Dynamic Data Display Overview
Version 2.0

This document gives overview of Dynamic Data Display library core algorithms and architecture. Topics covered include figure composition, coordinate transforms, palettes, legend and implementations of marker graph and heat map visualizations. Extensibility points are mentioned and references to samples in Interactive SDK are provided.

Contents

Introduction

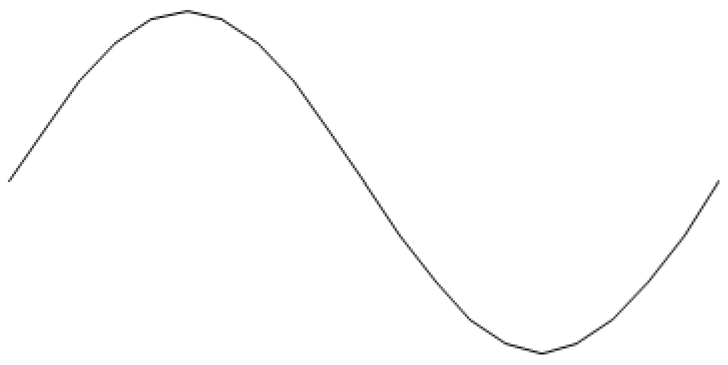
Dynamic Data Display or D^3 for short is a set of extensible controls for scientific visualization. Visualization is a graphical representation of specified input data; picture reflecting some aspects of input data. Here data comes as a set of points with additional optional information about some of the points. Data are considered dynamic and can change at any moment of time, but we suppose that all data changes are synchronized with main UI thread.

Scientific visualization is an interactive visualization of large amounts of dynamically changing data. Interactivity is important for exploratory purposes. Silverlight rendering performance is limited, so special techniques are required to achieve interactivity for great amounts of data: filtering, multithread rendering and gradual image improvements. Different visualization controls use different optimization techniques.

Basic definitions

A *plot* is an algorithm that maps data elements onto a set of graphic elements such as lines, markers or raster images. Location, size and other specific visual properties of the graphic elements could be either bound to the data in some way or have some fixed value. For example, line graph plot transforms series of points (X_i,Y_i) into a polyline with vertices coordinates equal to (X_i,Y_i). Source data and resulting plot are shown below.

X	***Y***
0	0
0,314159265	0,309016994
0,628318531	0,587785252
0,942477796	0,809016994
1,256637061	0,951056516
1,570796327	1
…	
5,654866776	-0,587785252
5,969026042	-0,309016994
6,283185307	0



Each plot requires a *coordinate transform,* a function that transforms data values *d* to screen coordinate x_s or y_s. Coordinate transforms in D^3 are always orthogonal, i.e. *x* and *y* axes transform independently of each other. Each axis transform is actually a composition of *data transform* and *plot transform.*

Data transform for a given data element *d* computes its vertical or horizontal coordinate *x* or *y* on *plot plane.* Plot plane can be thought of as an abstract sheet of graph paper which size is limited by representation of floating values only.

A *domain* of a given data transform is a possibly infinite segment on which the data transform function is defined and monotonically increasing, and thus invertible, too. Data transform takes into account some details about the nature of the data and can be non-linear. Values outside of domain must be transformed to System.Double.NaN. Same is true for back conversion: a plot coordinate value that doesn't match any data value is transformed to System.Double.NaN.

Data transform examples for vertical axis:

1. Identity: y = d; domain = (-inf,+inf).
2. Linear (see LinearDataTransform class): y = d * scale + offset; domain = (-inf,+inf); scale > 0
3. Logarithmic: y = log10(d); domain = [Double.Eps,+inf). Not supported in this release.
4. Mercator(see MercatorTransform class): y = S * log(tan(d)); domain = [-90 + Double.Eps, 90 – Double.Eps]

```
public class DataTransform
{
    public Range Domain { get; }
    public double DataToPlot(double d);
    public double PlotToData(double x);

    public static readonly DataTransform Identity;
}
```

Each plot occupies rectangular area on a screen and defines a window that maps portion of plot plane to this screen area. *Plot transform* is a transform from plot coordinates to screen coordinates. Plot transform is always a composition of Translate and Scale transforms, so it is independent along each axis and defined by four numbers xscale, yscale, xoffset, yoffset as

$x_s = x$ * xscale + xoffset,

$y_s = y$ * yscale + yoffset.

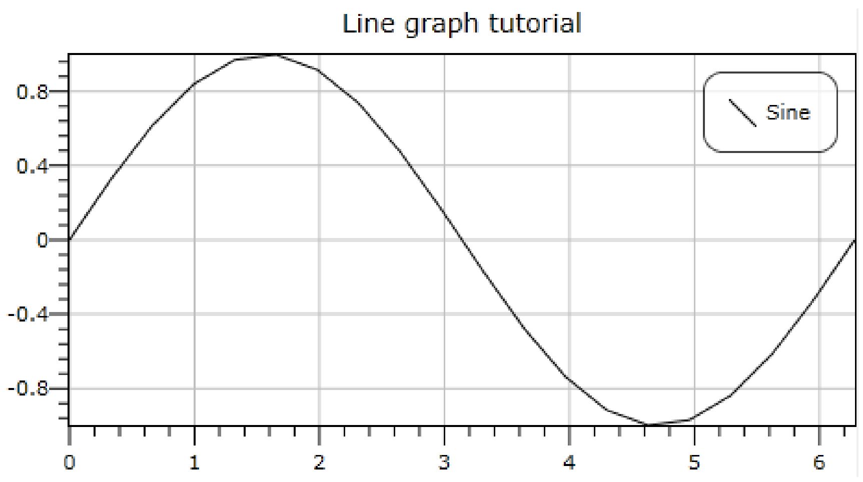
Without loss of generality we can assume that both xscale and yscale are positive.

Navigation is a process of interactive figure transform modification performed by a user. The goal of the navigation is to select a part of plot plane with features and graphical elements that are of interest to the user.

The ratio xscale/yscale is called *aspect ratio* of plot transform. Aspect ratio is important if user wants to maintain proportions of a picture.

A *figure* is a composition of Silverlight elements that performs dynamic data visualization. This composition includes plots with graphic elements, navigation and annotation elements. It is expected that all plots of a figure share exactly the same screen area. If this is not the case graphical elements of the plots may appear to be displaced against each other.

The following picture shows a figure with a single plot and several annotation elements such as vertical and horizontal axes, a grid, a legend and a title.



Plots

All visualizations in D^3 are performed using special controls derived from a class named PlotBase. To create a new visualization a user either uses existing elements such as Plot, LineGraph or Figure or creates new class derived from PlotBase.

PlotBase is a Silverlight panel that facilitates several tasks:

- Coordinate transformation between screen and user data
- AutoFit mode support
- Management of plot composition and coordinate transforms synchronization

Coordinate transformations

In D^3 a user doesn't have direct control over coefficients of transformation matrix. Instead he has two options. Either he defines the desired region of figure plane that should be visible in allocated screen space, or he enables AutoFit mode, when figure transform is automatically adjusted so that all graphic elements are always visible.

Four dependency properties PlotOriginX, PlotOriginY, PlotWidth, PlotHeight define a desired region of figure plane that should be entirely visible. Different compositions of figures are possible using these properties. For example, it is possible to bind PlotOriginX and PlotWidth properties of two figures to synchronize them along X axis (see 'Synchronized figures' sample in the D^3 Interactive SDK).

