Agile Software Visualization with Roassal

Alexandre Bergel, Vanessa Peña

University of Chile

April 26, 2013

Abstract

Giving a meaning to a large amount of data is challenging without adequate tools. Textual outputs are known to be limited in their expressiveness and support for interaction. The rapid growth of data sources has resulted in a surge of libraries and frameworks to visually analyze data. Roassal is an agile visualization engine, particularly adapted to software comprehension and reengineering.

1 Introduction

The cheer size of today software system has lead the software engineering community to produce efficient ways to get away from the textual representation of source code. The application range for new visual abstractions is broad: maintenance [1], dynamic analysis [2], test understanding [3], just to name a few.

Roassal is an agile visualization engine for the Pharo and VisualWorks programming languages. Roassal visualizes any arbitrary data and offers effective ways to interact, navigate and filter. Although Roassal is not tied to a particular application domain, Roassal is commonly used to visualize software systems. Roassal supports a number of features to address recurrent needs when visualizing software, namely designing and particularizing caches for optimization purposes; offering tools and mechanisms to build a visualization in a try-and-adjust fashion; closely connect a visual representation with its related software element.

As most graphical libraries, Roassal offers a flexible way to visually render data and offers a large set of interactions and visual representations. In addition, Roassal differs from traditional libraries by (i) transparently linking the domain objects and their visual representation; (ii) supporting a flexible way to build cache; (iii) providing an easel for interactive construction; (iv) offering the necessary tooling to supporting navigation between visual visualizations.

The design decisions we made for Roassal are based on the experienced gained with several other visualization engines (CodeCrawler [4], Mondrian [5], ZVTM [6]).

This paper presents the design points of Roassal and their benefices. This paper is structured as follows: Section 2 presents the design of Roassal. Section 3 compares Roassal with other similar work. Section 4 concludes this paper.

2 The Roassal Model

This section presents the main design decisions of Roassal. It also highlights benefits for particular situations for these decisions.

2.1 View, Elements, Shapes and Interactions

Roassal structures a visualization in terms of views, elements, shapes, interactions and animations. A *view* is simply a container of graphical elements and represents a layer that may be combined with other views. Combining views is interesting when a visualization has to be decomposed into group of semantically coherent graph elements (Section 2.5).

An *element* is a graphical representation of an arbitrary object. This representation is built by combining elementary *shapes*. A shape describes a visual primitive representation such as a box, a circle, a line or a textual label. Shapes may be combined to form more elaborated shapes. An end user sees elements and interacts with them by using the mouse and keyboard. A Roassal element is a compound object that contains (i) a two dimensional spacial location; (ii) a set of interactions; (iii) a set of shapes; (iv) a reference to a parent element to support hierarchical representation. To be visible, elements must have at least one shape and be added in a view.

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

Consider the following example that visually represent classes of the Pharo collection library:

```
1  | view element shape |
2  view := ROView new.
3  
4  Collection withAllSubclasses do: [ :cls |
5     element := ROElement model: cls.
6     shape := ROBorder new
7                        width:  [ :el | el model instVarNames size * 5 ];
8                        height: [ :el | el model methods size ].
9     element + shape.
10    element @ RODraggable @ ROPopup
11            @ (ROMenuActivable item: 'Browse class' action: [ :el | el model browse ]).
12    view add: element ].
13 
14 ROHorizontalLineLayout new gapSize: 2; on: view elements.
15 view open
```

Line 2 instantiates a new view, elements will be added to it in the subsequent lines. Line 4 iterates over all the subclasses of the class Collection and evaluates the provided block (`[ :cls | ...]`) for each of these subclasses. Line 5 creates an element for the class `cls`. Lines 6 - 9 creates and add to the element a border shape and specify the height and width of it. Line 10 - 11 add the `RODraggable`, `ROPopup` and `ROMenuActivable` interactions to the current element. Line 12 adds the element to the view Line 14 horizontally locate the nodes with a two-pixels gap between classes. Finally, Line 15 opens the view.

Figure 1 gives a screenshot of the visualization. Classes are horizontally lined up. Each class has a size that depends on the amount of instance variables and methods it defines. Each element gives the name of the represented class as a popup. The class `RunArray` has been dragged.

Relationships between elements may be expressed via edges, nesting or interaction (*e.g.,* color highlighting).

