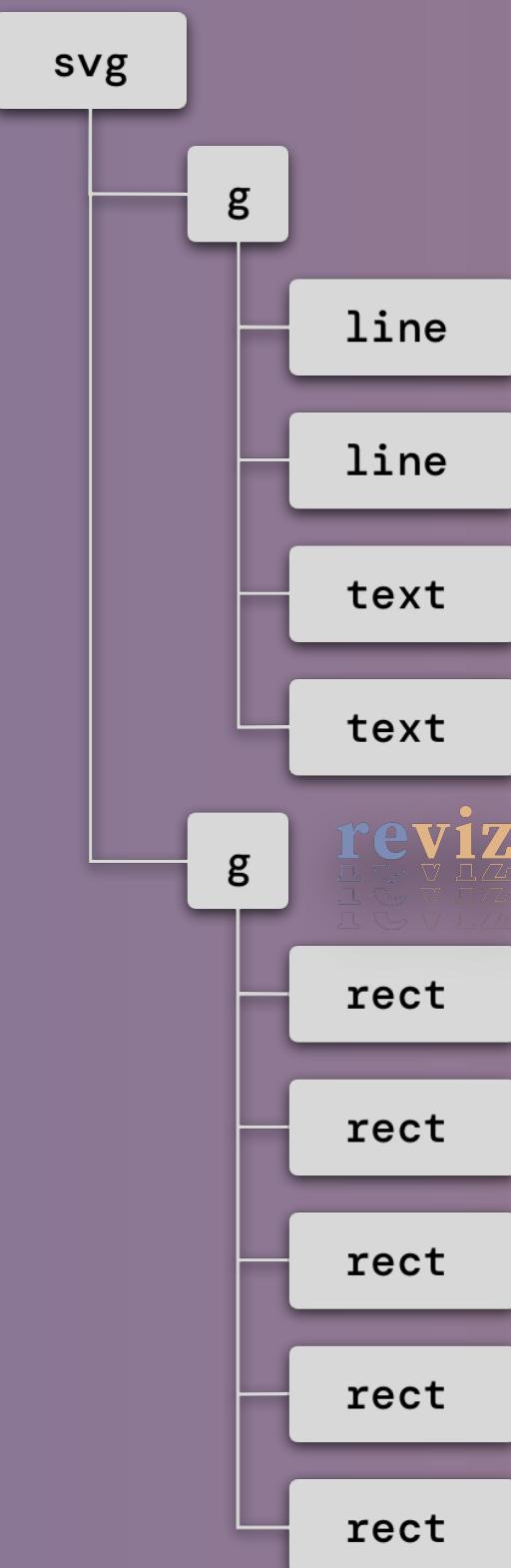


# What actually happens in this step?



1.

**Visit** each node in  
the **svg** subtree

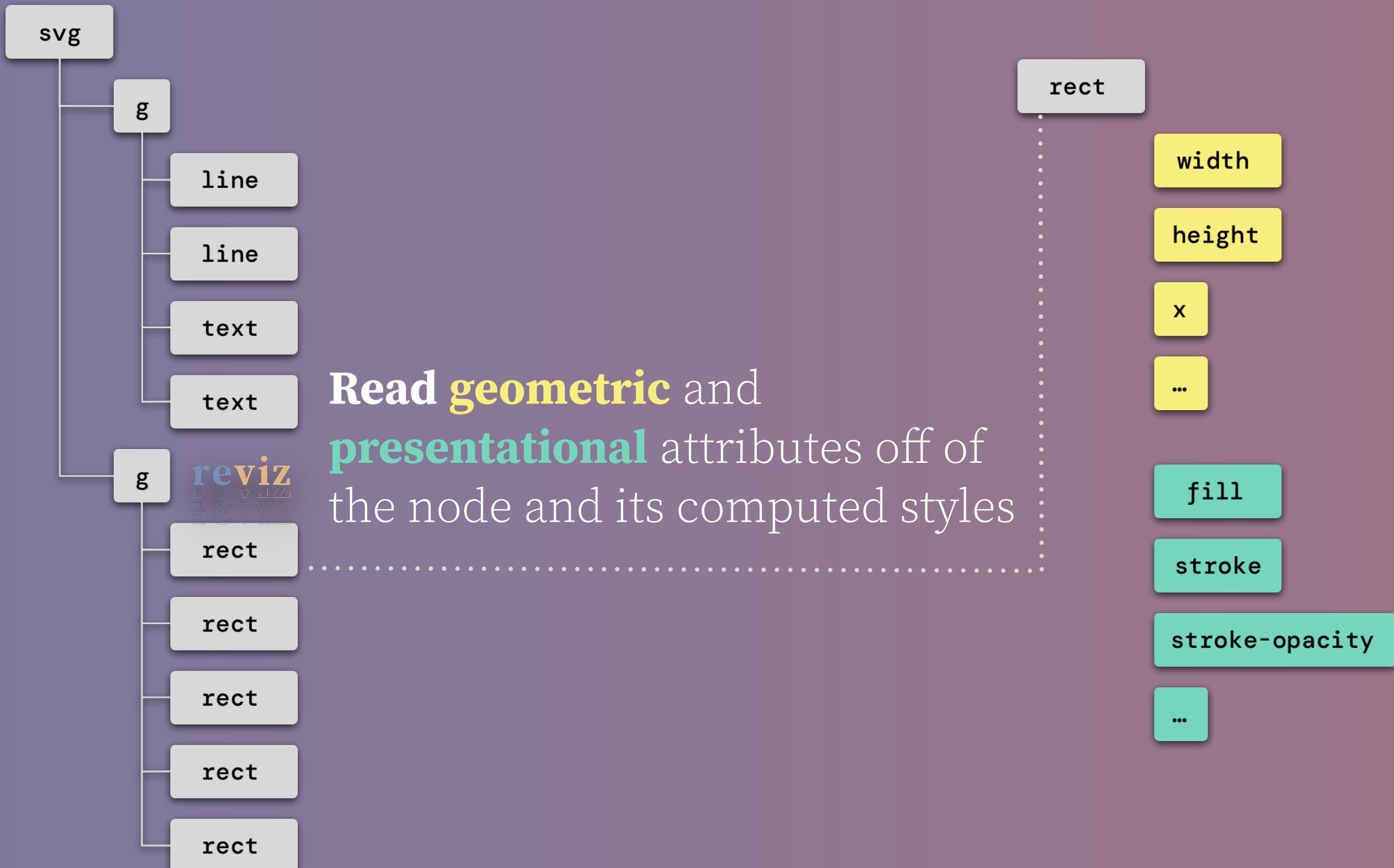


**Read geometric** and  
**presentational** attributes off of  
the node and its computed styles

# What actually happens in this step?

1.

**Visit** each node in  
the **svg** subtree



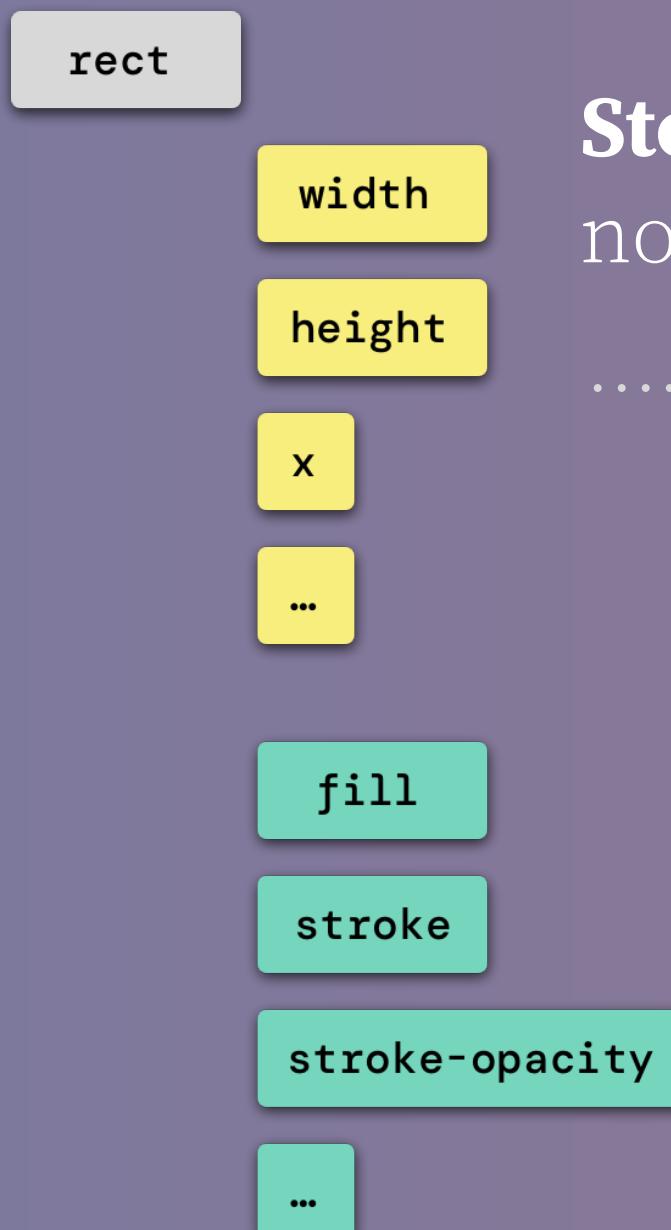
# What actually happens in this step?

