Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

Spring Semester 2024

An interactive Pharo Implementation of Voronoi Diagrams

Jeferson Morales Mariciano

Abstract

In the midst of the digital era, software systems have become ubiquitous, and their complexity has grown by orders of magnitude. The problem of understanding such systems, making them explainable, pressingly arises. Visualization tools are a great asset to understand the structure of software systems. Voronoi treemaps is the current state of the art regarding space-filling representation of hierarchical data. In this project, a purely object-oriented and interactive implementation of Voronoi diagrams is presented, leveraging the powerful Roassal visualization framework and the live environment offered by Pharo.

Advisor
Prof. Dr. Michele Lanza

Advisor's approval (Prof. Dr. Michele Lanza): Date:

Contents

List of Figures

Listings

1 Introduction

1.1 Motivation

Software systems are complex hierarchical structures consisting of thousands of entities and millions lines of code. Much data is either inherently hierarchical or purposefully organized in this manner for easier comprehension, abstraction, and interaction. The hierarchical relations can be represented in a rooted tree, where singleton sets of base elements form leaves, and each inner node represents the union of its children. Treemaps have been proposed as a space-filling representation of such hierarchy trees. Voronoi treemaps are the current state of the art in this field, and readers familiar with computational or combinatorial geometry will have noticed in the last few years the increasing interest in such geometrical construct, even in articles of natural science journals under different names specific to the respective area. Voronoi diagrams arise in nature in various situations and several natural processes can be used to define particular classes of Voronoi diagrams. Human intuition is often guided by visual perception: if one sees an underlying structure, the whole situation may be understood at a higher level. Second, its mathematical properties led several authors to believe that it is one of the most fundamental constructs defined by a discrete set of points.

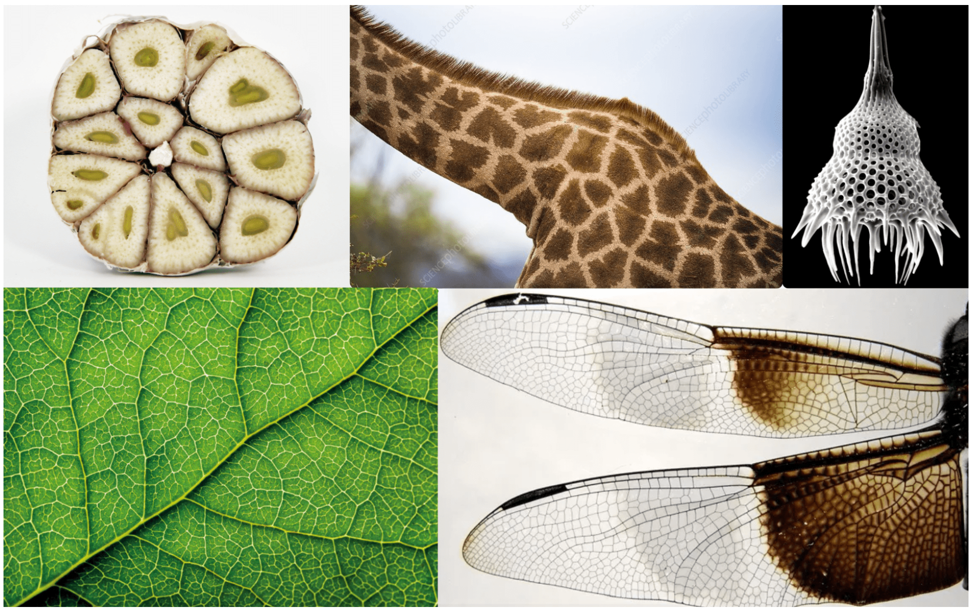


Figure 1. Voronoi diagram patterns in nature

1.2 Goal

Design a Voronoi diagram interactive and purely object-oriented implementation in Pharo with the aim of facilitating software systems explainability, making them more understandable by visualizing software metrics. Software metrics are a quantitave measure of the degree to which a software system, a component, or process possesses a given attribute [10].

1.3 Approach

At the beginning of the project, a top-down analysis followed the study of the state of the art of treemaps and Voronoi diagrams described in section 2, leading to gather the necessary requirements for the analysis of the domain of the topic in section 3. To tackle the problem space, the system must follow an object-oriented design as depicted in section 4. Then, the implementation of the system is detailed in section 5. Finally, the system limitations are discussed together with the wrap up summary of the project in section 6.

2 State of the Art

2.1 Previous Work on Treemaps

Treemaps are a space-filling method of visualizing large hierarchical data sets. They were first introduced by Shneiderman and Johnson in 1991 [8]. Originally designed to visualize files on a hard drive, treemaps have been applied to a wide variety of domains ranging from financial analysis [9], [15] to sports reporting [6]. The basic idea is to subdivide a given area without producing holes or overlappings. Therefore, the area is alternately divided horizontally and vertically according to the hierarchy of the objects and the given proportion between the considered objects. This treemap layout approach is called *Slice and Dice*, and an example is represented in Figure 2.

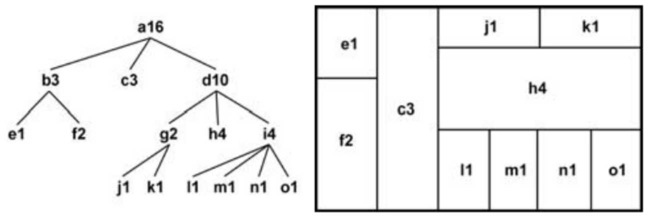


Figure 2. Slice-and-Dice treemap layout algorithm [10]

The treemap is constructed via recursive subdivision of the initial rectangle. The size of each sub-rectangle corresponds to the size of the node. The direction of the subdivision alternates per level: first horizontally, then vertically, again horizontally, etc. The initial rectangle is partitioned into smaller ones, so that the size of each rectangle reflects the size of the leaf. As a result of its construction, the treemap reflects the structure of the tree.

A negative effect in this layout is that the subdivision of each step is done in one dimension. As result, thin elongated rectangles with high aspect ratio between width and height emerge, if many objects or ones with high diversity in size are considered. Such long rectangles are difficult to see, select, compare in size and label as Figure 3 presents as example.

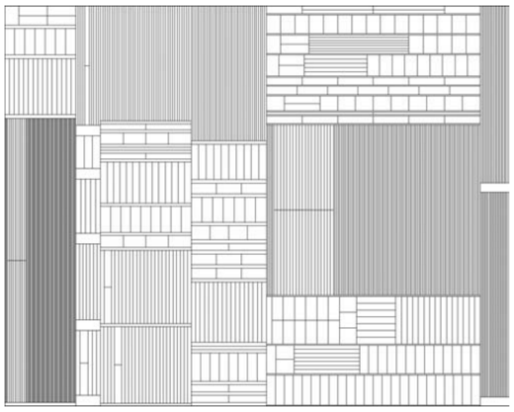


Figure 3. Aspect ratio problem with Slice-and-Dice treemap layout [10]

From 1999 to 2001, the issue was first tackled by Clustered Treemaps [16] where the aspect ratio problem was to employ both vertical and horizontal partitions at each level of hierarchy and place similar sized as a contiguous region, as shown in Figure 4.
Then, Squarified Treemaps [2] forced the aspect ratio of the rectangles to be close to one to resemble a square, and showed how frames can be used to improve the perception of structure. With squarification, the relative ordering of siblings is lost and images tend to be less regular, with less standard patterns, than standard treemaps. Nevertheless, when the structure of the tree is important, its usefulness is evident, and an example is shown on the left in Figure 5.
Finally, Ordered Treemaps [11] ensures items near each other in the given order will be near each other in the treemap layout. Thus, the improvement relies on preserving the ordering of the treemap, as figuratively shown in Figure 6.

The main optimization criterion of these layouts algorithms is the approximation of the sub-rectangles to the shape of a square, whereby the aspect ratio between width and height of each rectangle converges to one.

