Pharo Graphics

The Pillar team

March 28, 2024

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Vector graphics in Athens

There are two different computer graphics: vector and raster graphics. Raster graphics represent images as a collection of pixels. Vector graphics is the use of geometric primitives such as points, lines, curves, or polygons to represent images. These primitives are created using mathematical equations.

Both types of computer graphics have advantages and disadvantages. The advantages of vector graphics over raster are:

- smaller size,
- ability to zoom indefinitely,
- moving, scaling, filling, and rotating do not degrade the quality of an image.

Ultimately, pictures on a computer are displayed on a screen with a specific display dimension. However, while raster graphics doesn't scale very well when the resolution differs too much from the picture resolution, vector graphics are rasterized to fit the display they will appear on. Rasterization is the technique of taking an image described in a vector graphics format and transforming it into a set of pixels for output on a screen.

Morphic is the way to do graphics with Pharo. However, most existing canvases are pixel based, and not vector based. This can be an issue with current IT ecosystems, where the resolution can differ from machine to machine (desktop, tablet, phones, etc...)

Enter Athens, a vector-based graphic API. Under the scene, it can either use balloon Canvas, or the Cairo graphic library for the rasterization phase.

1.1 Example

By the end of this chapter, you will be able to understand the example below: SD: problem with parentheses.

```
| surface |
surface := AthensCairoSurface   extent: 400@400.

surface drawDuring: [ :canvas |

  | paint font |
  paint := (PolymorphSystemSettings pharoLogoForm) asAthensPaintOn:
    canvas.

  canvas setPaint: (
     (LinearGradientPaint from: 0@0  to: 400@400)
       colorRamp: {
        0 -> (Color red alpha: 0.8).
        0.166 -> (Color orange alpha: 0.8).
        0.332 -> (Color yellow alpha: 0.8).
        0.5 -> (Color green alpha: 0.8).
        0.664 -> (Color blue alpha: 0.8).
        0.83 -> (Color magenta alpha: 0.8).
        1 -> (Color purple alpha: 0.8).
     }).

canvas drawShape: (0@0 extent: 400@400).
paint maskOn: canvas.

font := LogicalFont familyName: 'Source Sans Pro' pointSize: 30.
canvas setFont: font.
canvas setPaint:(  (LinearGradientPaint from: 0@0  to: 100@150)
 colorRamp: {
  0 -> (Color white alpha: 0.9).
  1 -> (Color black alpha: 0.9).}).
canvas pathTransform translateX: 20 Y: 180 + (font
    getPreciseAscent); scaleBy: 1.1; rotateByDegrees: 25.

canvas drawString: 'Hello Athens in Pharo'

"canvas draw."
].

surface asForm
```

1.2 Athens details

`AthensSurface` and its subclass `AthensCairoSurface` will initialize a new surface. The surface represents the area in pixels where your drawing will

be rendered. You never draw directly on the surface. Instead, you specify what you want to display on the canvas, and Athens will render it on the area specified by the surface.

The class `AthensCanvas` is the central object used to perform drawing on an `AthensSurface` A canvas is not directly instantiated but used through a call like`surface drawDuring: [:canvas | .... ]`

The Athens drawing model relies on a three-layer model. Any drawing process takes place in three steps:

- First, a painting must be defined, which may be a color, a color gradient, or a bitmap.
- Then a `path` is created, which includes one or more vector primitives, i.e., lines, TrueType fonts, Bézier curves, etc... This path will define the shape that is then rendered.
- Finally, the result is drawn to the Athens surface, which is provided by the back-end for the output.

Paint

Paint can be:

- a single color, defined with the message `color:`
- A radial gradient paint, defined through object `RadialGradientPaint`
- a linear gradient paint, defined through object `LinearGradientPaint`
- a bitmap you can get by sending *asAthensPaintOn:* to a `Form`.

The way the paint is applied in the canvas is specified as *Fill* or *Stroke* that we will see in detail:

- `setPaint:` message will fill the Paint in the area defined by the path.
- `setStrokePaint:` message will set the paint as a virtual pen along the path.

Let's see some examples to better understand how it works.

Stroke paint (a pen that goes around the path)

The stroke operation takes a virtual pen along the path. It allows the source to transfer through the mask in a thin (or thick) line around the path

```
|surface|
surface := AthensCairoSurface extent: 200@200.

surface drawDuring: [ :canvas |
        surface clear: Color white.
```

```
        canvas setStrokePaint:  Color red.
        canvas drawShape: (20@20 extent: 160@160).
].

surface asForm
```

The paint can be customized like

```
|surface|
surface := AthensCairoSurface extent: 200@200.

surface drawDuring: [ :canvas |
        surface clear: Color white.
        canvas setStrokePaint:  Color red.
        canvas paint dashes: #( "fill"5   "gap" 15) offset: 5.
        canvas paint capSquare.
        canvas paint width: 10.
        canvas drawShape: (20@20 extent: 160@160).
].

surface asForm
```

Fill paint (a paint that fill the area defined by the path)

A gradient will let you create a gradient of color, either linear or radial.

