**ZSDraw**

The **ZSDraw** subsystem is a C++ class library for creating drawings consisting of two-dimensional graphical objects. The individual objects can exchange data with each other in simulation mode and change their states during runtime.

**Inhaltsverzeichnis**

[1. Introduction 3](#_Toc162860286)

[2. Image Size and Object Coordinates 4](#_Toc162860287)

[2.1. Pixel Drawings 4](#_Toc162860288)

[2.1.1. Rectangles 4](#_Toc162860289)

[2.1.2. Lines 5](#_Toc162860290)

[2.2. Metric Drawings 6](#_Toc162860291)

[2.2.1. Y Axis Scale Orientation Top Down 8](#_Toc162860292)

[2.2.1.1. Points 8](#_Toc162860293)

[2.2.1.2. Sizes (Width, Height, dx, dy) 9](#_Toc162860294)

[2.2.1.3. Lines (Points and Sizes) 10](#_Toc162860295)

[2.2.1.4. Rectangles (Points and Sizes) 11](#_Toc162860296)

[2.2.2. Y Axis Scale Orientation Bottom Up 12](#_Toc162860297)

[2.2.2.1. Points 13](#_Toc162860298)

[2.2.2.2. Lines (Points and Sizes) 14](#_Toc162860299)

[2.2.2.3. Rectangles (Points and Sizes) 15](#_Toc162860300)

[3. Coordinate Systems 16](#_Toc162860301)

[3.1. Graphics Items (internally used coordinates) **Fehler! Textmarke nicht definiert.**](#_Toc162860302)

[3.1.1. Convert CGraphObj physical value coordinates into local QGraphicsItem line coordinates. 18](#_Toc162860303)

[3.1.2. Convert local QGraphicsItem line coordinates into CGraphObj physical value coordinates. 20](#_Toc162860304)

[3.1.3. Items positioned without parent group on scene 20](#_Toc162860305)

[3.1.4. Items positioned in a group 22](#_Toc162860306)

[3.2. Transformations 24](#_Toc162860307)

[3.3. Resizing Groups 27](#_Toc162860308)

[3.3.1. Mouse Events 32](#_Toc162860309)

[3.3.1.1. Resizing Bounding Rectangle 32](#_Toc162860310)

[3.3.1.1.1. Calculating transformation values “straight forward” (wrong results) 33](#_Toc162860311)

# Introduction

# Image Size and Object Coordinates

## Pixel Drawings

5

9

0

5

9

Image.width = 10 px

Image.height = 10 px

Rect.topLeft

Rect.topRight

Invisible

Line

Invisible

Line

Rect.bottomRight

Rect.bottomLeft

Rect.width = 10 px

Rect.height = 10 px

Line.p1 = (3, 4)

Line.p2 = (7, 6)

Line.dx = 4

Line.dy = 2

Line.length = 4.47213..

Internally the QGraphics system uses QPointF, QLineF, QRectF and QSizeF to store the coordinates of the graphics items.

### Rectangles

When creating a rectangle with TopLeft = (0.0/0.0) and Width = 10.0 px and Height = 10.0 px, the “right” and “bottom” methods both return 10.0. If the rectangle is drawn onto an image with both width and height set to 10.0 px, the left and bottom border lines will not become visible as shown in the image above.

### Lines

The start and end points of a line are always on whole number multiple pixel positions. If the line is neither a horizontal or vertical line, the length and angle of the line got to be calculated by:

Length = sqrt(dx² + dy²)

Angle = arctan(dy/dx)

Very unlikely the length or the angle of an oblique line will become a whole number multiple as shown in the figure above. This implies that the length or angle of a QLineF object in a pixels drawing (image) cannot be set by the user.

## Metric Drawings

5

10 px

10 px

5

9

Image.Width = 10 mm

Image.Width = 10 px

ScreenResolution = 1.0 px / mm

0.0

9

5.0

10.0 mm

0

0.0

5.0

10.0 mm

Line0.width/px = 9

Line0

Line1

Line2

Line3

Line0.p1/px (0/0)

Line0.p2/px (9/0)

Line0.p1/mm (0/0)

Line0.p2/mm (10/10)

Line0.width/mm = 10.0

If the drawing is setup to use metric dimensions with a width of 10 mm and a height of 10 mm, the lines Line1 and Line2 must become visible. On the screen lines can only be drawn on whole number pixel positions but not between two pixels. To ensure that Line1 and Line2 become visible, one pixel must be truncated from the available pixel range to calculate the position in metric unit. On the screen pixel positions will be rounded to a whole number.

**xScaleRange/mm = Image.Width/mm = 10.0 mm**

**xRange/px = Image.Width/px = ScreenResolution/(mm/px) \* Image.Width/mm = 10 px**

**Transformation from pixel positions into metric positions:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (10.0 mm / 9 px) \* x/px

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xScreenPos/px | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| x/mm | 0.0 | 1.11 | 2.22 | 3.33 | 4.44 | 5.56 | 6.67 | 7.78 | 8.89 | 10.0 |

**Transformation from metric positions into pixel positions:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (9 px / 10.0 mm) \* x/mm

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x/mm | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 |
| x/px | 0.0 | 0.9 | 1.8 | 2.7 | 3.6 | 4.5 | 5.4 | 6.3 | 7.2 | 8.1 | 9.0 |
| xScreenPos/px | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 | 9 |

When using metric units, one pixel will be automatically added to the pixel range of the scene to draw lines at pixel positions.

This complies with the geometrical definition of a line to be infinitely thin and a point, which has no size. But to make a line and a point visible, you need at least one pixel.

When drawing a horizontal line, its “width” is 0 mm but one pixel is needed to make the line visible on the screen.

When drawing a point, its width and height is 0 mm but also one pixel is needed to make the point visible.

### Y Axis Scale Orientation Top Down

Image.Width = 10 mm

Image.Width = 10.0 mm \* 3.5 px/mm + 1 px = 36 px

ScreenResolution = 3.5 px/mm

0

5

10

15

20

25

30

350

0

5

10

15

20

25

30

350

0.0

5.0

10.0 mm

0.0

5.0

10.0 mm

Line0

Line1

Line3

Line2

Line0.width/px = 35

Line0.p1/px (0/0)

Line0.p2/px (35/0)

Line0.p1/mm (0/0)

Line0.p2/mm (10/0)

Line0.width/mm = 10.0

YScaleAxisOrientation = TopDown

Line4

Line4.p1

Line4.p2

Rect1

Rect0

Rect1.top

Rect1.bottom

#### Points

**xScaleRange/mm = Image.Width/mm = 10.0 mm**

**xRange/px = Image.Width/px = ScreenResolution/(mm/px) \* Image.Width/mm + 1 px = 36 px**

**Transformation from pixel positions into metric positions:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (10.0 mm / 35 px) \* x/px

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xScreenPos/px | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| x/mm | 0.0 | 1.43 | 2.86 | 4.29 | 5.71 | 7.14 | 8.57 | 10.0 |

**Transformation from metric positions into pixel positions:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = (35 px / 10.0 mm) \* x/mm

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x/mm | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 |
| x/px | 0.0 | 3.5 | 7.0 | 10.5 | 14.0 | 17.5 | 21.0 | 24.5 | 28.0 | 31.5 | 35.0 |
| xScreenPos/px | 0 | 4 | 7 | 11 | 14 | 17 | 21 | 25 | 28 | 32 | 35 |

#### Sizes (Width, Height, dx, dy)

When calculating sizes (width, height, length, dx, dy) between two points on the screen, the resulting distance is

dx = x2 – x1

dy = y2 – y1

to comply with QLineF and QRectF. You can imagine it as if the start and end points are in the middle of the screen pixel and therefore half a pixel has to be subtracted from each.

