Chapter 1

Overview of Roassal

Roassal is build on a very simple model to structure visualizations. Roassal is built on the notion of objects, elements, shapes, interaction, and view.

An *element* is a wrapper to an external provided domain object (e.g., a number, a file, your object). To an element, you can add a *shape* (composed or not), and interaction. Elements are added in a *view*. This is necessary to see things on screen. Making elements activable from user actions, it is necessary to specificy some *interaction* on elements.

This chapter gives an extended overview of the Roassal API. It covers the essential concepts of Roassal, such as the view, elements, shapes, and interactions.

Builders are at a higher level. Builders offer API to build more sophisticated elements, shapes, interactions and layouts. But it all boils down into shapes and interaction added to elements, and elements added to a view. The builder infrastructure is described in the second part of the book.

1.1 View, Elements, Shapes and Interactions

Roassal structures a visualization in terms of view, elements, shapes, interactions, and animations. A *view* is a container of graphical elements. Elements can be added and removed from a view.

An *element* is a graphical representation of an arbitrary object. An end-user sees elements and interacts with them using the mouse and keyboard. An element contains *shapes* that define its visual representation. A shape describes a primitive visual representation such as a box, a circle, a line or a textual label. Shapes can be combined to form more elaborate shapes.

More specifically, a Roassal element is a compound object that contains

(i) a two dimensional spatial location; (ii) a set of interactions; (iii) a combination of shapes; (iv) a model, which is any arbitrary object that belongs to the domain model provided by the user. To be visible, elements must have at least one shape. One key aspect of Roassal is that shapes may hold metrics or numerical values that define their visual dimensions (*e.g.,* height, width, color, border width). Instead of giving absolute numerical values to define the visual aspect of the graphical element (as with most visualization frameworks), functions that calculate the metrics may also be provided.

An *interaction* is a particular action the user may trigger. The range of supported interactions is broad: dragging elements, popping up an element on hover, highlighting other elements, getting a menu by right-clicking on an element.

The following example renders a tree, where each node is a number (Figure 1.2):

```
v := RTView new.
shape := RTBox new color: (Color blue alpha: 0.3).
elements := shape elementsOn: (1 to: 50).
v addAll: elements.
elements @ RTPopup.

RTEdgeBuilder new
   view: v;
   objects: (1 to: 50) from: [ :i | i // 3 ].

RTTreeLayout on: elements.
v
```

The first assignment to the variable `v` creates a new view that is empty. A colored box shape is then created. The expression `Color blue alpha: 0.3` specifies its color to be the blue color with 30% of transparency. The shape is then used as a factory of elements, creating a shape for each element of a collection of numbers (a range from 1 to 50). The expression `@ RTPopup` makes each element react when the mouse hovers over it: a popup is displayed that shows the model of the element (*i.e.* the number).

The relations between the elements are then expressed using the class `RTEdgeBuilder`. An instance of `RTEdgeBuilder` is created and then parametrized with the view. This is done in order for it to be able to retrieve the elements between which the edges will be drawn. Then the code specifies that for each number *i* between 1 and 50, an edge is drawn between *i* and *i // 3* (the remainder of the division *i / 3*).

The following code is a a slighly more elaborated example that extracts data from a Tab Separated Values (TSV) table (Figure 1.2):

```
"Processing a small TSV table"
tab := RTTabTable new input:
```

Figure 1.1: Linking numbers.

```
'id value1  value2  parent
1   10  20  1
2   5   12  1
3   8   17  1
4   9   13  3'.
tab removeFirstRow.
tab convertColumnsAsInteger: (1 to: 4).

v := RTView new.
shape := RTBox new
            width: [ :entry | entry second ];
            height: [ :entry | entry third ].
elements := shape elementsOn: tab values.
v addAll: elements.

elements @ (RTPopup new text: [ :entry |
   'id = ', entry first asString, String cr,
   'value1 = ', entry second asString, String cr,
   'value2 = ', entry third asString, String cr ]).

elements @ (RTLabelled new text: [ :entry | entry first ]).

RTEdgeBuilder new
   view: v;
   objects: tab values from: [ :entry | tab values at: entry fourth ].

RTHorizontalTreeLayout new
   verticalGap: 30; on: elements.
v
```

We extract and visualize data from a TSV table as typical spreadsheet applications are able to generate such documents. In this example, we assume that each row contains a numerical identifer, two values, and the identifer of a parent table entry.

The class RTTabTable converts the textual description of the table (*i.e.,* the

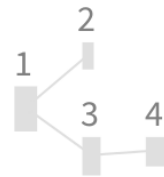

Figure 1.2: Linking TSV elements.

raw file content) into actual values that are usable by Roassal. The first row of the table is removed since it simply names the columns. The four columns are then converted into integer numerical values.