The color ramp is a collection of associations with keys - floating point values between 0 and 1 and values with Colors, for example:

```
{0 -> Color blue . 0.5 -> Color white. 1 -> Color red}.
```

Full example with all paints:

Paint bitmap needs to be moved to the place you want to have them drawn. Otherwise, they will stay in the top left corner of the surface area. The message `loadIdentity` is necessary for the next paint as the previous `paintTransform` will continue to apply to the new paint definition.

We can use the same paint definition with stroke paint move. However, in this situation, the paint will only appear in the thickness of the stroke line.

```
|surface|
surface := AthensCairoSurface extent: 200@200.

surface drawDuring: [ :canvas |
surface clear: (Color purple alpha: 0.3).
"linear gradient fill"
    canvas setPaint:  ((LinearGradientPaint from: 0@0  to: 100@100)
     colorRamp: {  0 -> Color white. 1 -> Color black }).
    canvas drawShape: (0@0 extent: 100@100).

"plain color fill"
```

```
    canvas setPaint:  (Color yellow alpha: 0.9).
    canvas drawShape: (100@0 extent: 200@100).

"Bitmap fill"
    canvas setPaint: (PolymorphSystemSettings pharoLogoForm
    asAthensPaintOn: canvas ).
    canvas paintTransform translateX: 0 Y: 135.
    canvas paintTransform scaleBy: 0.25.
    canvas drawShape: (0@100 extent: 100@200).

"Radial gradient fill"
    canvas paintTransform loadIdentity.
    canvas setPaint: ((RadialGradientPaint new) colorRamp: { 0 ->
    Color white. 1 -> Color black }; center: 150@150; radius: 50;
    focalPoint: 180@180).
    canvas drawShape: (100@100 extent: 200@200).

 ].
surface asForm
```

Start and stop points are references to the current shape being drawn. Example: Create a vertical gradient

```
canvas
    setPaint:
        (canvas surface
            createLinearGradient:
                {(0 -> Color blue).
                (0.5 -> Color white).
                (1 -> Color red)}
            start: 0@200
            stop: 0@400).
    canvas drawShape: (0@200 extent: 300@400).
```

Create a horizontal gradient:

```
canvas
    setPaint:
        (canvas surface
            createLinearGradient:
                {(0 -> Color blue).
                (0.5 -> Color white).
                (1 -> Color red)}
            start: 0@200
            stop: 300@200).
    canvas drawShape: (0@200 extent: 300@400).
```

Create a diagonal gradient:

```
canvas
    setPaint:
        (canvas surface
            createLinearGradient:
                {(0 -> Color blue).
                (0.5 -> Color white).
                (1 -> Color red)}
            start: 0@200
            stop: 300@400).
    canvas drawShape: (0@200 extent: 300@400).
```

1.3 Path

Athens always has an active path.

Use `AthensPathBuilder` or `AthensSimplePathBuilder` to build a path They will assemble segments for you

The method `createPath:` exists in all important Athens class: `AthensCanvas`,`AthensSurface`, and `AthensPathBuilder`. The message `createPath: aPath`

Using it returns a new path:

```
surface createPath: [:builder |
  builder
   absolute;
   moveTo: 100@100;
   lineTo: 100@300;
   lineTo: 300@300;
   lineTo: 300@100;
   close ].
```

Here are some helper messages in `AthensSimplePathBuilder`:

- `pathStart`
- `pathBounds` gives the limit of the bounds associated with the path

If you want to build a path using only a straight line, you can use the class `AthensPolygon`.

path builder Messages	Object Segment	comment
ccwArcTo: angle:	AthensCCWArcSegment	counter clock wise segment
cwArcTo:angle:	AthensCWArcSegment	clock wise segment
lineTo:	AthensLineSegment	straight line

moveTo:	AthensMoveSegment	start a new contour
curveVia: to:	AthensQuadSegment	quadric bezier curve
curveVia: and: to:	AthensCubicSegment	Cubic bezier curve
reflectedCurveVia: to:	AthensCubicSegment	Reflected cubic bezier curve
string: font:		specific to cairo
close	AthensCloseSegment	close the current contour

1.4 Coordinate class: Absolute or Relative

Absolute: absolute coordinate from surface coordinate

This will draw a square in a surface whose extent is 400@400 using an absolute move.

```
builder absolute;
   moveTo: 100@100;
   lineTo: 100@300;
   lineTo: 300@300;
   lineTo: 300@100;
   close
```

relative: each new move is relative to the previous one. This will draw a square in a surface whose extent is 400@400 using relative move.

```
 builder relative ;
  moveTo: 100@100;
  lineTo: 200@0;
  lineTo: 0@200;
  lineTo: -200@0;
  close
```

moving

- moveTo: -> move path to a specific point to *initiate* drawing.

straight line

- lineTo: line path from previous point to target point.

arcs

- `ccwArcTo: endPt angle: rot`. Add a counter-clockwise arc segment, starting from current path endpoint and ending at andPt. Angle should be specified in radians.