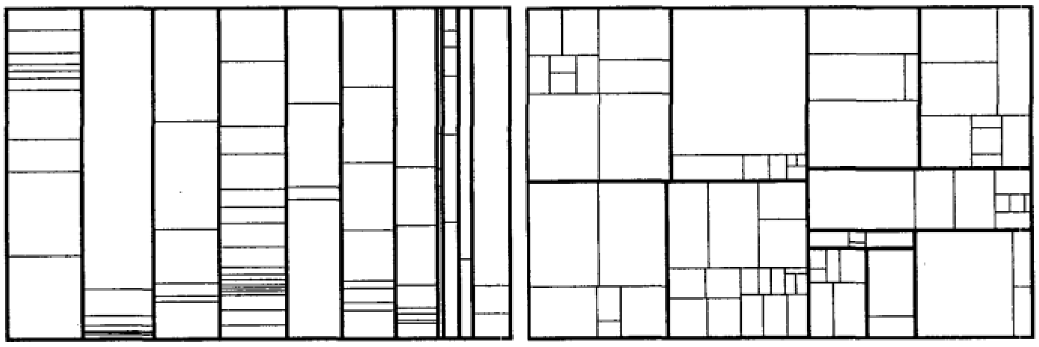


Figure 4. Hierarchical improvement from S&D to Clustered Treemap layout [16]

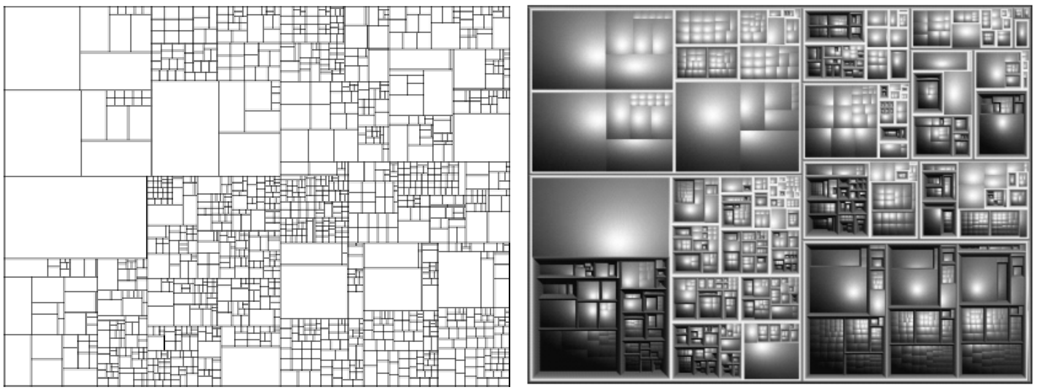


Figure 5. Framed Squarified Treemap layout [2]

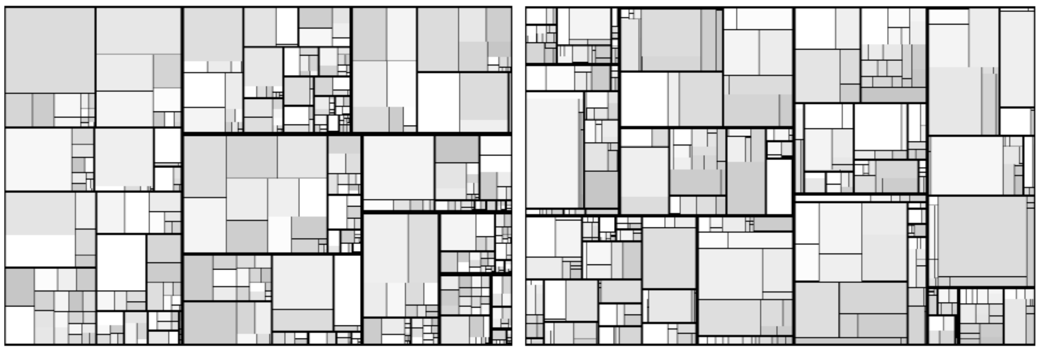


Figure 6. Ordered improvement from Squarified to Ordered Treemap layout [11]

Other criteria are considered as well: for example, the order of the objects or the nearness to a given point in the initial area, or other criteria often closely related to the respective application domain.

The persisting problem is the **difficulty to differentiate if two neighbor objects** are siblings or far away in the hierarchy. The problem is provoked by the square-like shape of the rectangles, and because the edges are only horizontally and vertically aligned, whereby the edges of the different objects appear to run into each other. Figure 5 shows that this effect may be reduced, but can not be prevented by using frame borders around the rectangles and/or Cushion Treemaps [13] with their shading cushion effect. So far all treemap layouts are **restricted to axis-aligned rectangular shapes**.
Voronoi treemaps are polygon-based treemaps allowing support for non-regular shapes.

2.2 Voronoi Diagrams

Intuitively, given a number of points in the plane, their **Voronoi diagram** divides the plane according to the *nearest-neighbor rule*: each point is associated with the region of the plane closest to it.

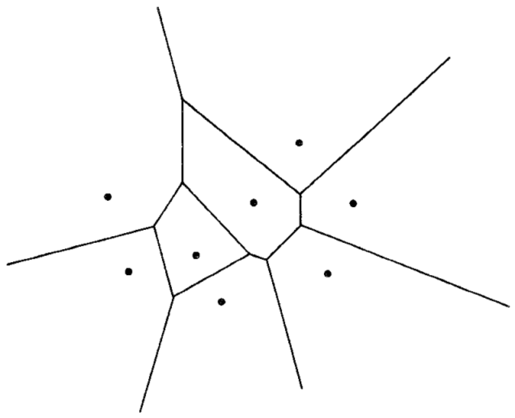


Figure 7. Voronoi diagram for eight sites in the plane [1]

A Voronoi diagram is formally defined as follows: given a set S of n distinct points called **sites**, the corresponding Voronoi diagram divides the plane into **regions**, one for each site. Each region, called a Voronoi cell, consists of exactly those points that have the same closest site.

Since display space is usually bounded, we consider bounded Voronoi diagrams. Formally, for a convex area $\Omega \subset \mathbb{R}^2$ and a set of sites $S = \{s_1, s_2, \dots, s_n\}$, $\forall s_i \in S$ its associated cell $\mathcal{V}_i$ is defined as:

$$\mathcal{V}_i = \{p \in \Omega : \|p - s_i\| < \|p - s\| \; \forall s \in S - s_i\} \tag{1}$$

where $\|p_1 - p_2\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ is the Euclidean distance of points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. Each cell $\mathcal{V}_s = \mathcal{V}(s)$ is bordered by a polygon $\overline{\mathcal{V}_s}$ of points that have equal distance to at least two sites, or belong to the boundary of Ω. Furthermore the area of a cell (region) is denoted by $A(V_s)$.

An ordinary Voronoi diagram is thus defined as the collection of cells,

$$\mathcal{V}(S) = \{\mathcal{V}(s_1), \dots, \mathcal{V}(s_n)\} \tag{2}$$

Voronoi treemaps objects satisfy the following properties:

1. the distribution of objects completely utilizes the given area without holes and overlappings
2. objects distinguish themselves by their irregular shapes and edges not running into each other
3. objects are compact with aspect ratio between width and height converging to 1

Voronoi cell objects use polygons to satisfy the properties. Polygons, defined as a closed plane figure with n sides, can be divided into smaller polygons, satisfying the first property. Then, polygons have arbitrary shapes and large amount of edges which can potentially approximate curvatures, satisfying the 2nd and 3rd properties.

The main idea is to consider the objects of the top level in the hierarchy, which are distributed in the given area. The output will be a set of polygons. For the next hierarchy level, this algorithm is performed recursively within the according polygons of the considered objects in the hierarchy, and so on.

2.3 Pharo

I invented the term Object-Oriented, and I can tell you I did not have C++ in mind.

Alan Kay

Pharo is a **reflective** purely **object-oriented** language supporting **live programming** inspired by Smalltalk. It's more than a language: it ships a powerful environment (IDE) for developing software, focused on **simplicity** and **immediate feedback** [3].

Let's break down the terms by briefly argumenting them. The simplicity lies on the fact that the language is minimalistic: there are no constructors, type declarations, interfaces, primitive types.

The language is purely and uniformly object-oriented because both system and language are composed from objects and, most imporantly, messages *all the way down.* E.g. all of the listed elements are objects: nil (aka null), true and false, intergers, errors, stacktraces, closures, method implementations, classes, metaclasses and even the system itself.

The live programming experience is achieved by supporting advanced debugging techniques: on-the-fly inspection and immediate objects identity swapping during runtime. The latter replaces all references to the old object in the running environment to the new object. E.g. create or change methods on the fly and receive immediate feedback of changes.

The reflection nature exposes to the programmer every object in the system, which can be examined and changed. E.g. fast objects enumeration of all existing instances of a particular class can aid to detect memory leaks, or customize **metaclasses**, which are classes of classes, to change the behavior of the system. The IDE, VM and compiler are extensively written in Pharo itself, and the language has **garbage collection** mechanism for memory managament. The language is **duck typed**, meaning that the type of an object is determined by its behavior, and not by its inheritance hierarchy. **Image based persistence** in Pharo leverages the reflection capabilities of the language, by storing the system state in a platform independent file image format. For example, during a debugging breakpoint, you can save the image, experiment with code changes, and if unsuccessful, reload the image to revert to the previous state, allowing you to seamlessly resume the debugging session later.

It breaks the traditional development paradigm: code, compile, run, debug. Basically, Pharo has all these capabilities for the ability of the system to evolve without the need to restart the application, thus avoiding to interrupt the development flow.

Moreover, the versatility of the language allows to create other languages: the focus is completely shifted to the problem domain and not the complexity of the language itself, allowing to create **Domain Specific Languages** (DSLs) following the **Domain Driven Design** (DDD) principles.

Traditionally, the smalltalk legacy introduced the world to the language virtual machine (VM), which allow software to be platform independent. Recent languages like Java and C# adopted this concept. Smalltalk pioneered JIT (Just-In-Time) compilation, a technique for improving the performance of bytecode software such as Java. It also introduced the concept of **Integrated Development Environment** (IDE) in a GUI windowed system with a mouse, including a text editor, a system class browser, object inspector and debugger. In 1979, a visionary 24 years old Steve Jobs, the founder of Apple, took a look at what Xerox PARC had done with Smalltalk [7]:

> "[...] I was so blinded by the first thing they showed me, which was the graphical user interface [...] and within 10 minutes, it was obvious to me that all computers would work like this someday."