Since each table entry contains two values, we define a graphical box of wich the width and the height are directly mapped to the first and second value, respectively. A popup is then defined that gives more details about the table entry. This is followed by labeling each element with the first numerical entry value, the identifier.

Lastly, edges are built, linking each entry to its parent, and all elements are laid out as a horizontal tree. Note that we assume here that (i) entries are ordered in the table according to their identifier and (ii) the parent of the first entry is the entry itself.

1.2 Shapes

In Roassal, a multiple of primitive shapes are offered, and these are classified in the two families of *elements* and *edges*.

Element shapes

Six element shapes are meant to cover typical usage:

- RTBox: a rectangular box, *e.g.,* RTBox new width: 10; height: 20
- RTEllipse: an ellipse, *e.g.,* RTEllipse new width: 10; height: 20
- RTLabel: a textual label, *e.g.,* RTLabel new text: 'hello world'. The text may be multi-lined.
- RTPolygon: a polygon, *e.g.,* RTPolygon new vertices: { 25 @ 0 . 10 @ 50 . 50 @ 20 }
- RTBitmap: an image., *e.g.,* RTBitmap new form: RTIcon roassal

- RTSVGPath: an SVG path, *e.g.,* RTSVGPath new path: 'M150 0 L75 200 L225 200 Z'
- RTArc: arc portion, *e.g.,* RTArc new externalRadius: 100; innerRadius: 20; outerAngle: 45 innerAngle: 90

For each of the examples above, filling in the ... in the following code template will produce a visualization of the shape.

```
v := RTView new.
shape := ... .
v add: shape element.
v
```

An element shape understands the following messages, amongst others:

- size: sets both width and height to the argument given
- width: and height: set the width and the height, respectively
- color: sets the fill color of the shape
- borderColor: sets the border color of the shape

The complete set of messages that may be used with element shapes is declared in the classes RTShape and RTAbstractElementShape.

Edge shapes

Edges are typically drawn between two elements. Similar to elements, an edge may have a shape.

- RTLine: a straight line between its extremities, *e.g.,* RTLine new color: Color blue
- RTArrow: an arrowed line between its extremities, *e.g.,* RTArrow new color: Color blue
- RTDirectedLine: a directed line, *e.g.,* RTDirectedLine new. Direction is given by the curving of the line.
- RTMultiLine: segmented lines, *e.g.,* RTMultiLine new orthoVertical; color: Color blue

> **Book Todo:**
> Unclear: Direction is given by the curving of the line.

The following template shows the use of edge shapes. As above, simply replace ... by an expression given above. At the end of the example code both shapes are set to be draggable, which serves to illustrate how edges are dynamically updated when shapes are dragged around the canvas.

```
v := RTView new.

elementShape := RTEllipse new size: 20; color: (Color blue alpha: 0.3).
e1 := elementShape element.
e2 := elementShape element.

shape := ... .
v add: (shape edgeFrom: e1 to: e2).

v add: e1.
v add: e2.
e1 @ RTDraggable.
e2 @ RTDraggable.
v
```

Functions and metrics as parameters

A graphical element acts as a visual facade for the object that the element represents, *i.e.* for its model. Roassal transparently associates each element to this model object. Typically these objects are provided by the user. For example, In the case given previously of the number of the TSV table entries, each Roassal element represents an entry from the table. Each element has a particular size that is computed from the table entry.

The visual representation given by an element's shapes and the interactions offered to the end user depends on the model object. In Figure 1.2, all the elements have the same shape: a RTBox object with two functions used to compute the width and the height of an element. These functions are evaluated against the table entry given as a parameter when the element is created.

The benefit of having a model object for each element is a transparent update mechanism. As soon as this object is modified (and different caches are properly reset), the visual representation of the element and its interactions are updated accordingly.

Shape composition

Book Todo:
change example.

Figure 1.3: Example of composing shapes.

More elaborate shapes may be obtained by composing the primitive shapes given above. For example, the shape RTMultiCompositeShape is a composite of shapes.

Consider the following example that paints four national European flags (Figure 1.3):

```
view := RTView new.
shape := RTMultiCompositeShape new.
shape add: (RTBox new color: #first; height: 20; width: 15).
shape add: (RTBox new color: #second; height: 20; width: 15) translateBy: 15 @
    0.
shape add: (RTBox new color: #third; height: 20; width: 15) translateBy: 30 @ 0.

flags := Dictionary new.
flags at: 'France' put: (Array with: Color blue with: Color white with: Color red).
flags at: 'Belgium' put: (Array with: Color black with: Color yellow with: Color red).
flags at: 'Romania' put: (Array with: Color blue with: Color yellow with: Color red).
flags at: 'Italia' put: (Array with: Color green with: Color white with: Color red).

els := shape elementsOn: flags values.
view addAll: els.

RTGridLayout new on: els.
view
```

Composed shapes are often employed to give a title to elements. The following example visualizes some of the classes defined in Roassal (Figure 1.4):

```
v := RTView new.
shape := RTMultiCompositeShape new.
shape add: (RTLabel new color: Color gray).
shape add: (RTBox new color: Color lightRed;
            width: [ :cls | cls numberOfVariables * 8 ];
            height: #numberOfMethods).
shape vertical.
es := shape elementsOn: RTShape withAllSubclasses.

v addAll: es.
```

Figure 1.4: Giving a title to some boxes.

