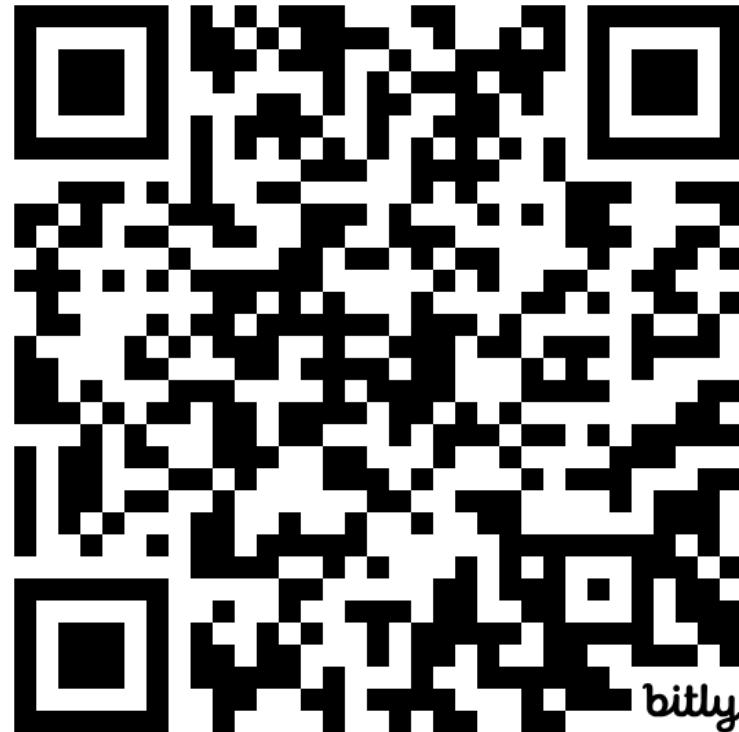


Introduction to Dynamic SVG in Core Visuals

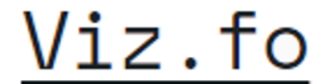
Power BI Next Step | 13 September 2024

Download this workbook

<https://bit.ly/next-step-svg>



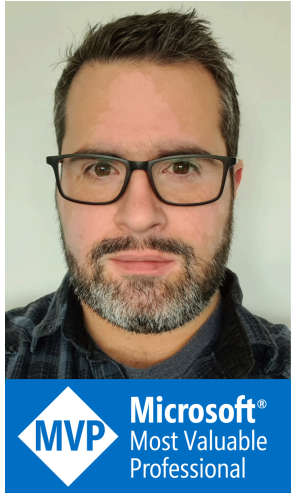
Thanks to our sponsors



Raffle prizes from our sponsors



**Power BI
Next Step**



Daniel Marsh-Patrick



@the_d_mp



daniel-m-p



@dm-p.nz



coacervo.co



daniel@coacervo.co



Power BI
Community super user

Free and open-source AppSource visuals



Deneb: Declarative
Visualization in Power BI



HTML Content



HTML Content (lite)



Violin Plot



Small Multiple Line Chart






Agenda

- Data-bound images in Power BI: concepts
- Intro to SVG
- What the above two concepts mean for fun with core visuals
- Wrap-up
- Q&A

Data Binding: Image URLs

- You can use images in core visuals - Table, Matrix, New Card, New Slicer*
- We can do this with an image URL (remote address)
- Needs to be a 'public'/anonymous address [[source](#)]

* Currently has some issues with dynamic images; hopefully it's going to be fixed soon, but for now, assume it's off the table

Table visual with image columns - using web URL		
Country	Country Flag Web URL	Country Flag Web URL (Image)
Canada	https://www.html-content.com/images/flags/CA.png	
France	https://www.html-content.com/images/flags/FR.png	
Germany	https://www.html-content.com/images/flags/DE.png	
Mexico	https://www.html-content.com/images/flags/MX.png	
United States of America	https://www.html-content.com/images/flags/US.png	

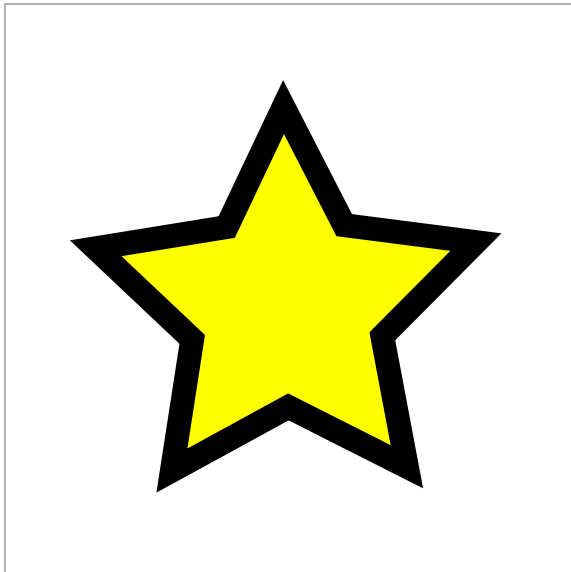
Data Binding: Data URIs

- You can also 'embed' images using a data URI
- Doesn't use a web request
- Image is stored in the workbook rather than online
- Will inflate your model size (and columns have a 32KB limit)