1.

**Visit** each node in  
the **svg** subtree

2.

Read **geometric** and **presentational** attributes off of the node and its computed styles



# Store attributes of all nodes in **attribute sets**

```
{  
  width: {'20'},  
  height: {'10', '13', '27', '3', ...},  
  x: {'133', '353', '27', ...}  
  fill: {'#7b16ff'},  
  stroke: {'none'},  
  strokeOpacity: {'1'}  
}
```

# What actually happens in this step?

1.

**Visit** each node in the **svg** subtree

2.

**Read geometric** and **presentational** attributes off of the node and its computed styles

```
{
  width: {'20'},
  height: {'10', '13', '27', '3', ...},
  x: {'133', '353', '27', ...}
  fill: {'#7b16ff'},
  stroke: {'none'},
  strokeOpacity: {'1'}
}
```

Apply a set of **predicate functions** associated with a **visualization type**

{ type: "BarChart" }

hasMarkType('rect') ✓

hasConsistentGeomAttr('width') ✓

hasXScaleType('discrete') ✗

3.

**Store** attributes of all nodes in **attribute sets**

# What actually happens in this step?

4.

Apply a set of  
**predicate  
functions**  
associated with a  
**visualization type**

Compute the “**true rate**”  
associated with a  
**visualization type**

```
{ type: "BarChart" }
```

```
hasMarkType('rect') ✓
```

numPredicates = 3

```
hasConsistentGeomAttr('width') ✓
```

truePredicates = 2

```
hasXScaleType('discrete') ✗
```

trueRate = 0.666

# What actually happens in this step?

4.

Apply a set of  
**predicate**  
**functions**  
 associated with a  
**visualization type**

```
{ type: "BarChart" }
```

```
hasMarkType('rect') ✓
```

```
hasConsistentGeomAttr('width') ✓
```

```
hasXScaleType('discrete') ✗
```

```
numPredicates = 3
```

```
truePredicates = 2
```

```
trueRate = 0.666
```

5.

Compute the “**true**  
**rate**” associated  
 with a  
**visualization type**

```
{ type: "Scatterplot" }
```

```
hasMarkType('circle') ✗
```

```
hasConsistentGeomAttr('r') ✗
```

```
numPredicates = 2
```

```
truePredicates = 0
```

```
trueRate = 0
```

```
{ type: "Histogram" }
```

```
hasMarkType('rect') ✓
```

```
hasConsistentGeomAttr('width') ✓
```

```
hasXScaleType('continuous') ✓
```

```
numPredicates = 3
```

```
truePredicates = 3
```

```
trueRate = 1
```

# What actually happens in this step?

4.

Apply a set of **predicate functions** associated with a **visualization type**

- { type: "Histogram" }
- 
- hasMarkType('rect') ✓
- 
- hasConsistentGeomAttr('width') ✓
- 
- hasXScaleType('continuous') ✓

Select the **visualization type** with the highest "**true rate**"

5.

Compute the "**true rate**" associated with a **visualization type**

- { type: "BarChart" }
- hasMarkType('rect') ✓
- hasConsistentGeomAttr('width') ✓
- hasXScaleType('discrete') ✗

- { type: "Scatterplot" }
- hasMarkType('circle') ✗
- hasConsistentGeomAttr('r') ✗

# What actually happens in this step?

4.

Apply a set of **predicate functions** associated with a **visualization type**

**Visualization type**

```
{ type: "Histogram" }
```

Combine the **visualization type** and **attribute sets** into an IR

5.

Compute the “**true rate**” associated with a **visualization type**

**Attribute sets**

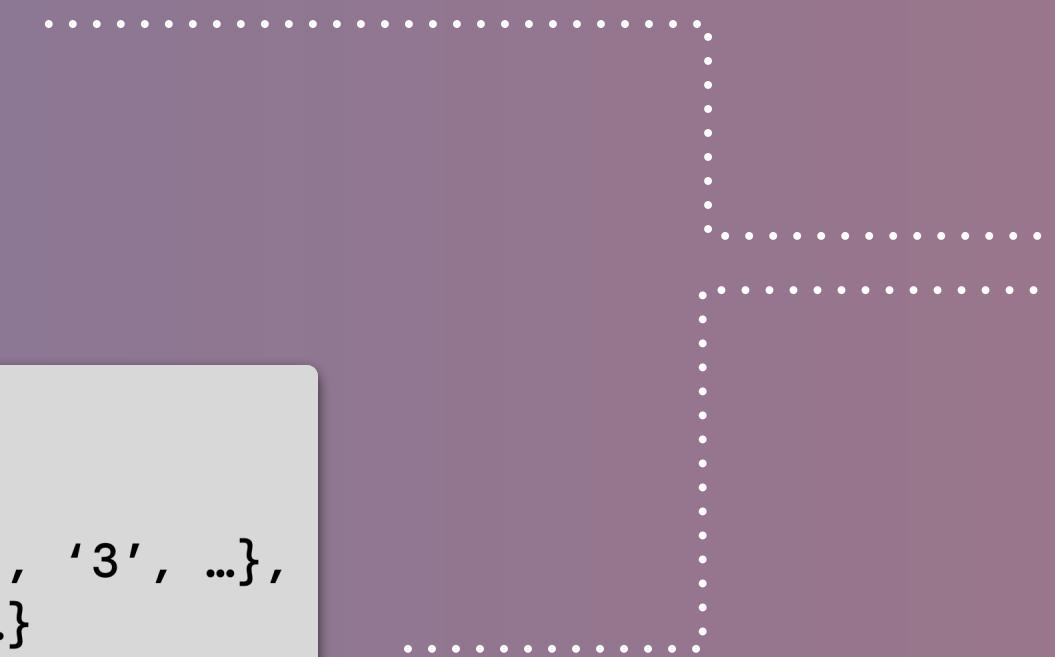
```
{
  width: {'20'},
  height: {'10', '13', '27', '3', ...},
  x: {'133', '353', '27', ...}
  fill: {'#7b16ff'},
  stroke: {'none'},
  strokeOpacity: {'1'}
}
```

6.