Figure 1.5: Elements may be rotated.

```
RTGridLayout on: es.
v
```

Each class is represented as a composite shape. A title is first added to the composite shape. Then a box is added, its width is set as the number of variables and the number of methods is set as its height.

1.3 Element transformation

An element may be translated in the view using the translateBy: and translateTo: messages, both taking a point as argument.

Elements may be rotated by sending rotateByDegrees:. This message accepts as argument a numerical value (*e.g.,* rotateByDegrees: 30), a symbol or a block, as in the following example (Figure 1.5):

```
v := RTView new.
shape := RTBox new width: 5; height: 25.
es := shape elementsOn: (1 to: 90 by: 5).
es rotateByDegrees: #yourself.
RTHorizontalLineLayout new gapSize: 2; on: es.
v addAll: es.
v
```

18 elements are added in the view. Each element represents a value between 1 and 90, with an interval of 5. The variable es represents a group of elements. By sending the message rotateByDegrees: to a group, each element of the group is rotated as specified in the argument of the message.

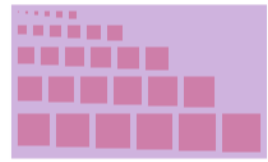

Figure 1.6: Nesting boxes into a larger box.

1.4 Group of elements

The class RTGroup is an ordered collection aimed to contains elements. A group is a composite of elements and dispatches to each contained element received messages. A group may be resized, rotated, and scaled.

Book Todo:
give example

1.5 Nesting elements

Expressing containment is critical as soon as the data to be represented is hierarchical. Being able to embed some elements into other elements is an easy and cheap way to express encapsulation. Roassal offers several options to express nesting or encapsulation, all rooted in the class RTNest.

The message on: element nest: someElements is probably the most commonly used. This message nests someElements into element. Consider the following example:

```
v := RTView new.
el := (RTBox new width: 80; height: 40; color: (Color purple alpha: 0.3)) element.
shape := RTBox new color: (Color red alpha: 0.3); size: #yourself.

innerElements := (1 to: 30) collect: [ :i | shape elementOn: i ].
v addAll: innerElements.

RTGridLayout new on: innerElements.
RTNest new
   on: el nest: innerElements.
v add: el.
v
```

The message on:nest: takes as the first argument an element, onto which you wish to nest the elements provided as second argument. The nesting

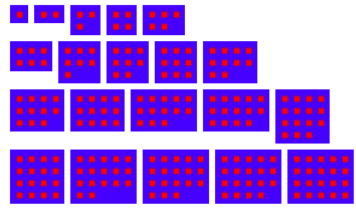

Figure 1.7: Using for:add: to nest elements.

element (*i.e.,* the first argument provided to on:nest:, el in our example) is resized to encapsulate the nested elements. Dragging using the mouse the nesting elements also drag on the nested elements.

The precedent example first set the layout for the inner elements, then nest the element (RTGridLayout on: inner. RTNest new on: el nest: inner.). A layout may be specified when using RTNest. This may be written

```
RTNest new
    layout: RTGridLayout new;
    on: el nest: innerElements.
```

The message for:add: helps recurring the nesting. Consider the example (Figure 1.7):

```
v := RTView new.
es := (RTBox new color: Color blue) elementsOn: (1 to: 20).
v addAll: es.

RTNest new
    for: es add: [ :group :model |
        group addAll: ((RTBox new color: Color red) elementsOn: (1 to: model)).
        RTGridLayout on: group ].

RTGridLayout on: es.
v
```

The message for:add: takes as first argument the group of elements to go over. The second argument is a block that follows the pattern [:group :model | ...]. The variable group is a group in which elements to be nested have to be added into. The variable model is the model represented by the nesting node. The model is often key to compute the elements to be nested.