- `cwArcTo: endPt angle: rot`. Add a clockwise arc segment, starting from current path endpoint and ending at andPt. Angle should be specified in radians.

`cwArcTo:angle:` and `ccwArcTo: angle:` will draw circular arc, connecting previous segment endpoint and current endpoint of given angle, passing in clockwise or counter clockwise direction. The angle must be specified in Radian.

Please remember that the circumference of a circle is equal to 2 Pi R. If R = 1, half of the circumference is equal to PI, which is the value of half a circle.

curves

- curveVia: cp1 and: cp2 to: aPoint. Add a cubic bezier curve starting from the current path endpoint, using control points cp1, cp2 and ending at aPoint.
- curveVia: cp1 to: aPoint. Add a quadric bezier curve, starting from the current path endpoint, using control point cp1, and ending at aPoint.
- reflectedCurveVia: cp2 to: aPoint. Add a reflected cubic bezier curve, starting from the current path endpoint and ending at aPoint. The first control point is calculated as a reflection from the current point, if the last command was also a cubic bezier curve. Otherwise, the first control point is the current point. The second control point is cp2.

curveVia: to: and |curveVia: and: to

This call is related to the bezier curve. A Bézier curve consists of two or more control points, which define the size and shape of the line. The first and last points mark the beginning and end of the path, while the intermediate points define the path's curvature.

More detail on Bezier curve on available at: https://pomax.github.io/bezier-info

Close the path

- close. Close the path contour, between the initial point, and the last reached point.

Here is a full example with all family of paths.

```
| surface |
surface := AthensCairoSurface extent: 100@100.

surface drawDuring: [ :canvas |
 surface clear: Color white.
```

```
   canvas setStrokePaint: Color red.
 canvas paint width: 5.
   canvas drawShape: (
  canvas createPath: [ :builder |
       builder
        absolute;
      moveTo: 25@25;
      lineTo: 50@37.5;
   relative;
      lineTo: 25@(-12.5);
   absolute;
   cwArcTo: 75 @ 75 angle: 90 degreesToRadians;
   curveVia: 50@60 and: 50@90 to: 25@75;
      close
  ]
 ).
].
surface asForm
```

Path transformation

A path can be rotated, translated, and scaled so you can adapt it to your needs. For example, you can define a path in your own coordinate system, and then scale it to match the size of your surface extent.

```
can pathTransform loadIdentity.
can pathTransform translateX: 30 Y: 30.
can pathTransform rotateByDegrees: 30.
can pathTransform scaleBy: 1.2.
```

1.5 Drawing

Either you set the shape first and then you call **draw**, or you call the convenient method **drawShape:** directly with the path to draw as argument

drawing text

```
font := LogicalFont familyName: 'Arial' pointSize: 10.
canvas setFont: font.
canvas setPaint: Color pink.
canvas pathTransform translateX: 20 Y: 20 + (font getPreciseAscent);
    scaleBy: 2; rotateByDegrees: 25.
canvas drawString: 'Hello Athens in Pharo'
```

Drawing using a mask

Athens mask will paint the canvas. It fills the area with the current fill pattern and blends it by whatever alpha is for that pixel on the mask.

Here, the mask the the Pharo

Note RDV: this API is part of AthensCairoPatternPaint*but is not in the standard messages of Athens API. I'm wondering if we should include it.*

```
| surface |
surface := AthensCairoSurface   extent: 400@120.

surface drawDuring: [ :canvas | |paint font|

paint := (PolymorphSystemSettings pharoLogoForm) asAthensPaintOn:
    canvas.

canvas setPaint: (
 (LinearGradientPaint from: 0@0  to: 400@120)
 colorRamp: {
  0 -> (Color red alpha: 0.8).
  1 -> (Color yellow alpha: 0.8).
 })..


canvas drawShape: (0@0 extent: 400@120).
paint maskOn: canvas.
].

surface asForm
```