Select the **visualization type** with the highest “**true rate**”

**Intermediate Representation**

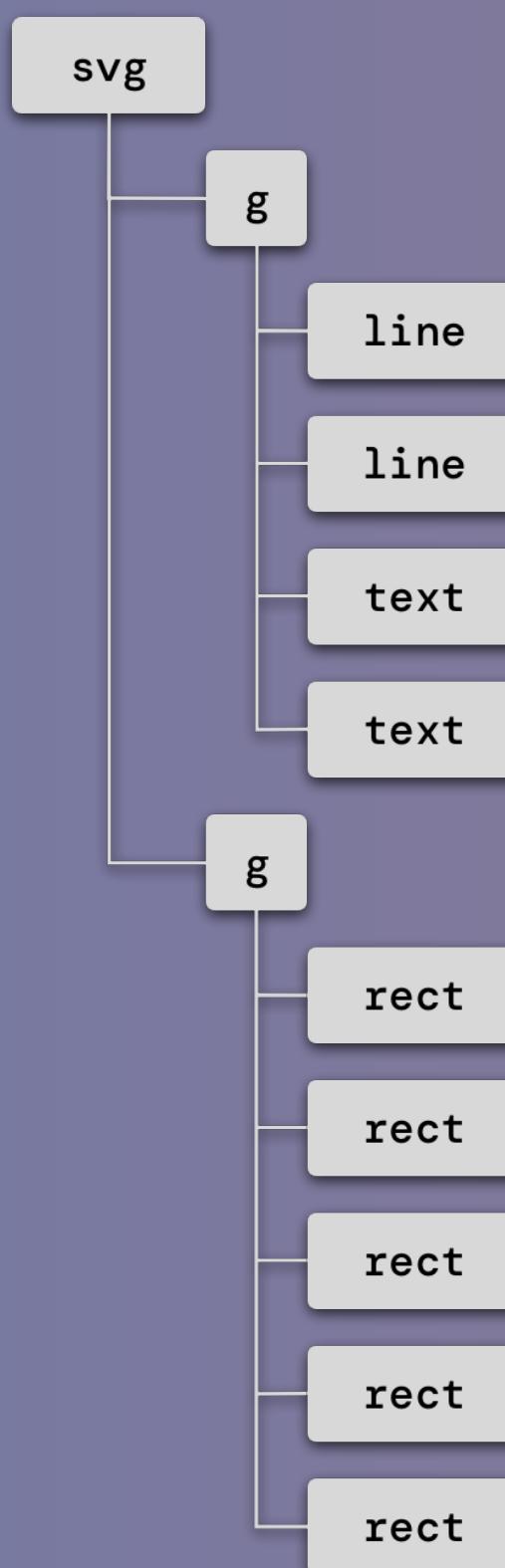
```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```



# Where are we now?



svg subtree



The *frontend* of the reviz compiler

Intermediate Representation

```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

# What actually happens in this step?



## Intermediate Representation

```
{  
  "type": "Histogram",  
  "width": 20,  
  "fill": [  
    "#7b16ff"  
  ],  
  "fill-opacity": [  
    "1"  
  ],  
  "stroke": [  
    "none"  
  ],  
  "stroke-opacity": [  
    "1"  
  ],  
  "stroke-width": [  
    "1px"  
  ]  
}
```

The *backend* of the reviz compiler

.....●

## Partial Observable Plot Program

```
1 const plot = Plot.plot({  
2   marks: [  
3     Plot.barY(  
4       data,  
5       Plot.binX(  
6         { y: "count" },  
7         {  
8           x: "??",  
9           fill: "steelblue",  
10          fillOpacity: 1,  
11          stroke: "none",  
12          strokeOpacity: 1,  
13          strokeWidth: 1,  
14        }  
15      ),  
16    ),  
17  1,  
18});  
19
```

Program

# Where are we now?

## Intermediate Representation

```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

Apply **contextual semantics**  
to rewrite terms in the IR to  
the output program

```
`${camelCase(attrName)}: ${val}${!isLastAttr ? ` , ${EVAL_HOLE}` : ''}`
```

```
`Plot.barY(data,
  Plot.binX(
    {
      y: 'count',
    },
    {
      x: '${PROGRAM_HOLE}',
      ${EVAL_HOLE}
    }
  )
)`
```

# Where are we now?



## Intermediate Representation

```
{  
  "type": "Histogram",  
  "width": 20,  
  "fill": [  
    "#7b16ff"  
  ],  
  "fill-opacity": [  
    "1"  
  ],  
  "stroke": [  
    "none"  
  ],  
  "stroke-opacity": [  
    "1"  
  ],  
  "stroke-width": [  
    "1px"  
  ]  
}
```

## Program

```
`${EVAL_HOLE}`
```

Start with an empty program,  
represented by a “hole”.

# Where are we now?



## Intermediate Representation

```
{  
  "type": "Histogram", .....  
  "width": 20,  
  "fill": [  
    "#7b16ff"  
  ],  
  "fill-opacity": [  
    "1"  
  ],  
  "stroke": [  
    "none"  
  ],  
  "stroke-opacity": [  
    "1"  
  ],  
  "stroke-width": [  
    "1px"  
  ]  
}
```

Apply the **rewrite** for the first key in the IR.

## Program

```
`Plot.barY(data,  
          Plot.binX(  
            {  
              y: 'count',  
            },  
            {  
              x: '${PROGRAM_HOLE}',  
              ${EVAL_HOLE} .....  
            }  
          )`
```

- **Leave a hole** signaling where evaluation should continue.

# Where are we now?

## Intermediate Representation

```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

**Keep applying rewrite rules**  
for keys in the IR.

## Program

```
`Plot.barY(data,
  Plot.binX(
    {
      y: 'count',
    },
    {
      x: '${PROGRAM_HOLE}',
      fill: '#7b16ff',
      ${EVAL_HOLE}
    }
  )
)`
```

- **Keeping holes** signaling where evaluation should continue.

# Where are we now?

## Intermediate Representation

```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

Keep applying  
rewrite rules  
for keys in the  
IR.

## Program

```
`Plot.barY(data,
  Plot.binX(
    {
      y: 'count',
    },
    {
      x: '${PROGRAM_HOLE}',
      fill: '#7b16ff',
      fillOpacity: 1,
      ${EVAL_HOLE}
    }
  )
)`
```

- **Keeping holes**  
leaving holes  
signaling where  
evaluation  
should  
continue.

# Where are we now?

## Intermediate Representation

```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

Return the program once we've iterated through all keys in the IR.

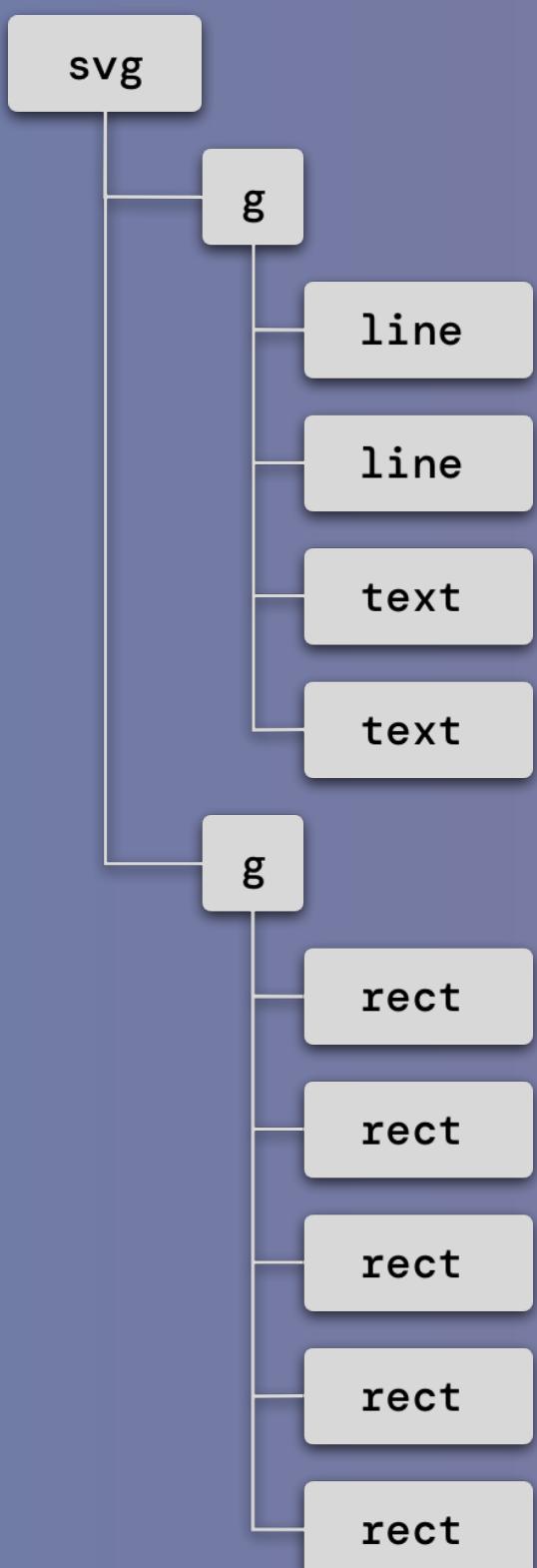
## Program

```
`Plot.barY(data,
  Plot.binX(
    {
      y: 'count',
    },
    {
      x: '${PROGRAM_HOLE}',
      fill: '#7b16ff',
      fillOpacity: 1,
      stroke: 'none',
      strokeOpacity: 1,
      strokeWidth: 1
    }
  )
)`
```