As a further example, consider the following script:

```
v := RTView new.
es := (RTBox new color: Color white; borderColor: Color lightGray) elementsOn: {
    RTLayout . RTShape . RTBuilder }.
```

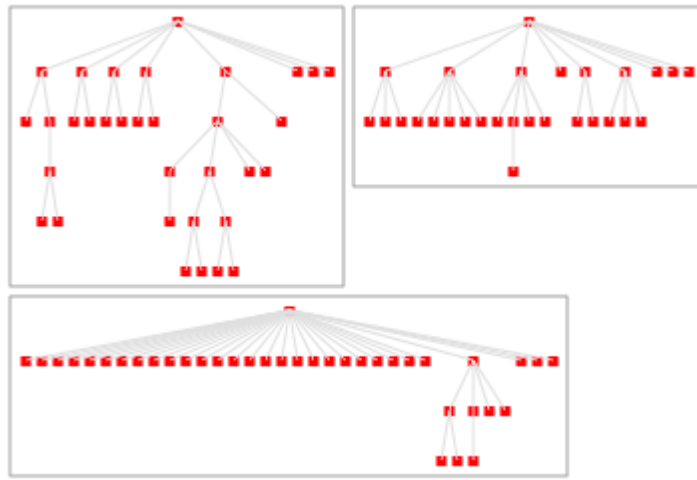

Figure 1.8: Using for:add: and edges.

```
v addAll: es.
es @ RTDraggable.

RTNest new
    for: es add: [ :group :model |
        elements := (RTBox new color: Color red) elementsOn: model
    withAllSubclasses.
        group addAll: elements.
        edges := RTEdgeBuilder new
            view: group;
            objects: model withAllSubclasses
            from: #superclass.
        RTTreeLayout on: elements edges: edges .
    ].

RTGridLayout on: es.
v
```

Figure 1.8 illustrates the usage of for:add: in which edges are added in the group.

Book Todo:
Different form of nesting. for:inShape:add:, on:inShape:nest:layout:

Book Todo:
Nesting in a popup

1.6 Giving behavior to element with interactions

Interaction are particular objects that set a particular behavior on Roassal elements. Such behavior is typically actionnable by a particular user action (*e.g.,* mouse click, mouse overring, key stroke). Interaction may be set to a view, elements or edges.

Three interactions may be set to a view:

- RTDraggableView to set a view draggable. *E.g.,* view @ RTDraggableView.
- RTHorizontalDraggableView to make a view horizontally draggable.
- RTVerticalDraggableView to make a view horizontally draggable.

A number of interactions are available for elements:

- RTDraggable makes an element draggable.
- RTDraggableChildren makes an element draggable. When being dragged, connected elements with an edge are also dragged.
- RTPopup dynamically adds a text to the element pointed by the mouse.
- RTLabelled adds a label above an element. The label may be particularized using text: if the default string representation is not sufficient. Consider:

```
v := RTView new.
e := (RTEllipse new size: 30) elementOn: 42.
v add: e.
e @ (RTLabelled new text: [ :value | 'My value is ', value asString ]).
v
```

- RTShowLabel adds a label on a set of elements when the mouse enters a particular element. RTLabelled is used for the labelling. When the mouse leaves the element, all the labels are removed.
- RTShowEdge adds edges on one particular element when the mouse is above the element. Edges are removed when the mouse leaves the element.
- RTSetEdgeAlpha temporarily decreases the transparency of the incoming and outgoing edges for a given element when the mouse enters the element. When added to an elements, connected edges are made transparent.

Event may be propagated to other elements using RTEventForwarder. This is handy in case objects have to forward some particular actions. Consider the following example

```
v := RTView new.

box1 := (RTBox new size: 20) element.
box2 := (RTBox new size: 20) elementOn: 'hello'.

box2 translateBy: 50 @ 0.
box2 @ RTDraggable.

box1 @ RTDraggable.
box1 @ (RTEventForwarder new toObject: box2).

v add: box1.
v add: box2.
v
```

Dragging the left box forward the dragging events to the right box. Another example of RTEventForwarder is in case elements are above other elements. Consider the following example:

```
v := RTView new.

inner := (RTBox new color: Color green; size: 30) elementOn: 'world'.
outter := (RTBox new color: Color blue) elementOn: 'hello'.

inner @ (RTHighlightable new highlightedColor: Color yellow).
outter @ RTPopup.
inner @ (RTEventForwarder new toObject: outter).

v add: outter ; add: inner.
RTNest new on: outter nest: { inner }.
v
```

The element inner, located above outter, had an interaction RTHighlightable. Without this interaction, locating the mouse above inner displays the popup for outter. However, since inner has an interaction already, the elements answer events RTMouseLeave and RTMouseEnter. The runtime does not search for elements answering these events which may be located below it. RTEventForwarder is here used to make inner answer the highlighting interaction *and* triggering the popup of outter.

1.7 Normalizer

Being able to quickly compare data elements is essential in a visualization. *Pre-attentive processing* refers to the ability of the human brain to uncounsciously accumulate information from the surrounding. Without even realizing what is front of our eyes, the brain filters and processes what is important. This phenomen is particular relevant in data visualization. Our brain is able to immediately spot a different element or a stark color.