1.6 Some example

```
"canvas pathTransform loadIdentity.  font1 getPreciseAscent. font
    getPreciseHeight"
   surface clear.
   canvas
    setPaint:
     ((LinearGradientPaint from: 0 @ 0 to: self extent)
      colorRamp:
       {(0 -> Color white).
       (1 -> Color black)}).
   canvas drawShape: (0 @ 0 extent: self extent).
   canvas
    setPaint:
     (canvas surface
      createLinearGradient:
       {(0 -> Color blue).
       (0.5 -> Color white).
```

```
      (1 -> Color red)}
     start: 0@200
     stop: 0@400). "change to 200 to get an horizontal gradient"
  canvas drawShape: (0@200 extent: 300@400).
  canvas setFont: font.
  canvas
   setPaint:
    (canvas surface
     createLinearGradient:
      {(0 -> Color blue).
      (0.5 -> Color white).
      (1 -> Color red)}
     start: 50@0
     stop: (37*5)@0). "number of caracter * 5"
  canvas pathTransform
   translateX: 45 Y: 45 + font getPreciseAscent;
   scaleBy: 2;
   rotateByDegrees: 28.
  canvas
   drawString: 'Hello Athens in Pharo/Morphic !!!!!!!'.
```

```
renderAthens
 surface
  drawDuring: [ :canvas |
   | stroke squarePath circlePath |
   squarePath := canvas
    createPath: [ :builder |
     builder
      absolute;
      moveTo: 100 @ 100;
      lineTo: 100 @ 300;
      lineTo: 300 @ 300;
      lineTo: 300 @ 100;
      close ].
   circlePath := canvas
    createPath: [ :builder |
     builder
      absolute;
      moveTo: 200 @ 100;
      cwArcTo: 200 @ 300 angle: 180 degreesToRadians;
      cwArcTo: 200 @ 100 angle: Float pi ].
   canvas setPaint: Color red.
   canvas drawShape: squarePath.
   stroke := canvas setStrokePaint: Color black.
   stroke
    width: 10;
    joinRound;
    capRound.
   canvas drawShape: squarePath.
   canvas drawShape: circlePath.
```

```
circlePath := canvas
 createPath: [ :builder |
  builder
   relative;
   moveTo: 175 @ 175;
   cwArcTo: 50 @ 50 angle: 180 degreesToRadians;
   cwArcTo: -50 @ -50 angle: Float pi ].
canvas drawShape: circlePath ]
```

1.7 Practicing Athens drawing

To help you practice your Athens drawing, you can use Athens sketch, migrated from SmalltalkHub which is available atAthens Sketch: https://github.com/rvillemeur/AthensSketch

Part I

Bloc & BlElement

CHAPTER 2

introduction

Bloc is a powerful and innovative graphical framework designed specifically for Pharo. Initially developed by the **Pharo** team, it has received valuable contributions from the Feenk team for **GToolkit** integration. These combined efforts are now being merged back into **Pharo**, paving the way for a significant step forward in its graphical capabilities.

2.1 Evolution Beyond Morphic

Bloc is poised to become the primary graphical framework for Pharo, gradually replacing the well-established but aging Morphic framework. This transition promises to bring numerous advantages, including:

- Enhanced performance and efficiency
- Greater flexibility and customization options
- Modernized development experience
- Improved compatibility with various platforms and technologies

To install it in Pharo 11, simply type in the playground

```
EpMonitor disableDuring: [
  Author useAuthor: 'Load' during: [
    [ Metacello new baseline: 'Toplo'; repository:
    'github://pharo-graphics/Toplo:master/src';
        onConflictUseIncoming;
        ignoreImage;
        load.
    ] on: MCMergeOrLoadWarning do: [ :warning | warning load ].
  ].
]
```

Bloc distinguishes itself by prioritizing object composition over inheritance as its core design principle. This means that instead of relying heavily on complex inheritance hierarchies, **Bloc** encourages building user interface components by combining and customizing basic building blocks.

2.2 BlElement: The Foundation of Bloc Components

Every visual element within **Bloc** stems from the fundamental class**BlElement**. This versatile class serves as the foundation upon which you can construct more intricate components. By directly customizing and combining**BlElement** instances, you gain granular control over the appearance and behavior of your UI elements.

2.3 Navigating Bloc's Spatial Landscape: BlUniverse and BlSpace

Bloc introduces two key concepts for managing the visual environment:**BlUniverse** and **BlSpace**. Imagine **BlUniverse** as a container housing a collection of individual **BlSpace** instances. Each **BlSpace** represents a distinct operating system window where your Pharo application unfolds. If you have multiple windows open simultaneously, they'll be neatly organized within the **BlUniverse**, providing a clear overview of your active spaces.

2.4 Ready to Build: Creating Your First Bloc Component

```
BlElement new
geometry: BlRectangleGeometry  new;
size: 200 @ 100;
background: Color blue;
openInNewSpace
```

1. **Start with a blank canvas:** Begin by creating a new BlElement. This serves

as the foundation for your user interface element, initially appearing invisible.

1. **Define its shape:** In Bloc, the element's visual representation is

determined by its geometry. In this example, we'll use a simple rectangle, but more complex shapes are also possible (explored in further detail later).

1. **Set its dimensions and appearance:** Specify the element's size and color

to customize its visual characteristics.