**xScaleRange/mm = Image.Width/mm = 10.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 35 px**

**Transformation from pixel sizes into metric sizes:**

dx/mm = (xDistance/mm / xRange/px) \* dx/px = (10.0 mm / 35 px) \* dx/px

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dx/px | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| dx/mm | 0.0 | 1.4 | 2.9 | 4.3 | 5.7 | 7.1 | 8.6 | 10.0 |

**Transformation from metric sizes into pixel sizes:**

dx/px = (xDistance/px / xScaleRange/mm) \* dx/mm = (35 px / 10.0 mm) \* dx/mm

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dx/mm | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 |
| dx/px | 0.0 | 3.5 | 7.0 | 10.5 | 14.0 | 17.5 | 21.0 | 24.5 | 28.0 | 31.5 | 35.0 |

**xScaleRange/mm = Image.Width/mm = 100.0 mm**

**xRange/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 350 px**

**Transformation from pixel sizes into metric sizes:**

dx/mm = (xScaleRange/mm / xDistance /px) \* dx/px = (100.0 mm / 350 px) \* dx/px

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dx/px | 0 | 1 | 10 | 50 | 100 | 150 | 175 | 200 | 250 | 300 | 350 |
| dx/mm | 0.0 | 0.3 | 2.9 | 14.3 | 28.6 | 42.9 | 50.0 | 57.1 | 71.4 | 58.7 | 100.0 |

**Transformation from metric sizes into pixel sizes:**

dx/px = (xDistance /px / xScaleRange/mm) \* dx/mm = (350 px / 100.0 mm) \* dx/mm

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dx/mm | 0.0 | 1.0 | 10.0 | 20.0 | 30.0 | 40.0 | 50.0 | 60.0 | 70.0 | 80.0 | 90.0 | 100.0 |
| dx/px | 0.0 | 4.0 | 35.0 | 70..0 | 105.0 | 140.0 | 175.0 | 210.0 | 245.0 | 280.0 | 315.0 | 350.0 |

#### Lines (Points and Sizes)

**xScaleRange/mm = Image.Width/mm = 10.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 35 px**

**Transformation from pixel points and sizes into metric positions and sizes:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (10.0 mm / 35 px) \* x/px

dx/mm = (xScaleRange/mm / xDistance/px) \* dx/px = (10.0 mm / 35 px) \* dx/px

**Transformation from metric positions and sizes into pixel points and sizes:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (35 px / 10.0 mm) \* x/mm

dx/px = (xDistance/px / xScaleRange/mm) \* dx/mm = (35 px / 10.0 mm) \* dx/mm

**Line 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 0 | 0 | 0.0 | 0.0 |
| P2 | 35 | 0 | 10.0 | 10.0 |
| Center | 18 | 0 | 5.0 | 0.0 |
| distance | 35 | 0 | 10.0 | 0.0 |
| Length | 35 | | 10.0 | |
| Angle | 0.0 ° | | | |

**Line 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 10 | 10 | 2.9 | 2.9 |
| P2 | 25 | 25 | 7.1 | 7.1 |
| Center | 17.5 | 17.5 | 5.0 | 5.0 |
| distance | 15 | 15 | 4.3 | 4.3 |
| Length | 21 | | 6.1 | |
| Angle | 315° | | | |

**xScaleRange/mm = Image.Width/mm = 100.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 350 px**

**Transformation from pixel points and sizes into metric positions and sizes:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (100.0 mm / 350 px) \* x/px

dx/mm = (xScaleRange/mm / xDistance/px) \* dx/px = (100.0 mm / 350 px) \* dx/px

**Transformation from metric positions and sizes into pixel points and sizes:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (350 px / 100.0 mm) \* x/mm

dx/px = (xDistance/px / xScaleRange/mm) \* dx/mm = (350 px / 100.0 mm) \* dx/mm

**Line 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 0 | 0 | 0.0 | 0.0 |
| P2 | 350 | 0 | 100.0 | 100.0 |
| Center | 175 | 0 | 50.0 | 0.0 |
| distance | 350 | 0 | 100.0 | 0.0 |
| Length | 350 | | 100.0 | |
| Angle | 0.0 ° | | | |

**Line 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 100 | 100 | 28.6 | 28.6 |
| P2 | 250 | 250 | 71.4 | 71.4 |
| Center | 175 | 175 | 50.0 | 50.0 |
| distance | 150 | 150 | 42.9 | 42.9 |
| Length | 212 | | 60.5 | |
| Angle | 315° | | | |

#### Rectangles (Points and Sizes)

**xScaleRange/mm = Image.Width/mm = 100.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 350 px**

**Transformation from pixel points and sizes into metric positions and sizes:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (100.0 mm / 350 px) \* x/px

dx/mm = (xScaleRange/mm / xDistance/px) \* dx/px = (100.0 mm / 350 px) \* x/px

**Transformation from metric positions and sizes into pixel points and sizes:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (350 px / 100.0 mm) \* x/mm

dx/px = (xDistance/px / xScaleRange/mm) \* x/mm = (350 px / 100.0 mm) \* x/mm

**Rect 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| TopLeft | 0 | 0 | 0.0 | 0.0 |
| TopRight | 350 | 0 | 100.0 | 0.0 |
| BottomRight | 350 | 350 | 100.0 | 100.0 |
| BottomLeft | 0 | 300 | 0.0 | 100.0 |
| Center | 175 | 175 | 50.0 | 50.0 |
| Width/Height | 350 | 350 | 100.0 | 100.0 |

**Rect 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| TopLeft | 70 | 180 | 20.0 | 51.4 |
| TopRight | 130 | 180 | 37.1 | 51.4 |
| BottomRight | 130 | 320 | 37.1 | 91.4 |
| BottomLeft | 70 | 320 | 20.0 | 91.4 |
| Center | 100 | 250 | 28.6 | 71.4 |
| Width/Height | 60 | 140 | 17.1 | 40.0 |