Table visual with image columns - data URIs		
Country	Country Flag Base64 URL	Country Flag Base64 URL (Image)
Canada	data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAMwAAAB4CMAAACJp+...	
France	data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAMwAAAB4AgMAAADDF/...	
Germany	data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAMwAAAB4AgMAAADDF/...	
Mexico	data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAMwAAAB4CMAAACJp+...	
United States of America	data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAMwAAAB4CMAAACJp+...	

SVG Images

- We just saw binary encoded images (.png)
- SVG = **S**calable **V**ector **G**raphics
- Defined using XML and known element types (human-readable)
- Potential for data-driven images w/DAX



```
<svg xmlns='http://www.w3.org/2000/svg'
      width='100%' height='100%' viewBox='0 0 6.3499999 6.35'>
  <path fill='#ffff00'
        stroke='#000000'
        stroke-width='0.3'
        d='M 3.4251989,3.1568064 1.9374211,2.4051437 0.47477525,3.2046102
          0.72990042,1.5573729 -0.48241983,0.41336268 1.1630339,0.14697683
          1.8764247,-1.3595268 2.6382459,0.12307493 4.2914659,0.33601464
          3.1168435,1.5186988 Z'
        transform='matrix(1.0114715,0,0,1.0114715,1.3219352,2.2649803)'
  />
</svg>
```


A Basic DAX Pattern

- A very basic SVG container element can be drawn as an inline data URL
- Commented part (denoted by `<!-- -->`) is where we would add our main content
- Not all SVG elements use % as their measurement unit (this is fine for now)

```
VAR __svgBase = "data:image/svg+xml;utf8,
    <svg
        xmlns='http://www.w3.org/2000/svg'
        width='100%'
        height='100%'
    >
        <!-- our content will go here -->
    </svg>"
RETURN
__svgBase
```

Our Star: With Simple Bindings to Measures

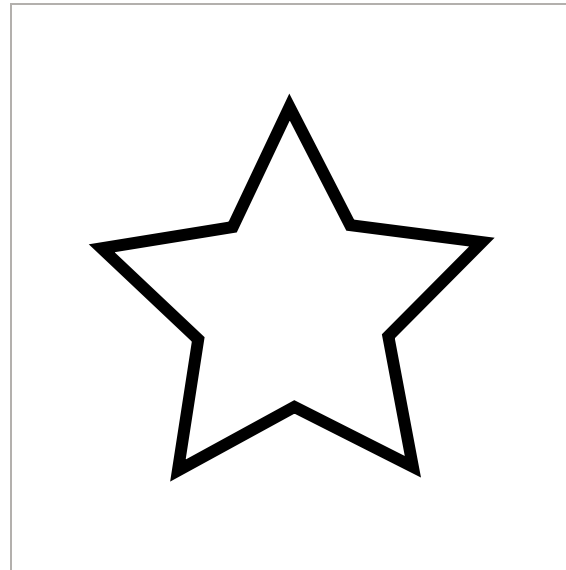
```
VAR __fill = [Selected Colour]
VAR __strokeWidth = [Percentage Value]
VAR __svgBase =
  "data:image/svg+xml;utf8,
  <svg
    xmlns='http://www.w3.org/2000/svg'
    width='100%'
    height='100%'
    viewBox='0 0 6.3499999 6.35'
  >
    <path
      fill='" & __fill & "'
      stroke='#000000'
      stroke-width='" & __strokeWidth & "'
      d='M 3.4251989,3.1568064
        1.9374211,2.4051437
        0.47477525,3.2046102
        0.72990042,1.5573729
        -0.48241983,0.41336268
        1.1630339,0.14697683
        1.8764247,-1.3595268
        2.6382459,0.12307493
        4.2914659,0.33601464
        3.1168435,1.5186988 Z'
      transform='matrix(1.0114715,
        0,0,1.0114715,1.3219352,
        2.2649803)'
    />
  </svg>"
RETURN
__svgBase
```

fill

- ☒ white
- ☐ black
- ☐ blue
- ☐ red
- ☐ #118DFF
- ☐ #118DFF80
- ☐ rgb(255, 99, 71)
- ☐ rgba(255, 99, 71, 0.5)

stroke-width

0.15



Thinking Beyond What We Just Saw

or: "There's potential, but also here be dragons"

- Most Power BI visuals are SVG elements
- You can apply this thinking to what we just saw and enrich core visuals
- This can be a huge boon for the efficacy of your reports

However...