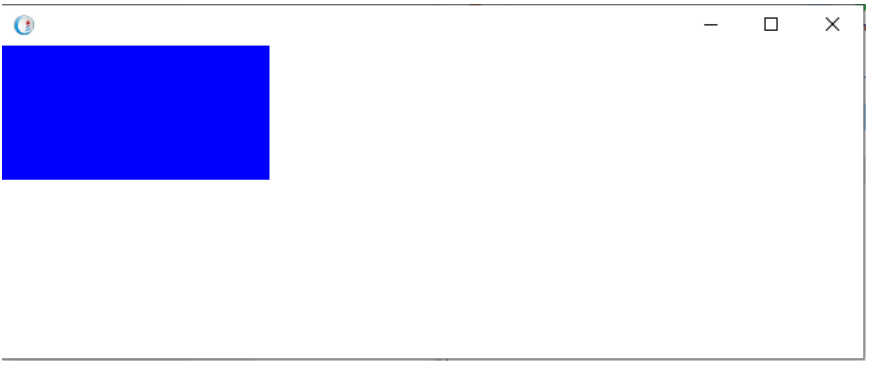

Figure 2-1 basic element

1. **Bring it to life:** Finally, open the element in a new space, making it visible on the screen.

CHAPTER 3

element shape & color

3.1 geometry of BlElement

In Bloc, the visual form and boundaries of your UI elements are determined by their geometries. Each element can only possess a single geometry, essentially acting as a blueprint for its shape and size. You can visualize an element as a specific geometry encapsulated within an invisible rectangular container, representing its overall *bounds*.

Bloc provides a diverse range of pre-defined geometry shapes accessible through`BlElementGeometry allSubclasses`. This comprehensive library empowers you to construct elements of varying complexities, from basic rectangles and circles to more intricate forms.

Bloc excels in facilitating the creation of custom components with advanced layout possibilities. Imagine building complex layouts by strategically arranging various elements, each defined by its unique geometry, to form a cohesive whole.

While Alexandrie canvas provides a foundational set of building drawing primitives, Bloc offers a richer library of pre-defined shapes and the flexibility to construct even more intricate geometries.

- **Annulus** : `BlAnnulusSectorGeometry new startAngle: 225; endAngle: 360; innerRadius: 0.3; outerRadius: 0.9);`
- **bezier** : `BlBezierCurveGeometry controlPoints: { 5@0. 25@80. 75@30. 95@100 }`
- **circle** : `BlCircleGeometry new matchExtent: 100 @ 50`
- **ellipse** : `BlEllipseGeometry new matchExtent: 100 @ 50)`

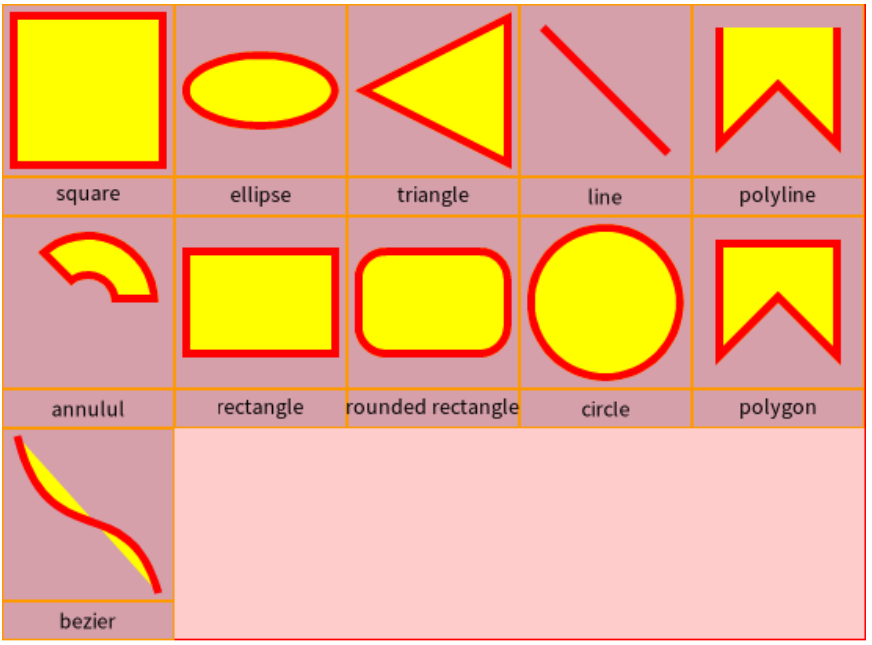

Figure 3-1 base geometry

- **line** : `BlLineGeometry from: 10@10 to: 90@90`
- **Polygon** : `BlPolygonGeometry vertices: {(10 @ 10). (10 @ 90). (50 @ 50). (90 @ 90). (90 @ 10)}`
- **Polyline**: `BlPolylineGeometry vertices: {(10 @ 10). (10 @ 90). (50 @ 50).(90 @ 90). (90 @ 10) }`
- **Rectangle** : `BlRectangleGeometry new`
- **Rounded rectangle** : `BlRoundedRectangleGeometry cornerRadius: 20`
- **square** : `BlSquareGeometry new matchExtent: 70 @ 70`
- **triangle** : `BlTriangleGeometry new matchExtent: 50 @ 100; beLeft`