The class RTMetricNormalizer normalizes one or more visual dimensions of a set of Roassal elements. Consider the following example (Figure 1.9):

```
v := RTView new.
es := (RTEllipse new color: Color blue) elementsOn: #(4 5 1 2 3 5).
v addAll: es.

es @ RTPopup.

RTMetricNormalizer new
   elements: es;
   alphaColor: 0.3;
   normalizeSize: #yourself min: 5 max: 30.

RTHorizontalLineLayout new alignCenter; on: es.

es @ RTLabelled.

v
```

The script above creates six elements, each representing one of the values in #(4 5 1 2 3 5). Elements are then added to the view and a popup interaction is given to each element.

The class RTMetricNormalizer is then instantiated. Elements to be normalized are set in the metric normalizer using elements:. Elements are first made translucent (note that strictly speaking the message alphaColor: is not a normalization, but rather a convenience). The message normalizeSize:min:max: normalizes the size of each element against the set of elements provided with elements:.

Similarly, the position along the X-axis may be normalized. Consider the example below (Figure 1.10):

```
v := RTView new.
es := (RTEllipse new size: 15; color: Color blue) elementsOn: #(4 5 1 2 3 5).
v addAll: es.

es @ RTPopup.
```

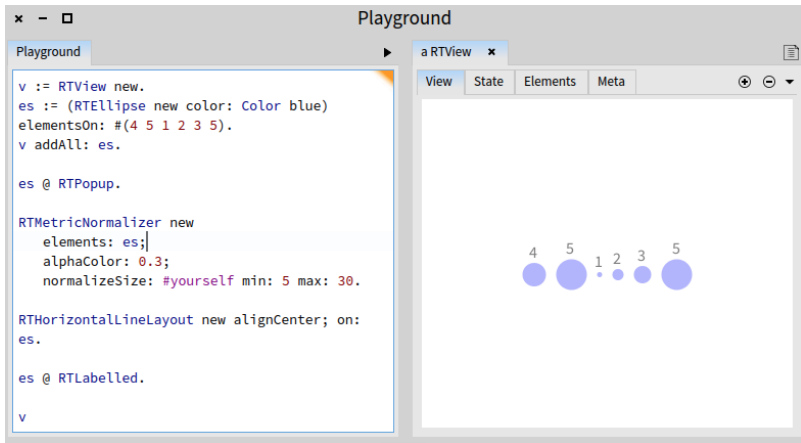

Figure 1.9: Normalizing element size.

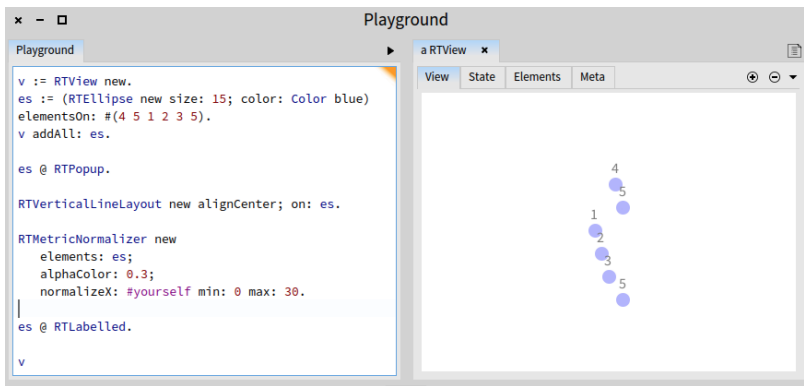

Figure 1.10: Normalizing element size.

```
RTVerticalLineLayout new alignCenter; on: es.

RTMetricNormalizer new
   elements: es;
   alphaColor: 0.3;
   normalizeX: #yourself min: 0 max: 30.

es @ RTLabelled.

v
```

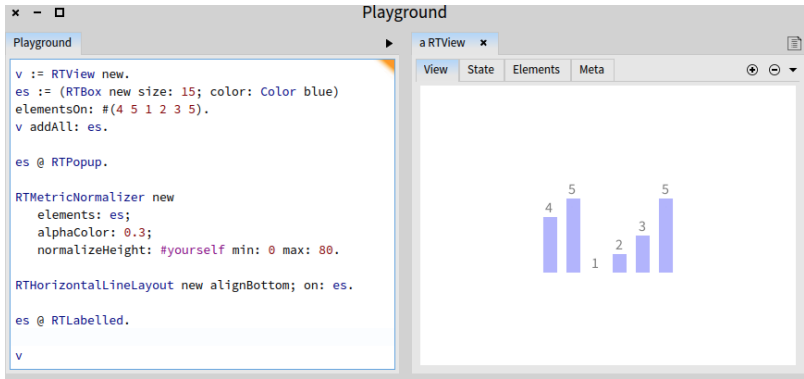

Figure 1.11: Normalizing element height.