- SVG is a *graphics* specification, **not** a *graphing* specification
- Visuals often use JavaScript libraries that handle bridging the two concepts (e.g. D3)
- You don't have this in DAX, so you need to be thorough
- You now have (or your team or successor has) more to support and maintain

An 'Enhanced' Data Bar






- We have options in Power BI to add data bars to tables
- These have limited formatting capabilities and have some gotchas
- We could use SVG to design and implement a cleaner, more actionable version
- Break ideas down into components (bar, label) and look up SVG equivalents (rect, text)
- Gotchas: you're working in each row, but have to consider all data (scaling of bars)






Native Data Bars in a Table	
Country	\$ Sales
United States of America	\$1,026,166,934
Canada	\$976,155,769
France	\$897,584,018
Mexico	\$879,000,063
Germany	\$758,125,278

Using SVG to Make a New Bar Type	
Country	Sales (mUSD)
United States of America	\$1,026
Canada	\$976
France	\$898
Mexico	\$879
Germany	\$758

An 'Enhanced' Sparkline

- Power BI has native sparklines, but again, very minimal
- If we're inspired by other examples (e.g., Stephen Few), we can replicate using SVG
- Not all SVG elements use % as their measurement unit (this is fine for now)
- Highest / lowest and **most recent** month

Native Sparkline in a Table	
Country	\$ Sales by Month Start
Canada	
France	
Germany	
Mexico	
United States of America	

Using SVG to Make a Stephen Few-Inspired Sparkline	
Country	\$ Sales (mUSD)
Canada	 <div><div>\$147.8</div><div>\$28.7</div><div>\$147.8</div></div>
France	 <div><div>\$105.9</div><div>\$13.5</div><div>\$92.0</div></div>
Germany	 <div><div>\$86.5</div><div>\$16.1</div><div>\$59.4</div></div>
Mexico	 <div><div>\$105.0</div><div>\$17.6</div><div>\$71.1</div></div>
United States of America	 <div><div>\$103.7</div><div>\$23.8</div><div>\$74.5</div></div>

Axis Scaling

☒ Independent

☐ Synchronised

Session Evaluation



Homework / Further Learning

- We covered a lot of ground in this session, very fast
- This is an approach with many ways to approach and implement
- It's hard to know everything in a short/intro session
- The following pages break down the learning and get towards the advanced examples in a more step-wise manner
- Recommended if you want to learn more on your own time or experiment

Drawing a simple rectangle

Here, we're starting out simple, by drawing a [<rect>](#) (rectangle) element on the canvas. This is an easy shape to understand, and is typically the building block of a bar chart in SVG.

In the measure opposite, we size the SVG container to 100% of the width and height of the space that Power BI allocates for images (which can be changed by using the **Height** and/or **Width** properties in the Image size property menu. If you adjust these properties, you will see the rectangle stretch or shrink to fit accordingly.

Note that these properties currently have a lower limit of **24px** and an upper limit of **150px**.

<SVG> 01.01 Rect



Layering a circle on top of the rectangle

Now we're taking the previous example and adding a `<circle>` element. This is to illustrate two things early on:

1. The draw order of SVG elements

Elements are placed on the canvas in the order they are declared. In this case, because the `<circle>` is the last thing in our SVG definition, it is drawn on top of the `<rect>`. There is no concept of "z index" in SVG, so if something isn't where you think it is, be sure to check the order in which everything is being declared.

2. The coordinate system

SVG works using an x/y coordinate system, and this isn't necessarily the same as how you will read a chart. If you're considering using SVG to design a chart, consider that charts are (mostly) a drawing in a 2-dimensional space and you will need to convert logical values (measures, categories etc.) to physical dimensions (the SVG canvas).

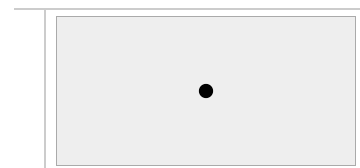
The x-axis runs **from left to right**, starting at **0**.

The y-axis runs **from top to bottom**, also starting at **0**.

