Interactive Charts & Graphs Toolkit

1. Introduction

Interactive Charts Toolkit is a versatile and powerful Unreal Engine plugin designed to simplify the creation of dynamic, visually rich data visualizations. It provides a comprehensive collection of chart types, enabling developers to represent complex data in an intuitive and interactive way.

With support for multiple chart formats such as bar charts, line graphs, pie charts, doughnut charts, and more, the toolkit is built to handle a wide range of use cases—from in-game analytics and dashboards to UI-driven data displays. Each chart is highly customizable, allowing control over colors, gradients, labels, animations, and layout to match any project's design.

The plugin focuses on performance and flexibility, making it suitable for both real-time applications and static presentations. Its interactive features, such as hover effects, tooltips, and dynamic updates, enhance user engagement and make data exploration seamless.

Whether you're building a game, simulation, or data-driven application, Interactive Charts Toolkit provides the tools that are needed to transform raw data into meaningful visual insights.

The system includes:

- Full Customization for all the charts.
- Demo charts for direct usability for the user.
- Easy to add and edit data for all the charts.

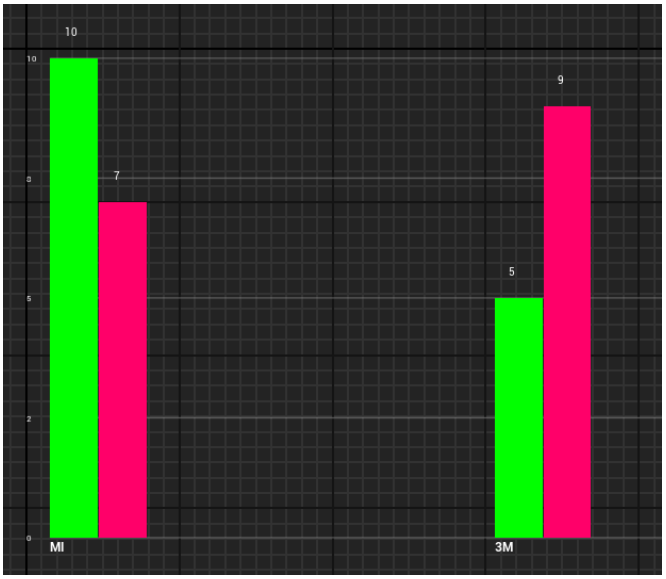
2. System Overview

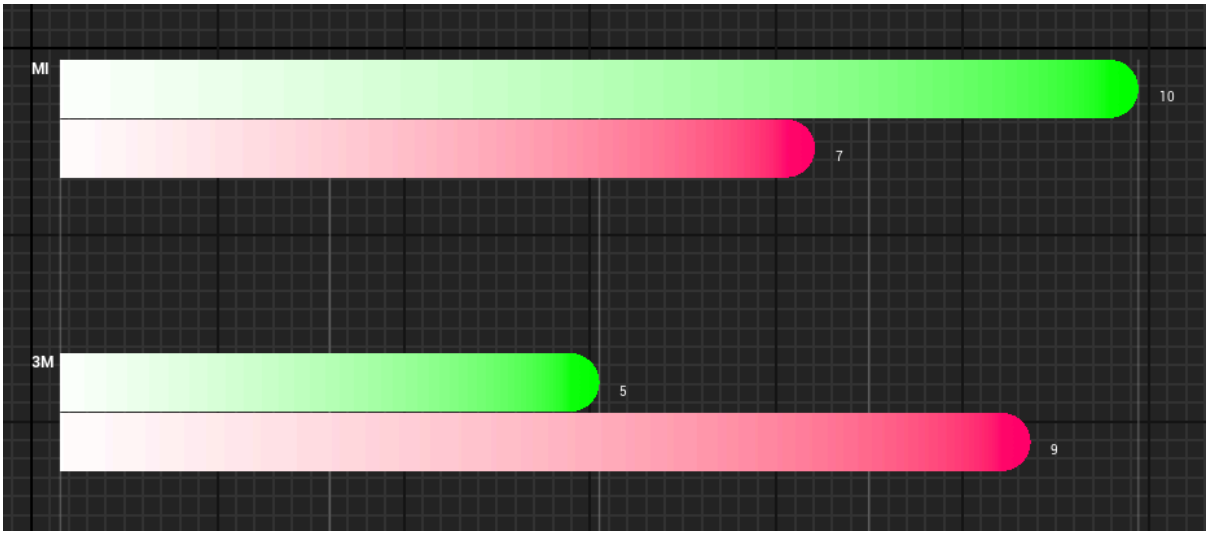
In this system there are different sorts of charts created and represented which can be used in games and other representations. Below are the detailed explanations of all the charts that are there in the plugin:

2.1 Bar Chart

Overview

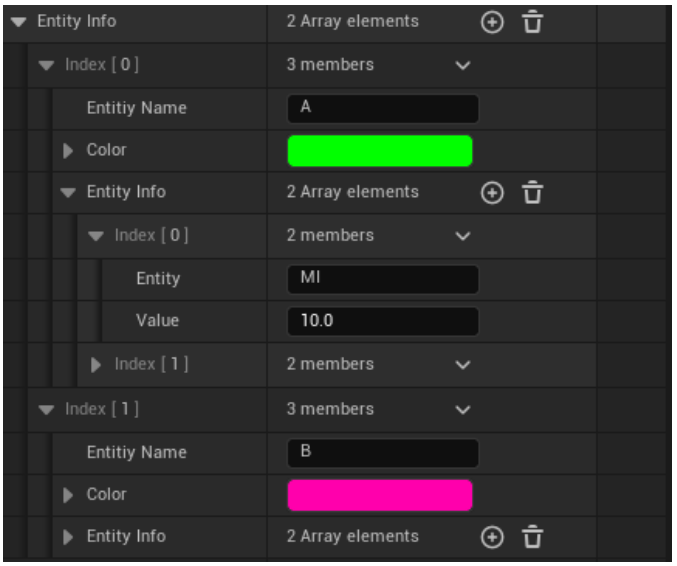
Bar charts visualize categorical data using rectangular bars. This section describes the data models, configuration options, and rendering behaviour for the Bar Chart widget.





2.1.1 Data Structures

Single Data Point



Represents an individual value within a dataset.

- **Entity** *(FString)*
  Label for the data point.
- **Value** *(float)*
  Numerical value of the data point.
  - Clamped to minimum 0.

Dataset / Series

Represents a single dataset.

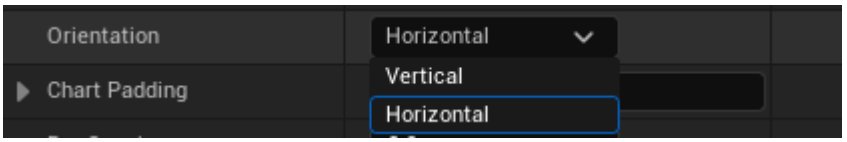
Entity Name *(FString)*
Name of the dataset.

- **Color** *(FLinearColor)*
  Color used to render this dataset's bars.
- **EntityInfo** *(TArray<FBarItem>)*
  Collection of values belonging to this dataset.

2.1.2 Enum: EBarOrientation

Defines chart direction:

- **Vertical** → Bars grow upward
- **Horizontal** → Bars grow sideways



2.1.3 UBarChart Properties

Data Source

- **EntityInfo**
  Main data input.
  - Supports **multi-dataset grouped bar charts**.