### Y Axis Scale Orientation Bottom Up

Image.Width = 10 mm

Image.Width = 10.0 mm \* 3.5 px/mm + 1 px = 36 px

ScreenResolution = 3.5 px/mm

0

5

10

15

20

25

30

350

0

5

10

15

20

25

30

350

0.0

5.0

10.0 mm

0.0

5.0

10.0 mm

Line0

Line1

Line3

Line2

Line0.width/px = 35

Line0.p1/px (0/0)

Line0.p2/px (35/0)

Line0.p1/mm (0/10)

Line0.p2/mm (10/10)

Line0.width/mm = 10.0

YScaleAxisOrientation = BottomUp

Line4

Line4.p1

Line4.p2

Rect1

Rect0

Rect1.top

Rect1.bottom

#### Points

**xScaleRange/mm = Image.Width/mm = 10.0 mm**

**xRange/px = Image.Width/px = ScreenResolution/(mm/px) \* Image.Width/mm + 1 px = 36 px**

**Transformation from pixel positions into metric positions:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (10.0 mm / 35 px) \* x/px

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xScreenPos/px | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| x/mm | 0.0 | 1.43 | 2.86 | 4.29 | 5.71 | 7.14 | 8.57 | 10.0 |

y/mm = (yScaleRange/mm / (yRange/px – 1 px)) \* (yMin/px - y/px) = (10.0 mm / 35 px) \* (35 px - y/px)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| yScreenPos/px | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| y/mm | 10.0 | 8.57 | 7.14 | 5.71 | 4.29 | 2.86 | 1.43 | 0.0 |

**Transformation from metric positions into pixel positions:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (35 px / 10.0 mm) \* x/mm

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x/mm | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 |
| x/px | 0.0 | 3.5 | 7.0 | 10.5 | 14.0 | 17.5 | 21.0 | 24.5 | 28.0 | 31.5 | 35.0 |
| xScreenPos/px | 0 | 4 | 7 | 11 | 14 | 17 | 21 | 25 | 28 | 32 | 35 |

y/px = yMin/px - ((yRange/px – 1 px) / yScaleRange/mm) \* y/mm = 35 px - (35 px / 10.0 mm) \* y/mm

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| y/mm | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 |
| y/px | 35.0 | 31.5 | 28.0 | 24.5 | 21.0 | 17.5 | 14.0 | 10.5 | 7.0 | 3.5 | 0.0 |
| yScreenPos/px | 0 | 32 | 28 | 25 | 21 | 18 | 14 | 11 | 7 | 4 | 0 |

#### Lines (Points and Sizes)

**xScaleRange/mm = Image.Width/mm = 10.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 35 px**

**Transformation from pixel points and sizes into metric positions and sizes:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (10.0 mm / 35 px) \* x/px

y/mm = (yScaleRange/mm / (yRange/px – 1 px)) \* (yMin/px - y/px) = (10.0 mm / 35 px) \* (35 px - y/px)

dx/mm = (xScaleRange/mm / xDistance/px) \* dx/px = (10.0 mm / 35 px) \* dx/px

**Transformation from metric positions and sizes into pixel points and sizes:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (35 px / 10.0 mm) \* x/mm

y/px = yMin/px - ((yRange/px – 1 px) / yScaleRange/mm) \* y/mm = 35 px - (35 px / 10.0 mm) \* y/mm

dx/px = (xDistance/px / xScaleRange/mm) \* dx/mm = (35 px / 10.0 mm) \* dx/mm

**Line 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 0 | 0 | 0.0 | 10.0 |
| P2 | 35 | 0 | 10.0 | 10.0 |
| Center | 18 | 0 | 5.0 | 10.0 |
| distance | 35 | 0 | 10.0 | 0.0 |
| Length | 35 | | 10.0 | |
| Angle | 0.0 ° | | | |

**Line 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 10 | 10 | 2.9 | 7.1 |
| P2 | 25 | 25 | 7.1 | 2.9 |
| Center | 17.5 | 17.5 | 5.0 | 5.0 |
| distance | 15 | 15 | 4.3 | 4.3 |
| Length | 21 | | 6.1 | |
| Angle | 315° | | | |

**xScaleRange/mm = Image.Width/mm = 100.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 350 px**

**Transformation from pixel points and sizes into metric positions and sizes:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (100.0 mm / 350 px) \* x/px

y/mm = (yScaleRange/mm / (yRange/px – 1 px)) \* (yMin/px - y/px) = (100.0 mm / 350 px) \* (350 px - y/px)

dx/mm = (xScaleRange/mm / xDistance/px) \* dx/px = (100.0 mm / 350 px) \* dx/px

**Transformation from metric positions and sizes into pixel points and sizes:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (350 px / 100.0 mm) \* x/mm

y/px = yMin/px - ((yRange/px – 1 px) / yScaleRange/mm) \* y/mm = 350 px - (350 px / 100.0 mm) \* y/mm

dx/px = (xDistance/px / xScaleRange/mm) \* dx/mm = (350 px / 100.0 mm) \* dx/mm

**Line 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 0 | 0 | 0.0 | 100.0 |
| P2 | 350 | 0 | 100.0 | 100.0 |
| Center | 175 | 0 | 50.0 | 100.0 |
| distance | 350 | 0 | 100.0 | 0.0 |
| Length | 350 | | 100.0 | |
| Angle | 0.0 ° | | | |

**Line 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| P1 | 100 | 100 | 28.6 | 71.4 |
| P2 | 250 | 250 | 71.4 | 28.6 |
| Center | 175 | 175 | 50.0 | 50.0 |
| distance | 150 | 150 | 42.9 | 42.9 |
| Length | 212 | | 60.5 | |
| Angle | 315° | | | |

#### Rectangles (Points and Sizes)

**xScaleRange/mm = Image.Width/mm = 100.0 mm**

**xDistance/px = Image.Right/px – Image.Left/px = ScreenResolution/(mm/px) \* Image.Width/mm = 350 px**

**Transformation from pixel points and sizes into metric positions and sizes:**

x/mm \* = (xScaleRange/mm / (xRange/px – 1 px)) \* (x/px – xMin/px) = (100.0 mm / 350 px) \* x/px

y/mm = (yScaleRange/mm / (yRange/px – 1 px)) \* (yMin/px - y/px) = (100.0 mm / 350 px) \* (350 px - y/px)

dx/mm = (xScaleRange/mm / xDistance/px) \* dx/px = (100.0 mm / 350 px) \* x/px

**Transformation from metric positions and sizes into pixel points and sizes:**

x/px = xMin/px + ((xRange/px – 1 px) / xScaleRange/mm) \* x/mm = 0 px + (350 px / 100.0 mm) \* x/mm

y/px = yMin/px - ((yRange/px – 1 px) / yScaleRange/mm) \* y/mm = 350 px - (350 px / 100.0 mm) \* y/mm

dx/px = (xDistance/px / xScaleRange/mm) \* x/mm = (350 px / 100.0 mm) \* x/mm

**Rect 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| TopLeft | 0 | 0 | 0.0 | 100.0 |
| TopRight | 350 | 0 | 100.0 | 100.0 |
| BottomRight | 350 | 350 | 100.0 | 0.0 |
| BottomLeft | 0 | 300 | 0.0 | 0.0 |
| Center | 175 | 175 | 50.0 | 50.0 |
| Width/Height | 350 | 350 | 100.0 | 100.0 |