SVG has the concept of viewports, and you can introduce `viewBox` attributes to change how the physical space can convert to a logical coordinate system. This is outside the scope of this talk and we will be avoiding them as part of the introduction. For further reading, [it is suggested that you start here to understand the concept of positions](#) in further detail (and there are many more guides online that can walk through this in a more prescribed manner than we can accommodate here).

If you edit the `cx` (centre-x) and `cy` (centre-y) attributes of the `<circle>`, you can see how they affect the position of the element relative to the coordinate system.

<SVG> 01.02 Rect + Circle



Making a data bar from a rectangle

Here, we layer a second rectangle and give it some properties to make it appear like a data bar. The rectangle starts from the left-most portion of the viewport and stretches to a value that is driven by a variable in the accompanying DAX statement.

Here, we're introducing a very simple concept of scaling our bar to the width of our viewport, and also for our data value. We're sticking with percentages to keep things simple.

We use the following variables to declare the maximum value for our "x-axis":

```
VAR __xValueMax = 100
```

As our coordinate system starts at **0**, we're currently assuming this as a minimum value, and our maximum value should be **100**. Next, we declare a variable representing the value of our bar's "measure":

```
VAR __xValue = 30
```

We then introduce a variable to handle the positioning of where the bar should stretch to (its width) based on the percentage of this "measure" to the maximum value:

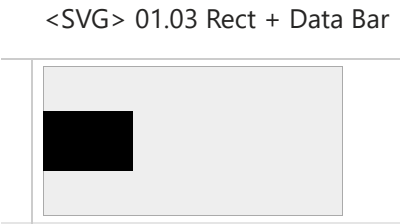
```
VAR __xValuePos = DIVIDE( __xValue, __xValueMax) * 100
```

Because our maximum value might not be as nice a round number as 100 (which is also convenient for percentages), this pattern allows us to modify the above variables and we should always be able to calculate the percentage of the width that our measure would take up.

We then feed this result into the <rect> element that we want to draw:

```
<rect
  width='' & __xValuePos & '%'
  y='30%'
  height='40%'
/>
```

This is nice, because our bars will always start from 0 (keeping them honest!), but what happens if our measure value is negative? We will need to think how this fits into our available width, and we'll extend this measure design to factor this in on the next page.



Extending our data bar to be "range safe"

We'll be building on the previous example to ensure that our axis range can span from negative to positive values and ensure that our bar is positioned and sized as we would expect it to.

Our first change is to introduce a minimum value for our axis, e.g.:

```
VAR __xValueMin = -20
```

Now we have a minimum and maximum value, we calculate the "span" of these values, i.e. how much of the percentage of our width a unit of our available range will occupy (effectively scaling it):

```
VAR __xSpan = __xValueMax - __xValueMin
VAR __xValueScale = DIVIDE( 100, __xSpan)
```

We then need to take steps to "convert" our measure values to the physical (scaled) position.

Firstly, where the zero value would occur along our axis (we'll use this to validate our approach):

```
VAR __xPosZero = __xValueScale * ( 0 - __xValueMin )
```

And then, where our measure value will actually be positioned (which follows the same calculation as above, just for the measure value):

```
VAR __xPosValue = __xValueScale * ( __xValue - __xValueMin )
```

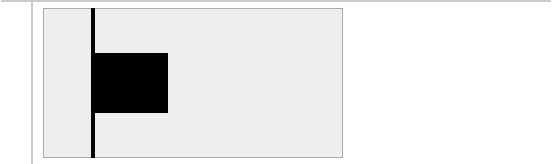
With <rect> elements, we can only set the initial x position and the width, so we're going to create two variables, __x1 and __x2, which work out the "start" and "end" of our bar based on whichever are the smallest and largest (out of zero and our measure), so that we don't end up creating negative width:

```
VAR __x1 = MIN( __xPosZero, __xPosValue )
VAR __x2 = MAX( __xPosZero, __xPosValue )
VAR __xWidth = __x2 - __x1
```

The __x1 and __xWidth variables are substituted into our <rect>:

```
<rect
  y='30%'
  height='40%'
  x=' ' & __x1 & '%'
  width=' ' & __xWidth & '%'
.
```

<SVG> 01.04 Rect + Data Bar (range-safe)



Binding our bar to a measure

Now, we're taking our previous example and binding it to a measure in our data model. Because this will potentially be used in conjunction with a category, we need to think about our axis minimum and maximum values. This is because, if we work this out based on current row context, our bar will only know about its current value and all bars will essentially be the same length (making them useless for comparison purposes).