Layout & Appearance

- **Orientation**
  Controls whether bars are vertical or horizontal.
- **ChartPadding**
  Space around the chart area (Left, Top, Right, Bottom).
- **BarSpacing**
  Space between individual bars.
- **MaxBarThickness**
  Maximum width (or height for horizontal) of each bar.
- **GroupSpacing**
  Space between groups (categories).
- **InnerSpacing**
  Space between bars within the same group.

Grid & Scaling

- **GridLines**
  Number of grid divisions on the value axis.

Styling Options

Gradient

- **GradientEffect**
  Enables gradient shading on bars.
- **GradientColor**
  Gradient overlay color.

- **GradientOpacity**
  Opacity of the gradient effect.

Bar Shape

- **bRoundedTips**
  Enables rounded ends on bars.
- **CornerRadius**
  Radius of rounded corners.
  - `0` = flat edges
  - `50` = highly rounded depending on the thickness of the bar.
- **CornerSegments**
  Controls smoothness of rounded edges.

Offsets

- **BarStartOffset**
  Offset from the chart origin before bars begin rendering.
- **TextHeightOffset**
  Adjusts label position vertically.
- **TextWidthOffset**
  Adjusts label position horizontally.

2.1.4 Rendering (NativePaint)

virtual int32 NativePaint(...) const override;

This function is responsible for **custom drawing of the chart** using Slate.

Responsibilities:

- Calculates layout based on:
  - Orientation
  - Padding
  - Spacing
- Normalizes values relative to the maximum value
- Draws:
  - Grid lines

    - Bars (grouped or single)
    - Optional gradients
    - Rounded corners (if enabled)
- Positions labels and text

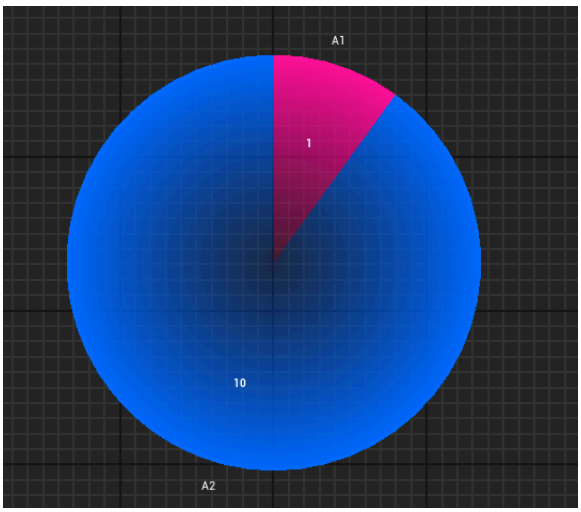
2.1.5 Key Features

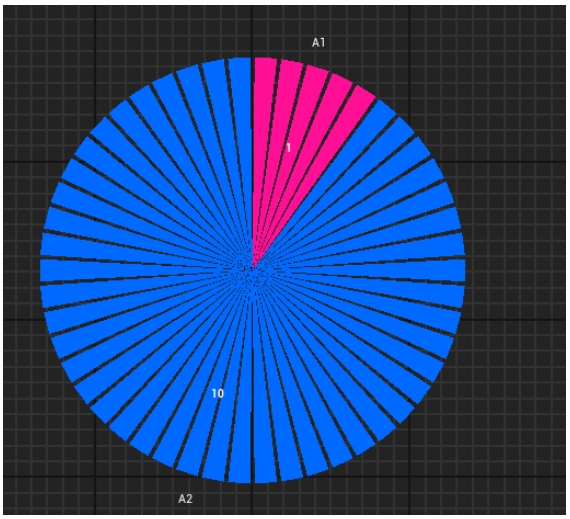
- Supports **multi-dataset grouped bar charts**
- Configurable spacing and layout
- Vertical & horizontal orientations
- Gradient rendering
- Rounded bar styling
- Fully Blueprint editable
- Custom Slate rendering for performance

2.2 Pie Chart

Overview

Pie charts show part-to-whole relationships by dividing a circle into proportional slices. This section covers the pie chart data structures, configurable properties, and rendering behaviour.



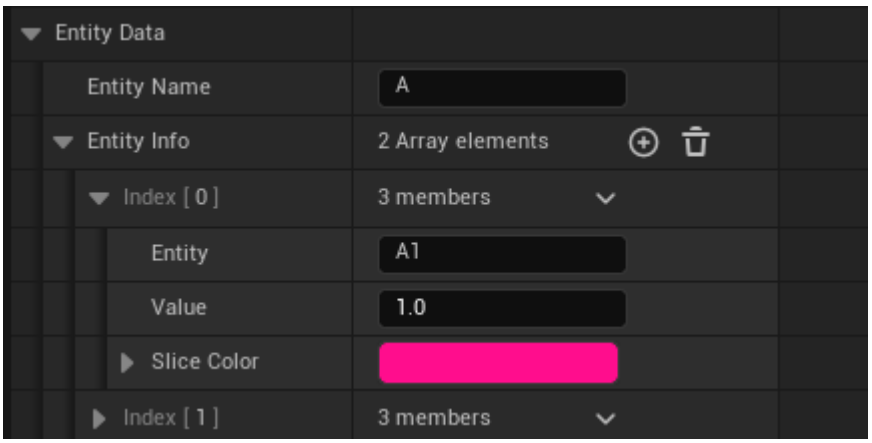


2.2.1 Data Structures

FPie (Single Slice)

Represents an individual slice of the pie chart.

- **Entity** *(FString)*
  Label for the slice (e.g., category name).
- **Value** *(float)*
  Numerical value of the slice.
    - Clamped to minimum `0`.
- **SliceColor** *(FLinearColor)*
  Color used to render this slice.



FInfo (Dataset)

Represents the full dataset for the pie chart.

- **EntityName** *(FString)*
  Name of the dataset (optional descriptive label).
- **EntityInfo**
  Collection of all slices in the chart.

2.2.2 UPieChart Properties

Data Source

- **EntityData**
  Main input data for the chart.
    - Unlike bar charts, pie charts typically use a **single dataset**.

Chart Geometry

- **CircleSmoothness** *(int32)*
  Controls how smooth the circle appears.
  - Higher values = smoother edges
  - Lower values = more polygonal look
- **ChartPadding** *(FMargin)*
  Space around the chart area.

Label Positioning

- **TextPosition** *(float)*
  Controls how far labels are placed from the center.
  - `0` = center
  - 1 = edge of the chart
- **LetterPosition** *(float)*
  Offset used for fine-tuning text placement.

2.2.3 Styling Options

Gradient Effects

- **GradientEffect** *(bool)*
  Enables gradient shading across slices.
- **InnerGradient** *(float)*
  Controls gradient intensity toward the center.
- **OuterGradient** *(float)*
  Controls gradient intensity toward the outer edge.
- **AlphaGradient** *(float)*
  Alpha value applied to the lower gradient portion.

Tip: Set **InnerGradient = OuterGradient** for a solid color look.

Slice Spacing

- **bUseSliceGap** *(bool)*
  Enables spacing between slices.
- **SliceGapAngle** *(float)*
  Gap size between slices (in degrees).

Segmented Mode

- **bSegmented** *(bool)*
  Splits slices into multiple smaller segments for a stylized effect.
- **SegmentGapAngle** *(float)*
  Gap between each segment.
- **TotalSegments** *(int32)*
  Total number of segments used for rendering.