**Rect 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | x/px | y/px | x/mm | y/mm |
| TopLeft | 70 | 180 | 20.0 | 48.6 |
| TopRight | 130 | 180 | 37.1 | 48.6 |
| BottomRight | 130 | 320 | 37.1 | 8.6 |
| BottomLeft | 70 | 320 | 20.0 | 8.6 |
| Center | 100 | 250 | 28.6 | 28.6 |
| Width/Height | 60 | 140 | 17.1 | 40.0 |

# Coordinate Systems

Internally graphics items live in their own local coordinate system. The item’s shape points (and it’s bounding rectangle) are defined relative to the item’s coordinate system with the coordinate origin point at **(0/0)** and are always given in pixels. Their coordinates are usually centered around its center point **(0, 0)**, and this is also the center for all transformations.

All graphics items methods, returning item coordinates, are returning those coordinates in the items local coordinate system. There is only one exception: “pos” which returns the position of the graphics item relative to the center point of its parent item. If the item does not have a parent, the position is returned relative to drawing scenes top left corner.

Example for a pixels drawing whose scenes rectangle has the size of 800\*600 pixel. Eight lines (Line0, .. Line7) are drawn onto the scene. The coordinates are shown for the vertical Line0 which does not have a parent item.

(0/0)(0/0)

Line0.p1 = (0.0/-50.0)

300

400

200

500

600

700

Line0.p2 = (0.0/50.0)

Line0.pos = (300.0/300.0)

100

200

300

400

500

Line0.origin = (0.0/0.0)

100

200

300

400

500

Y-Scale: TopDown

Y-Scale: BottomUp

A user does not expect to enter the values in the local coordinates of the graphics items and also not relative to the origin (center) point of its parent but, depending on the Y-Axis-Scale Orientation, either relative to the top left or bottom left corner of the items parent.

For Line0, for example, the user wants to get and set the following coordinates:

**Line 0 (vertical line)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Local Coordinates | | Physical Values relative to Parent’s TopLeft or BottomLeft Corner | | | | | |
|  | Pixels Drawing  Y-Scale TopDown | | Metrics Drawing  Y-Scale TopDown | | Metrics Drawing  Y-Scale BottomUp | |
|  | x/px | y/px | x/px | y/px | x/mm | y/mm | x/mm | y/mm |
| P1 | 0 | -50 | 300 | 250 | 300 | 250 | 300 | **350** |
| P2 | 0 | 50 | 300 | 300 | 300 | 300 | 300 | **250** |
| Center | 0 | 0 | 300 | 300 | 300 | 300 | 300 | 300 |
| distance | 0 | 100 | 0 | 100 | 0 | 100 | 0 | **-100** |
| GraphicsItem ::pos |  | | (300/300) px | | (300/300) px | | (300/300) px | |
| CGraphObj ::getPos |  | | (300/300) px | | (300/300) mm | | (300/300) mm | |
| Length | 100 | | 100 | | 100 | | 100 | |
| Angle | 270 ° | | 270 ° | | 270 ° | | 270 ° | |

## Convert CGraphObj physical value coordinates into local QGraphicsItem line coordinates.

If the coordinates are set in physical values relative to the parent’s TopLeft or BottomLeft corner (“CGraphObjLine::setline(CPhysVal)”), they have to be converted into the graphics items local coordinates to set the QGraphicsItem object coordinates and position within the parent QGraphicsItem.

(0/0)

Line0.p1 = (0.0/-50.0)

300

400

200

500

600

700

Line0.p2 = (0.0/50.0)

Line0.pos = (300.0/300.0)

100

200

300

400

500

Line0.origin = (0.0/0.0)

100

200

300

400

500

Y-Scale: TopDown

Y-Scale: BottomUp

Line0.p1 = (300.0/250.0)

Line0.p2 = (300.0/350.0)

Line0.p1 = (300.0/350.0)

Line0.p1 = (300.0/250.0)

physValLine(physValPointP1(300/250), physValPointP2(300/350))

CGraphObjLine::setline(physValLine)

* QLineF lineF = **mapFromParent**(physValLine)
  + QLineF lineF = DrawingScene::convert(physValLine, Units.Length.px).toQLineF()
  + QPointF ptPos = lineF.center()
  + QPointF ptP1 = lineF.p1() + ptPos
  + QPointF ptP2 = lineF.p3() + ptPos
  + return QLineF(ptP1, ptP2)
* QGraphicsItem::setline(lineF)
* QLineF lineF = DrawingScene::convert(physValLine, Units.Length.px).toQLineF()
* QPointF ptPos = lineF.center()
* QGraphicsItem::setPos(ptPos)

To avoid calling DrawingScene::convert twice, mapFromParent is not called:

CGraphObjLine::setline(physValLine)

* QLineF lineF = DrawingScene::convert(physValLine, Units.Length.px).toQLineF()
* QPointF ptPos = lineF.center()
* QPointF ptP1 = lineF.p1() + ptPos
* QPointF ptP2 = lineF.p3() + ptPos
* QGraphicsItem::setline(lineF)
* QGraphicsItem::setPos(ptPos)

## Convert local QGraphicsItem line coordinates into CGraphObj physical value coordinates.

If the line coordinates are set in local coordinates (“QGraphicsLineItem::setline(QLineF)”), they must be converted into physical values relative to the parent’s TopLeft or BottomLeft corner to provide the coordinates to the user.

## Items without parent group

**Scene** and **local** coordinates of items (checkmark and small rectangle) if the items don’t belong to a group and are positioned on the scene (unit of drawing in pixels, YScale aligned TopDown). The coordinates provided to the user of the checkmark and rectangle items are the **blue** scene coordinates.

