

The Tree Visualization Application is organized as follows:

The user interface as shown on startup of the application is defined in the app.fxml file which basically describes the tree tabs ‘Data Source‘, ‘Visualization‘ and ‘Tree‘. The JavaFX controller organizing these taps is the AppController.

The user interface of the ‘Data Source’ and ‘Visualization’ tab are defined in a separate fmxl files and organized by the controllers ‘DatasetSelectionController’ and ‘ChartController’.

DatasetSelectionController

The DatasetSelectionController triggers the generation, rendering of the input tree as well as the option to drop nodes above a specified level. The generation of the tree is delegated to the controller corresponding to selected ‘Data Source’. The recent implementation supports four data sources: Directory Tree Generator, Random Tree Generator, UDC Tree Generator, Tree Dump File. For each of those a separate user interface is displayed below the buttons ‘Generate Tree’ and ‘Rendering tree’. This UI allows the user to configure settings of the generation. By example of the Directory Tree Generator, the user is able to select the directory to be visualized. The user interfaces for the settings are created according to fxml files and organized by corresponding controllers which have to implement the java interface IDatasetGenerationController.

The basic control flow works as follows:

1. The user selects a Data source
2. The user configures the generation using the settings e.g. by selecting a directory
3. The user presses the button ‘Generate tree’
   1. The generation is delegated to the selected IDatasetGenerationController
   2. Statistics about the generated tree are displayed in the text view below the settings user interface.
4. The user presses the button ‘Render tree’. This triggers the generation of the hexagon map as explained in the paper ‘Enhanced Hexagon-Tiling Algorithm for Map-Like Information Visualisation’ by Muye Yang and Robert P. Biuk-Aghai. The respective algorithm is implemented in the class Method1. In case, the Regions/Countries should be rendered as one area instead of the rendering of single hexagons, the borders around each region is identified using the algorithm implemented in BorderCreater class.

Afterwards, the rendering of the map triggered.

Rendering of Regions

In order to render regions instead of one hexagon by another, the basic information required are the edges of the Hexagontiles defining border of the regions. Then, the edges can be connected to a path and rendered as the border of the region. Furthermore, this path along the region’s border defines a closed shape which can be filled with a color using the JavaFx framework.

1. **Method1**

For the rendering of regions, the output of the Method1 was extended by the creation of Region and LeafRegion objects.

**LeafRegions** correspond to the Leafs of the tree and are supposed to store Border objects. These Border objects store the edges of the hexagon tiles lying on the border of the region. Since e.g. the level of each border are not known while generation of the map, Method1 stores for each Region the Hexagon tiles lying on the border of the Region. The creation of the actual Border objects is done in the next step by the ‘BorderCreator’.

For the inner nodes of the tree ‘**Regions’** are created storing all ChildRegions just according to the tree.

1. **BorderCreator**

The BorderCreator basically creates Border objects which are then stored in the LeafRegions. Therefore, the output of Method1, which are the Hexagontiles lying on the border of the regions, are analyzed. The BorderCreator orders the edges of Hexagontiles, calculates the level of the edge and collects consecutive edges in one Border object. These Border objects are then stored within the LeafRegions. So after running the BorderCreator, each LeafRegion stores Border objects and each Border objects is a List of connected edges of Hexagontiles. Thereby, it is ensured that two LeafRegions separated by a border also share the same Border object. Thus, for each drawn border, only one Border object exists.

1. **RegionRenderer**

The GUI of the visualization allows the user to select the maximum level of Regions, Borders and Labels to render. This rendering is basically done by the RegionRenderer class. However, the RegionRenderer delegates rendering to the RegionAreaRenderer, RegionBorderRenderer and RegionLabelRenderer. Each of those delegates renders either area, border or the label of one Region, given the concrete shape points of the Region. The rendering of one Region is done in the following order: First, the RegionAreaRenderer will fill the area of the Region with color, then the RegionBorderRenderer renders the respective border as lines or curves of different (according to the level of the border) and finally the RegionLabelRenderer renders the name of the Region as defined in the tree.

The Border objects created by the BorderCreator store the border edges as a tuple of the position of the hexagon (the edges belongs to) and the direction of the edge (N, NE, SE, S, SW, NW). However, the rendering requires the absolute positions of the points on the canvas.

First, the regions to be rendered according to the configuration in the GUI are determined. This configuration is basically stored and organized by either the RampRegionColorStyler or the RandomRegionColorStyler.

* 1. Since only the LeafRegions store the concrete information about its borders to avoid redundancy, Regions (inner Nodes in the tree) are able to acquire the borders of their children recursively. The LeafRegion will then return only those borders which have a level lower or equal to the acquiring Region; because only these borders make up the borders defining of the Region to be rendered. Afterwards, for each of those Borders the absolute points on the canvas is calculated and stored in the **BoundaryShape** object. A BoundaryShape object can be regarded as a segment of the border of the Region which has a certain Border level.
  2. Rendering and filling an area with a color is done by rendering a path going along the entire border of the Region. Therefore, the corresponding BoundaryShape objects have to be ordered in a way, that the last point of a BoundaryShape/bordersegment is the same as the start point of the subsequent BoundaryShape/bordersegment in the ordered List. Furthermore, Method1 may have also generated Regions which have more than one border e.g. in case the Region contains lakes; there are inner and outer borders.
  3. The GUI allows the user to select different types of smoothing algorithms. These are implemented in the classes DirectBoundyShapeSmoother, MovingAverageBoundaryShapeSmoother, SimplifiedBoundaryShapeSmoother. To increase the degree of freedom for the smoothing algorithm, these algorithms first summarize BoundaryShapes of the same level to one BoundaryShape in case the current visualization doesn’t show borders between them (Depends on the maximum level for Regions and Borders).
  4. The **DirectBoundyShapeSmoother** doesn’t change the coordinates of the BoundaryShape. The **MovingAverageBoundaryShapeSmoother** calculates the midpoint between two subsequent border points of a BoundaryShape. The **SimplifiedBoundaryShapeSmoother** smooths the BoundaryShapes using the Douglas-Peucker algorithm. However, each smoothing algorithms ensure that the start- and endpoint of the border segments will be the same as in the original one. Only then other borders with the same start- or endpoint connect correctly. Furthermore, the current implementation stores the result of the smoothing in a map. Thus, after the area/borders of one Region were smoothed, all neighboring regions can reuse the smoothed, shared border segments when rendered.
  5. After the smoothing of the borders of the regions, three different types of rendering methods are available. These are DirectPolylineBoundaryShapeRenderer, QuadraticCurveBoundaryShapeRenderer and BezierCurveBoundaryShaperRenderer.

The **DirectPolylineBoundaryShapeRenderer** takes the smoothed coordinates of the border segments and connects them by straight lines.

The **QuadraticCurveBoundaryShapeRenderer** calculates the point in the middle of two subsequent border coordinates. Two middle points are then connected by a quadratic curve by using the original coordinates as control point.

The **BezierCurveBoundaryShapeRenderer** fits Bezier curves to the border segments. The calculation of the Bezier curves is done as explained in the following article:

<http://scaledinnovation.com/analytics/splines/aboutSplines.html>

Datasets:

Linux Kernel linux-4.2.3:

* Source:
  + <https://www.kernel.org/>
  + Access: 16.10.2015

UDC:

* Source:
  + <http://udcdata.info/>
  + Access 16.10.2015