2.2.4 Rendering (NativePaint)

Signature: `virtual int32 NativePaint(...) const override;`

Responsibilities:

- Calculates total value and slice proportions
- Converts values into angular segments
- Draws:
  - Pie slices (or donut segments)
  - Optional gaps between slices
  - Gradient shading
  - Segmented arcs (if enabled)
- Positions and renders labels dynamically

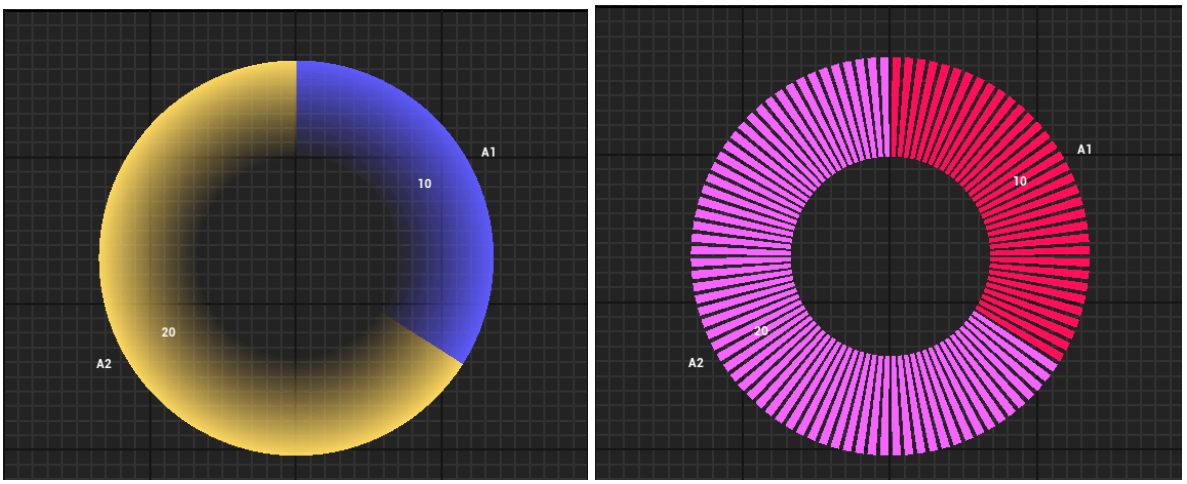
2.2.5 Key Features

- Standard pie chart visualization
- Donut chart support
- Gradient shading control
- Slice gap support
- Segmented rendering mode
- Custom label positioning
- Smooth or low-poly rendering control
- Fully Blueprint editable
- High-performance Slate rendering

2.3 Doughnut Chart

Overview

Doughnut charts are pie charts with a hollow centre, often used to emphasize the ring and support layered styling. This section documents the doughnut chart data model, properties, and rendering behaviour.



2.3.1 Data Structures

FDoughnut (Single Slice)

Represents an individual segment of the doughnut chart.

- **Entity** *(FString)*
  Label for the segment (e.g., category name).
- **Value** *(float)*
  Numerical value of the segment.
  - Clamped to minimum 0.
- **SliceColor** *(FLinearColor)*
  Color used to render this segment.

FDoughnutInfo (Dataset)

Represents the complete dataset for the chart.

- **EntityName** *(FString)*
  Name of the dataset.
- **EntityInfo** *(TArray<FDoughnutSubject>)*
  Collection of all segments.

2.3.2 UDoughnutChart Properties

Data Source

- **EntityData** *(FDoughnutInfo)*
  Main input data for the chart.
  - Typically, a **single dataset** representing proportions.

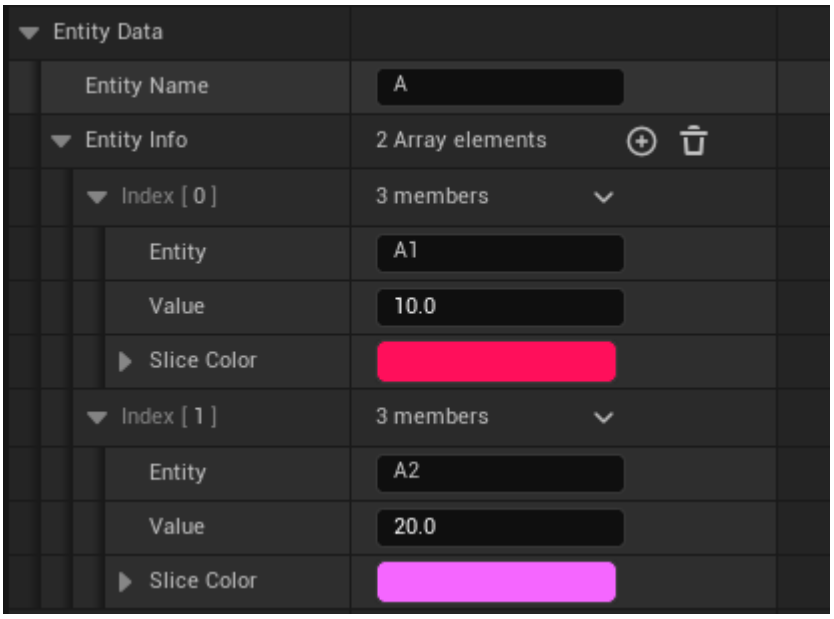


Chart Geometry

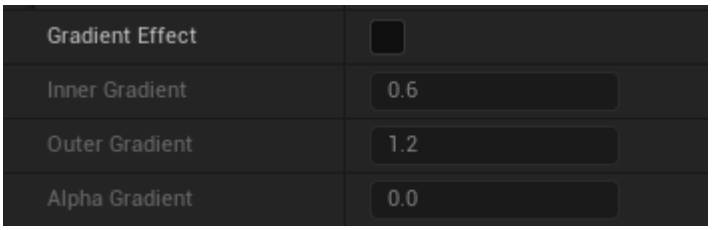
- **InnerRadiusPercent** *(float)*
  Controls the size of the hollow center.
  - `0.0` → behaves like a full pie chart
  - `0.5` → balanced donut
  - `0.8+` → thin ring
- **ChartPadding** *(FMargin)*
  Space around the chart.

2.3.3 Styling Options

Gradient Effects

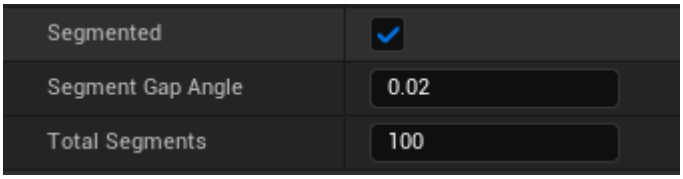
- **GradientEffect** *(bool)*
  Enables gradient shading across segments.
- **InnerGradient** *(float)*
  Gradient intensity toward the inner radius.
- **OuterGradient** *(float)*
  Gradient intensity toward the outer edge.
- **AlphaGradient** *(float)*
  Alpha applied to the lower gradient portion.

Tip: Set **InnerGradient = OuterGradient** for a flat solid color.



Segmented Mode

- **bSegmented** *(bool)*
  Splits the doughnut into multiple smaller arc segments for a stylized or progress-like effect.
- **SegmentGapAngle** *(float)*
  Gap between each segment (in radians).
- **TotalSegments** *(int32)*
  Total number of segments used for rendering.