Dependency property AspectRatio allows preserving proportions while transforming points from figure to screen coordinates. AspectRatio == 0 (default value) means than no restrictions are applied. Visible region specified by user using PlotOriginX, PlotOriginY, PlotWidth and PlotHeight properties may not have required aspect ratio. In this case it is enlarged so that resulting transform has specified aspect ratio. Enlarged rectangle is called actual plot rectangle and can be obtained from read only property ActualPlotRect.

To summarize, we can say that Plot visible region is computed from properties defined by a user and size of allocated screen space.

Change of any property PlotWidth, PlotHeight, PlotOriginX, PlotOriginY or AspectRatio results in new layout and rendering pass for all plots of entire figure.

Transform between figure and screen coordinates is performed by four methods: XFromLeft, YFromTop, LeftFromX,TopFromY. Here left and top mean distance in screen units from point to left and top edges of element's screen rectangle (similar to Canvas attached properties).

Event PlotTransformChanged is raised any time when transform from plot to screen is changed. This event is raised inside measure phase of layout update so it is possible to update Children collection of plot according to current transform. See 'Matrix display' example in Interactive SDK that shows matrix grid lines in high zoom levels and hides them when zooming out. This is simple example of *filtering* approach to rendering optimization when some elements became invisible when they are not required or too small.

```
public class PlotBase
{
    // Visible rectangle control
    public double PlotOriginX, PlotOriginY { get; set; } // Dep. prop.
    public double PlotWidth, PlotHeight { get; set; } // Dep. prop.
    public double AspectRatio { get; set; } // Dep. prop.

    public DataRect ActualPlotRect { get; }

    public event EventHandler PlotTransformChanged;

    // Data transforms are identity by default
    public DataTransform DataTransformX { get; set; } // Dep. prop
    public DataTransform DataTransformY { get; set; } // Dep. prop

    public double XFromLeft(double left);
    public double YFromTop(double top);
    public double TopFromY(double y);
    public double LeftFromX(double x);

    …
}
```

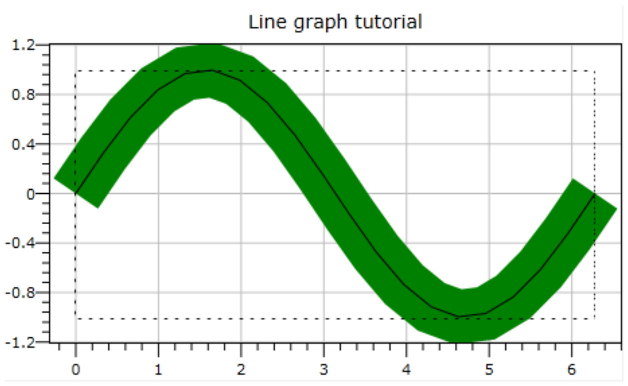
AutoFit mode support

One of most frequent and powerful navigation actions is fit-to-view which adjusts figure transform so that all figure graphic elements are visible. Auto fit may be triggered either by user action (Double-Click or press 'Home' key if default navigation is enabled) or because new graphic elements are produced by some plot of a figure, which may happen after data update. Auto fit mode is turned on by default.

Auto fit figure rectangle is computed from plot bounding rectangle and screen padding.

Plot bounding rectangle – minimal axis aligned rectangle in plot coordinates that contains all points of graphics elements produced by plot. In many cases bounding rectangle for plot is a bounding rectangle of all plot's input points in plot coordinates, but this is not true in general case. For example, convex hull for spline curve is not always a convex hull of its control points. Thus plot bounds computational algorithm is plot specific and it is defined by virtual ComputeBounds method. By default it returns empty rectangle. Plot bound rectangle is taken as [-0.5,0.5]x[-0.5,0.5] if ComputeBounds return empty value or [-0.5 + x, 0.5 + x] x [-0.5 + y, 0.5 + y] if ComputeBounds returns single point (x,y).

Screen bounding rectangle for a plot – minimal axis aligned rectangle that contains all screen representations of graphics elements produced by plot. In some cases screen bounding rectangle is larger than plot bounding rectangle transformed from figure to screen coordinates. For example, line graph drawn by stroke with thickness 5pt or points displayed with radius 10pt circles. This difference is approximated by screen padding. Screen padding is plot specific and is computed by virtual ComputePadding method. Default implementation returns value of Padding dependency property with default value (0,0,0,0).



On the figure above a line graph with a thick stroke is drawn. Dotted rectangle shows plot bounding rectangle. Padding here is a half of stroke thickness which helps to keep all pixels of line graph visible.

When performing an auto fit, transform coefficients are chosen with respect to aspect ratio so that plot bounding rectangle is entirely mapped into screen space decreased by padding.

Auto fit mode is controlled by a single dependency property IsAutoFitEnabled of type bool. When plot is in auto fit mode, its visible region is updated on each layout pass so that all graphics elements produced from data are visible. Plot implementations should trigger layout pass by calling InvalidateLayout on each data change to support auto fit mode.

Any manual adjustments of visible region by modifying PlotOriginX, PlotOriginY, PlotWidth and PlotHeight properties automatically turns off auto fit mode.

```
public class PlotBase
{
    // AutoFit control
    public bool IsAutoFitEnabled { get; set; } // Dep. prop.

    // Screen bounds computation
    protected virtual DataRect ComputeBounds();
    protected virtual Thickness ComputePadding()
        { return Padding; }
    public Padding { get; set; } // Dep. prop.
    …
}
```

Plot, Figure and Chart

Plot is ready to use control that defines attached properties X1, Y1, X2, Y2 and Points to specify plot coordinates of any Silverlight element. Plot control automatically computes plot bounds from plot coordinates of its children elements. See 'Drawing bounding lines' tutorial and 'Plotting Silverlight elements' sample in Interactive SDK for details.

```
<d3:Plot Padding="10,10,10,10">
    <Line d3:Plot.X1="-1" d3:Plot.Y1="0" d3:Plot.X2="1" d3:Plot.Y2="0"
          Stroke="Green" StrokeThickness="2"/>
    <Ellipse d3:Plot.X1="-0.8" d3:Plot.Y1="0.5" d3:Plot.X2="-0.3" d3:Plot.Y2="1.0"
             Stroke="Gold" StrokeThickness="2"/>
    <Polyline d3:Plot.Points="0.3,0.5,0.8,0.5,0.8,1.0,0.3,1.0,0.3,0.5"
              Stroke="Blue" StrokeThickness="2"/>
</d3:Plot>
```

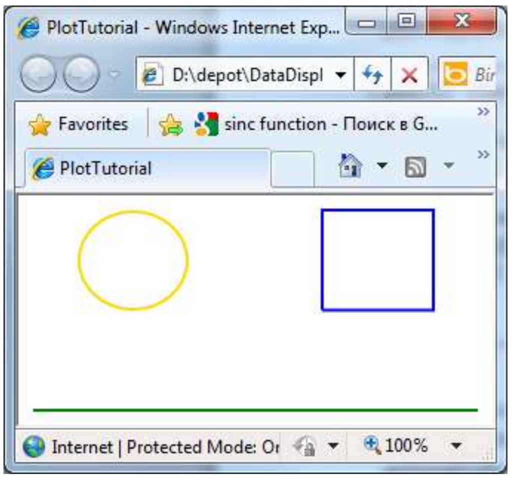
Results are shown below. Please notice that although all coordinates in plot elements were on figure place (in range -1..1) coordinate transformation is automatically adjusted to make resulting picture fit all available window space except for 10 pixel padding.