# Where are we now?



svg subtree



The *frontend* of  
the reviz compiler

Intermediate Representation

```
{  
  "type": "Histogram",  
  "width": 20,  
  "fill": [  
    "#7b16ff"  
  ],  
  "fill-opacity": [  
    "1"  
  ],  
  "stroke": [  
    "none"  
  ],  
  "stroke-opacity": [  
    "1"  
  ],  
  "stroke-width": [  
    "1px"  
  ]  
}
```

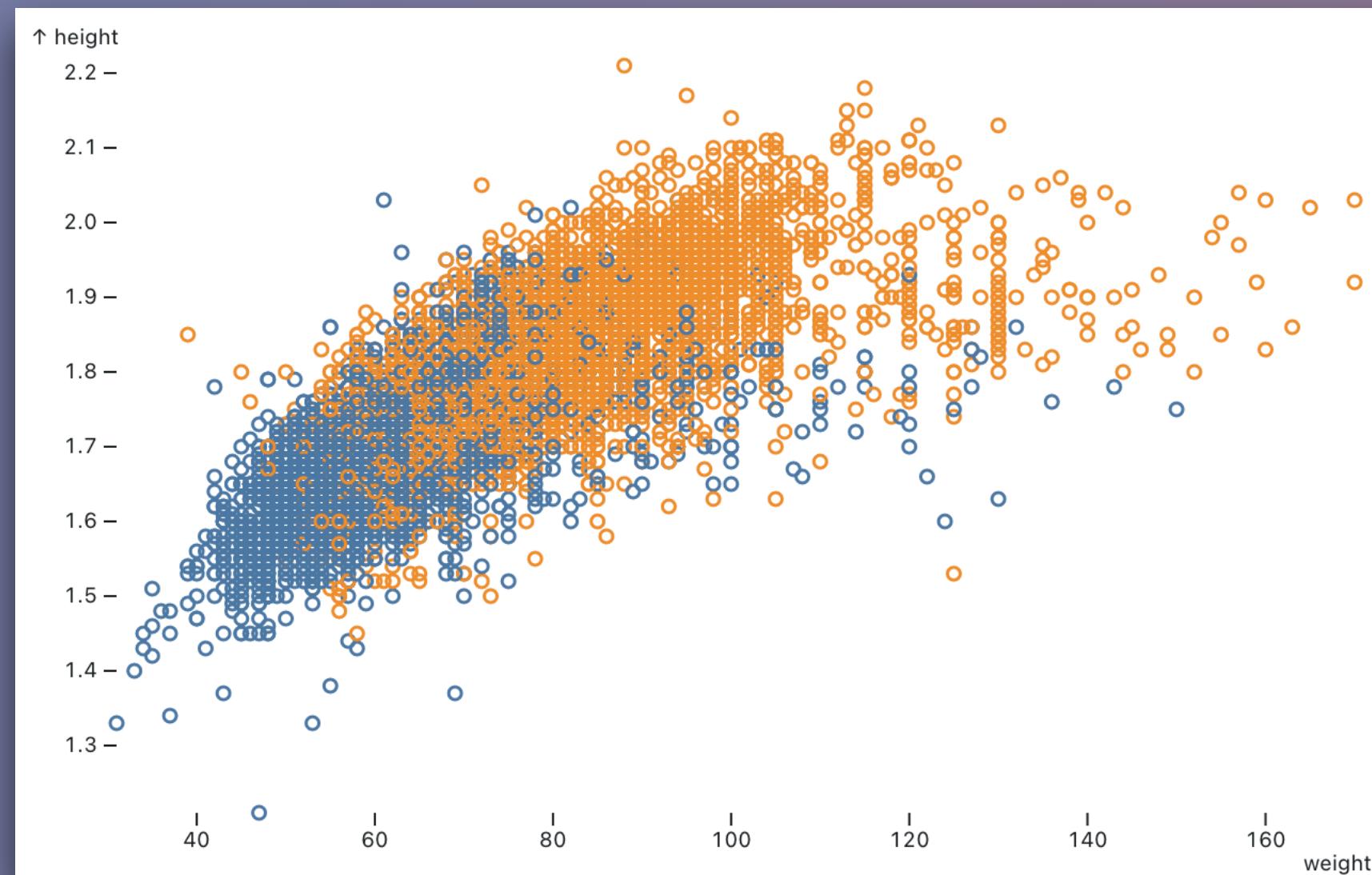
The *backend* of  
the reviz compiler

Partial Observable Plot Program

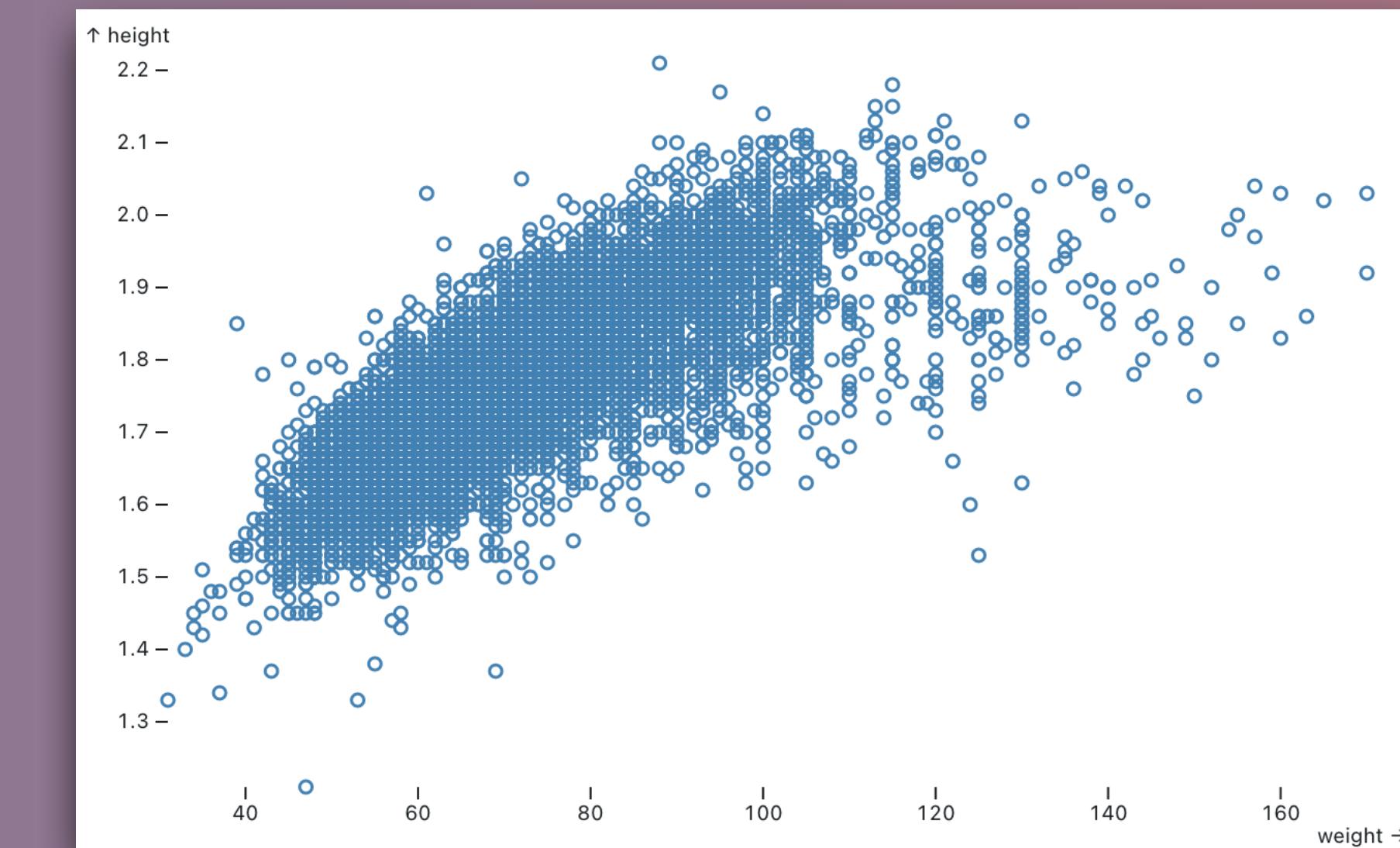
```
`Plot.barY(data,  
  Plot.binX(  
    {  
      y: 'count',  
    },  
    {  
      x: '${PROGRAM_HOLE}',  
      fill: '#7b16ff',  
      fillOpacity: 1,  
      stroke: 'none',  
      strokeOpacity: 1,  
      strokeWidth: 1  
    }  
  )`
```

# How do we decide where to leave holes?

stroke: “??”