3.2 element border

The geometry is like a an invisible line on which your border is painted. The painting is a subclass of **BlPaint**, and one of the three:

- solid color

- linear gradient color
- radial gradient color

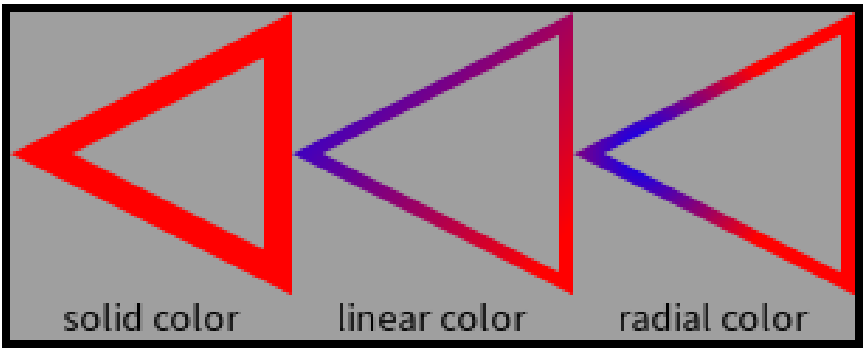

Figure 3-2 border color type

Your border opacity can be specified as well: `opacity: 0.5;`

By default, your border will be a full line, but it can also be dashed, with**dash array** and **dash offset**. Dash array define the number of element, and dash offset, the space between elements.

You also have pre-defined option, available in a single call:

- **dashed**
- **dashed small**

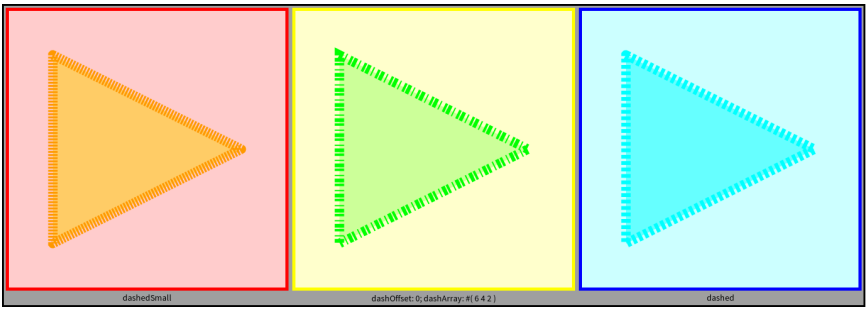

Figure 3-3 border dash

If the path is not closed, The style extent of your border can be defined with

- **cap square**
- **cap round**
- **cap butt**

Last, when the line of your border cross each other, you can define the style of the join:

- **round join**
- **bevel join**
- **mitter join**

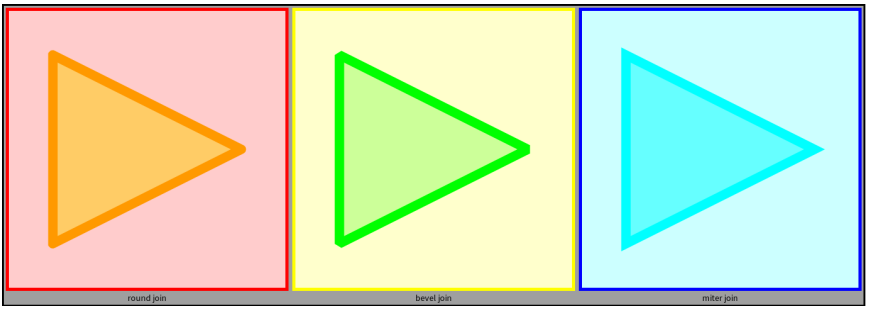

Figure 3-4 border join type

You have two option to define your border:

- short call : `border: (BlBorder paint: Color orange width: 5)`
- with a builder :`BlBorder builder dashed; paint: Color red; width: 3; build`

The first one is very helpfull for solid line definition. The builder let use customize all the detail of your border.

3.3 elements bounds and outskirts

Lets look at the diffent possible bounds of your element.

Layout bounds can be defined explicitly using **size:** method or dynamicaly Layout bounds are considered by layout algorithms to define mutual locations for all considered elements. You'll know more about layout later.

Geometry bounds area is defined by minimum and maximum values of polygon vertices. This does not take in account the border width

Visual bounds is an exact area occupied by an element. it takes strokes and rendering into account.