Pixel Drawing, Y Scale Top Down

**x**

**y**

**(0/0)**

**Local Coordinates**

**Checkmark**

**250**

**350**

**Checkmark and Rectangle Coordinates**

**Provided to the User**

**(-50/-50)**

**(50/50)**

**350**

**250**

**(5/5)**

**(-5/-5)**

**0**

**0**

**0**

**0**

**Local Coordinates**

**Rectangle**

**320**

**230**

**270**

**280**

Metrics Drawing, Y Scale Bottom Up

**x**

**y**

**(0/0)**

**300**

**300**

**Checkmark and Rectangle Coordinates**

**Provided to the User**

**400**

**350**

**600**

**250**

**200**

**350**

**250**

**280**

**270**

**325**

**320**

**330**

## Items with parent group

**Scene, group** and **local** coordinates of items (checkmark and small rectangle) if the items belong to a group. The group is positioned on the scene (unit of drawing in pixels, YScale aligned TopDown). The coordinates provided to the user of the checkmark and rectangle items are the **red** group coordinates relative to top left corner of the groups bounding rectangle. The coordinates provided to the user for the group are the **blue** scene coordinates. The local coordinates of the checkmark and rectangle remain the same and are not shown again.

Pixel Drawing, Y Scale Top Down

**x**

**y**

**(0/0)**

**Group Coordinates**

**Provided to the User**

**0**

**0**

**250**

**350**

**350**

**250**

**100**

**100**

**70**

**80**

**20**

**30**

**Checkmark and Rectangle**

**Coordinates**

**Provided to the User**

**Internally used**

**Group Coordinates**

**0**

**0**

**-25**

**-50**

**50**

**50**

**-50**

**25**

!! But internally the origin of the parents (groups) coordinate system is the center of the parents bounding rectangle. When positioning the items within the group for drawing operations on the graphics scene the coordinates are defined in internal group coordinates !!

Metrics Drawing, Y Scale Bottom Up

Assuming 1px = 1mm (X-Scale with orientation Left to Right: x/mm = x/px)

**x/mm = x/px**

**y/mm**

**(0/0)**

**300**

**300**

**Coordinates of group object  
with scene as parent  
in metric unit  
provided to the user**

**400**

**350**

**600**

**250**

**200**

**350**

**250**

**280**

**270**

**325**

**320**

**330**

**Coordinates of objects  
with group as parent  
in metric unit  
provided to the user**

**(0/0)**

**100**

**100**

**50**

**50**

**30**

**20**

**75**

**70**

**80**

**Coordinates of group object  
with scene as parent in pixels**

**300**

**200**

**250**

**0**

**350**

**400**

**100**

**y/px**

**500**

**500**

**100**

**600**

# Transformations

Transformations (scaling, rotating, shearing, moving) are used, if not the shape points are modified directly but for example the bounding rectangle is resized or the object is rotated or sheared.

Transformations are also used if a group is resized, rotated or sheared. The group must apply its geometry change to its children.

The transformation matrix is applied to the original coordinates of the item. This should avoid rounding errors. When rotating an object the rotation angle for example is applied to the original coordinates of the item. When resizing a group the group will apply its scale factor to the children.

## Rectangles

### CPhysValRect compared to QRectF

QRectF cannot be rotated but supports methods to set the top, bottom, left and right edge modifying the size of the rectangle. QRectF does also know nothing about different Y axis scale orientations.

CPhysValRect can be rotated but does not provide methods to set the top, bottom, left and right edge modifying the size of the rectangle. But CPhysValRect is aware of different Y axis scale orientations and supports conversion from pixel to metric unit and vice versa.

When drawing for example buildings, Y axis scale orientation from bottom to top is very likely used. The top of the roof is above the ground line. “top” is greater than “bottom” and “height” is positive. Transforming the building to the display screen using pixel coordinates, the roof will get lower values than the ground line. But the height in pixels is still greater than 0.

By modifying the corner points of the rectangle, width and height may temporarily become lower than 0. Also width and height may explicitly set to values less than 0. But the rectangle in this case will have an invalid size. Before applying the changes to the drawing scene, width and height must be corrected to become valid values greater than 0.

by simply chang topLeft and bottomRight will be exchanged so that width and height become values greater than 0 and the size of the rectangle is valid again. When resizing the rectangle on the screen by moving the selection points, this correction will be done immediately if for example the topLeft corner is moved below one of the bottom corners.

|  |  |  |
| --- | --- | --- |
|  | QRectF | CPhysValRect |
| setSize |  |  |
| setWidth |  |  |
| setHeight | Height can be negative |  |
| setTop | Modifies the height |  |
| setBottom |  |  |
| setLeft |  |  |
| setRight |  |  |
| setCenter |  |  |
| setTopRight |  |  |
| setTopLeft |  |  |
| setBottomRight |  |  |
| setBottomLeft |  |  |
| setAngle | not supported | supported |
|  |  |  |

### Rotation

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

α **=30°**

φ=26.6**°**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

r=111,8

**TR‘**

**BL‘**

Corner and other selection points of a rotated rectangle can be calculated from the center point, the size (width and height) and the rotation angle.

As time consuming trigonometric functions and the mathematical root function must be used to calculate the corner points, the corner points (and other selection points) are only calculated if needed. A dirty flag indicates for each point whether calculation is needed.

To rotate a point around another point by angle α, the distance (r) between the points and the original angle (φ) is needed. For the corners of rectangles (TL, TR, BR, BL) the distance (r) is the same and is calculated as follows:

r = sqrt((width/2)² + (height/2)²) = sqrt(100.0² + 50²) = 111.8

To calculate the original angle (φ) of the corner points, the quadrant of the corner point has to be taken into account. In addition you have to pay attention to that the trigonometric function of the stdlib math functions are counting counter clockwise whereas the graphics item coordinate system counts clockwise. This means that the rotation angle α has to be subtracted from the original angle φ before invoking the trigonometric functions of the stdlib.

Also the Y-Scale axis orientation has to be taken into account when invoking the trigonometric functions to get the original angle φ. The trigonometric functions of the stdlib are using a scale with orientation BottomUp where the y value of the top edge is greater than the y value of the bottom edge. Depending on whether the y value of top edge is greater or less than the y value of the bottom edge the height is positive or negative. Same applies to the width.

**Calculation of φ (Counter Clockwise)**

φ = arctan(abs(height/2) / abs(width/2)) = 26.6°

φTR = φ = 26.6°

φTL = 180° - φ = 153.4°

φBL = 180° + φ = 206.6°

φBR = 360° - φ = 333.4°

**Y Scale Top Down**

TL = 200, 250

BR = 400, 350

Pos = 300, 300

Width = 200, Height = 100

(Rotation angle α clockwise)

TR’.x = pos.x + r \* cos(φTR - α) = 300 + 111.8 \* cos(-3.4°) = 411.4

TR’.y = pos.y - r \* sin(φTR - α) = 300 **-** 111.8 \* sin(-3.4°) = 306.6

TL’.x = pos.x - r \* cos(φTL - α) = 300 + 111.8 \* cos(123.4°) = 238,5

TL’.y = pos.y - r \* sin(φTL - α) = 300 **–** 111.8 \* sin(123.4°) = 206,7

BL’.x = pos.x + r \* cos(φBL - α) = 300 + 111.8 \* cos(176.6°) = 188,4

BL’.y = pos.y + r \* sin(φBL - α) = 300 **–** 111.8 \* sin(176.6°) = 293,4

BR’.x = pos.x + r \* cos(φBR - α) = 300 + 111.8 \* cos(303.4°) = 361,5

BR’.y = pos.y + r \* sin(φBR - α) = 300 - 111.8 \* sin(303.4°) = 393,3

**Y Scale Bottom Up**

TL = 200, 350

BR = 400, 250

Pos = 300, 300

Width = 200, Height = -100

(Rotation angle α clockwise)