We take the previous pattern and update the variables that set axis minimum and maximum values to include ALLSELECTED for the grain of our data (in this case, the Country column from the Demographic table), e.g.:

```
VAR __xValueMin =
    MIN(
        0,
        MINX( ALLSELECTED( Demographic[Country] ), [$ Sales] )
    )
VAR __xValueMax =
    MAX(
        0,
        MAXX( ALLSELECTED( Demographic[Country] ), [$ Sales] )
    )
```

We sub in our measure value to the __xValue parameter, e.g.:

```
VAR __xValue = [$ Sales]
```

...and the rest is as before! The table on the right shows how our SVG measure looks when we bind it to the Country column, and we can see that our bars are comparable to each other.

Bonus: sorting

because SVG columns are sorted as text, this means that the raw SVG is used as the sort value. You can therefore create a text representation of the measure value with enough padded zeroes, e.g.:

```
VAR __SortValue = FORMAT( __xValue, REPT(0, 15) )
```

...and add this as an arbitrary attribute to the svg element. As long as it's not a part of the SVG specification, it will be ignored by the parser, but will still give your measure the precedence you need. Give it a try by sorting the table columns.

Country	\$ Sales	<SVG> 02.01 Rect + Data Bar (range-safe. bound)
Canada	\$976,155,769	<div></div>
France	\$897,584,018	<div></div>
Germany	\$758,125,278	<div></div>
Mexico	\$879,000,063	<div></div>
United States of America	\$1,026,166,934	<div></div>

Converting from percentage domain to pixel domain

So far, we've been using % as our unit to scale and position our SVG elements. However, not all elements support this unit, and you can guarantee that all will support pixels within our canvas.

While bars will work fine and you can use the previous patterns as much as you like for them, it's a useful exercise to understand how to scale in pixels too, so that you can start thinking in SVG for other use cases. As stated earlier, you can use the viewport and viewBox concepts to work around this by setting your own coordinate system within the SVG element, but we're sticking with pure values to get you used to the idea of scaling your measures to physical coordinates in the scope of this session.

The main difference in this version is that we declare a new variable called __width, and set this to match our value from the properties pane, e.g.:

```
VAR __width = 150
VAR __xValueScale = DIVIDE( __width, __xSpan)
```

Our DIVIDE calculation now uses this as the numerator rather than the fixed value of 100 (for percent) that we were doing previously.

The only other change is to remove all references to % where our SVG measures are bound downstream, and the result is the same, but more flexible going forward.

(note that wherever values have been hard-coded into the SVG output, these have been left as percentages, but this can be a good exercise for you to attempt conversion on if you wish to try and work it out)

Country	\$ Sales	<SVG> 02.02 Data Bar (pixels)
Canada	\$976,155,769	<div></div>
France	\$897,584,018	<div></div>
Germany	\$758,125,278	<div></div>
Mexico	\$879,000,063	<div></div>
United States of America	\$1,026,166,934	<div></div>

Swapping from x to y axis for our measure

Here, we're going to attempt to swap our measure from a bar to a column, which means we'll be using the y-axis (height) to bind our measure rather than the width. This is not useful for a single bar, but is useful when you want to think about using many bars (e.g. for a series along the x-axis).

This follows the same pattern as the previous example, except that we need to:






- Use a taller height than width for our image size in the table.
- Use height instead of width when calculating the measure bindings.

However, with height, we need to remember that our coordinates start from the top-down, rather than bottom-up, so we need to ensure that we update our scaling calculations accordingly. The most convenient way of doing this is by subtracting the starting position of the y-axis (whish is the `__height` variable in this example) from the scaled value, e.g.:

```
// Calculate the position of y=0
VAR __yPosZero = __height - __yValueScale
                * ( 0 - __yValueMin )

// Calculate the position of the measure value
VAR __yPosValue = __height - __yValueScale
                * ( __yValue - __yValueMin )
```

This is a common tripping point for why things may not be positioned where you expect them to be when dealing with "y-axis" values.

Country	\$ Sales	<SVG> 02.03 Data Bar (pixels, y version)
Canada	\$976,155,769	
France	\$897,584,018	
Germany	\$758,125,278	
Mexico	\$879,000,063	
United States of America	\$1,026,166,934	

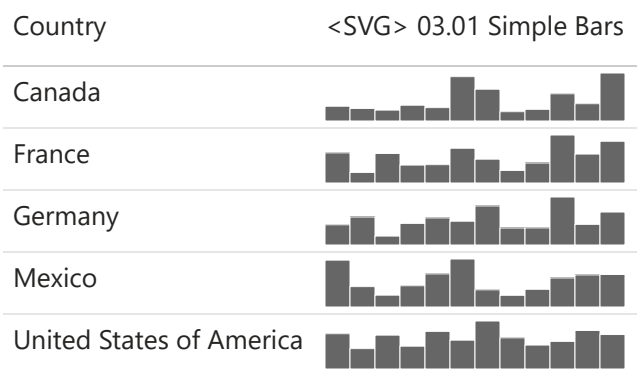
Simple spark bar (using rect + stroke)

Now, we're going to take everything we've learned so far and turn this into a simple spark bar, which breaks out our Sales by Country into Months for a selected year (by using the slicer). These bars are drawn using rect elements like before, but we give each element a small amount of white stroke (border) to give them a bit of room and appear distinct from each other.