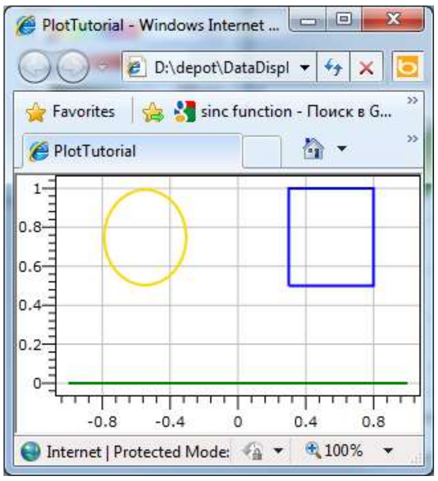
Plot handles infinity values is special way. They do not participate in plot bounds computation, "+Infinity" is translated to maximum visible value, "-Infinity" is translated to minimum visible coordinate. This is done in 'Drawing Silverlight elements' sample in Interactive SDK.

Figure class a panel is derived from PlotBase and provides special layout options that are often found in charts. It provides attached property Placement that allows to place child elements in center, left, top, right and bottom slots (see samples in 'Plot composition' section in Interactive SDK for Figure usage).

Usually plots are placed in the center of a Figure, axes and titles are placed in the side slots. Figure class provides two-pass algorithm that prevents well-known loop occurring on resize of figure with fixed aspect ratio: figure resize forces update of plot-to-screen transform which adjusts labels on the axes. Change of label size may result in change of central part size which again updates plot-to-screen transform which in turn leads to axes label updates and so on.

Chart element is a prepackaged figure with axis, grid lines, legend and title. It is enough to surround Plot element from previous listing with Chart tags. From technical point of view, Chart element is a ContentControl with special template that uses Figure class.

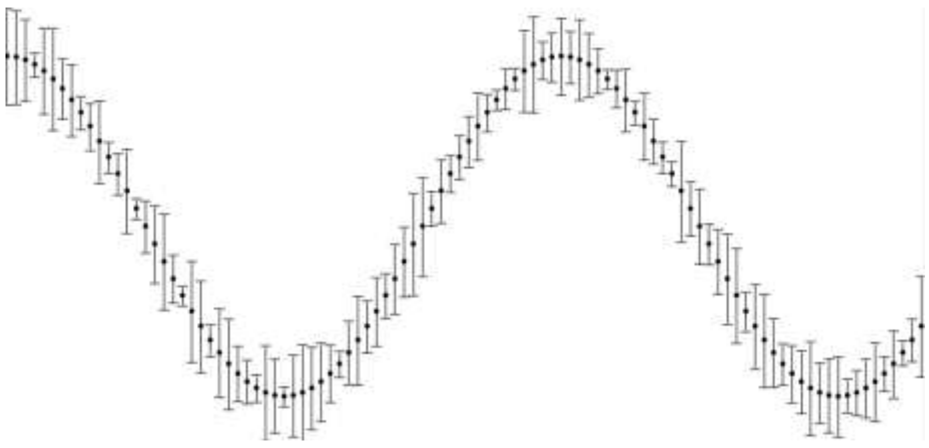

PlotTutorial - Windows Internet Exp...
D:\depot\DataDispl
Favorites
sinc function - Поиск в G...
PlotTutorial
Internet | Protected Mode: O
100%


PlotTutorial - Windows Internet ...
D:\depot\DataDispl
Favorites
sinc function - Поиск в G...
PlotTutorial
1
0.8
0.6
0.4
0.2
0
-0.8
-0.4
0
0.4
0.8
Internet | Protected Mode:
100%

```
<d3:Chart>
      <d3:Plot…/>
</d3:Chart>
```

Plot composition

Single plot is usually represented by single element derived from PlotBase. Complex figures usually contain more than one plot so it is important to synchronize transforms for all plots in figure to have coherent data display. Example is figure below, consisting of two plots: error bars showing error range and points indicating measured values.

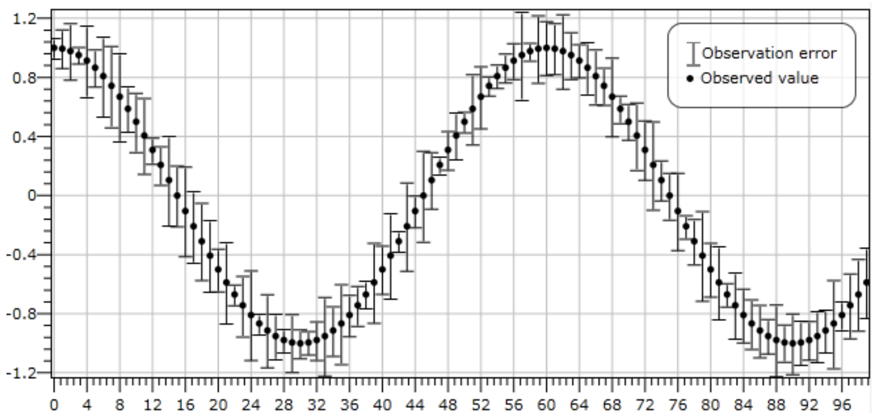


This effect can be achieved by making several plots children of another plot object. See XAML below:

```
<d3:Plot>
    <d3:ErrorBarGraph Size="10" Color="Black" …/>
    <d3:CircleMarkerGraph Description="Observed value" Size="5" Color="Black" …/>
</d3:Plot>
```

One of frequently used compositions is multiple child plots inside Chart. Chart element may have only single element as it content, so multiple plots are placed inside Grid element. This is illustrated by 'Drawing multiple graphs' tutorial.

```
<d3:Chart>
    <Grid>
        <d3:ErrorBarGraph Size="10" Color="Black"…/>
        <d3:CircleMarkerGraph Description="Observed value" Size="5" Color="Black…/>
    </Grid>
</d3:Chart>
```



It is very important to ensure that composition of plots has synchronized transforms, so they always coherent when performing navigation. In D^3 this is achieved by maintaining master-dependant relationship. Each PlotBase element can be either a *master* or *dependant*. Master-dependant relationship is defined according to visual element tree: PlotBase is a master if it has no other PlotBase panel as its ancestor element or its ForceMaster property is set to true; otherwise PlotBase is called a dependant of its ancestor PlotBase, which in turn could also be a dependent of another PlotBase. Therefore dependant plots form a tree within visual element tree with single master plot at the root. Master state is accessible through IsMaster property.

```
public class PlotBase
{
    // Master control
    bool ForceMaster { get; set; }
    bool IsMaster { get; }

    // Information about all plots in composition
    IEnumerable<PlotBase> RelatedPlots { get; }
    IObservable<PlotCompositionChange> CompositionChange { get; }

    // Composition data bounds and padding
    public DataRect GetEffectiveDataBounds();
    public Padding GetEffectivePadding();
    …
}

public class RelatedPlotsChange
{
    PlotBase[] RelatedPlots { get; }
}
```

Master PlotBase elements together with all its dependents are called *related plots* and are accessible using RelatedPlots property of each related plot. Changes in visual tree and/or setting ForceMaster property leads to immediate changes in RelatedPlots collection. These changes could be tracked by subscribing to CompositionChange observable. This observable generates event for each plot is composition.