TR’.x = pos.x + r \* cos(φTR - α) = 300 + 111.8 \* cos(-3.4°) = 411.4

TR’.y = pos.y - r \* sin(φTR - α) = 300 **+** 111.8 \* sin(-3.4°) = 293.4

TL’.x = pos.x - r \* cos(φTL - α) = 300 + 111.8 \* cos(123.4°) = 238,5

TL’.y = pos.y - r \* sin(φTL - α) = 300 **+** 111.8 \* sin(123.4°) = 393,3

BL’.x = pos.x + r \* cos(φBL - α) = 300 + 111.8 \* cos(176.6°) = 188,4

BL’.y = pos.y + r \* sin(φBL - α) = 300 **+** 111.8 \* sin(176.6°) = 306,6

BR’.x = pos.x + r \* cos(φBR - α) = 300 + 111.8 \* cos(303.4°) = 361,5

BR’.y = pos.y + r \* sin(φBR - α) = 300 + 111.8 \* sin(303.4°) = 206,7

### Resizing after Rotation

#### Order of Transformations Matters

The following figure should clarify the fact, that the order of the transformations is essential for the result. The position of the resulting rectangle is different depending on whether the rectangle is first rotated and then resized or first resized and afterwards rotated.

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

setAngle(30°)

setSize(300, 200)

setSize(300, 200)

setAngle(30°)

!! The order matters !!

**Pos‘**

**Pos‘**

**TL‘**

**TR‘**

**BL‘**

**BR‘**

!! The order matters !!

#### setSize

**Width and height greater than 0**

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

α **=30°**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**r‘‘**

**φ ‘‘**

**β ‘‘**

**w‘‘**

**h‘‘**

* The rotation angle α remains the same.
* The top left corner (TL’) remains unchanged.
* The size (w’’, h’’) is taken over as the new size.
* The new center point got to be calculated based on the distance r’’ from the top left corner TL’ to the center point which is calculated using the new size:  
  r’’ = sqrt((size.width/2)² + (size.height/2)²)
* The angle β’’ of the diagonal lines is also needed and can be calculated using the rotation angle α and the angle **φ’’** of the rectangle diagonals.   
  β’’ = **φ’’ +** α
* Using the radius r’’ and the angle β’’ the distances dx and dy between the top left corner point and the center point can be calculated as follows:  
  pos’’.x = r’’ \* cos(β’’)  
  pos’’.y = r’’ \* cos(β’’)

**Width and height less than 0**

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

α **=30°**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**r‘‘**

**φ ‘‘**

**β ‘‘**

**w‘‘**

**h‘‘**

* The rotation angle α remains the same.
* The top left corner (TL’) remains unchanged.
* The size (w’’, h’’) is taken over as the new size. In the example above width and height are both less than 0. This complies to a rotation of the rectangle around the top left corner by 180°.
* The new center point got to be calculated based on the distance r’’ from the top left corner TL’ to the center point which is calculated using the new size:  
  r’’ = sqrt((size.width/2)² + (size.height/2)²)
* The angle β’’ of the diagonal lines is also needed and can be calculated using the rotation angle α and the angle **φ’’** of the rectangle diagonals.   
  β’’ = **φ’’ +** α
* Using the radius r’’ and the angle β’’ the distances dx and dy between the top left corner point and the center point can be calculated as follows:  
  pos’’.x = r’’ \* cos(β’’)  
  pos’’.y = r’’ \* cos(β’’)
* As the width and height are less than 0, the rectangles size is invalid. To validate the rectangle, the method “adjustToValidSize” has to be called. The method keeps the rotation angle and the position and will just adjust the size to positive values. Afterwards when retrieving the position of the corner points the oppositive corners will have been changed (TL’’ becomes BR’’, TR’’ becomes BL’’).

#### setWidth

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos‘**

α **=30°**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**R‘‘**

**φ ‘‘**

**widthLine**

* The rotation angle α remains the same.
* The left edge (TL’, LC’, BL’) remains unchanged.
* The width is taken over into the size.
* Create the width line from left center to right center selection points.
* Set length of the width line to the given width.
* Get center point of width line and use this as the new center point of the rectangle

Another but more complex possibility would have been:

* The center point is calculated depending on the new size. The left edge of the rectangle remains unchanged and we can use the top left corner of the rectangle to get the new center point. If the center point is right of top left, the cosine of the angle φ’’ is negative. If the center point is left of top left, the cosine of angle φ’’ is positive. So the x value got to be subtracted from the x value of the top left corner.
  + R’’ = sqrt((width/2)² + (height/2)²) = sqrt(150² + 50²) = 158.1
  + φ’’ = arctan(abs(height/2) / abs(width/2)) = 18.4°
  + φTL’’ = 180° - φ = 161,6°
  + Pos’’.x = TL’.x - R’’ \* cos(φTL’’ – α) = 238,5 - 158.1 \* cos(131.6) = 343.3
  + Pos’’.y = TL’.y - R’’ \* sin(φTL’’ – α) = 206.7 + 158.1 \* sin(131.6) = 325.0

#### setWidthByMovingLeftCenter

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

**TL‘**

**Pos‘**

α **=30°**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**φ ‘‘**

**ptPosMoved**

**LC‘‘**

**RC‘**

**w‘‘**

**perpendicularLine**

**widthLine**

Given the position “ptPosMoved” first the resulting width got to be determined. For this the intersection point of the perpendicular line going through “ptPosMoved” to the width line through LeftCenter and RightCenter must be calculated. This intersection point will become the new left center selection point LC’’ of the resized rectangle. Having LC’’ the new width w’’ can be calculated as the length of the line from LC’’ to RC’.

* The rotation angle α remains the same.
* Create the width line from right center to left center selection points.
* Determine the perpendicular line to the width line going through ptPosMoved.
* Determine the intersection point LC’’ of the perpendicular line with the width line.
* Get length of line from LC’’ to RC’ and use this as the new width of the rectangle.
* Get center point of line from LC’’ to RC’ and use this as the new center point of the rectangle.

The resulting center point, when resizing the rectangle by moving the left edge to a new width of 100.0, is the same as setting the width to 300.0 by moving the right edge of the rectangle. But only in this special case.