Despite being a powerful tool, it does not mean it will prevent undisciplined coders from making a large mess very quickly. Still, the Pharo environment and language allow to implement elegant purely object oriented solutions, which is the main goal of the project.

2.4 Graphs properties

Some properties borrowed from graphs will be useful to understand the complexity of the visualization algorithms later on.

A **graph** is a mathematical structure consisting of a set of objects, called **vertices**, and a set of connections between them, called **edges**. It is defined as in Formula 3.

$$G = (V, E) \quad \text{where} \quad V \text{ is a set of vertices and } E \text{ is a set of edges} \tag{3}$$

A graph can be either directed or undirected: implying whether the edges have a direction or not. A graph can be either connected or disconnected. A connected graph is a graph in which there is a path between every pair of vertices, whereas a disconnected graph is a graph in which there is at least one pair of vertices for which there is no path between them.

A complete graph is a connected graph in which there is an edge between every pair of vertices, defined as in Formula 4.

$$K_n = (V, E) \quad \text{where} \quad V = \{v_1, v_2, \ldots, v_n\} \quad \text{and} \quad E = \{(v_i, v_j) : v_i, v_j \in V, i \neq j\} \tag{4}$$

The **number of edges** in a complete graph is given by the Formula 5.

$$E = \frac{n(n-1)}{2} \quad \text{where} \quad n = |V| \tag{5}$$

3 Project requirements and analysis

3.1 Requirements

After studying the state of the art of treemaps and Voronoi diagrams to understand the context of the problem world through a **top-down analysis**, a machine solution can be planned.

The **system-as-is** is the current state of visualization regarding our problem space shipped by Pharo and offered by the Roassal visualization framework: several examples of charts, plots, shapes, layouts, UI components and features as animations, event handling and interaction capabilities. The problem discussed in the motivation in Section 1 is the absence of an (interactive) Voronoi diagrams implementation in the Pharo ecosystem. Thus, the opportunity to fill this gap arises and lead to gather the domain knowledge necessary from documentation and expertise in the field from my advisors, concerning literature and the community.

The **system-to-be** objective is the implementation of an interactive purely object-oriented implementation of Voronoi diagrams in Pharo exploiting the Roassal visualization framework and the reflection of the language to offer inspection capabilities to the user.

Assuming no constraints of hardware and scalability, the **system requirements**, i.e. the prescriptive statements formulated in terms of environmental phenomena of the domain, are as follows:

- to visualize diagrams.
- to visualize diagrams interactively.
- to visualize Voronoi diagrams interactively.
- to visualize Voronoi treemaps interactively.

From Section 2, the **domain properties**, i.e. statements of the problem world expected to hold regardless of the system, are inferred and listed. From the Voronoi diagram definition: a system that presents a set of sites in the plane, which is divided into regions according to the nearest-neighbor rule. In addition, the treemap property: a tree is a connected acyclic graph with a root node, where each node has a parent, except the root, and children, except the leaves. The latter implies that the model can be safely implemented as a natural recursive data structure.

The **software requirements** found, i.e. statements to be enforced only by the software-to-be, are listed as follows:

- **Functional requirements**
  - create a voronoi site and display it on a canvas
  - given a collection of voronoi sites, display them on a canvas
  - generate from a treemap the following voronoi sites
  - given a collection of voronoi sites, compute its graph edges and visualize them
  - given a collection of voronoi sites, compute the bisectors from their edges and visualize them
- **Non-functional requirements**
  - Quality requirements
    - follow an object-oriented design
    - leverage design patterns
    - follow style guidelines suggested from Pharo IDE
  - Compliance requirements
    - the system must be developed entirely in Pharo, no external libraries in any other languages can be used, e.g. no Foreign Function Interfaces (FFIs) or serving an already computed input model from another language
    - do not use any previous implementation of Voronoi diagrams in Pharo
  - Architectural requirements
    - the system has a monolith architecture (inherently from single Pharo image instance)

4 Project design

4.1 System architecture

The system architecture is a monolith image-based instance divided in three main packages: **Model, View and Controller**. The Model package contains the Voronoi diagram model used to represent the diagram entities and its geometrical properties. The View package contains the visual representation objects of the diagram entities, e.g. a voronoi site will have its voronoi respective view object to represent it on a canvas. The Controller package contains the objects that will orchestrate the interaction between the model and the view, in order to perform computation and providing inspection insights, e.g. the object handling the right-click event triggering a context menu of a voronoi site view object.

The Model-View-Controller (MVC) architectural pattern is used to separate the concerns of the system. Smalltalk-79 introduced MVC as a way to separate the domain logic from the user interface. The model is responsible for managing the data and core functionality of the system. The view renders a representation of the model to the user in a particular format, in this case, shapes in a canvas. The controller handles user inputs to perform interactions on the model objects.

In Figure 8, the UML class diagram of the system was created using Roassal which leverages the reflection of the system, and the code snippet used is provided in Listings 19.

4.2 Diagram Visualizer

Roassal is a visualization engine for Pharo used to showcase the model and its properties in a visual representation. Every view object, which is a counterpart of a diagram entity model object, has an associated **glyph** attribute. A glyph is intended as the low level Roassal shape visual representation of a diagram entity which can be displayed in the canvas, e.g. a circle, a polygon, a segment, etc. Every view object uses composition, not inheritance, to encapsulate the glyph attribute and its behavior. The reason for it is to allow to maintainbly change the glyph representation of a view object without freezing it to a certain shape or behavior, e.g. a voronoi site view object could be sometimes represented as a circle or a square. To edit the glyph representation of a view object from within the system, i.e. excluding the user interaction handled through the controller, the view object provides wrapping methods of the original Roassal Shape API, and the system communicates with the glyphs only through the view objects.

Assuming the diagram has an outer rectangular bounding polygonal area, the following visualizations are proposed.

In Figure 9, the visualization of a thousand sites is showcased in a bounded region. The code used to generate the visualization is provided in Listings 1.

```
diagram := VDiagram new.
diagram addRandomSites: 1000 between: (1 to: 200).
diagram show
```

Listing 1. Snippet to visualize a thousand sites

In Figure 10, the visualization of all the possible edges of a hundred sites is showcased. The code used to generate the visualization is the same as in Listings 1, with a lowered sites count to ease the computation by recalling from Section 2, that the edges count of a complete undirected graph is $\frac{n(n-1)}{2} = \frac{100(100-1)}{2} = 4950$ where n is the number of sites, i.e. vertices.

In Figure 11, the visualization of all the possible edges and bisectors of a hundred sites is showcased. First with the edges overlayed with the bisectors, then only the bisectors are displayed. These kind of setting is interactive through a context menu furtherly explained in Section 5.

Without loss of generality, we will switch to a more simplified case with only 3 sites in the plane, from an example available online by Ian VanderSchee [14], displayed in Figure 12 with the snippet code to recreate it provided in Listings 20. The image displays the edges and bisectors, with the possibility to interactively ask for the intersections between the bisectors as shown on the right.

From this point, we can ask the system to hide the unrelevant information for the diagram, i.e. the edges, by selecting it from the context menu later explained. The result will left us with the first image of Figure 13, contains the bisectors, their intersections between each other and the bounding region. The second image is obtained by hiding the bisectors and requesting to see only the segments obtained by segmenting the bisectors along their intersections.

Finally, in Figure 14, the system allows to get the final voronoi diagram for the correspondent sites: by following the *nearest neighbor rule*, the bisector generating the intersection is bounded to 2 sites, and when