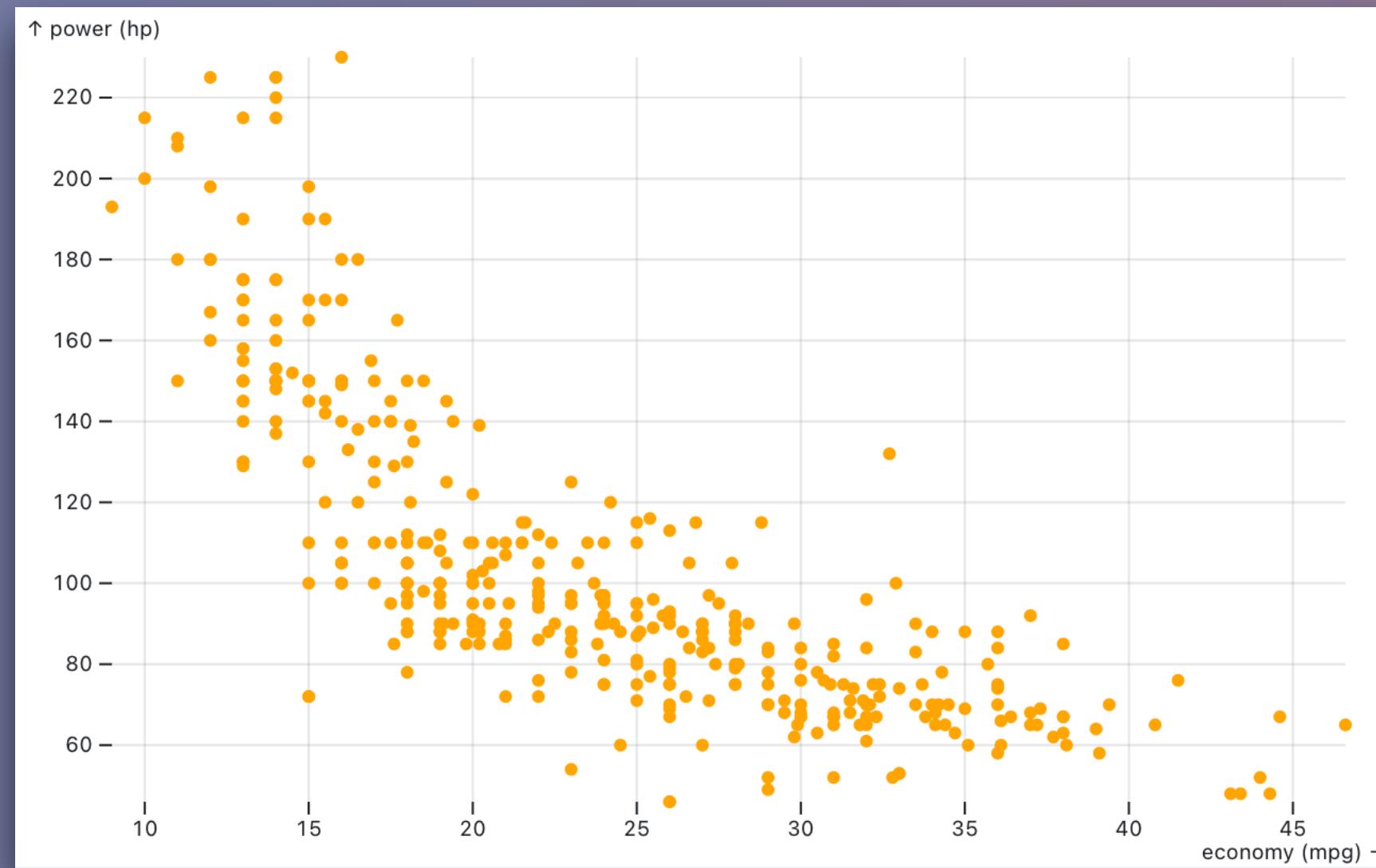
stroke: “steelblue”



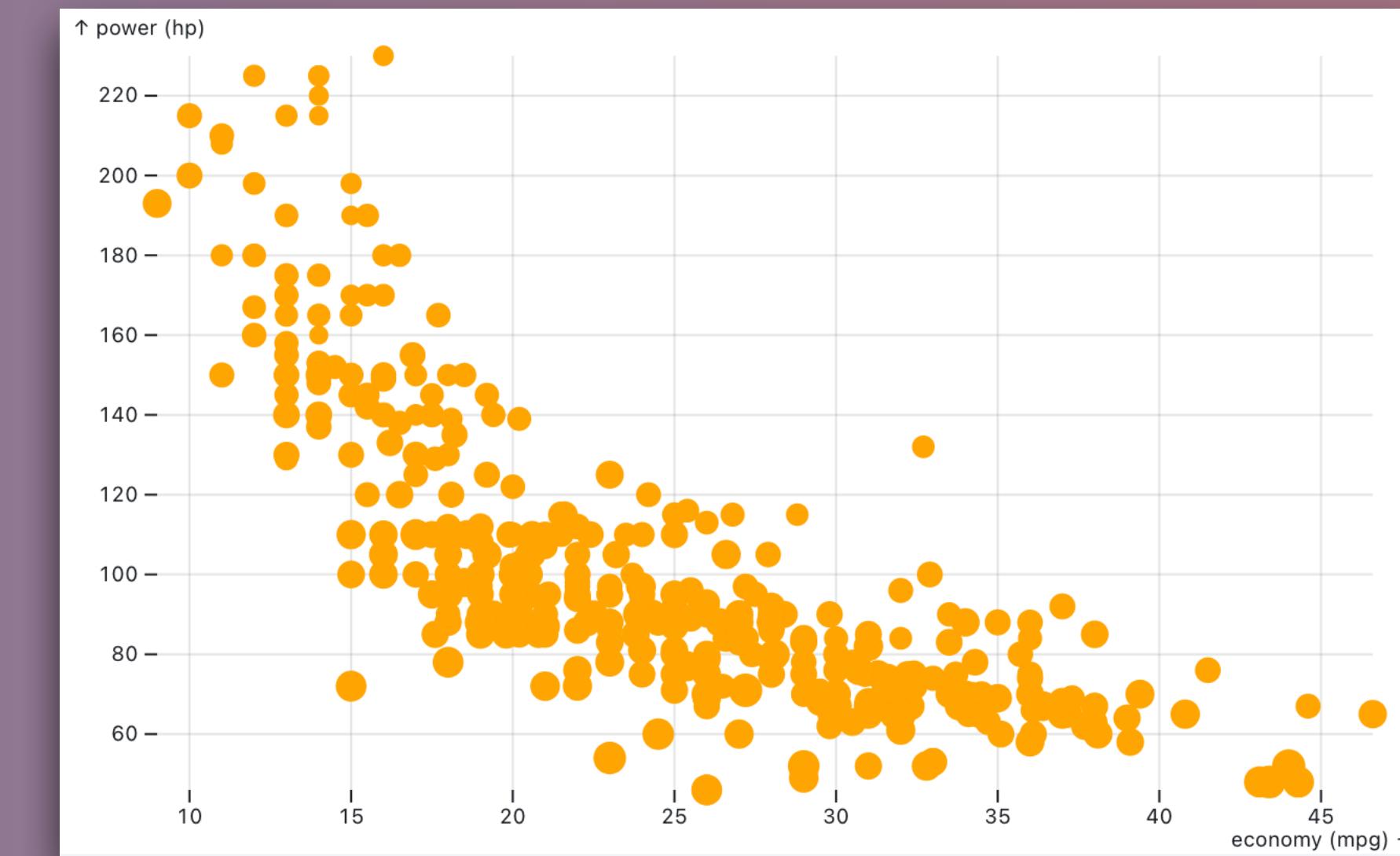
If there is variance of any kind in a visual variable, ***assume it is because that variable is mapped to a column in the data.***

# How do we decide where to leave holes?

r: 3



range: [0, 8], r: "??"