2.3.4 Rendering (NativePaint)

virtual int32 NativePaint(...) const override;

Responsibilities:

- Calculates total value of all segments
- Converts values into proportional angular spans
- Draws:
  - Doughnut arcs (with inner and outer radius)
  - Gradient shading (if enabled)
  - Segmented arcs (if enabled)
- Applies padding and layout constraints

2.3.5 Key Features

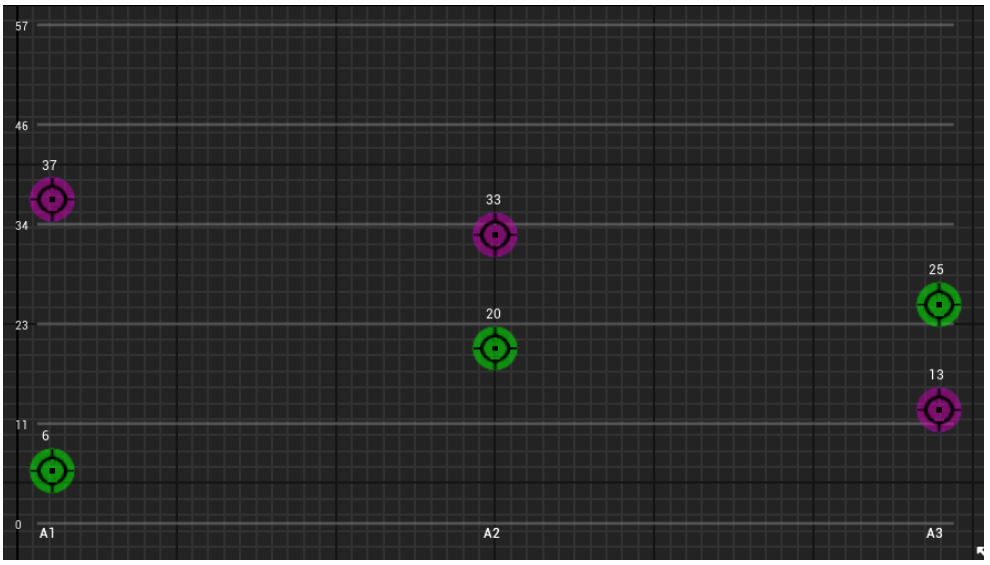
- Doughnut-style circular visualization

- Adjustable inner radius (controls thickness)
- Gradient shading support
- Segmented rendering mode
- Clean and modern UI representation
- Fully Blueprint editable
- High-performance custom Slate rendering

2.4 Point Chart

Overview

Point charts (scatter-style) plot discrete values as markers along an axis, optionally across multiple datasets. This section details the point chart data structures, layout, styling, and rendering behaviour.



2.4.1 Data Structures

FInsideInfo (Single Data Point)

Represents an individual point within a dataset.

- **Entity** *(FString)*
  Label for the data point (e.g., category or X-axis label).
- **Value** *(float)*
  Numerical value of the data point.
  - Clamped to a minimum of `0`.

FOutsideInfo (Dataset / Series)

Represents a dataset containing multiple points.

- **DatasetName** *(FString)*
  Name of the dataset.
- **PointColor** *(FLinearColor)*
  Color used to render all points in this dataset.
- **InsideData** *(TArray<FInsideInfo>)*
  Collection of data points.

2.4.2 UPointChart Properties

Data Source

- **DataPoints** *(TArray<FOutsideInfo>)*
  Main input data.
    - Supports **multiple datasets**, each rendered with a unique color.

2.4.3 Chart Layout

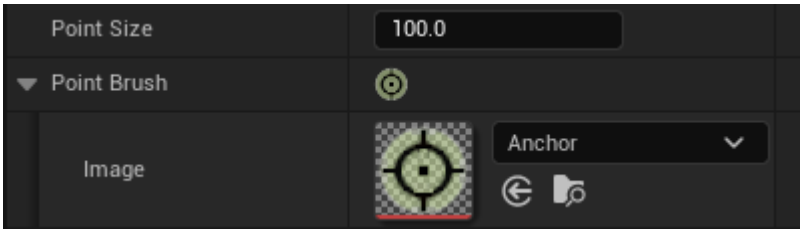
- **ChartPadding** *(FMargin)*
  Space around the chart area.
- **PointPadding** *(float)*
  Horizontal spacing between consecutive data points.

2.4.4 Grid & Scaling

- **GridLines** *(int32)*
  Number of grid divisions along the value axis.
  - Minimum value: 2
- **GetMaxValue()** *(function)*
  Returns the maximum value across all datasets.
  - Used internally for normalizing point positions.

2.4.5 Point Styling

- **PointSize** *(float)*
  Controls the size (radius) of each point.
- **PointBrush** *(FSlateBrush)*
  Custom brush used for rendering points.
  - Allows replacing default shapes with custom visuals.
- **PointColor** *(per dataset)*
  Defined in `FOutsideInfo`.



2.4.6 Label Positioning

- **HeightOffSet** *(float)*
  Vertical offset applied to text labels above each point.
  - Useful for avoiding overlap with the point marker.

2.4.7 Rendering (NativePaint)

virtual int32 NativePaint(...) const override;

Responsibilities:

- Determines chart bounds using padding
- Normalizes values using `GetMaxValue()`
- Calculates positions for each point
- Draws:
    - Grid lines
    - Data points (per dataset)
    - Optional labels above points
- Applies dataset-specific colors and styling

2.4.8 Key Features

- Multi-dataset support
- Scatter / point-based visualization
- Customizable point appearance
- Grid-based layout and scaling
- Label positioning control
- Blueprint editable
- Lightweight and performant rendering

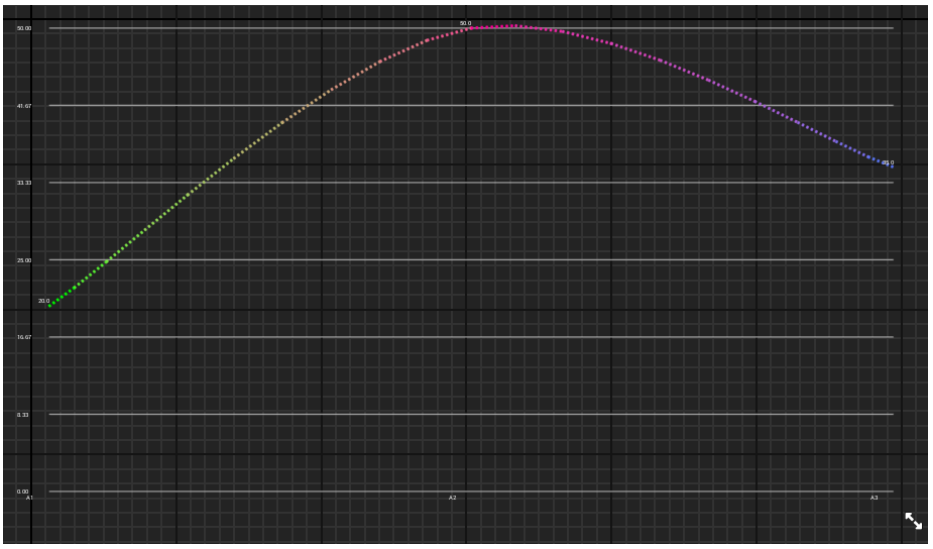
2.5 Line Graph

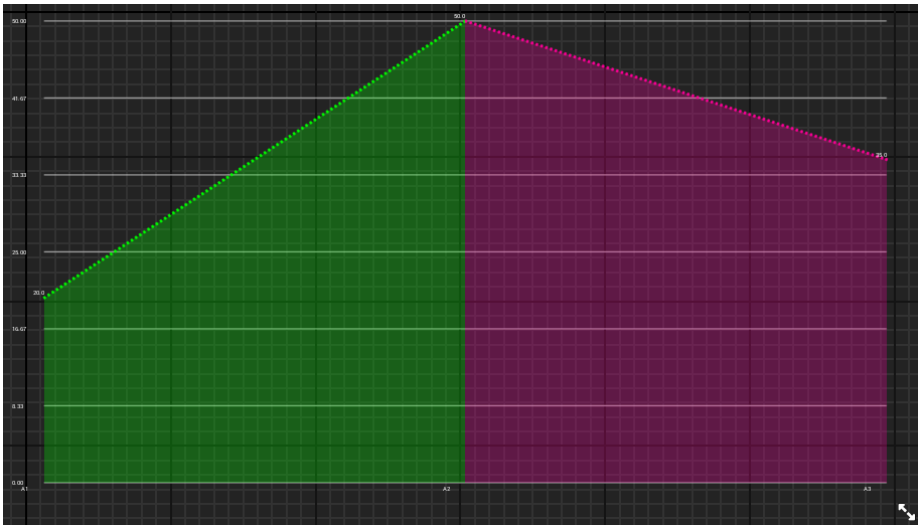
2.5.1 Data Structures

FChartPoint (Single Data Point)

Represents a single point in the line chart.