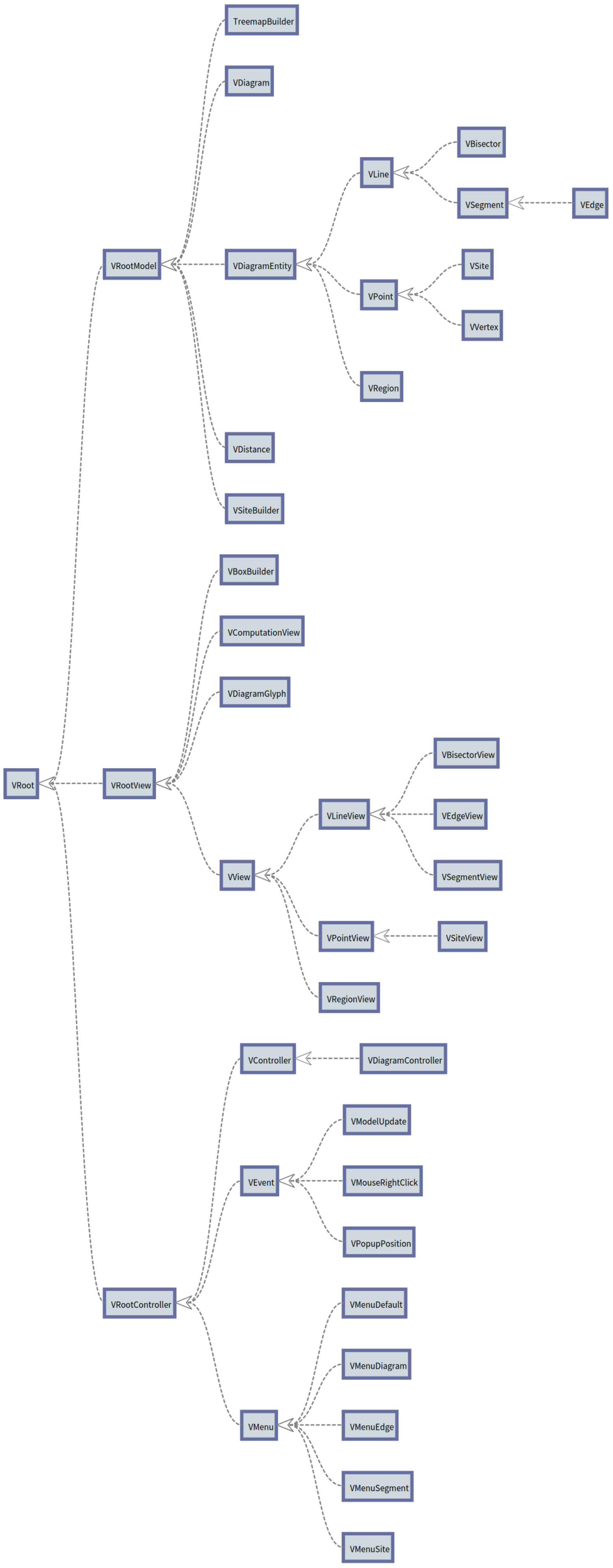


Figure 8. UML diagram of the system

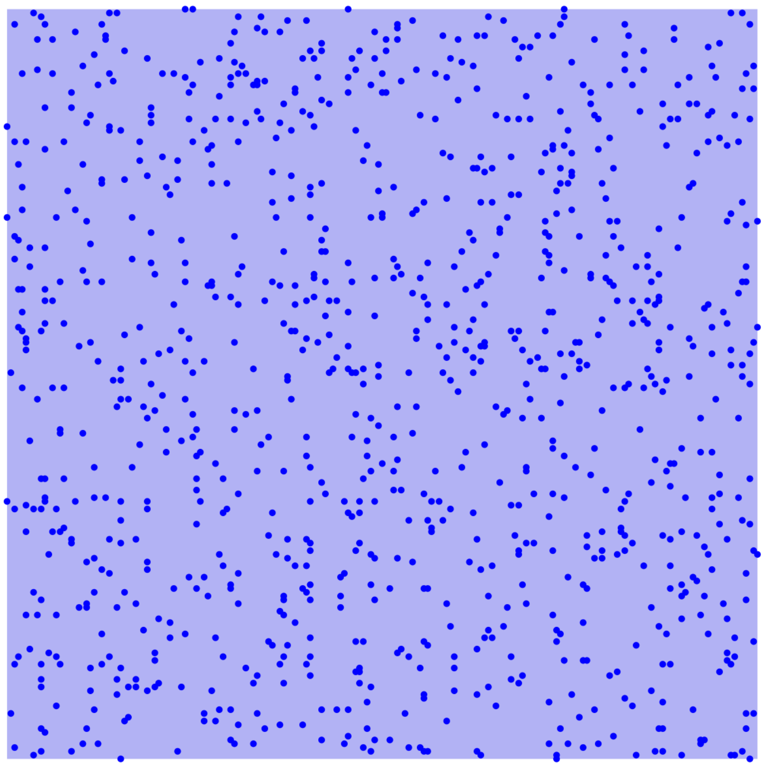


Figure 9. System showcasing a thousand sites

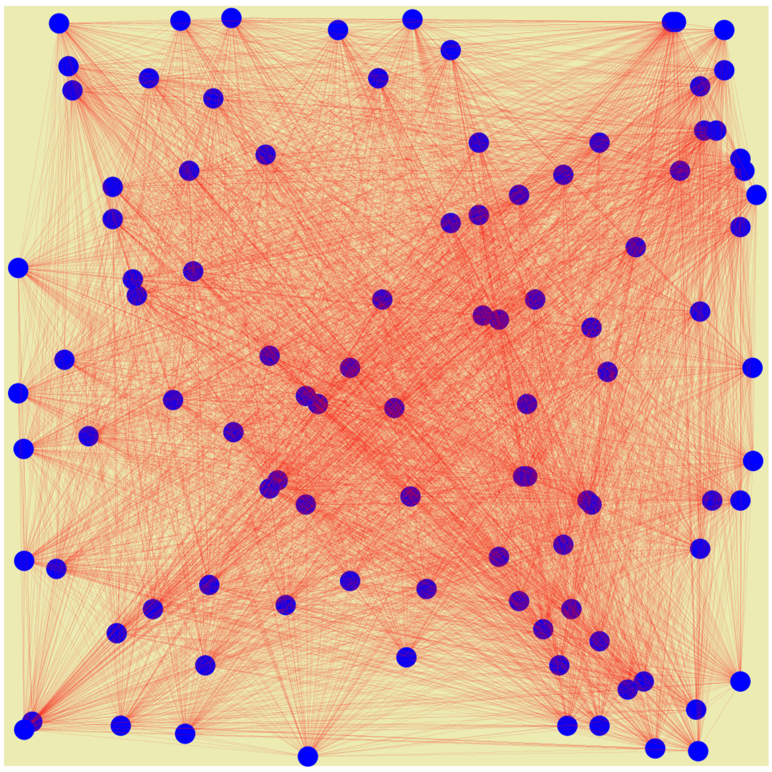


Figure 10. System showcasing all possible edges for a hundred sites

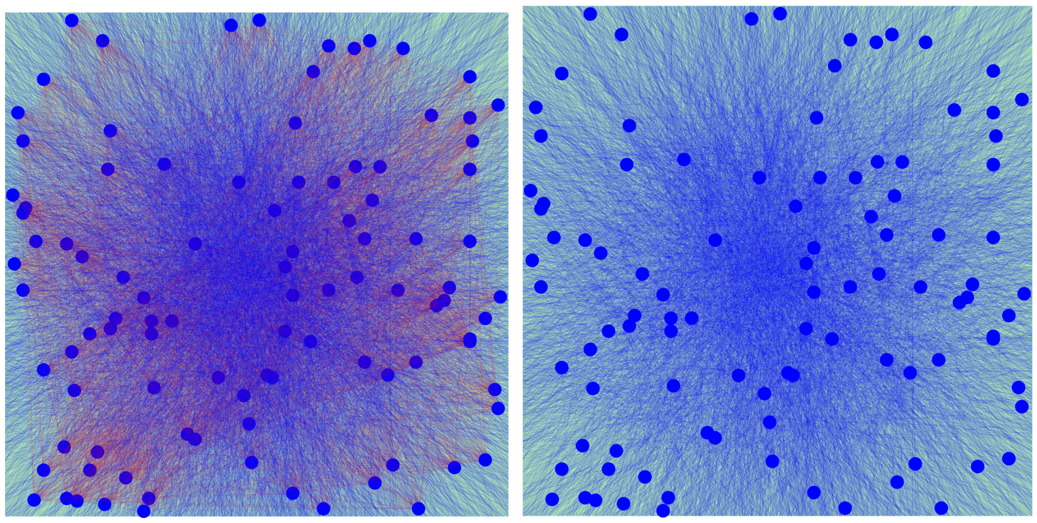


Figure 11. System showcasing all possible edges and bisectors for a hundred sites

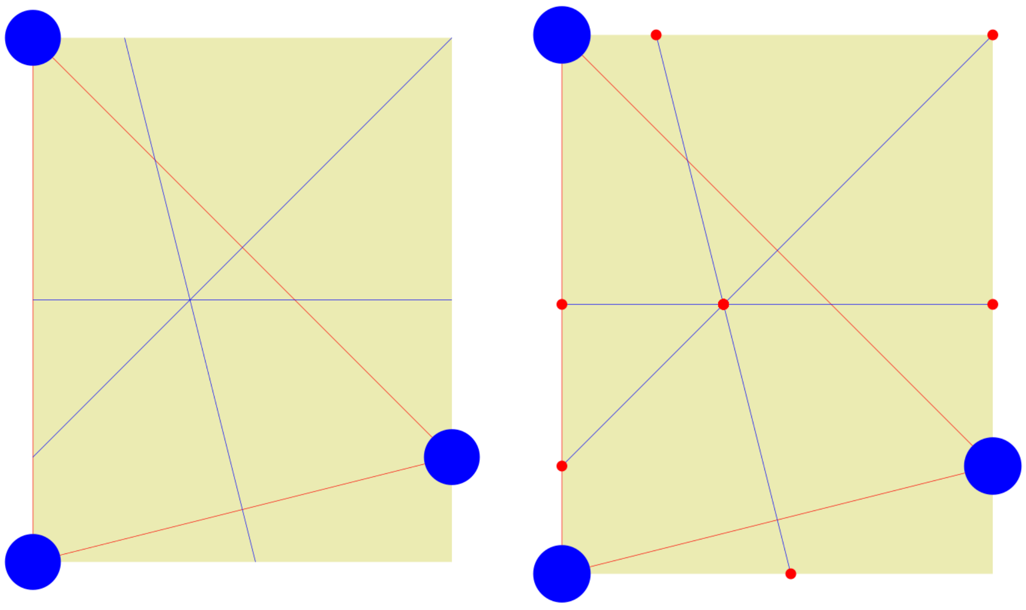


Figure 12. System showcasing edges, bisectors and intersections from online example [14]

