A Generative Layout Approach for Rooted Tree Drawings

Hans-Jörg Schulz* University of Rostock
Zabedul Akbar† University of Calgary
Frank Maurer‡ University of Calgary

ABSTRACT

In response to the large number of existing tree layouts, generic "meta-layouts" have recently been proposed. These generic approaches utilize layout design spaces to pinpoint a tree drawing with desired characteristics in the wealth of available drawing options and parameters. While design-space-based generic layouts work well for the confined set of implicit space-filling tree layouts, they have so far eluded their extension to explicit node-link diagrams.

In order to produce both, implicit and explicit tree layouts, this paper parts with the descriptive nature of the design spaces and instead takes a generative approach based on operators. As these operators can be combined into operator sequences and be used at different stages of the layout process, a small operator set already suffices to yield a large number of different tree layouts. To this end, we present a generic tree layout pipeline and give examples of suitable layout operators to plug into the pipeline. A prototypical implementation of our pipeline and operators is presented, and it is illustrated with space-filling and node-link examples. Furthermore, the paper presents results from a user study evaluating our generative approach as it is realized by the prototype.

Index Terms: I.3.3 [Computing Methodologies]: Computer Graphics—Picture/Image Generation

1 INTRODUCTION

Tree drawings are a standard diagram type that common visualization tools are expected to support. Yet, tree drawings subsume an entire family of diagrams with more than 250 variants of them [23] that can be generated through a variety of different algorithmic approaches [17, 22]. So on the one hand, the implementation of a substantial number of different layout algorithms is necessary to provide users with common layouts to draw their hierarchical data. On the other hand, a user's individual needs can hardly ever be fulfilled as each new hierarchical dataset may require a new layout approach for its adequate representation.

To counter this problem, more generic layout approaches have recently been proposed. They are able to produce a variety of concrete tree layouts according to their parametrization. While sometimes not explicitly stated, the existing generic layouts all rely on an underlying *design space*, whose dimensions are independent *design decisions* to be made and the value instances at each dimension are the concrete *design choices* available for a design decision. The so far existing generic tree layouts are in chronological order:

- Slingsby et al.'s *Hierarchical Visualisation Expressions* [26] (6 design dimensions, called *appearance operators*),
- Schulz et al.'s design space of implicit tree visualization [25] (4 design dimensions), and
- Baudel and Broeksema's design space of sequential subdivision techniques [3] (5 design dimensions).

*e-mail: hjschulz@informatik.uni-rostock.de
†e-mail: mzakbar@ucalgary.ca
‡e-mail: frank.maurer@ucalgary.ca

IEEE Pacific Visualization Symposium 2013
26 February - 1 March, Sydney, NSW, Australia
978-1-4673-4797-6/13/$31.00 ©2013 IEEE

Design spaces allow users to describe the desired output through a set of design decisions, but rarely permit to influence the process that generates the result. To automatically derive a layout procedure from the design decisions made, the design dimensions have to be independent of one another. While this is still possible for subclasses of tree layouts, it gets harder the more degrees of freedom one wants to capture in such a design space. That is why all of the existing generic approaches consider implicit layouts at most. While design spaces for general tree layouts have been proposed, none of them can practically be used to automatically derive concrete and possibly novel tree drawings from them. For this, they are either not fine-grained enough (e.g., the 3 design dimensions utilized in [14]) or their design choices are no longer independent (e.g., the 12 design dimensions proposed in [10]). The latter leads to the situation that design choices become intricately linked with one depending on another or even contradicting each other. These dependencies are far too complex to be automatically resolved and thus cannot serve as a basis for a generic tree drawing algorithm.