Plot rectangle properties (PlotOriginX, PlotOriginY, PlotWidth, PlotHeight), auto fit mode and aspect ratio became *synced* when plot became dependant. This means that:

1. These properties are set to master's values when PlotBase is added to visual children of master.
2. Change of property of any plot in composition results in immediate change of corresponding property of all related plots.
3. No changes are made to these properties when PlotBase became master.

Plot-to-screen transform properties (ScaleX, ScaleY, OffsetX, OffsetY) are read-only and their values are always values of corresponding master properties. This guarantees that all related plots sharing same screen rectangle will have same plot transform.

Data-to-plot transform properties (XDataTransform and YDataTransform) are not affected by master-dependant relationship. This allows different plots in composition to have different data transform. One useful scenario is to combine multiple graphs sharing same horizontal axis but with different ranges of values of vertical axis. See 'Two scales' sample in InteractiveSDK that combines $y = sinc(x)$ with *y* ranges from -0.2 to 1.0 and $y = x^3$ with *y* ranges from -1000 to 1000 graphs on single chart.

Auto fit implementation for plot composition takes aggregated plot bounds and aggregated screen padding into account. Aggregated plot bounds is an axis-aligned union of local bounds for all related plots and aggregated padding is a maximum of all paddings for all related plots. For composition of plots aggregated bounds and aggregating padding are used in fit-to-view algorithm. This ensures that all parts of all plots will be visible screen space allocated to master plot.

Palette

Palette maps numeric value into colours. D^3 supports normalized and absolute palettes.

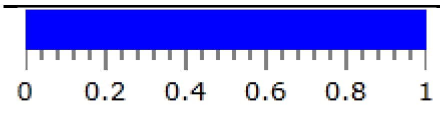
A normalized palette defines mapping from segment [0, 1]. Visualization is responsible for conversion of actual data range to [0, 1] segment. Heatmap and MarkerGraph classes use linear transform. A normalized palette is easy to use because it always covers the entire data range. The downside is that a change in the data may change which values the colours are mapped to. Consider normalized palette where Blue colour corresponds to 0, Green corresponds to 0.5 and Red corresponds to 1. When displaying data that ranges from -500 to 500 all values close to zero will be mapped to Green. But if the data change so that its values cover range from -500 to 1500 then zeroes change their colour to teal, and green shades correspond values about 500.

Absolute palette binds colours to fixed values. For example, a palette used to display elevation above sea level may map values close to 3000m to Brown and values below -5000m to dark blue.

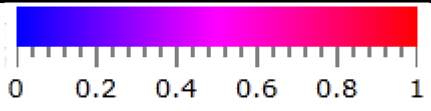
D^3 provides special language to define palettes as strings. For the first, we consider normalized palette.

Palette is defined by sequence of points. Each point has assigned colour and value in [0, 1] interval. In string representation point colours are defined by colour names (Red, Green, Blue and all other names supported by Silverlight) or hexadecimal notation #RRGGBB or ##AARRGGBB where A is transparency (00 – fully transparent, FF – opaque).

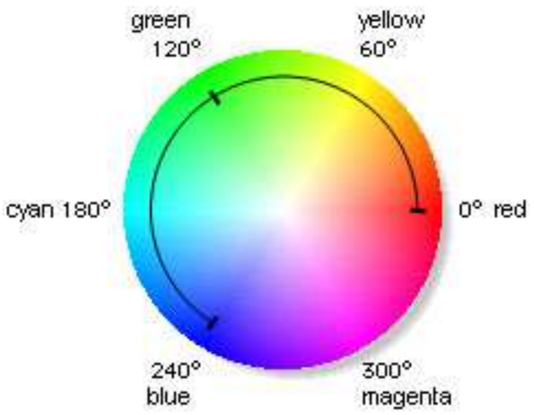
Simple palette maps all values to single colour

Palette	String representation
	"Blue"

More points can be added separated by comma:

Palette	String representation
	"Blue,Red"

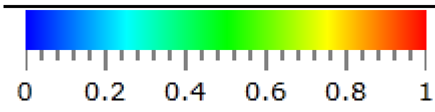
D³ uses HSL (Hue-Saturation-Lightness) model when interpolating between two colours. Hue, saturation and lightness components are interpolated independently. HSL model guarantees that all colours between brightest Blue (#0000FF) and brightest Red (#FF0000) will also have maximum brightness. The figure below shows interpolation path between red, green and blue colours. Yellow is in the middle between red and green, cyan is in the middle between green and blue.



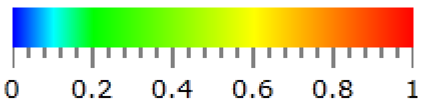
HSL definition is not standardized, so we put RGB to HSL conversion formulae used in D³ here:

$$M = \max(R, G, B),$$
$$m = \min(R, G, B),$$
$$C = M - m,$$
$$H = 60^o \times \begin{cases} undefined, C = 0, \\ \dfrac{G-B}{C} mod\ 6, M = R, \\ \dfrac{B-R}{C} + 2, M = G, \\ \dfrac{R-G}{2}, M = B, \end{cases}$$
$$L = \frac{M+m}{2},$$
$$S = \begin{cases} 0, C = 0, \\ \dfrac{1}{1 - |2L - 1|} \end{cases}.$$

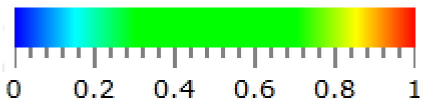
Values are assigned for points to be distributed uniformly. In next palette point between Blue and Red will have value of 0.5. Note that we use hexadecimal notation #00FF00 instead of name Green, because Silverlight Green colour has lower brightness.

Palette	**String representation**
	"Blue, #00FF00,Red"

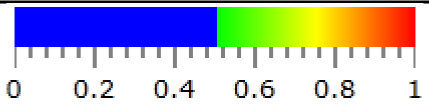
Values can be assigned to palette points explicitly. Put a value and '=' sign either from left or from right of colour. Note that compared to a previous one in the following palette has green colour shifted to the left.

Palette	**String representation**
	"Blue, #00FF00=0.2,Red" or "Blue,0.2=#00FF00,Red"

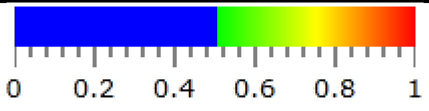
Regions of solid colours are defined by using both left and right value assignment:

Palette	**String representation**
	"Blue,0.3=#00FF00=0.7,Red"

It is possible to create palette point that has one colour from the left and another from the right. To do this connect point with left or right neighbor using '=' sign. Here point value 0.5 has blue colour on the left and linearly interpolated from green to red on the right.

Palette	String representation
	“Blue=0.5=#00FF00,Red”

Absolute palette is constructed almost in the same way, but its minimum and maximum values are specified at the left and right of border points. All values in the middle are also specified as absolute. Palette shown below is defined for values from -60 to 40, all negative values are shown as blue. All values below -60 will be shown as blue, all values above 40 – as red.