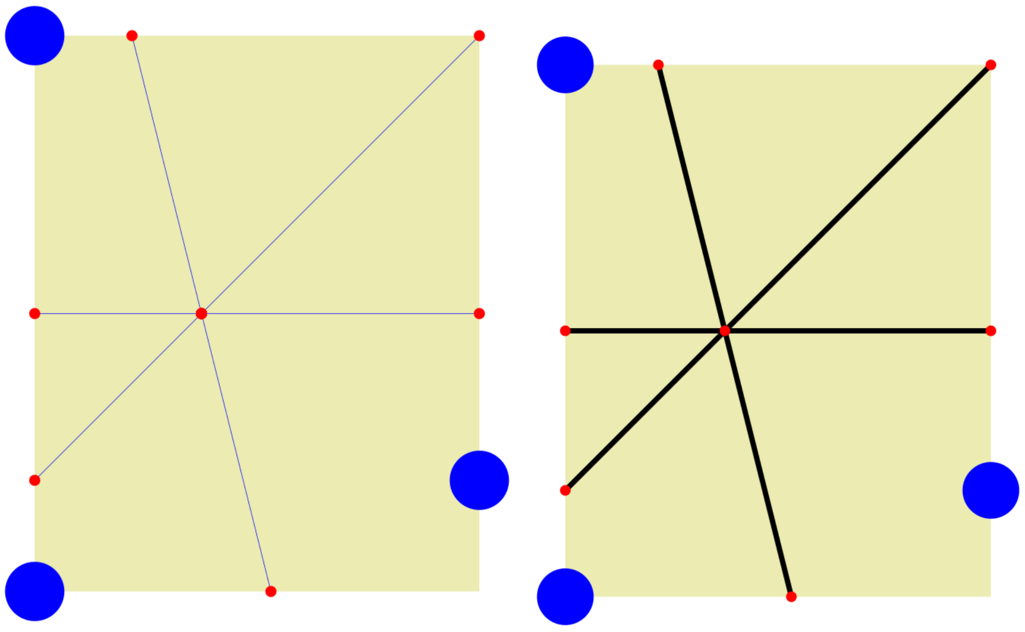


Figure 13. System showcasing bisectors ⊕ segments from online example [14]

confronted with another external site, if they are near to this third unbinded site then is considered not needed and deleted.

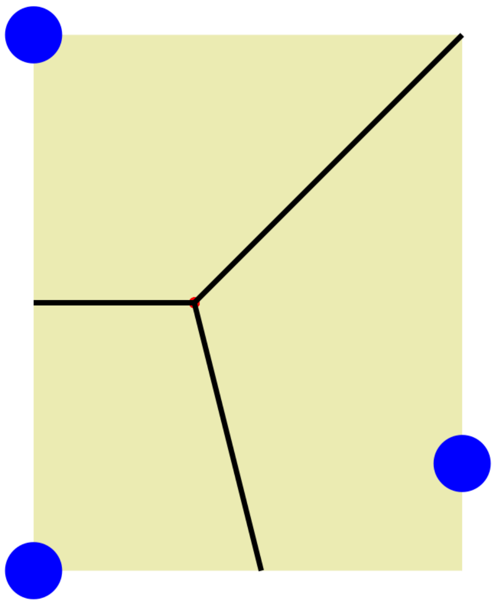


Figure 14. System showcasing the Voronoi diagram from online example [14]

Further examples of Voronoi diagrams for are provided in Figure (15, 16, 17). These examples have been generated from Listing 1 using as number of sites 7, 10 and 20 respectively. Two images for each example are provided: a version with and without the edges in the background, to hint about how the computation and the complexity increases with the number of sites. The intersection points in red are left in both images to show the user where the bisectors intersects and grasps the idea of the amount of segments generated by the intersections.

4.3 User Interaction

The techniques that allow the user to investigate the data and the system are dedicated popup labels, context menus and interactive zooming [10]. Context menus offer custom call-by-need computation for each diagram entity, providing introspection capabilities, and are triggered by right-clicking on a diagram entity over the canvas, besides the diagram menu, for which you need to click outside the diagram area. Examples for each

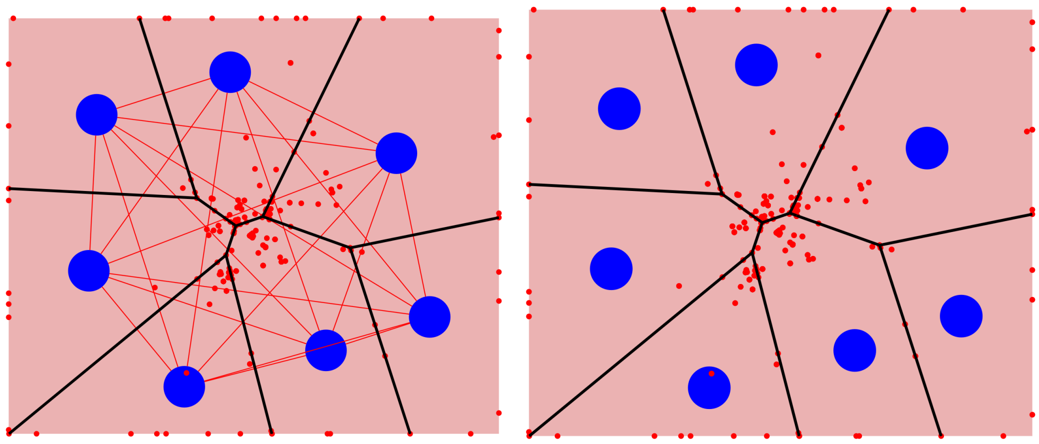


Figure 15. System showcasing a 7 sites voronoi diagram

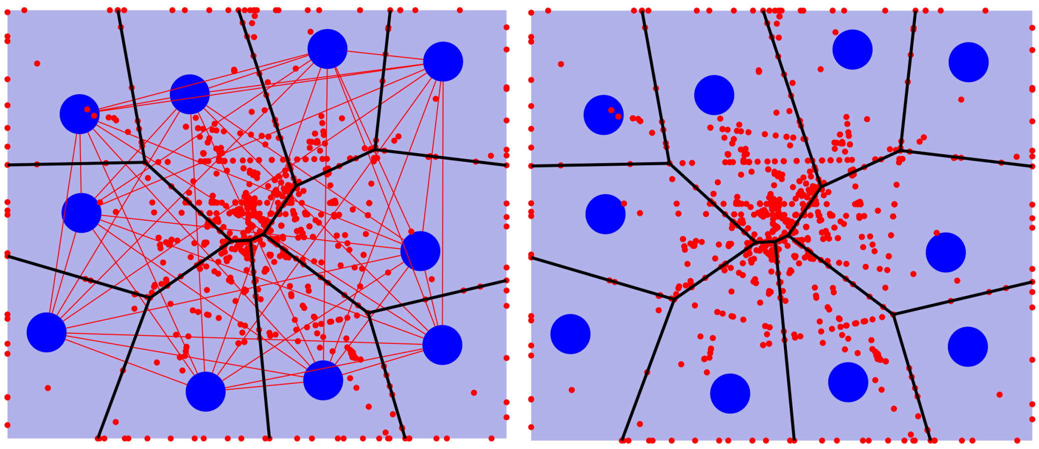


Figure 16. System showcasing a 10 sites voronoi diagram

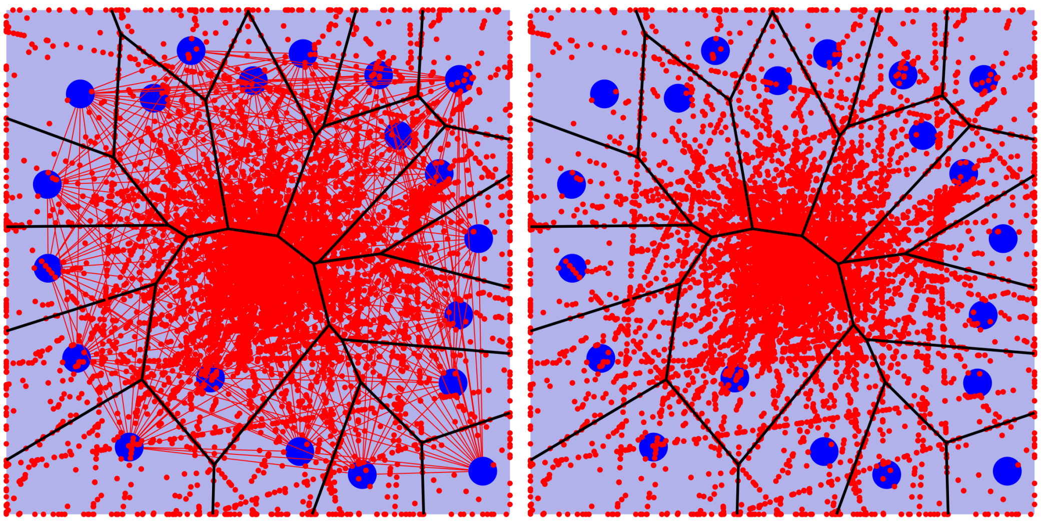


Figure 17. System showcasing a 20 sites voronoi diagram

diagram menu is provided in the computation of the voronoi diagram described the previous section *Diagram Visualizer*. The options in the site regarding the triangulation and circumscribed circle are not yet implemented and furtherly discussed in Section 6.