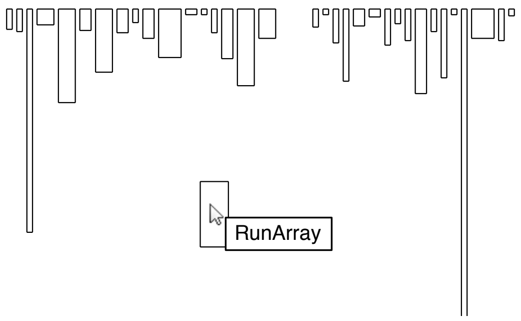


Figure 1: Polymetric views on classes

2.2 Visualization associated to a domain

A graphical element acts as a visual facade of the object the element represents. Roassal transparently associates each element to a user object, typically externally provided as with the example of the collection classes. In the example given previously, each Roassal element represents a class from the Pharo collection library. Each class has a particular size, computed from the amount of methods and instance variables of the represented class.

The visual representation given by an element's shapes and the interactions offered to the end user depends on the object model. In Figure 1, all the elements have the same shape which is a `ROBorder` object with two functions used to compute the width and the height of an element. These functions are evaluated against the object class given in parameter when rendering the element.

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

2.3 Customizable Optimizations

Making a visualization scalable necessitates visualization-specific optimization. In addition to the collection of elements that contains, a view has a separate queue of the elements to be rendered. This rendering queue contains a subset of elements, if not all of them, that have to be rendered at the same time. At each refresh, the queue of the rendering queue is run over.

In the case that a large number of elements are present, a visualization

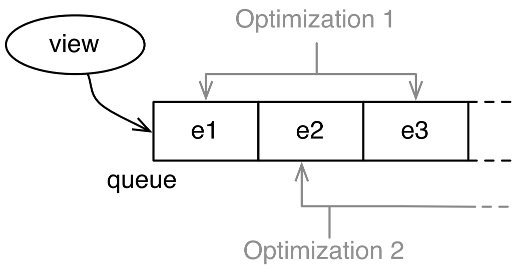


Figure 2: Optimizing by removing or adding in the queue of rendered elements.

is typically optimized and made exploitable by manipulating the queue of rendered elements. Instead of rendering all the elements at once, only a carefully selected portion of them are visualized. The hidden portion may be revealed upon particular actions, such as moving the window, the camera or the mouse.

Displaying only what is visible. In case a visualization is larger than what the end user can actually see implies that non-visible elements have to be excluded from the rendering loop in order to not unnecessarily consume resources. Moving the visible part of the hosting window (e.g., moving a window scrollbar) add or remove elements from the rendering queue to reflect the portion of the visualization that is effectively being seen.

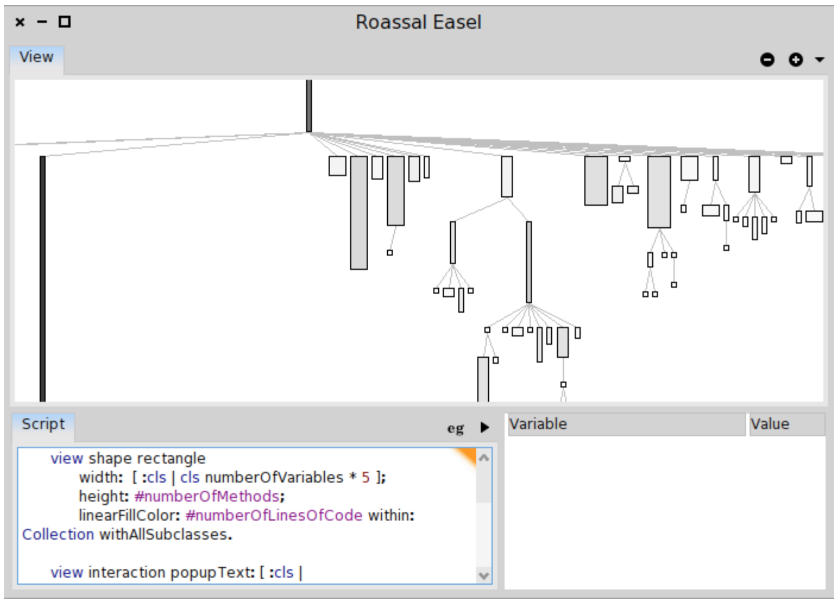
Such a mechanism may be tailored to particular needs. For example, in some cases, edges may be hidden if both extremities are not visible, or may be rendered if only one element extremity is not visible.

Semantic zooming. The intuition exploited with semantic zooming is that details are revealed when zooming in, and hidden out when zooming out. Elements contained in the rendering queue depends the camera altitude therefore. Moving the camera up and down may considerably reduce the amount of visible objects.

Edges on demand. Representing a software structure typically involves a high number of edges to represent dependencies between entities [7]. Being able to contextually adjust edges to be shown leads to a significant improvement of both clarity and usability of the visualization.