Another but more complex possibility would have been:

* The center point is calculated depending on the new size. As opposed to the “setWidth” call the right edge remains the same and we can use the right bottom corner of the rectangle to get the new center point. If the center point is right of bottom right, the cosine of angle φ’’ is negative. If the center point is left of bottom right, the cosine of angle φ’’ is positive. So the x value got to be added to the x value of the bottom right corner.
  + R’’ = sqrt((width/2)² + (height/2)²) = sqrt(50² + 50²) = 70.7
  + φ’’ = arctan(abs(height/2) / abs(width/2)) = 45.0°
  + φBR’’ = 360° - φ = 315,0°
  + Pos’’.x = BR’.x - R’’ \* cos(φBR’’ – α) = 361,5 - 70.7 \* cos(285.0) = 343.3
  + Pos’’.y = BR’.y - R’’ \* sin(φBR’’ – α) = 393.3 + 70.7 \* sin(285.0) = 325.0

#### setWidthByMovingRightCenter

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

**TL‘**

**Pos‘**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**ptPosMoved**

**RC‘‘**

**RC‘**

**w‘‘**

**perpendicularLine**

**widthLine**

Given the position “ptPosMoved” first the resulting width got to be determined. For this the intersection point of the perpendicular line going through “ptPosMoved” to the width line through LeftCenter and RightCenter must be calculated. This intersection point will become the new right center selection point RC’’ of the resized rectangle. Having RC’’ the new width w’’ can be calculated as the length of the line from RC’’ to LC’.

* The rotation angle α remains the same.
* Create the width line from right center to left center selection points.
* Determine the perpendicular line to the width line going through ptPosMoved.
* Determine the intersection point RC’’ of the perpendicular line with the width line.
* Get length of line from RC’’ to LC’ and use this as the new width of the rectangle.
* Get center point of line from RC’’ to LC’ and use this as the new center point of the rectangle.

#### setHeight

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos‘**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**heightLine**

* The rotation angle α remains the same.
* The top edge (TL’, TC’, TR’) remains unchanged.
* The height is taken over into the size.
* Create the height line from top center to bottom center selection points.
* Set length of the height line to the given height.
* Get center point of height line and use this as the new center point of the rectangle

#### setHeightByMovingTopCenter

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

**TL‘**

**Pos‘**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**ptPosMoved**

**TC‘‘**

**BC‘**

**h‘‘**

**perpendicularLine**

**heightLine**

Given the position “ptPosMoved” first the resulting height got to be determined. For this the intersection point of the perpendicular line going through “ptPosMoved” to the height line through TopCenter and BottomCenter must be calculated. This intersection point will become the new top center selection point TC’’ of the resized rectangle. Having TC’’ the new height h’’ can be calculated as the length of the line from TC’’ to BC’.

* The rotation angle α remains the same.
* Create the height line from top center to bottom center selection points.
* Determine the perpendicular line to the height line going through ptPosMoved.
* Determine the intersection point TC’’ of the perpendicular line with the height line.
* Get length of line from TC’’ to BC’ and use this as the new height of the rectangle.
* Get center point of line from TC’’ to BC’ and use this as the new center point of the rectangle.

#### setHeightByMovingBottomCenter

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

**TL‘**

**Pos‘**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Pos‘‘**

**ptPosMoved**

**BC‘‘**

**TC‘**

**h‘‘**

**perpendicularLine**

**heightLine**

Given the position “ptPosMoved” first the resulting width got to be determined. For this the intersection point of the perpendicular line going through “ptPosMoved” to the height line through TopCenter and BottomCenter must be calculated. This intersection point will become the new bottom center selection point BC’’ of the resized rectangle. Having BC’’ the new height h’’ can be calculated as the length of the line from BC’’ to TC’.

* The rotation angle α remains the same.
* Create the height line from top center to bottom center selection points.
* Determine the perpendicular line to the height line going through ptPosMoved.
* Determine the intersection point BC’’ of the perpendicular line with the height line.
* Get length of line from BC’’ to TC’ and use this as the new height of the rectangle.
* Get center point of line from BC’’ to TC’ and use this as the new center point of the rectangle.

#### setTopLeft

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Diagonal Line**

**φ ‘‘**

α

**β ‘‘**

**r‘‘**

**Pos‘‘**

* The rotation angle α remains the same.
* The opposite corner BR’ will not be changed.
* Create the diagonal line from the new top left corner TL’’ to the opposite, bottom right corner BR’.
* The new center point Pos’’ is the center point of the diagonal line.
* The distance (r) from the rectangles corner points to the center point is half the length of the diagonal line:  
  r’’ = lineDiagonale.length / 2.0
* The angle of the diagonal line from the center point (φ’’) to the corner points has been changed and must be newly calculated from the angle of the rotated diagonal line β’’ as follows:  
  β’’ = arctan(lineDiagonale.dy / lineDiagonale.dx)  
  φ’’ = β’’ - α
* Having the distance (radius r’’) and the angle φ’’ the width and height of the rectangle can be calculated as follows:  
  width = 2.0 \* r’’ \* cos(φ’’)  
  height = 2.0 \* r’’ \* sin(φ’’)

#### setTopRight

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Diagonal Line**

**φ ‘‘**

α

**β ‘‘**

**r‘‘**

**Pos‘‘**

* The rotation angle α remains the same.
* The opposite corner BR’ will not be changed.
* Create the diagonal line from the new top right corner TR’’ to the opposite, bottom left corner BL’.
* The new center point Pos’’ is the center point of the diagonal line.
* The distance (r) from the rectangles corner points to the center point is half the length of the diagonal line:  
  r’’ = lineDiagonale.length / 2.0
* The angle of the diagonal line from the center point (φ’’) to the corner points has been changed and must be newly calculated from the angle of the rotated diagonal line β’’ as follows:  
  β’’ = arctan(lineDiagonale.dy / lineDiagonale.dx)  
  φ’’ = β’’ + α
* Having the distance (radius r’’) and the angle φ’’ the width and height of the rectangle can be calculated as follows:  
  width = 2.0 \* r’’ \* cos(φ’’)  
  height = 2.0 \* r’’ \* sin(φ’’)

#### setBottomRight

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Diagonal Line**

**φ ‘‘**

α

**β ‘‘**

**r‘‘**

**Pos‘‘**

* The rotation angle α remains the same.
* The opposite corner TL’ will not be changed.
* Create the diagonal line from the new bottom right corner BR’’ to the opposite, top left corner TL’.
* The new center point Pos’’ is the center point of the diagonal line.
* The distance (r) from the rectangles corner points to the center point is half the length of the diagonal line:  
  r’’ = lineDiagonale.length / 2.0
* The angle of the diagonal line from the center point (φ’’) to the corner points has been changed and must be newly calculated from the angle of the rotated diagonal line β’’ as follows:  
  β’’ = arctan(lineDiagonale.dy / lineDiagonale.dx)  
  φ’’ = β’’ - α
* Having the distance (radius r’’) and the angle φ’’ the width and height of the rectangle can be calculated as follows:  
  width = 2.0 \* r’’ \* cos(φ’’)  
  height = 2.0 \* r’’ \* sin(φ’’)