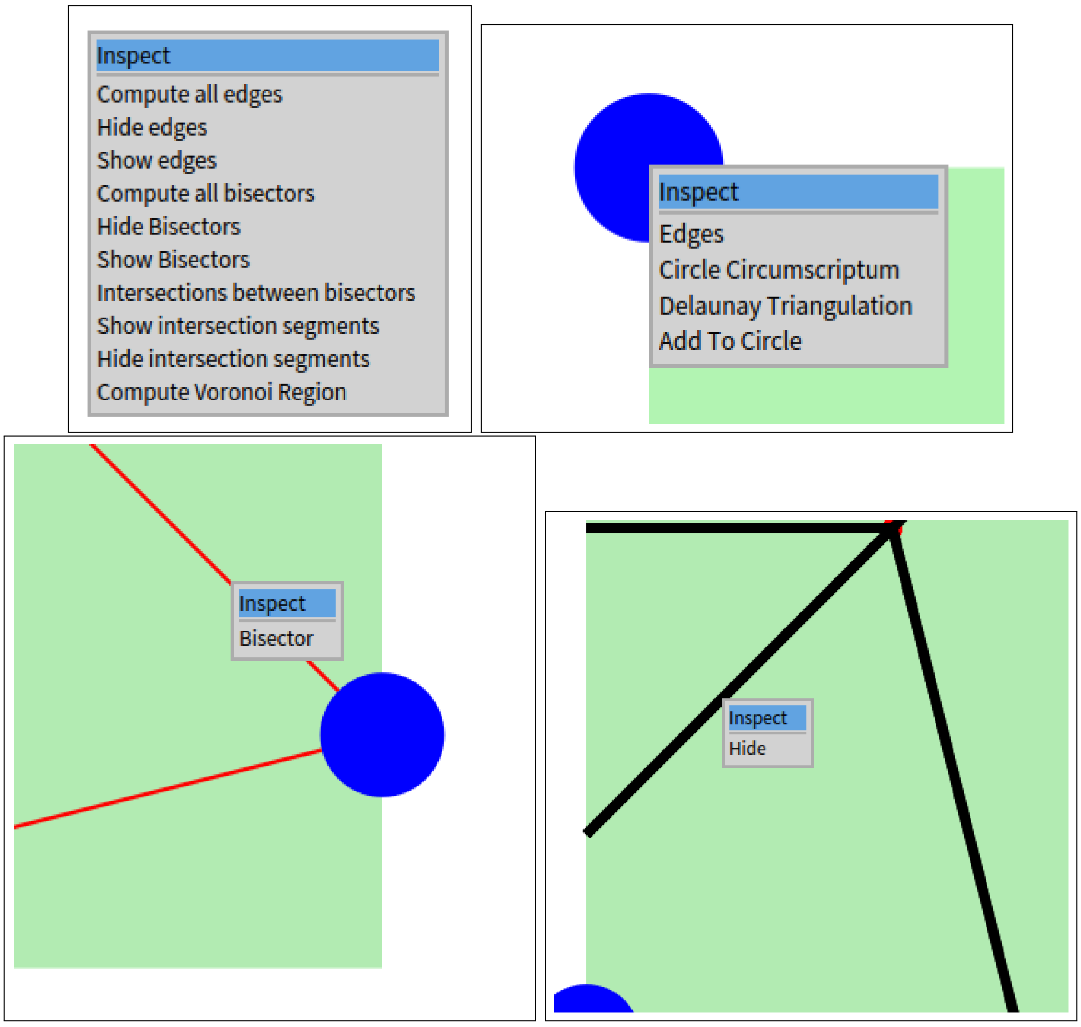


Figure 18. Context menu respectively for diagram, site, edge and segment entities

The menu styles are reported in Figure 18, Moreover, a popup label is displayed when hovering over a diagram point and its subclasses, showing its metadata and properties, as shown in Figure 19. The implementation overrides the *asString* method of the model object, which is then called by the view object to display the label by the Roassal Popup object.

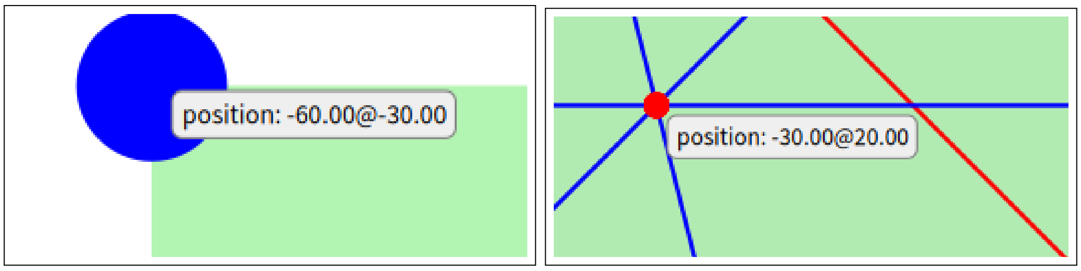


Figure 19. Popup label for site and point entities

5 Implementation

The system was developed using a **bottom-up implementation** strategy. First the most fundamentals classes of the model were spotted and designed. Then, their interaction through message passing was implemented. Builders and design patterns argumented afterwards were used to ease the development process. The visual representation followed the model design, and lastly the controller to allow user interaction was implemented.

5.1 Recursivity of the model

As mentioned in the requirements Section 3, the treemap is a natural recursive data structure. The Voronoi treemap has then to be a recursive data structure as well. Suppose only a single level of hierarchy, then the voronoi diagram would be a set of sites with its corresponding set of regions. In the general treemap case, there are as many diagram as the number of non leaf nodes in the tree. In Figure 20, the initial perspective would be seeing the root node as the diagram point of view and its sublayer children as the sites with regions displayed. E.g. the root node A would be a diagram with sites B, C, D, E, F, G. This is done recursively for each node which is not a leaf node in the tree. So the remaining diagrams would be from C visualizing H, from D displaying I, J, and from G showing K.

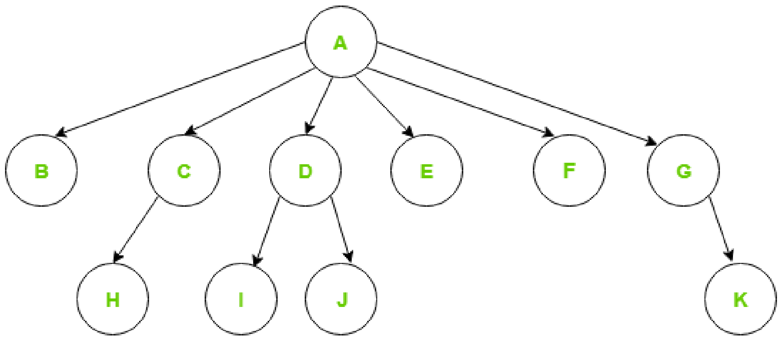


Figure 20. Generic N-ary tree

The diagram entity model *VRegion* has been designed to be a recursive data structure, where it has a *regions* attribute which is a collection of *VRegion* objects stricly mirroring the treemap model provided, for which you can compute the Voronoi diagram of the sites it contains over the defined polygonal area of the parent region.

5.2 Evolution of the Model

The design of the model underwent several evolutionary stages, each marked by a distinct paradigm: initially, the system was implemented in a algorithmic imperative style, as it was described in the academic pubblications. Such strictly top-down approach lead to a *playground* script based system with low maintainability, adaptability and object reusage due to its rigid linear structure.

To mitigate the issue, the system switched to a class-side procedural approach, which encapsulated the diagram entities in classes, but most of the methods were class-side resembling the signature of the algorithmical approach, leading to an **anemic model** in the object. Hence, despite the architectural improvement, object oriented principles were underutilized as the system relied in static methods without leveraging the potential of message polymorphism and objects inheritance.

Finally, with the guidance of my advisors, the system was refactored to an object-oriented approach, where methods and properties were encapsulated within the respective entities on the instance side, breathing life into the objects and making the model more expressive. The system was clearly divided in 2 main packages already: core and visualization. Still, the model was hard to use as many interactions between entities were verbosely long, e.g. to visualize a site, a canvas must exist, that concern is delegated to the view part of a diagram which must then exist, and so on. To allow a single site to be visualized, I needed it to not rely on any other entity than *itself*. As discussed later in Section 5, this led to a MVC architecture and the employment of design patterns, which vastly increased the adaptability and growth of the system.

The capabilities of the system at this points are shown in Figure 21. Beware, the algorithmical part of the geometrical properties of the Voronoi diagram were not implemented yet.