Using user actions to temporarily render edges is efficient at significantly reducing the cluttering of edges. For example, edges going out from an element are visible by having the mouse above the element. When the mouse leaves the element, edges are removed. Such addition and removal of elements from the rendering queue are triggered from particular user actions.

2.4 Easel for interactive construction



```
    view shape rectangle
        width:  [ :cls | cls numberOfVariables * 5 ];
        height: #numberOfMethods;
        linearFillColor: #numberOfLinesOfCode within:
Collection withAllSubclasses.

    view interaction popupText: [ :cls |
```

Figure 3: Easel for interactive sessions.

The Roassal easel is a tool to build a visualization by repeatedly try-and-adjust the defining script. The easel allows for an exploratory construction of the Roassal script by breaking the traditional edition/compilation/run cycle.

The easel may also be contextualized by offering a list of variables that can be used in the script.

2.5 Other features

Camera. Each Roassal view has a camera. The camera has a position and an altitude. Zooming effects are obtained by varying the altitude. Reducing the camera's altitude produces a zooming-in, and increasing it a zoom-out effect. The orientation of the camera is perpendicular to the view: a camera can move horizontally, vertically, going up and down.

Animations. Roassal offers a number of animations for elements, view or camera. Animation objects forms a sequence and are held by the view. Each refresh of the view executes an incremental step of each of the animations.

An earlier version of Roassal defined a thread per animation. However involving threads has many limitation, such as scalability (e.g., in case each node is animated, having over thousands of threads is not practicable) and non-determinism (scheduling of threads makes it difficult to know when an animation completes), among others.

Stack of views. Several views may be visible at the same time. Stacking views is particularly useful when a view offers a particular set of elements or has different interactions than the other views. One example is when a view displays a menu or a legend, and the visualization is given by another view. The legend and the menus typically should react to the user actions in a different way that the core of the visualization.

3 Related Work

Roassal has been greatly inspired from ZVTM [6][1]. For example, ZVTM offers the rendering queue and the camera. ZVTM has not been designed to support short step-wise visualization construction. By being made in Java, a program made with ZVTM has to go through the traditional edit-compile-run loop, while Roassal favors an edit-resume loop, a lightweight approach that reduces context switching and compilation time.

Processing[2] is a popular visualization library. Processing offers efficient rendering primitives with a transparent access to dedicated hardware. Its domain application is broad, ranging from art performance to massive data analysis. Processing supports the edit-compile-run loop promoted by Roassal. Rendering primitives given by Processing are rather low level. In addition, the programming environment of Processing does not offer a debugger to assist developer. To our knowledge, we are not aware of any use of Processing to visualize software.

[1]Zoomable Visual Transformation Machine – `http://zvtm.sourceforge.net`

[2]`http://www.processing.org`

4 Conclusion

This paper presents the main design decisions that have been made when developing Roassal. The range of visualizations supported by Roassal is wide and addresses common situations faced by researchers from the field of software understanding and reengineering.

References

[1] M. Lanza, S. Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.
URL http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf

[2] A. Bergel, F. B. nados, R. Robbes, W. Binder, Execution profiling blueprints, Software: Practice and Experience 42 (9) (2012) 1165–1192. doi:10.1002/spe.1120.
URL http://bergel.eu/download/papers/Berg11f-profiling.pdf

[3] A. Lienhard, T. Gîrba, O. Greevy, O. Nierstrasz, Test blueprints — exposing side effects in execution traces to support writing unit tests, in: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), IEEE Computer Society Press, 2008, pp. 83–92. doi:10.1109/CSMR.2008.4493303.
URL http://scg.unibe.ch/archive/papers/Lien08a-TestBlueprint.pdf

[4] M. Lanza, CodeCrawler — lessons learned in building a software visualization tool, in: Proceedings of CSMR 2003, IEEE Press, 2003, pp. 409–418. doi:10.1109/CSMR.2003.1192450.
URL http://scg.unibe.ch/archive/papers/Lanz03aLessonsLearned.pdf

[5] M. Meyer, T. Gîrba, M. Lungu, Mondrian: An agile visualization framework, in: ACM Symposium on Software Visualization (SoftVis'06), ACM Press, New York, NY, USA, 2006, pp. 135–144. doi:10.1145/1148493.1148513.
URL http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf

[6] E. Pietriga, A toolkit for addressing hci issues in visual language environments, IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 00 (2005) 145–152. `doi:10.1109/VLHCC.2005.11`.

[7] D. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data, Visualization and Computer Graphics, IEEE Transactions on 12 (5) (2006) 741–748. `doi:10.1109/TVCG.2006.147`.