Normalization is essential for graph and histogram charting (cf Chapter on Charter). Consider the script:

```
v := RTView new.
es := (RTBox new size: 15; color: Color blue) elementsOn: #(4 5 1 2 3 5).
v addAll: es.

es @ RTPopup.

RTMetricNormalizer new
   elements: es;
   alphaColor: 0.3;
   normalizeHeight: #yourself min: 0 max: 80.

RTHorizontalLineLayout new alignBottom; on: es.

es @ RTLabelled.

v
```

You can notice this histogram is not quite right. The bar for the element 1 is not visible. This is because it has a height of 0. This is expected since the minimum value of the elements model is given 0 pixel (*i.e.,* the value provided to min:). A proper version of our (primitive) histogram charting needs to specify a minimum and maximum value. Consider (Figure 1.12):

```
v := RTView new.
es := (RTBox new size: 15; color: Color blue) elementsOn: #(4 5 1 2 3 5).
v addAll: es.

es @ RTPopup.
```

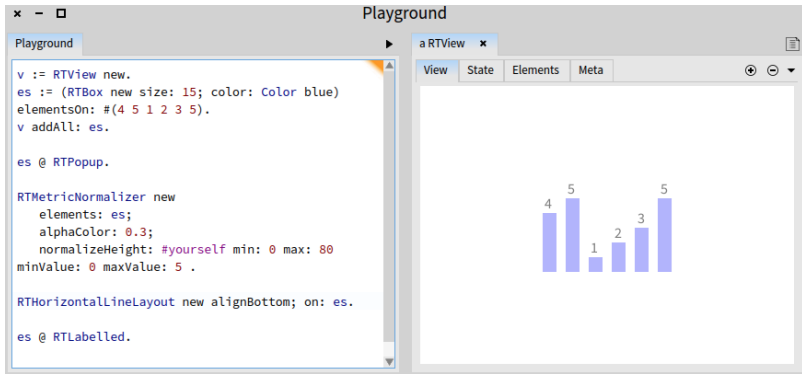

Figure 1.12: Normalizing element height.

```
RTMetricNormalizer new
   elements: es;
   alphaColor: 0.3;
   normalizeHeight: #yourself min: 0 max: 80 minValue: 0 maxValue: 5 .

RTHorizontalLineLayout new alignBottom; on: es.

es @ RTLabelled.

v
```

The message normalizeHeight:min:max:minValue:maxValue: takes as argument a minimum and maximum value.

1.8 Expressing constraints

> **Book Todo:**
> code below uses the code of Roassal. Change this to a real example.

Alignment

Adequately positioning some elements against other elements is often crucial. It freuently happens that elements need to be resized and positionned against other elements or the windows. The class TRConstraint offers a number of methods (located on the class side) dedicated to constraint the size or

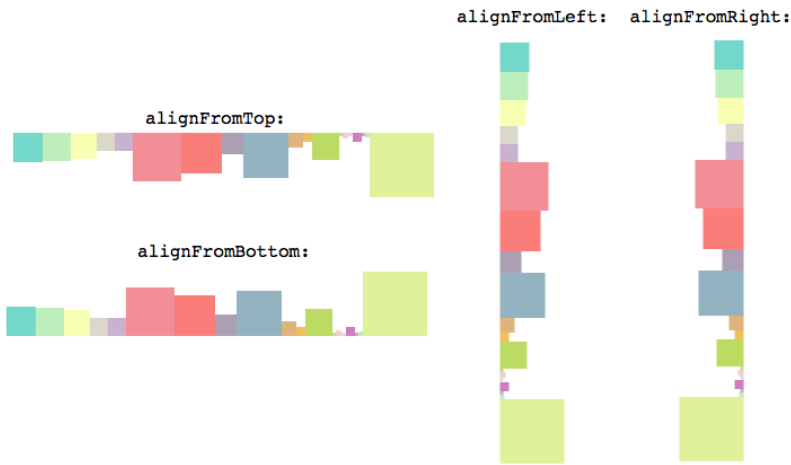

Figure 1.13: Use of alignment.

the position of some elements.

Consider the following example (Figure 1.13):

```
objects := TRShape withAllSubclasses.
v := RTView new.
n := RTMultiLinearColorForIdentity new objects: objects.
shape := RTBox new color: n; size: #numberOfMethods.
es := shape elementsOn: objects.
v addAll: es.
RTHorizontalLineLayout new gapSize: 0; on: es.
TRConstraint alignFromTop: es.
v
```

Figure 1.13 illustrates the different ways to line up elements using the class TRConstraint. Elements can be lined up against their lowest point (alignFromBottom:), against the left-most position (alignFromLeft:), Elements may be lined up against the right-most position using alignFromRight:. You may want to replace RTHorizontalLineLayout by RTVerticalLineLayout to try alignFromLeft: and alignFromRight:.

Aligment may also be defined against a particular element. The message use:alignFromBottom: and use:alignFromTop: will align the first element the collection of elements provided as the second argument.