Palette	String representation
	“60=Blue=0=#00FF00,Red=40”

D³ value to colour conversion is supported by IPalette interface and Palette class. Range property returns [0,1] for normalized palettes and range specified in palette definition for absolute palettes. Static method Parse allows constructing palette instances from strings. Value converter for using palette strings in XAML is also supported.

```
interface IPalette
{
    Color GetColor(double value);
    bool IsNormalized { get; }
    Range Range { get; }
}

class Palette : IPalette
{
    static Palette Parse(string s);
    …
}
```

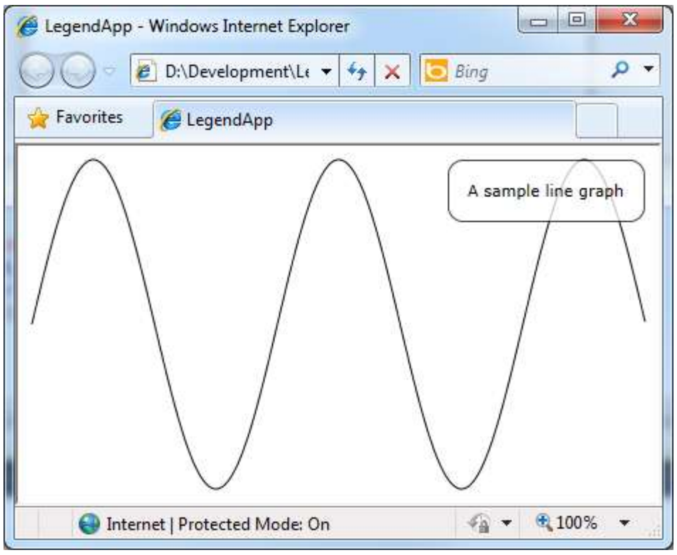
Legend

Legend shows extra information about figure such as plot descriptions and mapping between values and visual properties. Legends in D³ are displayed using two elements: Legend and LegendItemsPanel.

Legend element is a simple templated ContentControl that has predefined rounded rectangle border and is located in the right top of a parent control (e.g. Chart). Default content of Legend is LegendItemsPanel but it can be easily overridden in XAML.

```
<d3:Figure Padding="10,10,10,10">
     <d3:LineGraph x:Name="linegraph"/>
     <d3:Legend>A sample line graph</d3:Legend>
</d3:Figure>
```

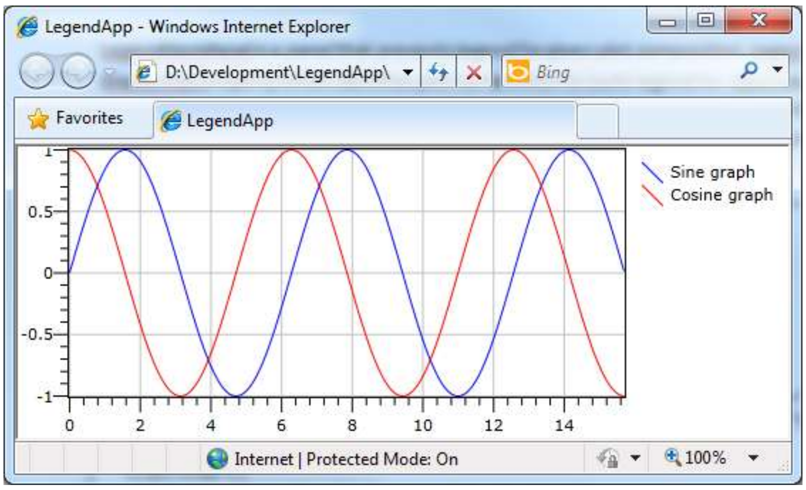
This XAML will show following application:


LegendApp - Windows Internet Explorer
D:\Development\Le
Bing
Favorites
LegendApp
A sample line graph
Internet | Protected Mode: On
100%

An instance of Legend control is present in Chart control. Its content is available and can be changed using Chart.LegentContent property.

LegendItemsPanel is a panel that presents legend for given plot composition. LegendItemsPanel provides MasterPlot property which identifies plot composition to build legend for. Next example shows how LegendItemsPanel can be used outside of Chart. Default chart legend is hidden using LegendVisibility property, new LegendItemsPanel is placed outside Chart and has its MasterPlot property is bound to plot composition of two line graphs.

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <d3:Chart LegendVisibility="Collapsed">
    <d3:Plot x:Name="plot">
        <d3:LineGraph x:Name="sin" Stroke="Blue" Description="Sine graph"/>
        <d3:LineGraph x:Name="cos" Stroke="Red" Description="Cosine graph"/>
    </d3:Plot>
  </d3:Chart>
  <d3:LegendItemsPanel Margin="10,10,10,10"
      Grid.Column="1" MasterPlot="{Binding ElementName=plot}"/>
</Grid>
```


LegendApp - Windows Internet Explorer
D:\Development\LegendApp\
Bing
Favorites
LegendApp
Sine graph
Cosine graph
Internet | Protected Mode: On
100%

LegendItemsPanel observes CompositionChange events of MasterPlot and rebuilds contents every time composition is changed, so any plot of composition can be assigned to MasterPlot.

Plot is included in legend if LegendItemsPanel has a *template* for it and plot doesn't have attached property Legend.IsVisible set to false. LegendItemsPanel has default templates for MarkerGraph, LineGraph and HeatmapGraph elements in its resource dictionary. New templates can be added and existing templates can be overrided by adding new DataTemplates with key equal to full type name to resource dictionary of LegendItemsPanel. 'Line graph legend' sample in Interactive SDK shows how to override legend appearance. Key moments are illustrated on the XAML below.

```
<d3:Chart>
  <!-- Replace default legend contents with customized ->
  <d3:Chart.LegendContents>
    <d3:LegendItemsPanel>
      <!-- New or overridden legend item templates ->
      <d3:LegendItemsPanel.Resources>
        <!-- Key is a full type name of plot element ->
        <DataTemplate
             x:Key="Microsoft.Research.DynamicDataDisplay.LineGraph">
          …legend template that has plot element as DataContext…
        </DataTemplate>
      </d3:LegendItemsPanel.Resources>
    <d3:Chart.LegendContents>
</d3:LegendItemsPanel>
```

Marker graph

Marker graph shows values of multiple variables as a collection of markers, where some variables define marker position and other variable define visual properties. Particular cases of marker graph include scatter plots, which show relationship between two variables by plotting points with coordinates defined by these variables, and bubble charts, where third variable defines size of point. More complex markers are possible, such as error bars or candlestick charts.

Marker graphs are plotted by instances of MarkerGraph class. To plot MarkerGraph one needs to specify data and visual appearance for each marker. Tooltip and legend visual representation can be optionally specified also. Several ready-to-use predefined marker graphs are provided by D3 library, see 'Marker templates', ' Interval plots', 'Bar chart' and 'Bubble chart' samples in Interactive SDK. All these classes are defined from MarkerGraph class.

```
public class MarkerGraph
{
    // Mandatory properties to draw a marker graph
    public DataCollection Sources { get; set; } // Dep. prop
    public DataTemplate MarkerTemplate { get; set; } // Dep. prop.

    // Optional properties
    public DataTemplate LegendTemplate { get; set; } // Dep. prop.
    public DataTemplate TooltipTemplate { get; set; } // Dep. prop.
    …
}
```

Further we'll discuss how to define marker graph from scratch using MarkerGraph class. You can find sample that plots custom marker graph in Interactive SDK.

Data model

MarkerGraph shows relationship between multiple variables or data series. One variable is represented by one instance of DataSeries class. Collection of DataSeries is specified using Sources property of marker graph. Each data series has properties to set variable data, description and key.