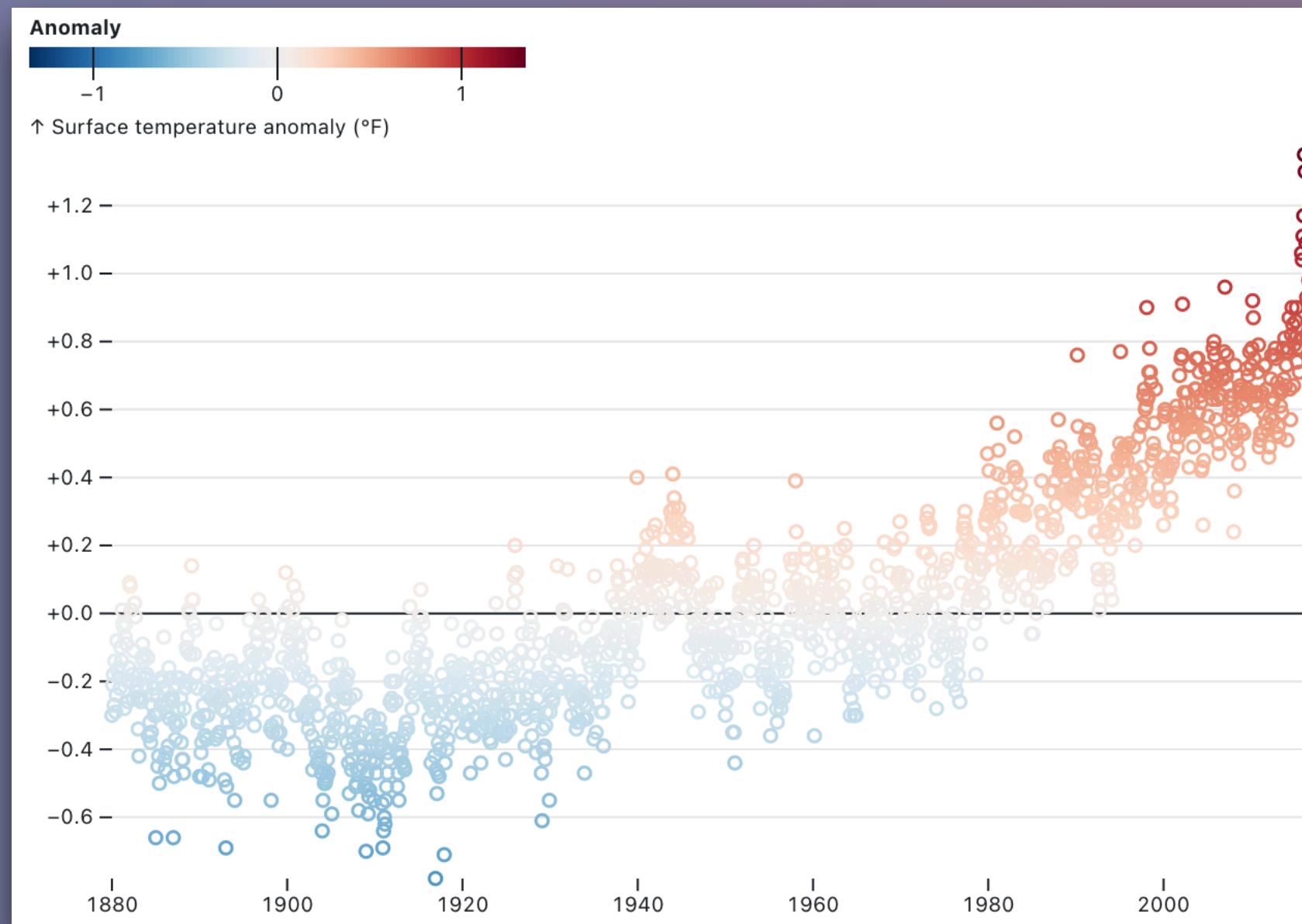
If there is variance of any kind in a visual variable, **assume it is because that variable is mapped to a column in the data.**

# How do we decide where to leave holes?

The harder case (in my opinion) is actually about  
**making semantic judgements**  
from  
**limited information!**

# How do we decide where to leave holes?

The harder case (in my opinion) is actually about **making semantic judgements** from **limited information!**



```
stroke: [
  'rgb(249, 200, 175)',
  'rgb(250, 213, 193)',
  ...,
  'rgb(249, 232, 222)'
]
```

```
color: {
  type: "diverging",
  scheme: "BuRd",
  legend: true
},
stroke: "???"
```

How do we **infer** that this is a **diverging** color scheme?

# Roadmap

1.

The **promise** of  
(and **problem** with)  
examples in data  
visualization

2.

What is **reviz**? How  
does it address the  
**problem**?

3.

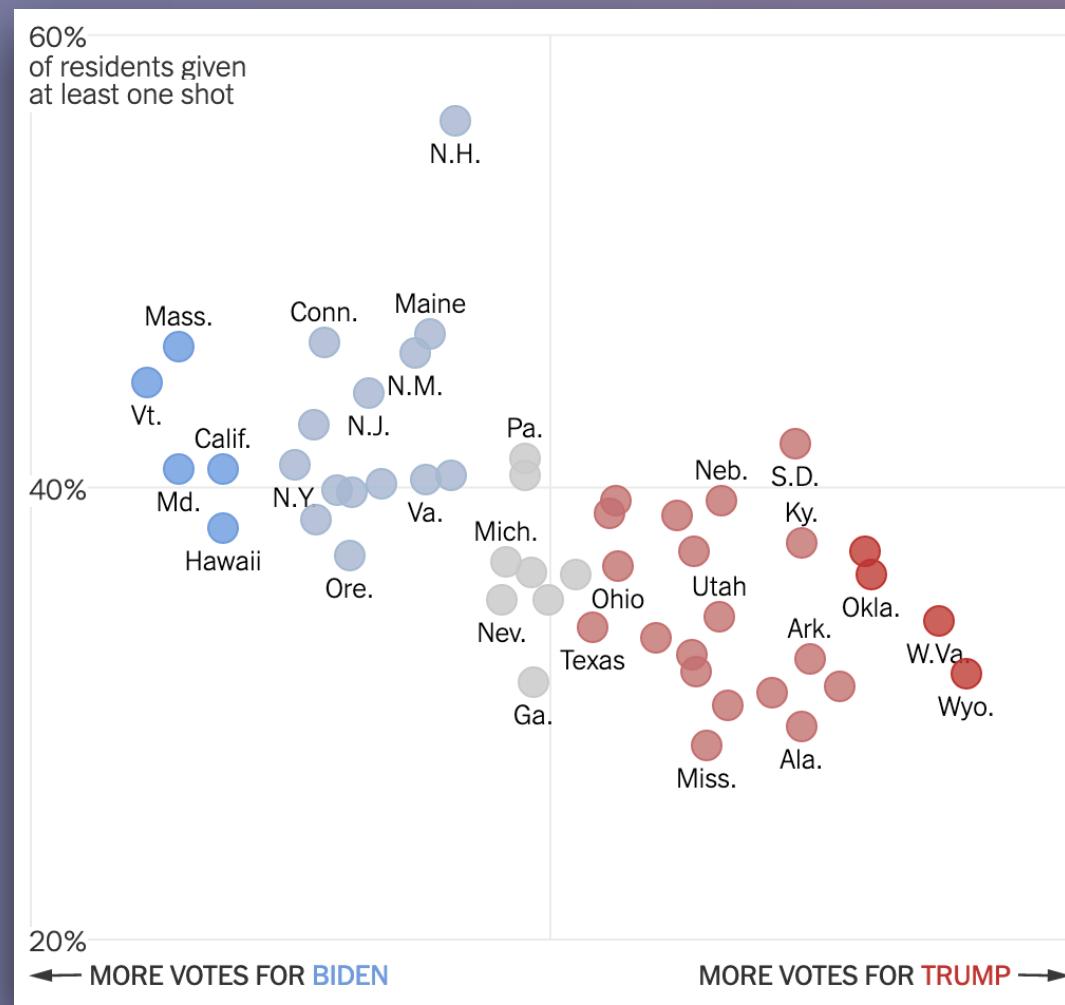
**Questions** and  
**curiosities** for  
future work