Most of the time, RTGroup are polymorphic to RTElement. This means that you can provide a group of elements where an element is expected. Consider the following example (Figure 1.14):

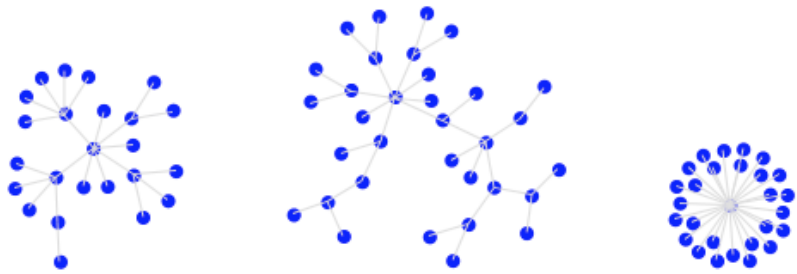

Figure 1.14: Group alignments.

```
rootClasses := { RTShape . RTLayout . RTBuilder }.

groups := rootClasses collect: [ :cls |
    g := RTGroup new.
    elements := (RTEllipse new size: 8; color: Color blue) elementsOn: cls
    withAllSubclasses.
    g addAll: elements.
    edges := RTEdgeBuilder new
        view: g;
        objects: cls withAllSubclasses from: #superclass.
    RTForceBasedLayout on: elements edges: edges.
    g ].

v := RTView new.
groups do: [ :aGroup | v addAll: aGroup ].
RTHorizontalLineLayout new gapSize: 30; on: groups.

TRConstraint alignFromBottom: groups.
v
```

The example above performs a alignFromBottom: on a groups made of element groups.

Resizing

Elements may have their width or height set to the largest dimension from a group of elements. Consider the following code (Figure 1.15):

```
objects := TRShape withAllSubclasses.
v := RTView new.
n := RTMultiLinearColorForIdentity new objects: objects.
shape := RTBox new color: n; size: #numberOfMethods.
es := shape elementsOn: objects.
```

Figure 1.15: All elements have the same height.

```
v addAll: es.
RTHorizontalLineLayout new gapSize: 0; on: es.
TRConstraint setAllToMaxHeight: es.
TRConstraint alignFromTop: es.
v
```

Similarly, setAllToMaxWidth: can be used. You may want to use RTVerticalLineLayout in this example.

Positioning

The class TRConstraint offers several methods (class side) to position elements against other.

- move:above: and move:below: moves a particular elements above or below another element
- move:between:and: and move:between:and:offset: to locate an element between two other elements
- move:onTheLeftOf:, move:onTheRightOf:, move:onTheTopLeftOf:, ... to position a particular element

Most of the methods contained in the class TRConstraint accepts a group instead of an element as argument. To illustrate this point, consider the following example:

```
v := RTView new.
n := RTMultiLinearColorForIdentity new
        numberOfColors: 20;
        colors: { Color red . Color gray }.
es1 := (RTEllipse new color: n; size: 15) elementsOn: (1 to: 20).
v addAll: es1.

n := RTMultiLinearColorForIdentity new
        numberOfColors: 20;
        colors: { Color yellow . Color purple }.
es2 := (RTEllipse new color: n; size: 15) elementsOn: (1 to: 20).
v addAll: es2.
```

Figure 1.16: Positioning a group above another group.

```
RTForceBasedLayout on: es1.
RTGridLayout on: es2.
TRConstraint move: es1 above: es2.
v
```

Figure 1.16 depicts the result. Two group of elementns are in play: es1 and es2. A force based layout is applied to the first group. A grid layout to the second group. The use of move:above: positions the first group above the second group.

1.9 Automatically View Refreshing

A view may be updated with new elements or modified elements in two different ways: either the view itself decide to refresh its content, or the view receives a signal triggered from a particular objects.

Consider the following script:

```
c := OrderedCollection new.

v := RTView new.
v canvas addMenu: '+1' callback: [
    c add: 42.
    v clean.
    v addAll: ((RTBox new size: 8; color: Color blue) elementsOn: c).
    RTGridLayout on: v elements.
    v signalUpdate ].
v
```

Pressing the *+1* button adds a blue color box. This is a simple solution to automatically add elements. An alternative, could be:

```
c := OrderedCollection new.

v := RTView new.
v canvas addMenu: '+1' callback: [ c add: 42 ].

RTActiveAnimation new
    blockToExecute: [
        (v numberOfElements ~~ c size) ifTrue:
            [ v clean.
              v addAll: ((RTBox new size: 8; color: Color blue) elementsOn: c).
              RTGridLayout on: v elements.
             ] ];
    inView: v.
v
```

This code has the same effect. There is a subtil and important difference between these two code snippets. The second one, check the size of the c collection at each screen refresh. No need to care about sending #signalUpdate.