We now have to consider that in addition to the current row context, we need to be able to look across both dimensions of Country and Month to determine our bar values. It's also a common pattern to consider whether bars should show relative to their own series, or across all data in the table (in order to perform correct comparisons as needed). Again, this toggle is made via a slicer, but this time it's a disconnected table and our logic will act on this to scale the y-axis and bars accordingly. Synchronised scaling is not currently possible with the built in data bars.

This measure is quite tricky to break out in this short amount of space, and the DAX is commented heavily enough for you to follow along, but the key workflow is as follows:

- Get all rows and categories from our 'table'. Note that as months are not part of the table's data set, we have to do this within a table variable in our DAX.
- Get minimum and maximum category values for our x-axis. This is the first and last month, and as time is read in a linear manner, we can sort them in our chart from left to right.
- Get all unique values of our category (month) from the first step. This is because sometimes a row may not contain data for a measure for a particular month, and we still need to think about creating a place for it on our x-axis for all categories so that it's easier and more consistent to read.
- Count how many unique values we have on our x-axis.
- Divide the width by our count of these values, so we know how wide each bar should be.
- Create a simple 'dataset' for each row of the table, based on our unique categories, their sort order and measure value.
- Calculate our bar heights, based on whether we are looking across all data, or just our current row. This is the same technique as we used previously, with an additional check for how much of the data we need to check through. This extends the 'dataset' variable with additional columns to help us out further downstream.
- Create our rectangle 'marks' from our 'dataset' table using CONCATENATEX to assign values to the SVG elements and flatten the table down to a text output.
- Embed these marks into our base SVG template.



Simple spark bar (adding bandwidth for spacing)

Note that using the stroke (border) in our previous example to create white space 'between' bars can sometimes mean that smaller values are more hidden than they should be, as the stroke is included in the calculation of the rectangle size.

We can therefore improve this this idea by introducing the concept of 'bandwidth' for our bars, which allows us to be more prescriptive about their spacing. The 'band' is the space we allocate for each bar, and the 'bandwidth' is how much of this space we wish to occupy for our bars.

The example builds on the previous example, and is again heavily commented if you wish to dissect the measure on your own, but we'll speak about the specifics for bandwidth that we're introduced.

First, we introduce a parameter to govern the percentage of our band that we wish to use, e.g.:

```
VAR __bandwidth = 0.75
```

This is a value ranging from 0 - 1, so $0.75 = 75\%$ of the width for each category. You can change this in the measure and see how the white space for the bars will change.

Further down, we revise our bar width, based on the calculated category width and this value, e.g.:

```
VAR __categoryBandwidth = __categoryWidth * __bandwidth
```

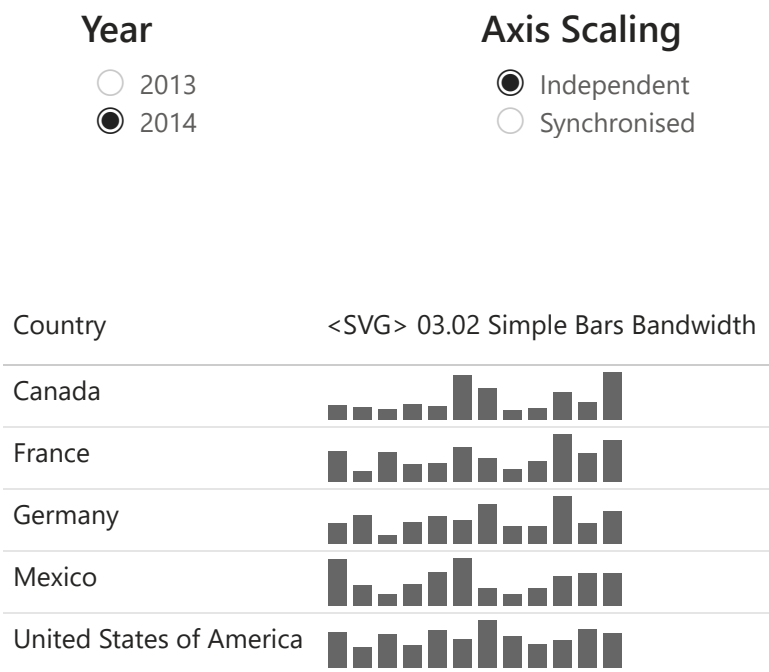
We then calculate the offset from the start position of our category, so we know where to position our bar within each band, e.g.:

```
VAR __categoryOffset = DIVIDE(
    __categoryWidth - __categoryBandwidth,
    2
)
```