- **Entity** *(FString)*
  Label for the point (e.g., time, category).
- **Value** *(float)*
  Numerical value of the point.
    - Clamped to a minimum of `0`.
- **Color** *(FLinearColor)*
  Color associated with the point.
    - Used for gradients or interpolated line coloring.

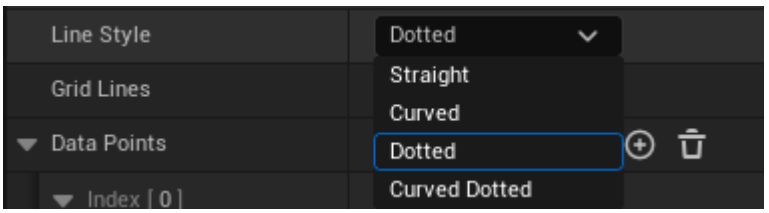




2.5.2 Enum: ELineStyle

Defines how the line is rendered:

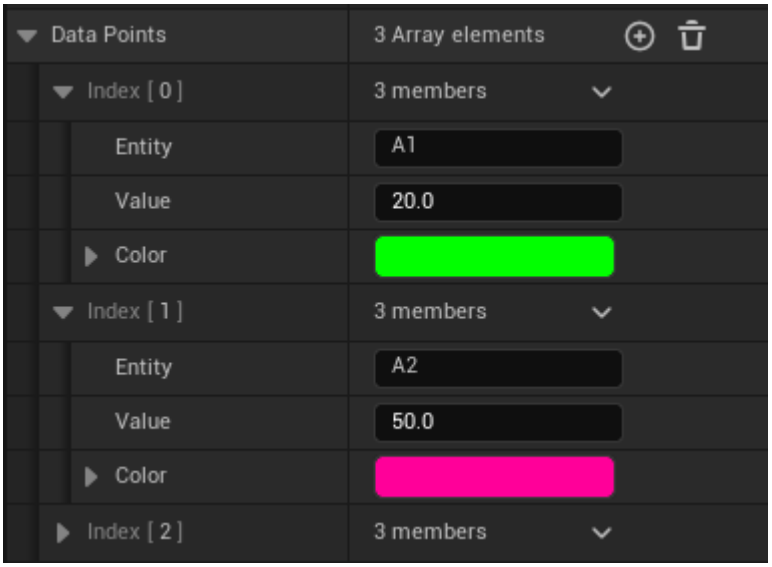
- **Straight** → Linear connections between points
- **Curved** → Smooth interpolated curves
- **Dotted** → Dashed straight lines
- **CurvedDotted** → Dashed curved lines



2.5.3 UW_LineChart Properties

Data Source

- **DataPoints** *(TArray<FChartPoint>)*
  Main dataset for the chart.
    - Points are rendered in sequence.



2.5.4 Chart Layout

- **ChartPadding** *(FMargin)*
  Defines spacing around the chart.

2.5.5 Grid & Scaling

- **GridLines** *(int32)*
  Number of horizontal grid divisions.
    - Minimum value: 2
- **StepsCount** *(int32) (non-UPROPERTY)*
  Number of interpolation steps used for curve smoothing.
    - Higher values → smoother curves

2.5.6 Line Styling

- **LineStyle** *(ELineStyle)*
  Controls overall rendering style of the line.
- **Thickness** *(float)*
  Thickness of the line.

Dotted Line Settings

(Only active when using dotted styles)

- **DashGap** *(float)*
  Space between dashes.
- **DashLength** *(float)*
  Length of each dash segment.

2.5.7 Area Chart Option

- **bEnableAreaChart** *(bool)*
  Fills the area under the line.
    - Useful for emphasizing magnitude or volume.

2.5.8 Rendering (NativePaint)

virtual int32 NativePaint(...) const override;

Handles all custom drawings using Slate.

Responsibilities:

- Calculates chart bounds using padding
- Normalizes values relative to the maximum
- Converts data points into screen positions
- Draws:
    - Grid lines
    - Line paths (straight or curved)
    - Dotted segments (if enabled)
    - Filled area (if enabled)
- Applies thickness and color interpolation

2.5.9 Helper Functions

GenerateCurvePoints

void GenerateCurvePoints(const TArray<FVector2D>& InPoints, TArray<FVector2D>& OutPoints) const;

- Generates smooth interpolated points for curved lines
- Uses `StepsCount` to control smoothness

GetInterpolatedColor

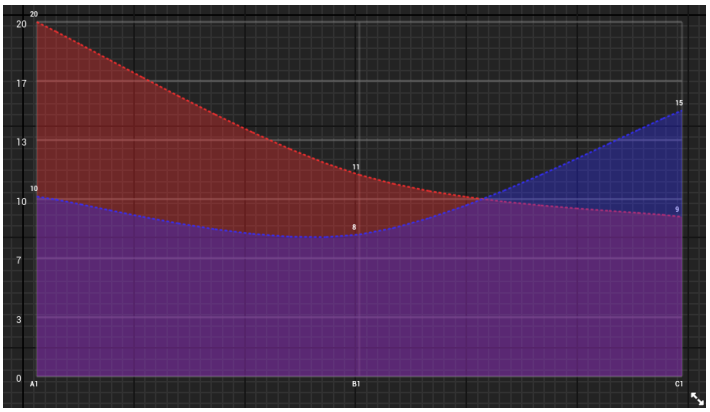
FLinearColor GetInterpolatedColor(float T) const;

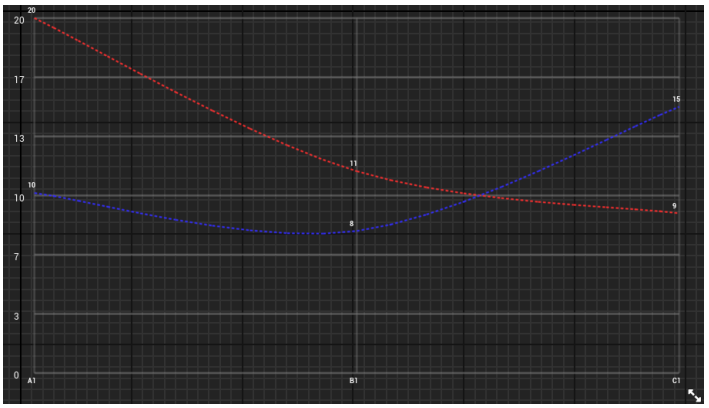
- Returns a color based on interpolation factor T
- Enables gradient transitions along the line

2.5.10 Key Features

- Straight and curved line rendering
- Dotted and dashed line styles
- Area chart (filled graph) support
- Smooth curve interpolation
- Per-point color support with interpolation
- Adjustable thickness and spacing
- Blueprint editable
- High-performance Slate rendering