# Roadmap

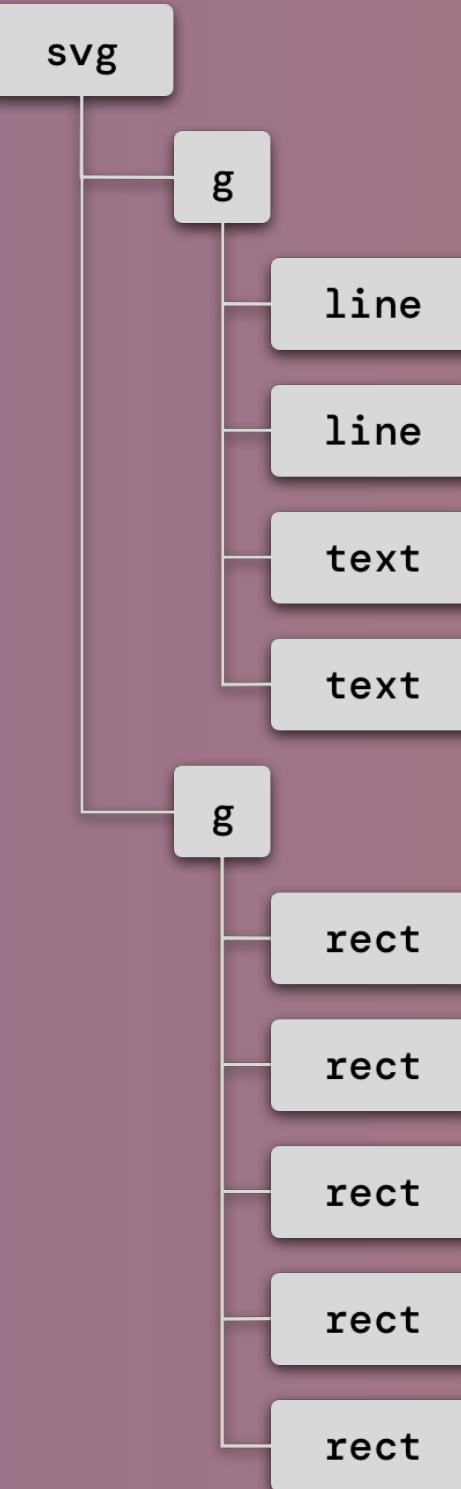
3.

**Questions** and  
**curiosities** for  
future work

# What is the right interaction model?



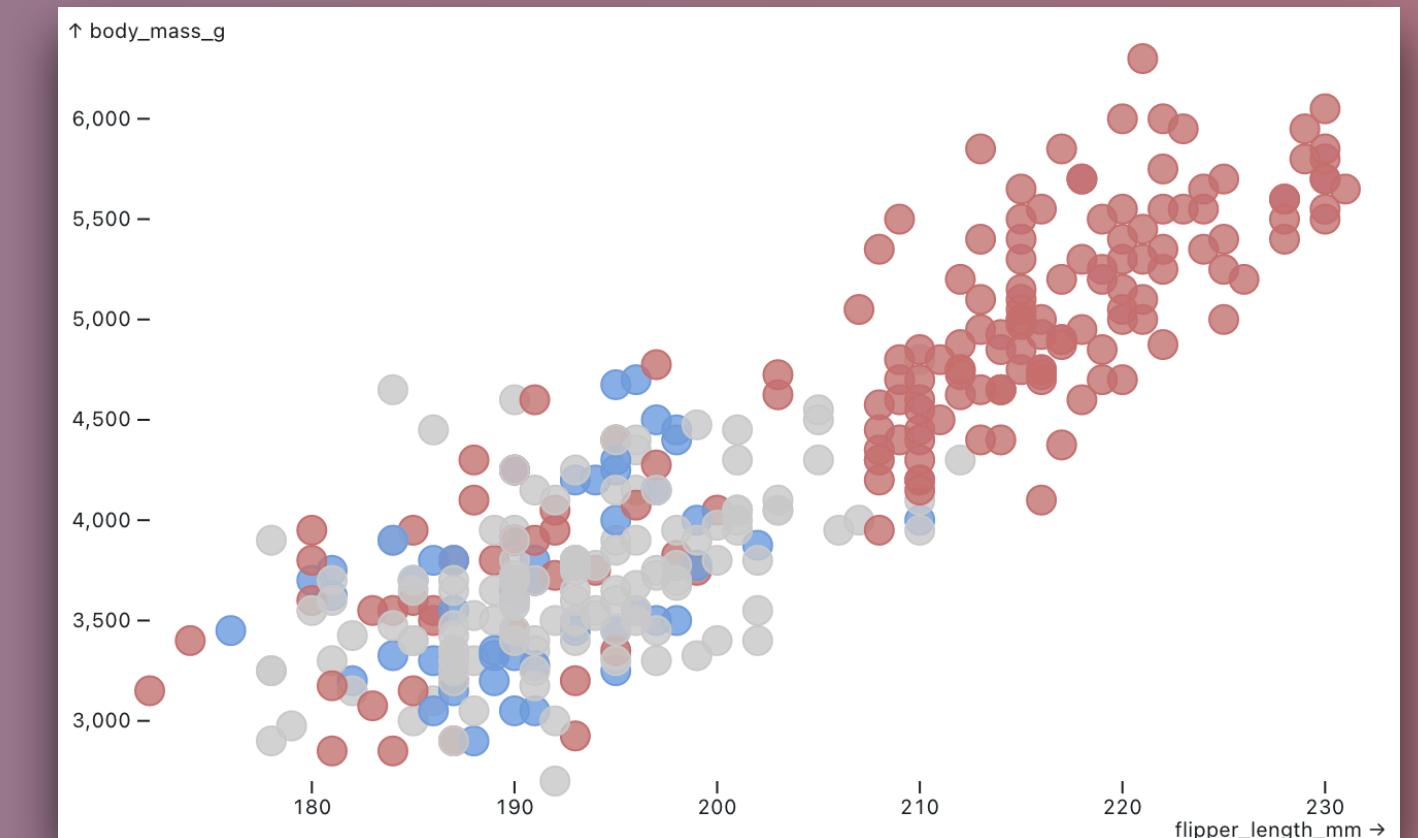
How do we make **extracting  
the svg subtree** as simple as  
possible?



# What is the right interaction model?

```
1 const plot = Plot.plot({  
2   color: {  
3     type: "categorical",  
4     range: ["#C67371", "#ccc", "#709DDE", "#A7B9D3", "#C23734"],  
5   },  
6   marks: [  
7     Plot.dot(data, {  
8       x: "??",  
9       y: "??",  
10      r: 7,  
11      fill: "??",  
12      fillOpacity: 0.8,  
13      stroke: "??",  
14      strokeOpacity: 1,  
15      strokeWidth: 1,  
16    }),  
17  ],  
18});  
19
```

Program  
.....  
How do we make **filling in holes** as simple as possible?