We divide by 2, so that we can position this evenly, i.e. in the centre of the band. This is then added to the @xStart column in our table variable that we calculated in the previous example, e.g.:

```
"@xStart", [ @xOrder ]
    * __categoryWidth + __categoryOffset
```

This approach can eke out as much space as possible for the data, while giving you more control over the amount of white space in the final design.



Simple sparkline

We can take the principles we've learned so far to convert this to a sparkline. This follows a very similar approach to the previous example, with some small modifications that make sense for the positioning of points in a line, rather than rectangles representing bars. Lines also don't always need a zero reference like bars should ideally have.

Firstly, we don't need the bandwidth concept, but we do need to position our points in the middle of where our band would occur. This is a case of modifying the logic for our category offset as follows, which will just divide the current width allocation for a band by 2, e.g.:

```
VAR __categoryOffset = DIVIDE( __categoryWidth, 2 )
```

Rather than using `CONCATENATEX` to combine separate `rect` elements, we will be using a [polyline](#) element for each series. As such, we just need to use the approach to flatten the computed coordinates down, e.g.:

```
VAR __points = CONCATENATEX (
    __primaryPositions,
    [@xStart] & "," & [@yValue],
    " ",
    [@xOrder]
)
```

This is then embedded into a single `polyline` element, e.g.:

```
VAR __line = "<polyline
    stroke='" & __stroke & "'
    stroke-width='" & __strokeWidth & "'
    fill='" & __fill & "'
    points='" & __points & "'
/>"
```

In turn, this gets embedded into our base `SVG` element as normal.

You may notice that in some cases, the line appears 'clipped', and this is because we're using the full width and height of our container. Sometimes, the elements we're drawing can fall outside these ranges and the `SVG` element will cut them off. The next example shows how we can extend our scaling to cater for this more easily.

Year

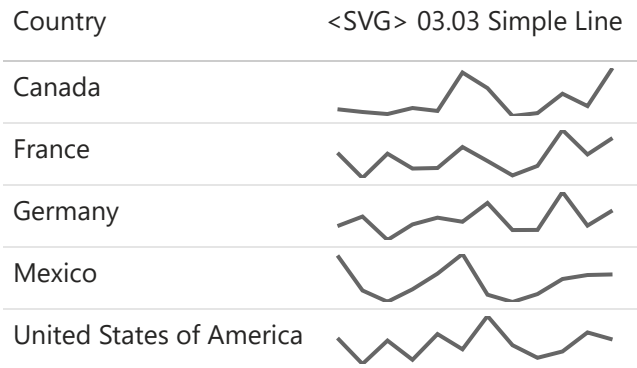
2013

2014

Axis Scaling

Independent

Synchronised



Simple spark line, with padding

To mitigate the clipping issue we may have observed in the previous example, we can introduce a variable to allow us to specify padding around the are for our chart, and this is then used to calculate the space for our axes. In this example, we're just using it on the y-axis, but this approach can be used on either axis, or even individual sides of an axis (for scenarios like needing to accommodate labels or non-data elements, without interfering with the bulk of the plot).

In this example, we've introduced a variable to manage the padding, and this is set to the same as the stroke width for our line, e.g.:

```
VAR __yPadding = 2
```

We then introduce a variable to calculate the amount of space (height) needed for our y-axis, i respect of our padding (which we are going to assume is evenly distributed at the top and bottom):

```
VAR __yAxisHeight = __height - ( __yPadding * 2 )
```

This is then used in our scale factor calculation rather than the height:

```
VAR __yValueScale = DIVIDE( __yAxisHeight, __ySpan )
```

We also modify the axis start variable to factor in the padding:

```
VAR __yAxisStart = __height - __yPadding
```

Our final adjustment is to modify the calculation of the scaled y-value in our table variable that handles positioning so that we use the axis start as a reference, rather than the height:

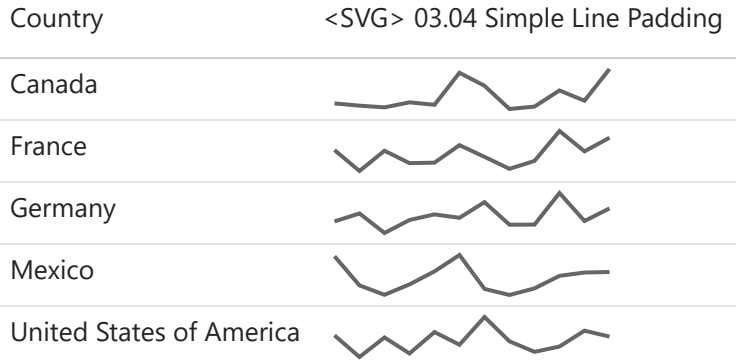
```
"@yValue", __yAxisStart - __yValueScale
    * ( [ @Measure ] - __yValueMin )
```

Year

- ☐ 2013
- ☒ 2014

Axis Scaling

- ☒ Independent
- ☐ Synchronised



Stephen Few inspired sparkline (revised for April 2023 update)

This example takes everything we've learned (plus a bit more!) to create a sparkline design that is inspired by the versions introduced in Stephen Few's book, *"Information Dashboard Design"*.

This is our line, evolved to show minimum and maximum points (in grey), and the most recent (in red). Our label shading corresponds to this also. We handle the situation where the most recent point may be the same as the highest or lowest, by ensuring that we draw those points first, and then the the most recent one, so that it is plotted above the other two. You could of course introduce filtering into your logic to reduce the number of redundant elements, but we're introducing enough new stuff here that you have enough to think about.

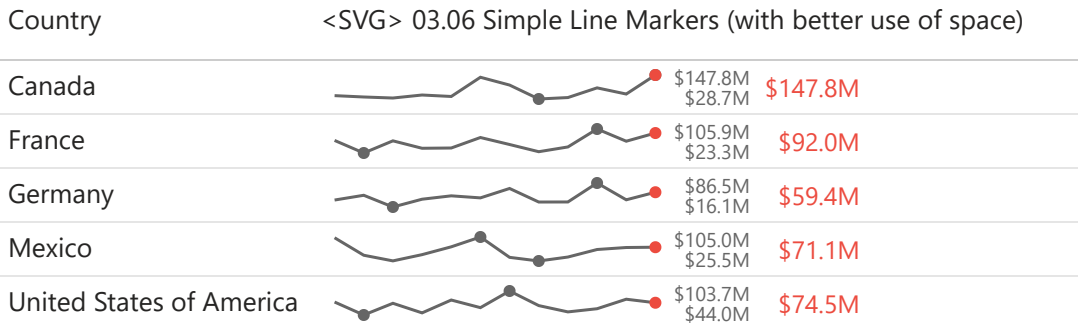
This is one that is probably best dissected by going through the measure and comments to see what's happening, but it should be evident as to how things are being derived, and laid out (notably we're using a parameter specifically to handle the padding for the right hand side vs. the rest of the chart, so that our labels can fit).

Year

- ☐ 2013
- ☒ 2014

Axis Scaling

- ☒ Independent
- ☐ Synchronised



Bonus: waffle chart concept

As we've focused on designs that extend the possibilities of existing data bar or sparkline/bar capabilities, here's an example that's a little bit different: visualising a percentage value as a [waffle chart](#).

This also introduces the concept of a simple [animation](#), where all squares are progressively drawn over a period of 0.5 seconds. Note that any changes on the page (cross-filtering, slicing etc.) cause the image to be re-drawn, so if considering animation, you will need to think about how your visual will look in these situations (it's not possible, for example, to transition from one state to the next).

Year/Month

- ▼

2013
- ▲

2014
- Jan
- Feb
- Mar
- Apr
- May
- Jun
- Jul
- Aug
- Sep
- Oct
- Nov
- Dec

Product	\$ Sales	% of Total	<SVG> Waffle Country
Paseo	\$230.93M	70.59%	<div></div>
Montana	\$50.61M	15.47%	<div></div>
Amarilla	\$32.43M	9.91%	<div></div>
VTT	\$6.01M	1.84%	<div></div>
Velo	\$4.74M	1.45%	<div></div>
Carretera	\$2.45M	0.75%	<div></div>
Total	\$327.16M	100.00%	

Homework / Further Learning

- Laura GB - [SVG in Power BI](#)
- [Bas Dohmen / How to Power BI](#)
- [Christine Payton \(@bi-ome\)](#)
- Kerry Kolosko - [SVG templates](#)
- Štěpán Rešl - [intro to SVG in Power BI](#)
- [Power of BI](#) (Andrzej Leszkiewicz)
- Me - [longer version of this session](#)