2.6 MultiData-line Graph





2.6.1 Data Structures

FInsideData (Single Data Point)

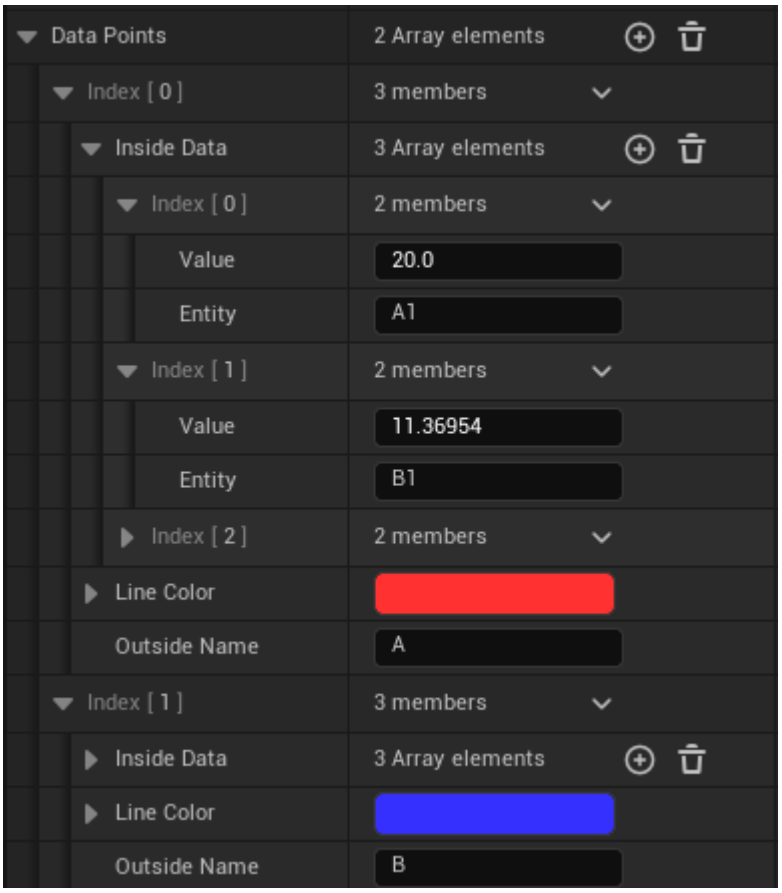
Represents an individual point within a dataset.

- **Value** *(float)*
  Numerical value of the data point.
    - Clamped to a minimum of `0`.
- **Entity** *(FString)*
  Label for the data point (e.g., time or category).

FOutsideData (Dataset / Series)

Represents a complete dataset (line) in the chart.

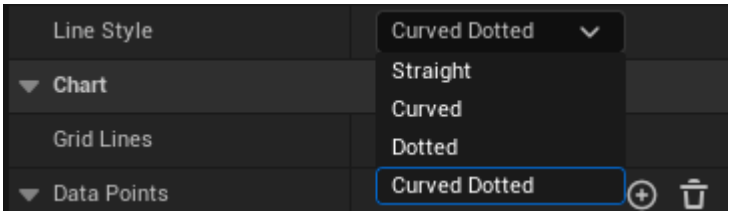
- **InsideData** *(TArray<FInsideData>)*
  Collection of points belonging to this dataset.
- **LineColor** *(FColor)*
  Color used to render the dataset line.
- **OutsideName** *(FString)*
  Name of the dataset (used for legends or identification).



2.6.2 Enum: EMultiLineStyle

Defines how each dataset line is rendered:

- **Straight** → Linear connections between points
- **Curved** → Smooth interpolated curves
- **Dotted** → Dashed straight lines
- **CurvedDotted** → Dashed curved lines



2.6.3 UW_MultidataLineChart Properties

Data Source

- **DataPoints** *(TArray<FOutsideData>)*
  Main input containing multiple datasets.
  - Each dataset is rendered as a separate line.

2.6.4 Chart Layout

- **ChartPadding** *(FMargin)*
  Defines spacing around the chart area.

2.6.5 Grid & Scaling

- **GridLines** *(int32)*
  Number of horizontal grid divisions.
  - Minimum value: 2
- **StepsCount** *(int32) (non-UPROPERTY)*
  Controls curve smoothness.
  - Higher values → smoother curves

2.6.6. Line Styling

- **LineStyle** *(EMultiLineStyle)*
  Defines rendering style for all datasets.

2.6.7. Area Fill

- **bDrawArea** *(bool)*
  Enables filling the area under each line.
  - Useful for highlighting trends
  - Works best with limited datasets to avoid overlap clutter

2.6.8. Rendering (NativePaint)

```
virtual int32 NativePaint(...) const override;
```

Handles all custom drawing using Slate.

Responsibilities:

- Calculates chart bounds and layout
- Finds maximum value across all datasets using `GetMaxValueOutOfAll()`
- Normalizes all points relative to global max value
- Iterates through each dataset:
  - Converts data points into screen positions
  - Generates curve points (if enabled)
  - Draws:
    - Lines (straight/curved/dotted)

        - Optional filled area under lines
- Applies dataset-specific colors

2.6.9. Helper Functions

GenerateCurvePoints

```
void GenerateCurvePoints(const TArray<FVector2D>& InPoints,
TArray<FVector2D>& OutPoints) const;
```

- Generates smooth interpolated points for curved lines
- Controlled by `StepsCount`

GetMaxValueOutOfAll

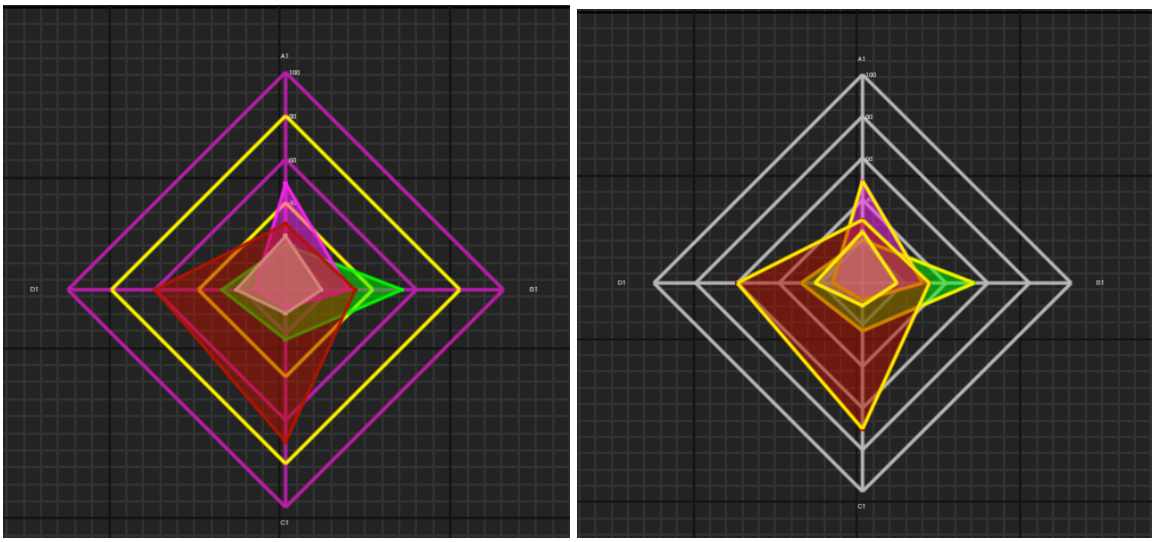
```
float GetMaxValueOutOfAll() const;
```

- Returns the highest value across all datasets
- Ensures consistent scaling across multiple lines