# How can we scale support for more visualization types?

For each **new visualization type**, we need to define **new predicates**.

```
{ type: "BarChart" }
```

```
hasMarkType('rect')
```

```
hasConsistentGeomAttr('width')
```

```
hasXScaleType('discrete')
```

# How can we scale support for more visualization types?

For each **new visualization type**, we need to define **new predicates**.

```
{ type: "BarChart" }
```

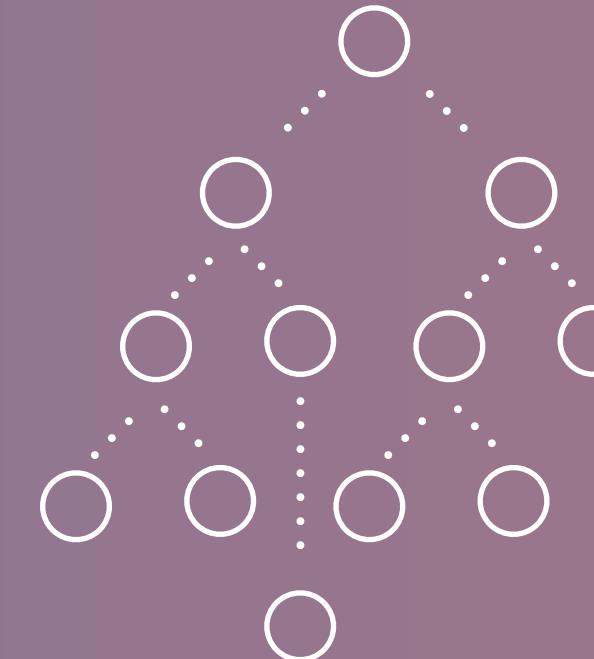
```
hasMarkType('rect')
```

```
hasConsistentGeomAttr('width')
```

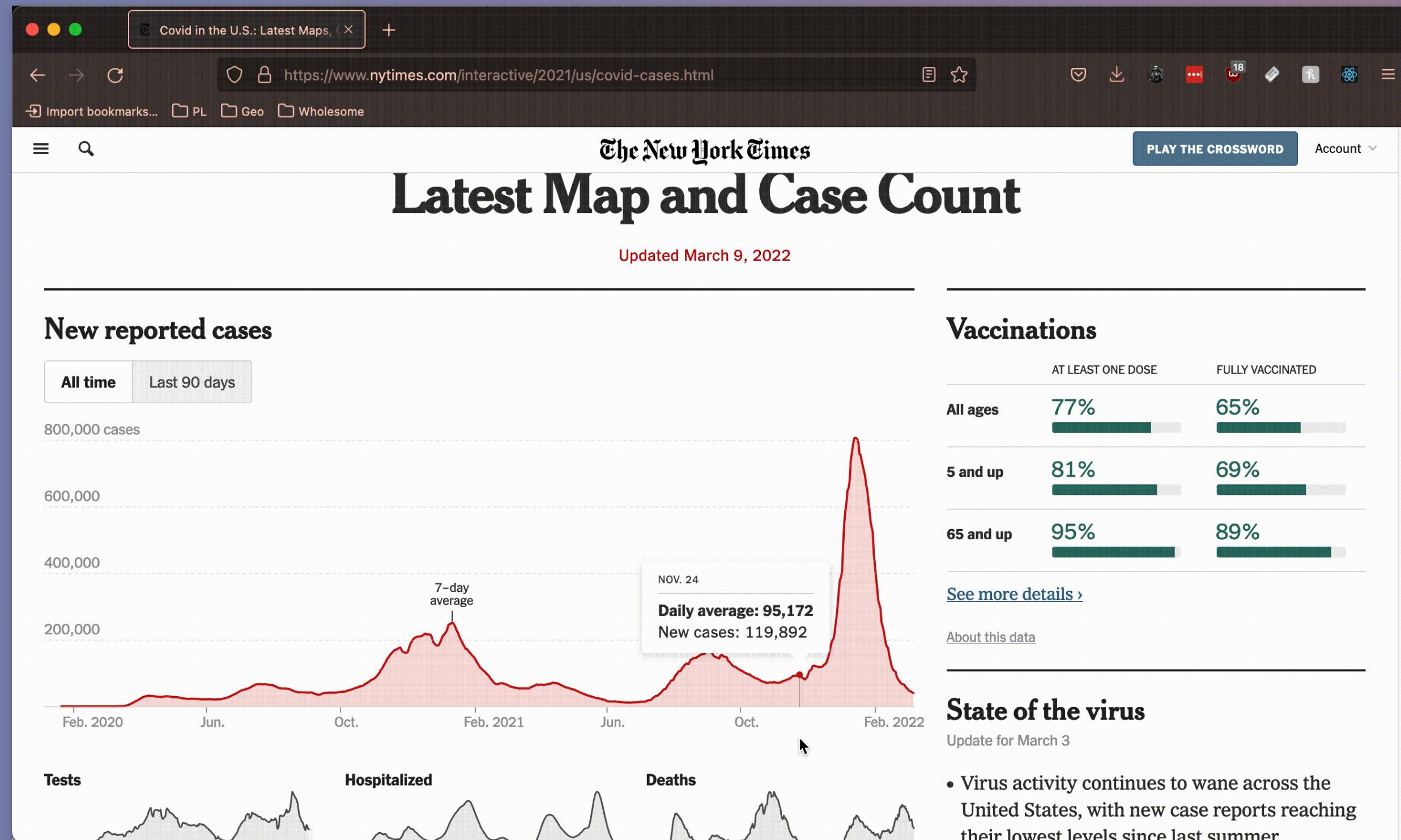
```
hasXScaleType('discrete')
```

Could we instead **learn these predicates** by training a model on **svg** subtrees?

```
{ type: "BarChart" }
```



# Can we reverse engineer interactivity?



Can we **detect interactive behavior** like tooltip rendering on hover?

If so, how might we **encode** this in our IR?

# Can we reverse engineer animation?

The screenshot shows a web browser window with the title "Learn D3: Animation / D3 / Observe". The URL is <https://observablehq.com/@d3/learn-d3-animation?collection=@d3/learn-d3>. The page content includes a text block: "Unlike graphics drawn on paper, computer graphics needn't be static; like Frankenstein's monster, they can come alive through animation! 🎉". Below this is a line chart visualization. The chart has a y-axis ranging from 0 to 600 and an x-axis with dates from July 2008 to April 2012. A blue line starts at approximately (July 2008, 100) and ends at (April 2012, 600), with a wavy pattern in between. To the right of the chart is a sidebar with icons for copy, link, search, and help. Below the chart is a code block:

```
{  
  replay; // references the button above, causing this cell to run on click  
  return htl.html`    ${d3.select(htl.svg`<path d="${line(data)}" fill="none" stroke="steelblue" stroke-width="1.5" stroke-miterlimit="1" stroke-dasharray="0,1"></path>`).call(reveal).node()}  
    ${d3.select(htl.svg`<g>`).call(xAxis).node()}  
    ${d3.select(htl.svg`<g>`).call(yAxis).node()}  
  </svg>`;  
}
```

At the bottom, there is a note: "The line chart above reveals progressively. This is somewhat gratuitous—motion should be used sparingly as it commands attention—but it at least reinforces that *x* represents time and introduces a touch of suspense to an otherwise ho-hum chart."

Can we **detect animation** like draw-on-render effects?

If so, how might we **encode** this in our IR?

# Thank You!



## Examples

A screenshot of the reviz.vercel.app website. It displays three examples of data visualizations. The first example is a bubble chart titled "Bubble" showing relationships between vehicle horsepower, fuel economy, and weight. The second example is a histogram titled "Histogram" showing the distribution of stopping times for the Collatz conjecture. The third example is a choropleth map titled "Map" showing the percentage of residents given at least one shot by state.

[reviz.vercel.app](https://reviz.vercel.app)

## Notebook

A screenshot of an Observable notebook titled "Hello reviz!". The notebook contains a histogram visualization titled "Histogram" showing the frequency distribution of stopping times for the first 1000 integers. The visualization has a blue color scheme. The notebook also includes a brief introduction to reviz and its capabilities.

[observablehq.com/@parkerziegler/hello-reviz](https://observablehq.com/@parkerziegler/hello-reviz)

## Source

A screenshot of the GitHub repository for reviz. The page shows the README.md file, which provides an overview of the library. It highlights reviz as a lightweight engine for reverse engineering data visualizations from the DOM. The README includes the reviz logo, installation instructions (yarn add @plait-lab/reviz), and links to the API documentation and a GitHub issue page.

[github.com/parkerziegler/reviz](https://github.com/parkerziegler/reviz)