The geometry is like a an invisible line on which your border is represented. The border drawing can happen outside (adding its border size to the size of your element), centered, or inside the geometry of the element. The final size (geometry + border width) will define the **bounds** of your element.

In this figure, the same exact star is drawned 3 time. The only difference is the outskirts definition between those 3.

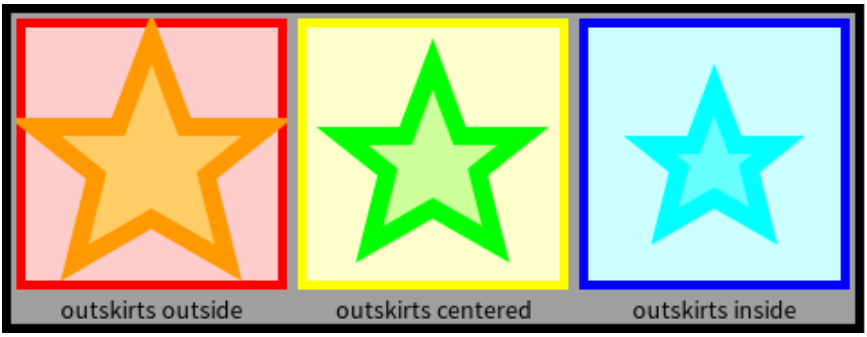

Figure 3-5 outskirts

If we specify BlOutskirts inside, visual bound and geometry bounds will be the same. But if BlOutskirts is outside, then visual bounds are larger than geometry bounds to take border width into its calculation.

3.4 element background

quick set-up: `background: (Color red alpha: 0.8);`

using rgb color

```
background: (Color r: 63 g: 81          b: 181     range: 255);
```

using linear gradient

```
background: ((BlLinearGradientPaint direction: 1 @ 1) from: Color
    red to: Color blue).
```

using radial gradient

```
background: (BlRadialGradientPaint new
stops: { 0 -> Color blue. 1 -> Color red };
center: largeExtent // 2;
radius: largeExtent min;
yourself);
```

Using dedicated *BlPaintBackground* object.

```
background: ((BlPaintBackground paint: fillColor asBlPaint) opacity:
    0.75; yourself);
```

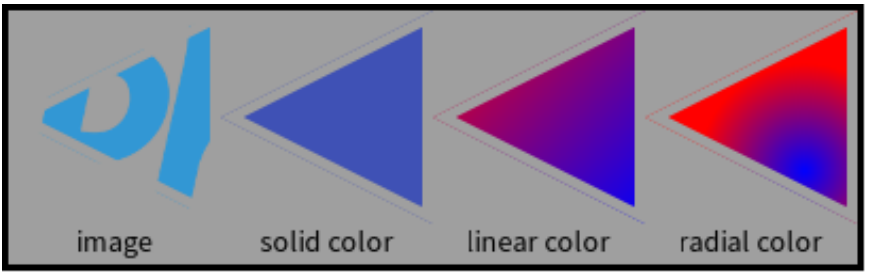

Figure 3-6 background color

3.5 element effect

```
BlElementEffect allSubclasses
```

```
BlElement new
        size: 200 @ 100;
        geometry: (BlRoundedRectangleGeometry cornerRadius: 2);
        background: (Color red alpha: 0.2);
        border: (BlBorder paint: Color yellow width: 1);
        outskirts: BlOutskirts centered;
        effect:
            (BlSimpleShadowEffect color: Color orange offset: -10 @
    -20)
```

Figure 3-7 simple shadow

effect: (BlSimpleShadowEffect color: (Color orange alpha: shadowAlpha) off-

set: shadowOffset);

```
BlElement new
        size: 300 @ 150;
        geometry: (BlRoundedRectangleGeometry cornerRadius: 2);
        background: (Color blue alpha: 0.5);
        border: (BlBorder paint: Color red width: 10);
        effect: (BlGaussianShadowEffect color: Color yellow offset:
    10@20 width: 5)
```

Figure 3-8 gaussian shadow

3.6 element opacity

`element opacity:`, value between 0 and 1, 0 meaning completely transparent You can apply opacity to background, border, or to your hole element.