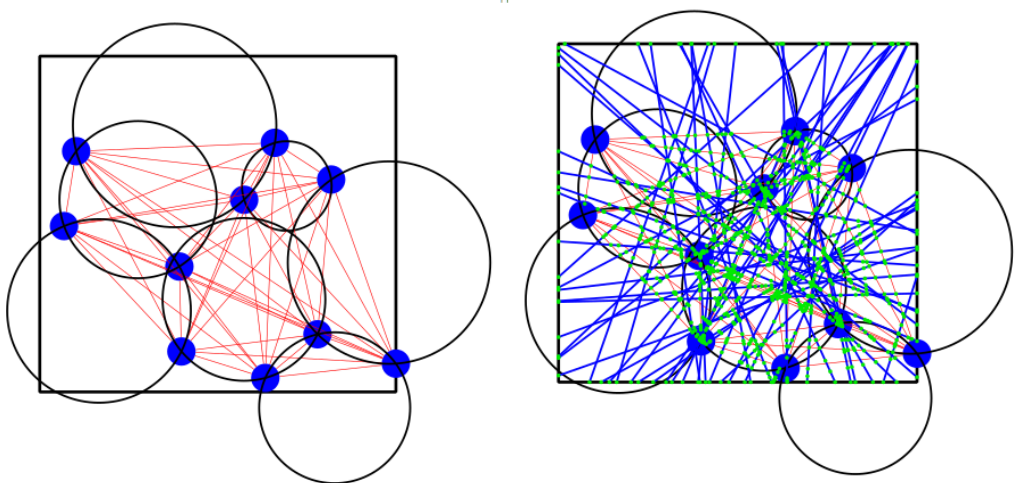


Figure 21. Showcase of diagram visualization with edges and circumscribed sites

5.3 Design Patterns

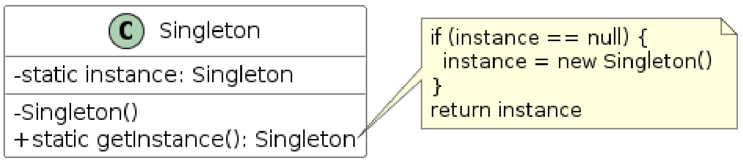
Historically, the evolution of design patterns is closely related with the rise of mainstream object oriented languages. The adoption of design patterns is profound in object oriented programming, where they provide a standardized approach to solve common design challenges and offering an high level abstraction to communicate complex design concepts. They are not solutions, but rather guidelines to design system architecture, promote code reusability and OOP principles.

5.3.1 Singletons in a live environment

Ensure a class only has one instance, and provide a global point of access to it [5]. Part of the Creational patterns, as shown in Figure 22.

Each diagram entity has its corresponding subclass from *VMenu* presenter to display the context menu when right-clicking on it. To ease scalability of the diagram visualizer, I used a singleton pattern to instantiate only one object per each Menu presenter, e.g. menu for sites, menu for edges, menu for bisectors, etc, as shown in Listings 2. In this way, the menu depending on the entity clicked only changed the diagram entity glyph, which is the receiver of the menu action, and the canvas, in case there is more than one diagram visualizer open, as shown in Listing 5.

The constructor of every singleton is private to avoid instantiation of the object from outside the class, as shown in Listing 6. When changing the source code of the menu presenter, since pharo is a live environment I needed to create a class side method to retrieve the instance from every subclass of the menu presenter and then explicitly every set it to *nil* in order for the garbage collector to dispose them, as shown in Listings (3,4). Thus, allowing a new instance to be created with the new source code.



```
if (instance == null) {
  instance = new Singleton()
}
return instance
```

Figure 22. UML class diagram for singleton pattern

```
getInstance
    ^ uniqueInstance ifNil: [ uniqueInstance := self basicNew initialize ]
```

Listing 2. Singleton get accessor in VMenu

```
delete
    uniqueInstance := nil
```

Listing 3. Dispose singleton instance of VMenu

```
deleteAll
    self allSubclassesDo: #delete
```

Listing 4. Dispose all singleton instance of VMenu subclasses

```
menu
    | menu |
    menu := VMenuSite getInstance.
    menu glyph: self glyph.
    ^ menu
```

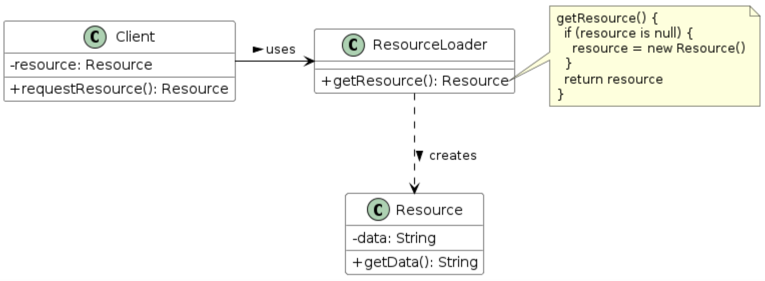
Listing 5. Singleton menu presenter retrieved VSiteView

```
new
    ^ self error: 'singleton, use getInstance'
```

Listing 6. Singleton class-side constructor in VMenu

5.3.2 Lazy Initialization

In Pharo the allocation in memory of an object is done by the VM, allowing us to use data structures that do not need to be initialized. The lazy evaluation is a strategy that delays the evaluation of an expensive expression until its value is needed, avoiding repeating evaluations, as shown in Figure 23. In the project, lazy initialization tactic is implemented to delay the instantiation of an object until the first time it is needed. It uses a special marker value (usually null) to indicate a field isn't loaded. Every access to the field checks the field for the marker value and if unloaded, loads it [4]. It is used for every view counterpart of each diagram entity, which is the expensive object handling its visual representation, as shown in Listing 7. Every diagram entity has a view attribute that is initialized lazily when it is visualized on a canvas, i.e. when the view getter is called. An example of its usage is provided in the diagram controller *updateView* method, where every site is loaded up in the canvas using their correspondent glyphs stored in their entity view, thus updating the current scenario, as shown in Listing 8.



```
getResource() {
  if (resource is null) {
    resource = new Resource()
  }
  return resource
}
```

Figure 23. UML class diagram for lazy loading pattern

```
view
    view ifNotNil: [ ^ view ].
    view := VSiteView newFrom: self.
    ^ view
```

Listing 7. Lazy initialization of view getter in VSite

```
...
siteGlyphs := self model sites collect: [ :site |
| siteGlyph |
siteGlyph := site glyph. "HERE"
siteGlyph radius: diameter / 2.
siteGlyph ].
...
```

Listing 8. Usage of lazy initialization in VDiagramController

5.3.3 Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations. [5]. Part of the Creational patterns, its commond UML class representation is shown in Figure 24.

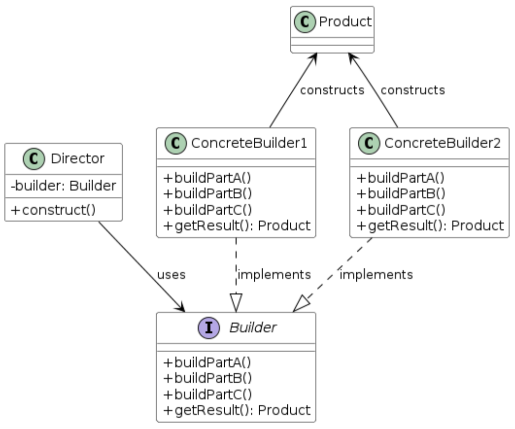


Figure 24. UML class diagram for builder pattern

The implementation of builder pattern was used to create an immediate process to instantiate multiple sites randomly sparsed within the diagram. It was also used to create a 1 layer treemap. Regarding the former, the site builder is initialized to not create any site yet with a range from $[1, 100]$ to spawn them randomly, as shown in Listing 9. It is possible to set the *diagram, interval and number of sites* that the builder needs to create. Then, the lazy *build* method is called to create the sites, where the range is iterated to create them and the randomly picked position from the interval and the diagram are set to each site. as shown in Listing 10. An example of its usage happens within the diagram, which offers an method to add a number of sites between a given interval, as shown in Listing 11.

```
initialize
    super initialize.
    numberOfSites := 0.
    interval := 1 to: 100
```

Listing 9. VSiteBuilder initialization

```
build
    | foreach sites |
    foreach := 1 to: self numberOfSites.
    sites := foreach collect: [ :i |
    | site |
    site := VSite newFromRandomPositionIn: self interval.
    site diagram: self diagram.
    site ].
    ^ sites
```

Listing 10. VSiteBuilder lazy build method for sites

```
addRandomSites: aNumberOfSites between: anInterval
    | builder vPoints boundingPoly |
    builder := VSiteBuilder new: aNumberOfSites.
    vPoints := builder
    interval: anInterval;
    diagram: self;
    build.
    vPoints do: [ :s | self addSite: s ].
    boundingPoly := GRectangle origin: 0 @ 0 corner: vPoints max.
    self rootRegion polygon vertices: boundingPoly vertices.
    ^ boundingPoly
```

Listing 11. Usage of site builder within VDiagram

5.3.4 Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure [5]. Part of the Behavioral patterns, its common UML class representation is shown in Figure 25.