```
public class DataSeries
{
    public string Key { get; set; } // Dep. prop.
    public object Data { get; set; } // Dep. prop.
    public string Description { get; set; } // Dep. prop.
    …
}
```

Key is a string value that identifies variable in bindings to visual element properties. Predefined marker shapes rely on predefined series keys, for example 'X' and 'Y' for Cartesian coordinates.

Data series key 'X' is special. If no data series with 'X' key present or its Data property is null, data series is assumed existent anyway and its values became ordinary number of item, e.g. 0,1,2….

Description is a readable description for values in the data series. Description appears in legend and tooltip.

Collection of data series is specified as content of MarkerGraph element.

```
<d3:MarkerGraph x:Name="markerGraph">
  <d3:DataSeries Key="X" Description="X coordinate"/>
  <d3:DataSeries Key="Y" Description="Y value"/>
  <d3:DataSeries Key="Z" Description="Another data series"/>
</d3:MarkerGraph>
```

Data property defines actual data of the data series. Data can be object of any type, but 1d arrays and IEnumerables (excluding strings) are treated as vector data and other types are treated as scalar. Data property can be null. Note that in Fig. 32 Data property for all three data series will have default value null and no markers will be displayed.

Not every combination of data series form valid data for marker graph visualization.

Combination of data series is called valid iff all data series with vector (array or IEnumerable) data has same number of elements and all data series except for 'X' has non-null values.

Number of markers to draw for valid non empty collection of data series is defined as length of any vector data series (if at least one vector data series exists) or' 1' if all data series are scalar.

Marker graphs are drawn only for valid data series. No exception is thrown and no drawing occurs for invalid combination of data series. Note that invalid data series not always indicate a bug in the code. For example, it often occurs when assigning new data, either in code or by bindings, to marker graph with at least two series: while second data series has old data with 'N' values, first data series is already has new value with 'M' values. MarkerGraph class provides way to assign all data to data series at once without going into invalid state by calling method MarkerGraph.Plot(params object[] data) that assigns data array elements to data of data series as atomic operation in the order they are specified in XAML.

Next few examples show how different combination of data series in marker graph defined in Fig. 32 result in different number of markers and values of their properties. Note how data series are accessed from code.

```
markerGraph.Sources['X'].Data = new int[] { -1,2,3,5,10 };
markerGraph.Sources['Y'].Data = new double[] { 0.2,0.1,3.5,-1.9,1.0 };
markerGraph.Sources['Z'].Data = new string[] { "A","B","C","D","E" };
```

	Marker #0	Marker #1	Marker #2	Marker #3	Marker #4
X	-1	2	3	5	10
Y	0.2	0.1	3.5	-1.9	1.0
Z	"A"	"B"	"C"	"D"	"E"

```
markerGraph.Sources['X'].Data = new int[] { -1,2,3,5,10 };
markerGraph.Sources['Y'].Data = new double[] { 0.2,0.1,3.5,-1.9,1.0 };
markerGraph.Sources['Z'].Data = new string[] { "Hello" };
```

	Marker #0	Marker #1	Marker #2	Marker #3	Marker #4
X	-1	2	3	5	10
Y	0.2	0.1	3.5	-1.9	1.0
Z	"Hello"	"Hello"	"Hello"	"Hello"	"Hello"

```
// Note: no data is assigned to ‘X’ data series
markerGraph.Sources[‘Y’].Data = new double[] { 0.2,0.1,3.5,-1.9,1.0 };
markerGraph.Sources[‘Z’].Data = new string[] { “Hello” };
```

	Marker #0	Marker #1	Marker #2	Marker #3	Marker #4
X	0	1	2	3	4
Y	0.2	0.1	3.5	-1.9	1.0
Z	“Hello”	“Hello”	“Hello”	“Hello”	“Hello”

Marker template

Each marker is displayed as composition of Silverlight elements called marker template. This composition is specified in XAML as DataTemplate. This template should be assigned to MarkerTemplate property of MarkerGraph. Next XAML code shows simple template of a circle marker with diameter 10.

```
<DataTemplate x:Key="Circle">
    <Ellipse d3:Plot.X1="{Binding X}"
             d3:Plot.Y1="{Binding Y}"
             Fill="{Binding C}"
             Width="10" Height="10"
             Stroke="{Binding Stroke}"
             StrokeThickness="{Binding StrokeThickness}">
       <Ellipse.RenderTransform>
           <TranslateTransform X="-5" Y="-5" />
       </Ellipse.RenderTransform>
    </Ellipse>
</DataTemplate>
```

Marker template is instantiated for each marker to be drawn. Instance of model view class derived from MarkerViewModel is set as DataContext of instantiated template. In addition to some predefined properties, MarkerViewModel class has dynamically generated properties with data series names that return value of corresponding data series for this marker. Stroke and StrokeThickness properties return values of similar properties of MarkerGraph class. Self property returns instance of view model object. It is required in case when some visual property of marker, such as height of VerticalIntervalBar marker, cannot be computed from single data series and requires special converter that took MarkerViewModel as a source. Although Silverlight provides special notation “.” to access DataContext itself, “.” notation doesn’t force update of binding when marker data changes.

```
class MarkerViewModel : INotifyPropertyChanged
{
    public Brush Stroke { get; }
    public double StrokeThickness { get; }
    public MarkerViewModel Self { get; }
    public DataSeries Series { get; } // Series itself
    …
}
```

Instantiated marker templates are put to panel, derived from Plot class. Plot coordinates of markers are usually specified using Plot.X1, Plot.Y1, Plot.X2, Plot.Y2 and Plot.Points attached properties. Marker template shown on Fig. 23 uses additional render transform to move center of the circle to point, specified by X and Y data series.

Special data series

Some data series perform conversion of user data and return converted values when accessing properties named by data series from marker template. Two examples supported by D^3 are ColorSeries and SizeSeries.

ColorSeries maps numeric values of data series to colours using palette. This mapping requires knowledge of values range for entire data series when using normalized palettes, so min value is mapped to the left colour of palette and max value is mapped to the right colour. SizeSeries maps numeric values of data series to screen size of markers which is also requires knowledge of values range for entire data series, so min value is mapped to screen size given by Min property, max value is mapped to Max screen size and values between are linear interpolated.

DataSeries class provide property called Converter which specify instance of IValueConverter used to convert data series values. DataSeries itself are passed as converter's parameter. Also DataSeries has properties Min and Max which return minimal and maximal numeric values in the series (Convert.ToDouble method is used for conversion of data series items). If data cannot be converted to numbers or data series has no elements NaN value is returned by Min and Max properties.