1.10 Visualizing a Domain

Roassal promotes a clear separation between the domain to be represented, and the visual representation. We refer to *domain* the input of the visualization, or, in a general sense, the objects you do care about. The domain should not be aware of the visualization. Whether your objects are visualized or not should not matter for these objects.

> **Book Todo:**
> give example

Roassal is built on a solid base. From a software engineering point of view, it is largely recognized about the benefits to separate a domain from a visualization: modularity, source code reduction, easiness for testing.

1.11 Code Snippets

This section contains a number of executable code snippets that illustrate various aspects of Roassal.

Edges with invisible extremities

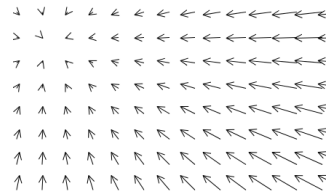

Figure 1.17: Invisible extremities.

```
v := RTView new.
edges := OrderedCollection new.

es := RTBox elementsOn: (1 to: 300).
RTGridLayout new gapSize: 30; on: es.

es do: [ :e |
    | te |
    te := RTBox element.
    te translateTo: e position + (0 @ 10).
    edges add: ((RTArrow new color: Color black) edgeFrom: e to: te).
].

v addAll: edges.

v when: TRMouseMove do: [ :evt |
    edges do: [ :edge |
        edge to
            translateTo: (evt position − edge from position) / 10 + edge from position.
    ].
    v signalUpdate.
].
v
```

Dynamically adding elements

```
v := RTView new.
v @ RTDraggableView.
stepping := RTSpringLayoutStepping new view: v.
stepping after: [ v canvas camera focusOnCenter ].
v addAnimation: stepping.

v canvas addMenu: 'Add circle' callback: [
    | el |
    el := (RTEllipse new size: 20; color: (Color blue alpha: 0.4)) element.
```

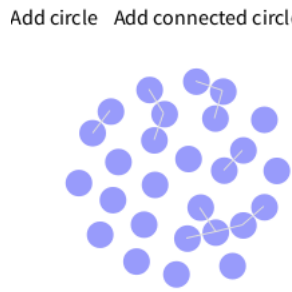

Figure 1.18: Dynamically adding elements.

```
    el @ RTDraggable.
    el translateTo: 5 atRandom @ 5 atRandom.
    v add: el.
    stepping addNode: el.
    v signalUpdate.
].

v canvas addMenu: 'Add connected circle' callback: [
    | el edge |
    el := (RTEllipse new size: 20; color: (Color blue alpha: 0.4)) element.
    el @ RTDraggable.
    el translateTo: 5 atRandom @ 5 atRandom.
    v add: el.

    edge := RTLine edgeFrom: el to: v elements atRandom.
    v add: edge.

    stepping addNode: el; addEdge: edge.
    v signalUpdate.
].
v
```

Orthogonal lines

```
v := RTView new.

v addAll: ((RTEllipse new size: 20; color: (Color red alpha: 0.3)) elementsOn: (1
    to: 20)).

shape := RTMultiLine new.
shape orthoVertical.
```

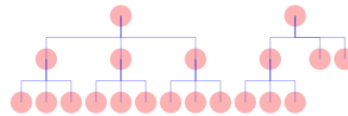

Figure 1.19: Orthogonal edges.

```
shape color: (Color blue alpha: 0.3).

RTEdge
   buildEdgesFromObjects: (1 to: 20)
   from: [ :n | n // 3 ]
   to: #yourself
   using: shape
   inView: v.

v elements @ RTDraggable @ RTPopup.

RTTreeLayout on: v elements.
v
```

Dynamically adding edges

```
v := RTView new.
es := (RTBox new color: (Color blue alpha: 0.4))
      elementsOn: (RTShape withAllSubclasses).
v addAll: es.

RTNest new
   for: es add: [ :group :model |
      group addAll: (RTBox new elementsOn: model methods).
      RTGridLayout on: group ].

es @ (RTShowEdge new
         connectTo: #dependentClasses;
         shape: (RTLine new color: Color red);
         yourself).
es @ (RTShowLabel new
         color: Color red;
         highlight: #dependentClasses; top; yourself).
RTGridLayout on: es.
v
```

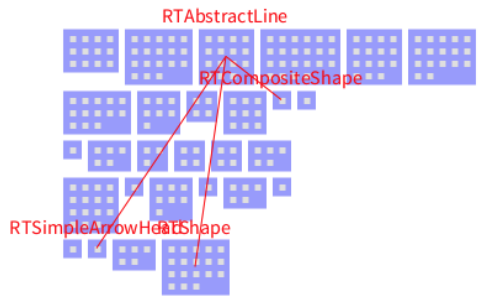

Figure 1.20: Dynamically adding edges.