#### setBottomLeft

300

400

200

500

600

700

100

200

300

400

500

Y-Scale: BottomUp

TL

BR

**TL‘**

**Pos**

**BR‘**

100

500

400

300

200

100

Y-Scale: TopDown

TR

BL

**TR‘**

**BL‘**

**TL‘‘**

**TR‘‘**

**BR‘‘**

**BL‘‘**

**Diagonal Line**

**φ ‘‘**

α

**β ‘‘**

**r‘‘**

**Pos‘‘**

* The rotation angle α remains the same.
* The opposite corner TR’ will not be changed.
* Create the diagonal line from the new bottom left corner BL’’ to the opposite, top right corner TR’.
* The new center point Pos’’ is the center point of the diagonal line.
* The distance (r) from the rectangles corner points to the center point is half the length of the diagonal line:  
  r’’ = lineDiagonale.length / 2.0
* The angle of the diagonal line from the center point (φ’’) to the corner points has been changed and must be newly calculated from the angle of the rotated diagonal line β’’ as follows:  
  β’’ = arctan(lineDiagonale.dy / lineDiagonale.dx)  
  φ’’ = β’’ + α
* Having the distance (radius r’’) and the angle φ’’ the width and height of the rectangle can be calculated as follows:  
  width = 2.0 \* r’’ \* cos(φ’’)  
  height = 2.0 \* r’’ \* sin(φ’’)

## Groups

When resizing a group all children of the group should be resized and positioned so that they keep their original relative positions and sizes within the group.

**x**

**y**

**(0/0)**

**0**

**0**

**250**

**350**

**350**

**250**

*100*

*100*

*70*

*80*

*20*

**40**

**Checkmark and Rectangle**

**Coordinates**

**Provided to the User**

**Internally used**

**Group Coordinates**

***0***

*0*

***-25***

*-50*

***50***

*50*

*-50*

*25*

**450**

**450**

**200**

**200**

**Group Coordinates**

**Provided to the User**

**140**

**160**

**60**

**-100**

**0**

**100**

**0**

**100**

**-100**

**-50**

**150**

By moving the bottom right corner of the group from (350/350) to (450/450) the following transformations have been applied to the group which again must be applied by the group to its children:

|  |  |
| --- | --- |
| **Group** | **Apply to Children** |
| Width scaled by 2.0. | Scale width by 2.0.  Move X position by factor 2.0. |
| Height scaled by 2.0. | Scale height by 2.0.  Move Y position by factor 2.0. |

To apply transformations to a graphical object the method “setGroupTransformations” is provided where all transformations may be passed at once as QVariants. If a transformation method should not be applied, an invalid QVariant value is passed.

X‘ = m11\*x + m21\*y + m31

Y‘ = m22\*y + m12\*x + m32

x = (1/m11)\*x’ – (m21/m11)\*y - m31/m11

y = (1/m22)\*y’ – (m12/m22)\*x - m32/m22

m11

m12

m13

m21

m22

m23

m31

dx

m33

m32

dy

ConnectionPoint5

ConnectionPoint6

ConnectionPoint7

ConnectionPoint8

In

Out1

Out2

Out3

CnctLine

ConnectionLine2

ConnectionLine3

ConnectionLine4

ConnectionLine5

Frame

Switch1

ConnectionPoint6

ConnectionPoint7

ConnectionPoint8

ConnectionLine3

ConnectionLine4

ConnectionLine5

Rect1

Group1

ConnectionLine1

ConnectionPoint5

ConnectionLine2

ConnectionPoint1

ConnectionPoint2

ConnectionPoint3

ConnectionPoint4

An item may be moved and rotated within the diagram scene’s coordinate system. Changing the size of an item does not change the scale factors but the real size of the item.

Transformations from the item’s coordinate system to the scene’s coordinate system is processed in two steps – first moving the object by changing it’s scene position and afterwards rotation the object around it’s bounding rectangles center point. To rotate the item three steps are needed – translating the item by the center point, rotating the item by the rotation angle and translating the item again back by the center point.

1. Move (GraphicsItem.setPos)

(0/0)

x

y

scenePos.x

scenePos.y

(0/0)

x

y

ptRotOrigin

1. Rotation

transform.translate( -ptRotOrigin )

(0/0)

x

y

(0/0)

scenePos.x

scenePos.y

transform.rotate( rotAngle\_deg )

(0/0)

x

y

(0/0)

scenePos.x

scenePos.y

transform.translate( ptRotOrigin )

(0/0)

x

y

(0/0)

scenePos.x

scenePos.y

ptRotOrigin

# Mouse Events

## Resizing Bounding Rectangle

The graphics item receives mouse press, mouse move and mouse release events. The mouse position is provided in scene coordinates, relative to the parent object (if any) and in item coordinates.

(0/0)

x

y

(0/0)

Item.

scenePos.x

Item

scenePos.y

mouseEv.scenePos

We use the mouse position in item coordinates to resize the bounding rectangle of the item. The shape points will be adjusted correspondingly within the item’s coordinate system. On pressing the mouse the current size, the rotation point and the shape points are temporarily stored.

As long as the item receives mouse move events the item’s bounding rectangle and shape points will be adjusted relative to the coordinates captured while pressing the mouse. The transformation values (“setPos”, “ptRotOrigin”, “rotAngle\_deg”) will not be changed (but newly applied for each move event).

## Calculating transformation values “straight forward” (wrong results)

**Calculate new Size**

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

mouseEv.pos

1. Calculating Bounding Rectangle and Shape Points on Move Events

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

mouseEv.pos

ptRotOrigin

1. Move (GraphicsItem.setPos)

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

(0/0)

ptRotOrigin

1. Rotation

transform.translate( -ptRotOrigin )

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

(0/0)

ptRotOrigin

transform.rotate( rotAngle\_deg )

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

(0/0)

ptRotOrigin

transform.translate( -ptRotOrigin )

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

mouseEv.scenePos

(0/0)

ptRotOrigin

1. On releasing the mouse the new transformation values will be adjusted.

(0/0)

x

y

Item.

scenePos.x

Item.

scenePos.y

mouseEv.scenePos

(0/0)

ptRotOrigin

The origin point for rotating the item (ptRotOrigin) and the scene position (Item.ScenePos) got to be newly calculated and adjusted. The new rotation origin point is simply the center point of the bounding rectangle of the item (in item coordinates). To get the new scene position:

* 1. the old scene position will be rotated using the previous rotation point as captured on pressing the mouse and
  2. the rotated scene position will then be rotated using the newly calculated rotation point.