```
class DataSeries
{
    public double Min { get; }
    public double Max { get; }
    public IValueConverter Converter { get; set; }
    …
}

class ColorSeries : DataSeries
{
    public IPalette Palette { get; set; } // Number -> color
    …
}

class SizeSeries : DataSeries
{
    public double Min { get; set; } // Screen size for min. value
    public double Max { get; set; } // Screen size for max. value
    …
}
```

Colour data series perform no conversion if data is scalar string or Brush or vector of strings or Brushes.

Tooltip and Legend templates

In addition to MarkerGraph plot information about markers is displayed in two places: in tooltip that appears when pointing by mouse to a marker and in legend. MarkerGraph class has two properties that define two DataTemplates for displaying tooltip and legend.

Corresponding instance of MarkerViewModel is set as DataContext when instantiating template for marker tooltip, so tooltip template has access to all values of data series and to data series itself, for example, to get data series description. However, that is not enough when working with data series that uses Converter, such as ColorSeries. Suppose that ColorSeries has key 'C'. In this case property 'C' will return not a value supplied by user, but colour taken from palette. That's why properties prefixed with 'Original' are generated at runtime in parallel with properties named after data series. Original properties will return untransformed value even if Converter is specified for data series.

Example below illustrates how Series property is used to get description of entire data series and how binding to 'OriginalC' is set to get original numeric value instead of converted colour for 'C' data series.

```
<DataTemplate x:Key="ColorTooltip">
    <Grid>
        <Grid.RowDefinitions>
           <RowDefinition/>
           <RowDefinition/>
           <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <!-- Data series descriptions !>
        <TextBlock HorizontalAlignment="Right"
            Text="{Binding Series.X.Description, StringFormat='\{0\}:'"/>
        <TextBlock HorizontalAlignment="Right" Grid.Row="1"
            Text="{Binding Series.Y.Description, StringFormat='\{0\}:'"/>
        <TextBlock HorizontalAlignment="Right" Grid.Row="2"
            Text="{Binding Series.C.Description, StringFormat='\{0\}:'"/>

        <!-- Data values ->
        <TextBlock Grid.Column="1" Margin="5,0,0,0"
            Text="{Binding Path=X}"/>
        <TextBlock Grid.Row="1" Margin="5,0,0,0" Grid.Column="1"
            Text="{Binding Path=Y}"/>
        <TextBlock Grid.Row="2" Margin="5,0,0,0" Grid.Column="1"
            Text="{Binding Path=OriginalC}"/>
    </Grid>
</DataTemplate>
```

DataContext for LegendTemplate is collection of all data series with dynamically generated properties with names equal to data series key to access individual data series. This allows drawing meaningful legend even when no data is supplied and no markers are available. However, information about values of data series which is available through MarkerViewModel instances is useful sometimes in legend template. Example is size and colour of markers if they are supposed to be similar for all markers (like in scatter plot). To access this information DataSeries class provides First property that returns view model for the first marker in series. First property may return null, if no markers exist. This is handled by FallbackValue attribute in XAML below.

```
 <DataTemplate x:Key="CircleLegend">
     <StackPanel Orientation="Horizontal">
         <Ellipse Fill="{Binding C.First, TargetNullValue=Black}"
                  Width="{Binding D.First, TargetNullValue=10}"
                  Height="{Binding D.First, TargetNullValue=10}"/>
         <TextBlock Margin="5,0,0,0" VerticalAlignment="Center"
                    Text="{Binding Y.Description}"/>
     </StackPanel>
</DataTemplate>
```

Here we defined a legend for scatter plot with data series X, Y, C and D. C and D series are supposed to be scalar and define colour and size of markers.

Rendering optimization

Silverlight has limited rendering performance and plotting more than a few thousands of elements takes signification amount of time which is enough to make application non-interactive. Scientific data sets often contain tens of thousands of data items. To overcome Silverlight limitation MarkerGraph uses bitmap caching and deferred rendering. You can see how it works in Many Markers sample in Interactive SDK. Markers appear on the screen in batches. Each new batch is rendered when application is idle, so when user navigated or interacts with UI in other manner markers rendering is suspended. When marker graph needs to be redrawn because of panning or zooming, all batches are instantly replaced with bitmaps which are translated and scaled according to user navigation. After that bitmaps are replaced with real markers batch by batch in application idle moments. You may see it clear when zooming in significantly. Markers became blurred because of bitmap scaling. Later, blurred markers are replaced with re-rendered ones which makes picture look sharp again.

User can control this process using two properties. MarkerBatchSize controls size of marker batch rendered at a single pass. Larger values can cause poor application resposiveness for complex marker templates, small values will result in increase of marker graph update time.

```
class MarkerGraph
{
    public int MarkerBatchSize { get; set; } // Dep. prop.
    public int MaxSnapshotSize { get; set; } // Dep. prop.

    public long Plot(params object[] data);
    IObservable<RenderCompletion> RenderCompletion;
    …
}
```

Marker graph rendering is restarted when data are changed. If data changes occur more frequently than application idle times when marker graph draws itself, most or even all of the data updates may be skipped and not shown. MarkerGraph provides technique to wait until current data is completely displayed. Plot method returns rendering task Id. When this task is either completed or skipped because new data arrives task id is observed on RenderCompletion subject. Using this technique, application can wait for markers complete draw before updating data. See 'Background rendering' sample in Interactive SDK.

Marker rendering in multiple passes raises following problem when implementing auto fit mode: plot bounds and screen margins must be known for all markers before all batches are rendered. This means that bounds cannot be computed from Plot's attached properties. Different shapes of markers may have different ways to calculate their plot rectangle and screen padding. For example, for vertical interval marker (X,Ymin,Ymax) plot bounds is rectangle [X,X] x [Ymin,Ymax] and screen padding is marker width. For bubble chart marker plot bounds in point rectangle [X,X] x [Y,Y] and screen padding half of its radius.

MarkerGraph looks for resources named 'PlotBoundsFunc' and 'ScreenBoundsFunc' which should be value converters that return DataRect and Thickness from instance of MarkerViewModel. See XAML below for definition of marker template with custom plot bounds and screen padding.

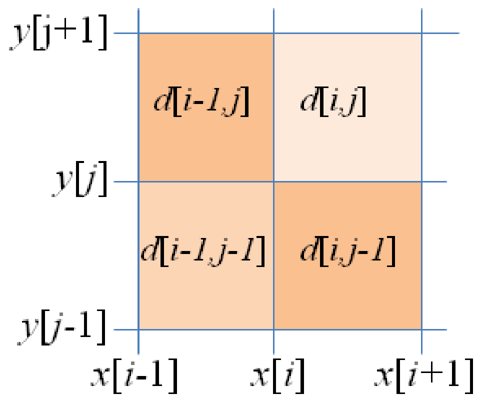
```
    <DataTemplate x:Key="VerticalInterval">
        <Grid …>
          <Grid.Resources>
                <d3:VerticalIntervalDataBoundsConverter
                    x:Key="DataBoundsFunc"/>
                <d3:ErrorBarScreenThicknessConverter
                    x:Key="ScreenBoundsFunc"/>
            </Grid.Resources>
            …
        </Grid>
    </DataTemplate>
```

Heat map

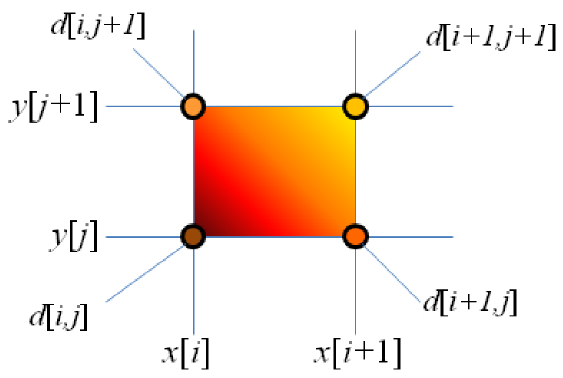
Bitmap and gradient modes

Heat map graph plots 2D array as a coloured rectangle, where colour of each point is determined by value of one to four nearest array items. More precisely, heat map graph support two visualization modes: bitmap and gradient.

Heat map requires three arrays: 2D data array and two 1D grid arrays that define heat map points location in plot coordinates. In *bitmap* mode lengths of grid arrays X[i] and Y[i] are greater by one than dimensions of data array D[i,j]. Rectangular area between points X[i],Y[j] and X[i+1],Y[j+1] is painted with solid colour mapped from D[i,j] by palette.



In gradient mode colours of points at (X[i],Y[j]), (X[i+1],Y[j]), (X[i],Y[j+1]), (X[i+1],Y[j+1]) are defined by data points D[i,j], D[i+1,j], D[i,j+1], D[i+1,j+1] respectively. For inner points data value is bi-linearly interpolated.



Palette class provides family of generic methods called Plot. In takes three parameters and defines rendering mode depending of lengths of X and Y parameters. Overloaded version of Plot takes additional parameter for *missing value.* Rectangle between X[i],Y[j] and X[i+1],Y[j+1] is not filled if D[i,j] equals to missing value in bitmap mode or at least one of values D[i,j], D[i+1,j], D[i,j+1], D[i+1,j+1] are missing value in gradient mode.