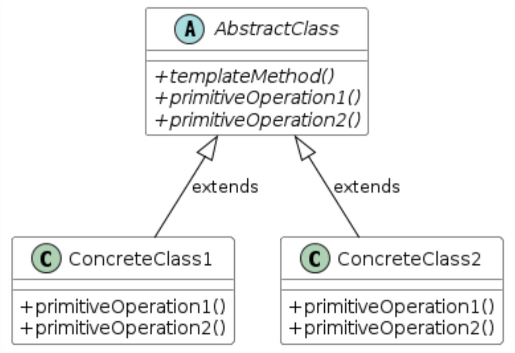


Figure 25. UML class diagram for template method pattern

The implementation of the template function was used to handle menus for each view object using the *menu* attribute. The *VView* class declared it as an abstract method delegating its implementation resposibility to its subclasses, as shown in Listing 12. Subclasses, e.g. VSiteView, implemented this method by calling the right menu, as shown in Listing 13. The menu object is a previous mentioned singleton object setting the current glyph of the menu to the glyph of the current viewed object.

```
menu
    ^ self subclassResponsibility
```

Listing 12. Template method for menu

```
menu
    | menu |
    menu := VMenuSite getInstance.
    menu glyph: self glyph.
    ^ menu
```

Listing 13. VSiteView menu

5.3.5 Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [5]. Part of the Behavioral patterns, its common UML class representation is shown in Figure 26.

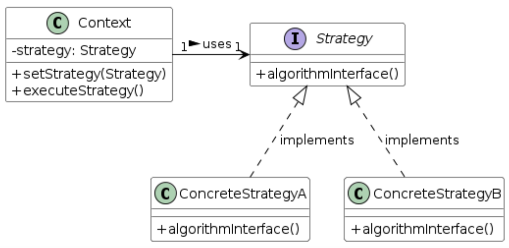


Figure 26. UML class diagram for strategy pattern

I implemented a strategy pattern to model the distance class called *VDistance*, containing algorithms that could be used for the computation of the diagram.

The distance object has only the *method* attribute. The method attribute is assigned to a block that takes 3 arguments: two VSite objects and a weight number. The snippet in Listing 14 shows the power weighted algorithm. In Listing 14, the method attribute is set to the power weighted algorithm at object initialization using the *useDistance* message. Finally, to compute the distance between two sites, the *distanceFrom:to:weight:* message is sent to the distance object. Optionally, if using the *distanceFrom:to:* message, the weight is set to nil, and the algorithm is called with weight value that will lead to the simple weightless distance. Listing (16, 17) shows the computation of the distance. A usage example of such object is shown in Listing 18.

```
powerWeighted
    ^ [ :aVSite :anotherVSite :anOptWeightNumber |
    | euclidean aWeight |
    aWeight := anOptWeightNumber ifNil: [ 0 ].
    euclidean := aVSite position distanceTo: anotherVSite position.
    euclidean - aWeight ]
```

Listing 14. Power Weighted distance

```
initialize
    super initialize.
    method := self useDistance: self powerWeighted
```

Listing 15. Initialization of the distance object

```
distanceFrom: aVPoint to: anotherVPoint weight: aWeightNumber
    ^ self method value: aVPoint value: anotherVPoint value: aWeightNumber
```

Listing 16. Computation of the weighted distance

```
distanceFrom: aVPoint to: anotherVPoint
    ^ self distanceFrom: aVPoint to: anotherVPoint weight: nil
```

Listing 17. Computation of the unweighted distance

```
distance := VDistance new distanceFrom: aSite to: anotherSite
```

Listing 18. Usage of the distance object

6 Conclusions, Limitations & Future Work

6.1 Limitations

The system is constrained at a single level of hierarchy in treemaps, meaning only voronoi diagrams can be visualized. Despite recursion is supported in the model, it is not implemented for the remaining parts.

Test-Driven Development (TDD) in Pharo promotes writing tests before the code implementation by allowing developers to create tests before the functionality even exists. By running the tests, an exception about the missing functionality is raised, and from within the exception window, we can start the implementation and continously re-run only the part of tests that are failing, until it passes, exploiting the reflective nature of the language argumented in Section 2. While Pharo's rapid development cycle accelerates a new codebase creation, this fast pace with frequent changes in the model would have make a strictly TDD approach challenging to maintining the test coverage. As it may be difficult to keep tests aligned with the evolving architecture, specially for newcomers to the language and the framework.

The reflection capabilities of the language can be intimidating for newcomers, as it allows to change the system in a way that could potentially break it, e.g. editing a common named message over the wrong scope.

Triangulations are not automatically implemented, but as seen in Figure 21, the system is capable of visualizing the circumscribed circles of a triplet of sites by manually using the context menu discussed in Section 4.

6.2 Future work or Possible Developments

I see much potential in spreading the adoption of Voronoi treemaps in industry if incorporated within the Continuous Integration part of the operation processes in the modern software development lifecycle of the devops methodology. Plugins for SonarQube could be developed and the progressive computing nature of the Voronoi diagrams allows for incremental computation as the codebase grows. Such incremental steps, which could be associated with e.g. commits [12], could be saved in artifact repositories. This could potentially address the before mentioned proposal.

Otherwise, it would be interesting to explore the possibility to integrate such visualization tool in mainstream IDEs e.g. Visual Studio Code or IntelliJ IDEA through plugins. Granting the user a live changing diagram would be a great asset.

6.3 Conclusions

In this document, I reported the development of a package in Pharo aiming to visualize Voronoi diagrams in a treemap context. Understanding the context of the problem was crucial to design the model, and building a background of the previous work in the field was a great asset. The problem was tackled with multiple approaches, and interesting design implementation choices were chosen to make the system more scalable and maintainable. Pharo as a language is a great tool providing immense introspection capabilities, which combined with the Roassal visualization engine, allowed to create a powerful system to visualize diagrams. Such power can be used to create elegant solutions to complex problems, but can also be intimidating when so much freedom and power is given to the developer. This is a positive feedback for our initial intuition and a good motivation for future work.

Appendices

A Toolkits

Hereby, it is provided all the code that is not strictly related to the implementation of the system, but was used to create the visualizations and the UML diagrams about the technologies used.

PlantUML scritps used to generate the UML class diagrams for the design patterns are available at the GitHub repository of the thesis, on the `latex/images` folder.

The Pharo script used to generate the system UML custom class diagram was partly obtained by the script in Listings 19, and partly by modifying the image code to remove the attributes and methods of the classes.

```
builder := RSUMLClassBuilder new.
builder classes: VRoot withAllSubclasses.
marker := (RSShapeFactory arrow
extent: 20@25;
noPaint;
withBorder) asMarker offset: -7.
builder renderer edgeBuilder: (RSLineBuilder horizontalBezier
width: 2;
dashArray: #(4);
capRound;
attachPoint: (RSHorizontalAttachPoint new startOffset: 20);
markerStart: marker).
builder layout horizontalTree
verticalGap: 50;
horizontalGap: 100.
builder build.
builder canvas open
```

Listing 19. Snippet to create class diagram of the system

The Pharo script to reimplement the online example provided from Ian VanderSchee [14] is the following:

```
diagram := VDiagram new.
vA := VSite newFrom: (-6 @ 7) * 10.
vB := VSite newFrom: (-6 @ -3) * 10.
vC := VSite newFrom: (2 @ 5) * 10.
boundPoly := GPolygon vertices: { vA . vB . vC }.
boundRect := boundPoly encompassingRectangle.
diagram rootRegion polygon vertices: boundRect vertices.
diagram addSite: vA; addSite: vB; addSite: vC.
diagram show.
```

Listing 20. Snippet to recreate the online example [14]

References

[1] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, sep 1991.

[2] M. Bruls, K. Huizing, and J. J. van Wijk. Squarified treemaps. In W. C. de Leeuw and R. van Liere, editors, *Data Visualization 2000*, pages 33–42, Vienna, 2000. Springer Vienna.

[3] P. Consortium. Pharo, 2024.

[4] M. Fowler. Lazy load, 2002.

[5] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[6] L. Jin and D. C. Banks. Tennisviewer: A browser for competition trees. *IEEE Comput. Graph. Appl.*, 17(4):63–65, jul 1997.

[7] S. Jobs. The lost interview, 1995.

[8] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proceeding Visualization '91*, pages 284–291, Oct 1991.

[9] A. Jungmeister. Adapting treemaps to stock portfolio visualization. *UMD HCIL*, 03 1991.

[10] C. L. M. Balzer, O. Deussen. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 165–172, New York, NY, USA, 2005. ACM.

[11] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. *IEEE Symposium on Information Visualization, 2001. INFOVIS 2001.*, pages 73–78, 2001.

[12] D. P. Tua, R. Minelli, and M. Lanza. Voronoi evolving treemaps. *2021 Working Conference on Software Visualization (VISSOFT)*, pages 1–5, 2021.

[13] J. Van Wijk and H. Van de Wetering. Cushion treemaps: visualization of hierarchical information. In *Proceedings 1999 IEEE Symposium on Information Visualization (InfoVis'99)*, pages 73–78, 1999.

[14] I. VanderSchee. Constructing voronoi diagrams, 2020.

[15] M. Wattenberg. Map of the market, 1998.

[16] M. Wattenberg. Visualizing the stock market. In *CHI '99 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '99, pages 188–189, New York, NY, USA, 1999. Association for Computing Machinery.