2.6.10. Key Features

- Multi-dataset support (multiple lines)
- Straight and curved line rendering
- Dotted and dashed styles
- Area fill under lines
- Shared axis normalization
- Smooth curve interpolation
- Blueprint callable utility functions
- High-performance Slate rendering

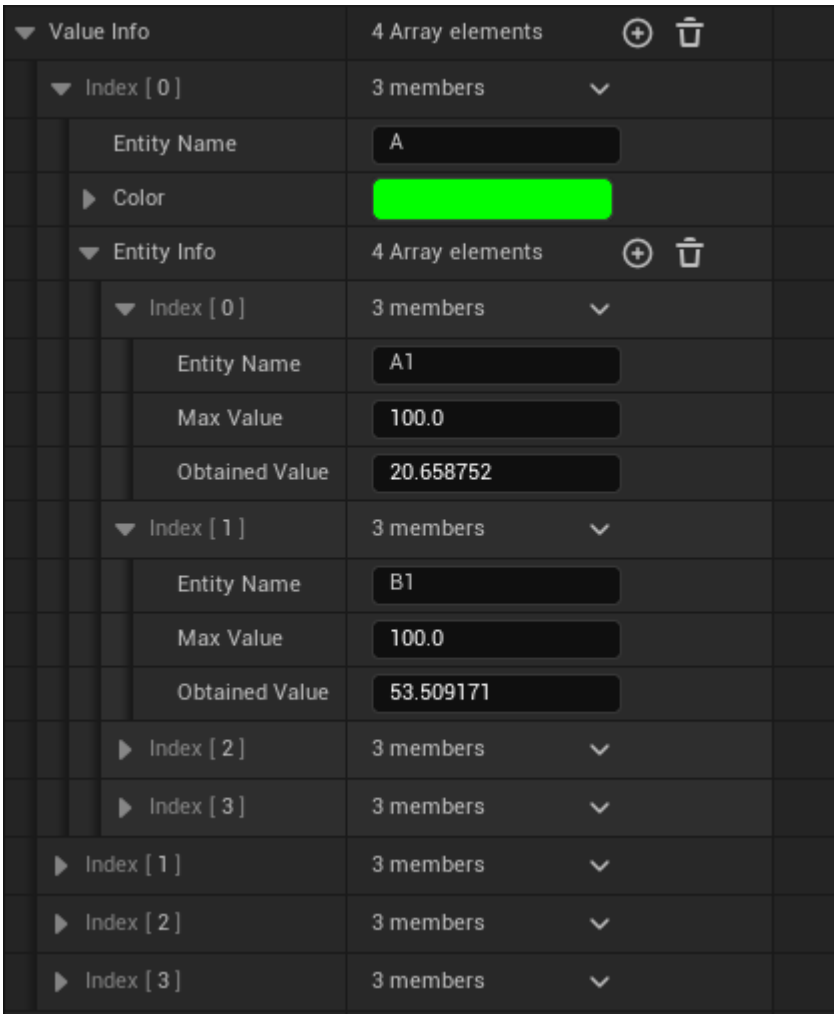
2.7 Radar Graph



2.7.1 Data Structures

Represents a single axis (dimension) in the radar chart.

- **EntityName** *(FString)*
  Name of the axis (e.g., Speed, Strength, Accuracy).
- **MaxValue** *(float)*
  Maximum possible value for this axis.
  - Default: `100`
- **ObtainedValue** *(float)*
  Actual value for this axis.
  - Clamped to a minimum of `0`



Represents a dataset plotted on the radar chart.

- **EntityName** *(FString)*
  Name of the dataset (e.g., player or subject name).
- **Color** *(FLinearColor)*
  Color used for rendering the dataset polygon.
- **EntityInfo**
  Collection of axis values for this dataset.
  - Minimum size: 2 (though typically ≥ 3 for proper radar shape)

2.7.2 UW_RadarChart Properties

Data Source

- **ValueInfo**
  Main input containing one or more datasets.

- Each dataset is rendered as a polygon.

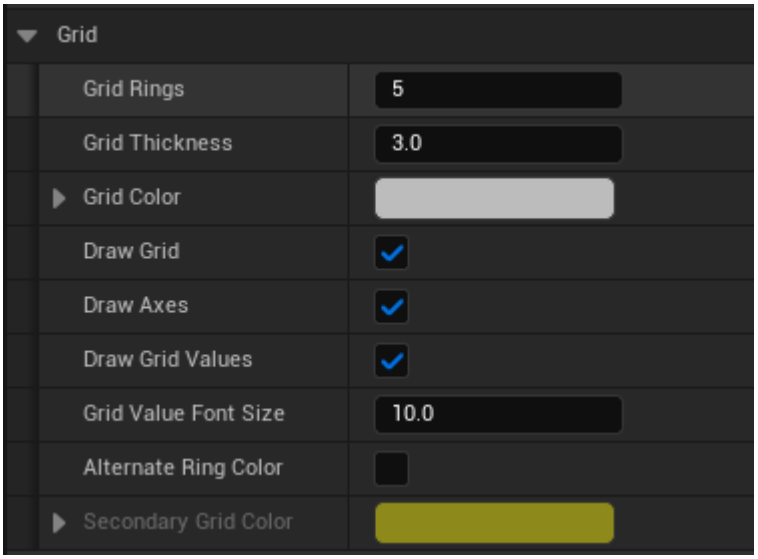
2.7.3 Chart Layout

- **ChartPadding** *(FMargin)*
  Space around the radar chart.
- **BoxSize** *(float)*
  Size of markers or points drawn on each axis.

2.7.4 Grid Configuration

Grid Structure

- **GridRings** *(int32)*
  Number of concentric rings in the chart.
  - Minimum value: 2
- **GridThickness** *(float)*
  Thickness of grid lines.
- **GridColor** *(FLinearColor)*
  Primary grid color.
- **bAlternateRingColor** *(bool)*
  Enables alternating colors between rings.
- **SecondaryGridColor** *(FLinearColor)*
  Secondary color used for alternating rings.



Grid Visibility

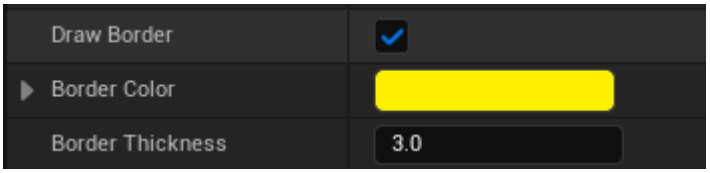
- **bDrawGrid** *(bool)*
  Toggles rendering of grid rings.
- **bDrawAxes** *(bool)*
  Toggles radial axis lines from centre.
- **bDrawGridValues** *(bool)*
  Displays numeric values along grid rings.
- **GridValueFontSize** *(float)*
  Font size for grid value labels.

2.7.5 Dataset Styling

- **FillColorOpacity** *(float)*
  Opacity of the filled polygon area.

Border Settings

- **bDrawBorder** *(bool)*
  Enables outline for dataset polygons.
- **BorderColor** *(FLinearColor)*
  Color of the polygon border.
- **BorderThickness** *(float)*
  Thickness of the border lines.



2.7.6 Rendering (NativePaint)

NativeConstruct

```
virtual void NativeConstruct() override;
```

- Called when the widget is initialized
- Used for setup and initial state handling
- **bIsInitialized** *(bool)*
  Internal flag to track initialization state

NativePaint

```
virtual int32 NativePaint(...) const override;
```

Handles all custom rendering using Slate.