```
class HeatmapGraph
{
    long Plot<T,A>(T[] data, A[] x, A[] y);
    long Plot<T,A>(T[] data, A[] x, A[] y, T missingValue);
    IPalette Palette { get; set; } // Dep. prop.
    IObservable<RenderCompletion> RenderCompletion { get. }
}
```

Background rendering

Heat map graph uses background rendering to improve application responsiveness when handling large data. Plot method is asynchronous and doesn't perform actual calculations. Instead, data filtering and palette lookups take place in separate thread where bitmap representing heat map is created. Bitmap is instantly displayed on the screen when it is ready.

Plot method returns before data is actually displayed, so it is possible to call Plot while background rendering is in progress. In this case results of background rendering are dropped and new data are being rendered. This means that when data updates are frequent not all or event none of the heat maps will appear on the screen. HeatmapGraph provides interface to asynchronously wait until heat map for specific data is complete and displayed. Plot method returns rendering task id. When heat map is completed this id is observed on RenderCompletion observable. You can see how it works in 'Heatmap background' rendering sample in Interactive SDK.

Heatmap class is derived from BackgroundBitmapRenderer class that provides infrastructure for background rendering to bitmap and can be a base for new bitmap-based visualizations, such as fractals or bitmap-based vector field display. BackgroundBitmapRender provides two methods: QueueRenderTask which queues new background rendering task and virtual method RenderFrame that takes RenderTaskState structure with visible rectangle in plot and screen coordinates and produces a bitmap along with information how to map it to the plot coordinates (see RenderResult structure). RenderTaskState structure provides IsCancelled property which should be checked from time to time during long render operations to stop rendering if new data appears and current task should be dropped.

```
class BackgroundBitmapRenderer
{
    virtual RenderResult RenderFrame(RenderTaskState state);
    long QueueRenderTask();
    IObservable<RenderCompletion> RenderCompletion { get }
    …
}
```

Tooltip layer

Heat maps are displayed using Silverlight Image control. Thus it is impossible to use standard tooltip services because it will produce similar tooltip for every point of image. Another limitation is that if two heat maps are combined using transparency standard tooltip will appear only for top heat map. HeatmapTooltip class overcomes this limitation. HeatmapTooltip finds nearest PlotBase in the chain of visual parents and maintain actual list of all HeatmapGraphs in composition by monitoring CompositionChanged observable. After mouse cursor is still for some time, tooltip layer enumerates all heat maps under cursor and calls HeatmapGraph.TooltipContentFunc. Results are arranged as vertical stack of ContentControls. Tooltip contents are updated if any of the heat maps raises RenderCompletion event which may indicate change of data.

Return value of HeatmapGraph.TooltipContentFunc is shown as content of ContentControl, so it may be a simple string or UI element. Default implementation show data and coordinate values of nearest point in gradient mode or coordinates and value exactly under cursor in bitmap mode.

```
class HeatmapGraph
{
   Func<Point,object> TooltipContentFunc { get; set; }
    …
}
```

Line graph

Line graph is simple control that plot lines connecting points in plot coordinates. Line graph uses internally Polyline Silverlight element with attached Plot.Points property. Line graph adds Plot methods, Description property and legend support to existing Polyline functionality. All line graph has black stroke of thickness '1' by define.

```
class LineGraph : Plot
{
   string Description { get; set; } // Dep. prop
   Brush Stroke { get; set; } // Dep. prop
   double StrokeThickness { get; set; } // Dep. prop
   …
   void PlotY(IEnumerable y);
   void PlotXY(IEnumerable x, IEnumerable y);
}
```

Navigation

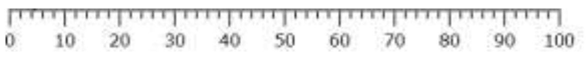
Navigation is performed by user to select area on the plot plane that contains interesting details at required zoom level. Navigation changes PlotOriginX,PlotOriginY,PlotWidth,PlotHeight, AspectRatio and IsAutoFitEnabled properties.

D^3 provides default controls for mouse and keyboard navigation: MouseNavigation and KeyboardNavigation. They are typically placed in the central slot of Figure. These controls are already placed inside Chart control.

D^3 controls support consistent Tab navigation model and Focus logic. When MouseNavigation intercepts mouse click it locates nearest Control in the chain of parent elements and makes it focused. The only tab-stoppable control inside a Chart is KeyboardNavigation, so after mouse click all keyboard input is routed to KeyboardNavigation element. Chart control has two visual states for focused and unfocused appearance.

Axes

Axes controls show how numeric values of different nature correspond to screen coordinates. Labels are drawn for each major tick. Shorter minor ticks divide space between major ticks into five segments. Here is how an axis looks when its properties set to Placement = Bottom, Range = [0, 100].



Range property defines range of values to show on axis. Min value corresponds exactly to right edge of axis, max value – to left edge. Orientation which may be 'Left', 'Top', 'Right', 'Border' define orientation of axis and placement of ticks and labels. DataTransform allows to specify additional data transform between values of Range and Ticks property: values specified in Range are considered plot coordinates and values displayed as labels are considered data coordinates. DataTransform is identity by default. Ticks property can be used to obtain plot coordinates of major ticks.

```
class Axis
{
   Range Range { get; set; } // Dep. prop
   double[] Ticks { get; } // Dep. prop
   AxisOrientation Orientation { get; set; } // Dep. prop
   DataTransform DataTransform { get; set; } // Dep. prop
    …
}
```

Ticks are generated in data coordinates, so plot-to-data transform (see DataTransform.PlotToData method) is applied to Range property and Ticks property specify ticks in data coordinates. Formula for tick value t_i is shown below. Parameter p is chosen from (1, 2, 5) and q is chosen from integer numbers to achieve optimal major ticks placement (not too dense, not to sparse).

$$t_i = i \cdot p \cdot 10^q$$

PlotAxis control is a templated ContentControl that has Axis control in its template. PlotAxis control is aware about plot compositions and automatically sets Range of axis to reflect ActualPlotRange. Content of PlotAxis is placed above Axis control. Chart control places instances of MouseNavigation over Axis to support individual navigation over vertical and horizontal axis.