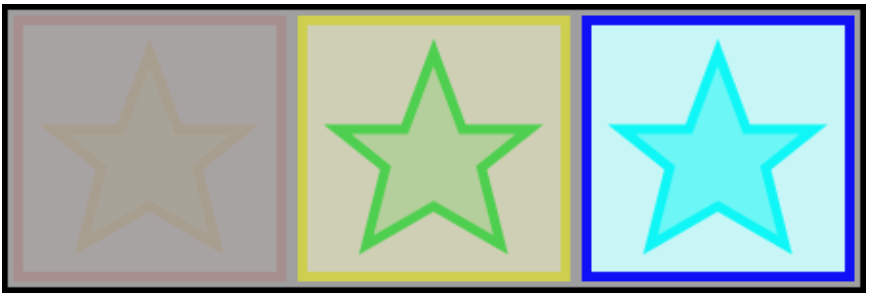

Figure 3-9 element opacity

CHAPTER 4

element transformation

You can apply transformation to a BlElement:

- rotation
- translation
- Scaling
- reflection
- etc...

transformDo: [:b | b scaleBy: 0.2; translateBy: -25 @ -15];

```
aContainer := BlElement new
                    layout: BlFrameLayout new;
                    constraintsDo: [ :c |
                        c horizontal fitContent.
                        c vertical fitContent ];
                    padding: (BlInsets all: 20);
                    background: (Color gray alpha: 0.2).

node := BlElement new
            geometry: (BlRoundedRectangleGeometry cornerRadius: 4);
            border: (BlBorder paint: Color black width: 2);
            background: Color white;
            constraintsDo: [ :c |
                c frame horizontal alignCenter.
                c frame vertical alignBottom ];
            size: 20 @ 20.

aContainer transformDo: [ :t |
    t
        scaleBy: 2.0;
```

```
        rotateBy: 69;
        translateBy: 50 @ 50 ].
aContainer addChild: node.

aContainer forceLayout.
```

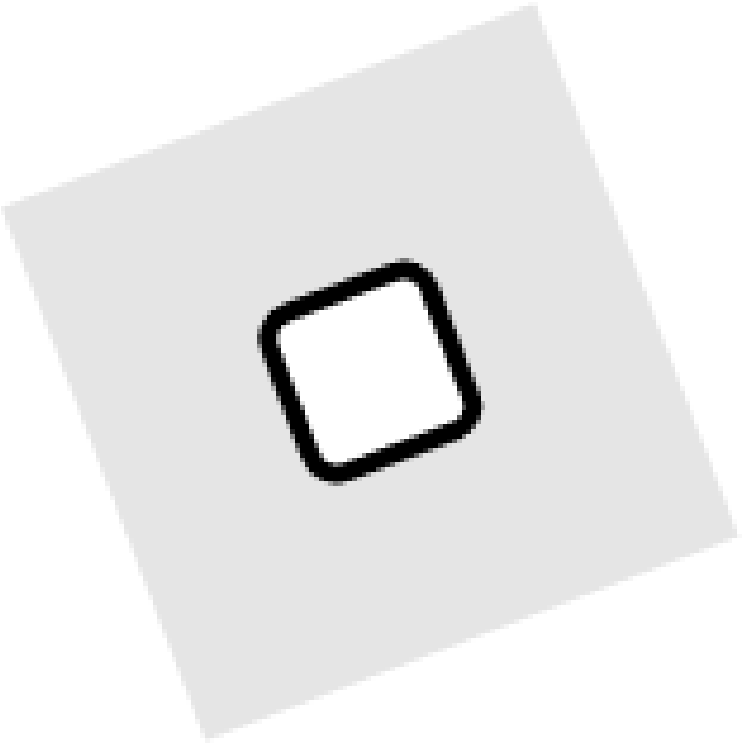

Figure 4-1 transform example

A few hint on using transormDo:

- transformDo: can be applied at any moment during the life of an object.
- you can use any static or pre-compute properties with transformDo:
- if you want to use dynamic layout properties (like size) with *transformDo:*, you need to wait for layout phase to be completed.

CHAPTER 5

Bloc styles

CHAPTER 6

element custom Painting

Bloc really favor BlElement composition to create your interface. Most of the time, you will not have to create a custom painting of your element widget. You can already do a lot with existing geometry.

Ultimately, you can define drawing methods on a canvas, but once drawn, a canvas cannot be easily inspected for its elements. However, Bloc element composition create a tree of elements, that can be inspected, and shaped dynamically.

creating and drawing your own block => subclass BlElement => Custom drawing is done with drawOnSpartaCanvas: method. =>

BlElement » aeFullDrawOn: aCanvas "Main entry point to draw myself and my children on an Alexandrie canvas."

self aeDrawInSameLayerOn: aCanvas.

self aeCompositionLayersSortedByElevationDo: [:each | each paintOn: aCanvas].

Element geometry is taken care by: BlElement » aeDrawGeometryOn: aeCanvas Painting is done on an Alexandrie Canvas, then rendered on the host: BARenderer (BlHostRenderer) » render: aHostSpace, display on a AeCairoImageSurface

Drawing is done through method 'drawOnSpartaCanvas', which receive a sparta (vector) canvas as an argument.

1. aeDrawChildrenOn:
2. aeDrawOn:
3. aeDrawGeometryOn:

CHAPTER 7

UI Building

<https://github.com/OpenSmock/Pyramid/tree/main>

Bibliography