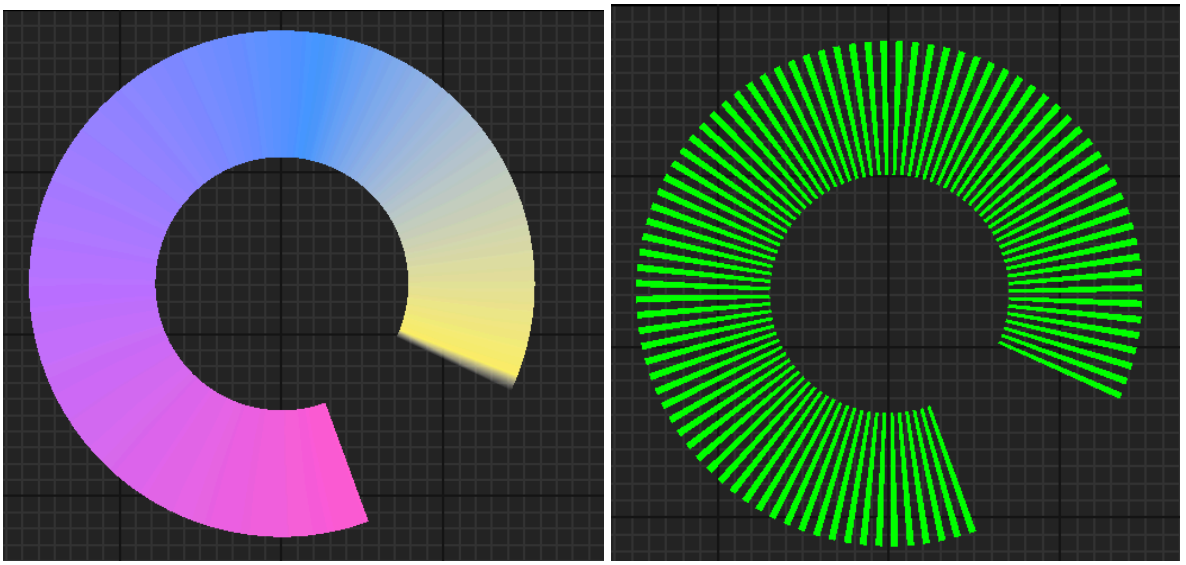
Responsibilities:

- Calculates center and radius based on padding
- Determines number of axes from dataset
- Draws:
    - Grid rings (concentric polygons or circles)
    - Radial axes
    - Axis labels and grid values
- For each dataset:
    - Normalizes values using `MaxValue`
    - Converts values into radial positions
    - Constructs polygon points
    - Draws filled polygon
    - Draws border (if enabled)
    - Renders point markers

2.7.7 Key Features

- Multi-axis data visualization
- Multi-dataset comparison (multiple polygons)
- Customizable grid (rings, axes, colors)
- Optional alternating ring colors
- Filled polygon rendering with opacity control
- Border styling for datasets
- Axis and value labelling
- Blueprint editable
- High-performance Slate rendering

2.8 Radial Progress Graph

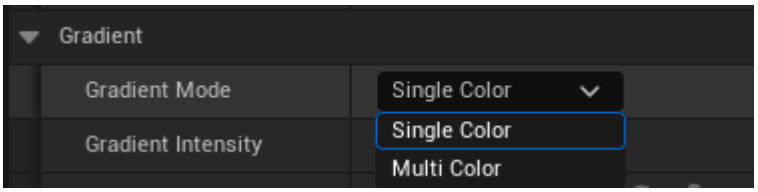


2.8.1 Gradient Mode

EGradientMode

Defines how color is applied:

- **SingleColor** → Uses one color with intensity variation
- **MultiColor** → Uses multiple colors for gradient transitions

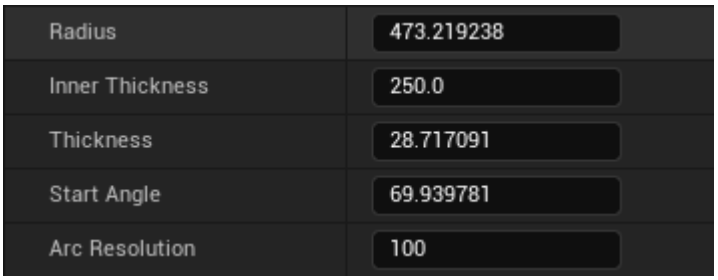


2.8.2 URadialProgressChart Properties

Chart Configuration

- **ChartType** *(EChartType)*
  Determines whether the chart is rendered as a pie or doughnut.
- **Radius** *(float)*
  Controls the overall size of the chart.
- **Thickness** *(float)*
  Thickness of the outer ring.

- **InnerThickness** *(float)*
  Controls inner cutout size (for doughnut charts).
  - Lower value → thicker ring
  - Higher value → thinner ring
- **StartAngle** *(float)*
  Starting angle of the progress arc (in degrees).
  - Default: `-90°` (top of the circle)
- **ArcResolution** *(int32)*
  Controls smoothness of the arc.
  - Higher values → smoother curves



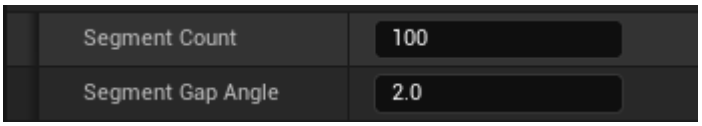
2.8.3 Progress Values

- **Value** *(float)*
  Current progress value.
- **MaxValue** *(float)*
  Maximum possible value.

Progress is calculated as:
`Value / MaxValue`

2.8.4 Segment Settings

- **SegmentCount** *(int32)*
  Number of segments dividing the circle.
- **SegmentGapAngle** *(float)*
  Gap between segments (in degrees).
- **GapAngle** *(float) (non-UPROPERTY)*
  Internal/general gap control for arcs.



2.8.5 Colors & Styling

Base Colors

- **FillColor** *(FLinearColor)*
  Color of the progress portion.
- **BackgroundColor** *(FLinearColor)*
  Color of the unfilled portion.

Gradient Settings

- **GradientMode** *(EGradientMode)*
  Selects gradient behaviour.
- **GradientIntensity** *(float)*
  Controls strength of gradient effect.

Multi-Color Gradient

- **GradientColors** *(TArray<FLinearColor>)*
  Used when `GradientMode = MultiColor`.
  - Defines a color sequence across the arc

2.8.6 Rendering (NativePaint)

```
virtual int32 NativePaint(...) const override;
```

Handles all custom drawing using Slate.

Responsibilities:

- Calculates normalized progress (`Value / MaxValue`)

- Determines total angle span based on progress
- Splits the arc into segments (if enabled)
- Draws:
    - Background arc (full circle)
    - Foreground progress arc
    - Segmented arcs with gaps
- Applies:
    - Gradient shading (single or multi-color)
    - Arc smoothing using `ArcResolution`
- Supports both pie and doughnut rendering modes

2.8.7 Key Features

- Circular progress visualization
- Pie and doughnut modes
- Segmented arc rendering
- Adjustable arc smoothness
- Single-color and multi-color gradients
- Customizable start angle
- High-performance Slate rendering
- Blueprint editable

3. Conclusion

The **Interactive Charts Toolkit** brings powerful and versatile data visualization directly into Unreal Engine. With a wide variety of chart types and extensive customization capabilities, it allows developers to transform raw data into meaningful, visually compelling experiences.

Designed for performance and flexibility, this toolkit is suitable for everything from in-game stats and HUDs to complex dashboards and simulations. It serves as a

complete solution for anyone looking to enhance their project with professional-grade charting.