Therefore in this paper, we do not burden ourselves with a design space and take a different approach by focusing only on the functional aspects that actually generate tree drawings. As a result, our approach trades in the insights, which a complete and consistent design space can yield, for the prospect of "getting the job done" and producing a large variety of different rooted tree layouts. To achieve this, we rely on functional building blocks, which we call *layout operators* in accordance with the aforementioned existing generic approaches that all rely on operators as well. Yet, we go beyond the existing approaches by considering operators not as being independent (as a design space would require) but as being strung together in a sequence that captures the actual layout process in a *layout pipeline*. By performing different layout operations at different stages of this pipeline, different layouts are produced – implicit space-filling layouts and explicit node-link layouts, likewise.

Fulfilling the requirements outlined in Section 2, the layout operators and the layout pipeline form the main contribution of this paper as described in Section 3. We have further implemented a prototype of the pipeline, as well as a number of operators to be used with it. We describe this implementation and examples of its use in Section 4. In addition, we conducted a qualitative evaluation of our approach by means of a user study with our prototype. Our findings and observations from this evaluation are described in Section 5, before concluding this paper in Section 6.

2 REQUIREMENTS FOR A GENERIC TREE LAYOUT

Despite its perceived simplicity, the tree layout process is neither self-evident nor self-explanatory, if considered in general. There are many visual features to be captured by a generic layout, while the procedure to capture them should not be overly complex. For design spaces, these two aspects translate to the requirements of *completeness* (capture all possible layouts) and *consistency* (a few independent design dimensions). As the related work has shown them to be hardly achievable for general tree layouts, we make concessions to both by requiring only to capture a large part of the commonly used, canonical tree layouts with just as much specification complexity as is needed to do so. After surveying the wealth of existing layouts, we found that the majority of standard tree drawings can be generated, when the following set of layout requirements is fulfilled:

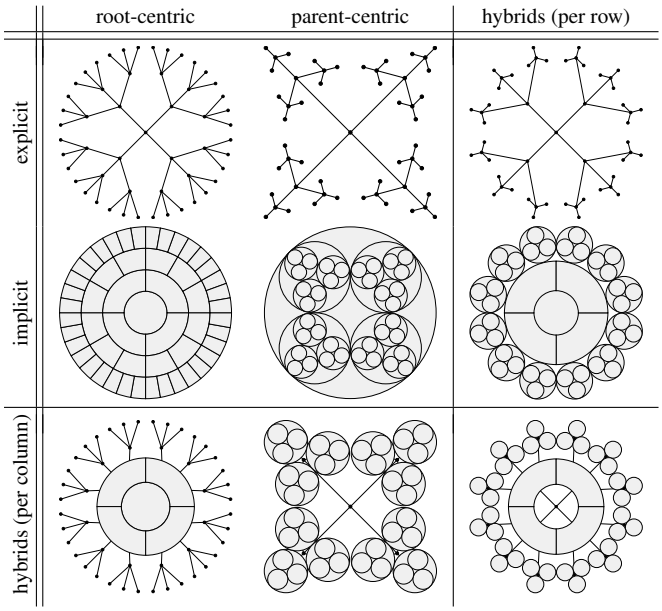


Figure 1: Radial examples for the different layout requirements LR1-LR3 and their interplay.

[LR1] It must be possible to create both, implicit space-filling layouts and explicit node-link layouts.

[LR2] It must allow for generating *root-centric* layouts (all layout operations are made w.r.t. the tree's root), as well as *parent-centric* layouts (all layout operations are made w.r.t. a node's parent).

[LR3] It must support not only one global layout for the entire tree, but be able to locally assign different layouts in order to create hybrid tree representations combining different layouts.

These three requirements are schematically depicted in Figure 1, illustrating them and their interplay to give an impression of the diversity of layouts that can be produced if all three requirements are fulfilled. We bound the necessary specification complexity to handle this diversity and to pinpoint a concrete layout among the many possible ones by defining three requirements for the layout specification as well:

[SR1] The specification should be as concise as possible. It should only require a few specification actions or operations, which are taken from a small set of possible layout operations.

[SR2] Creating the layout in different traversal orders, i.e., *top-down* (starting from the root) or *bottom-up* (starting from the leaves), should not require different ways of specifying them.

[SR3] The individual specification operations should be agnostic to the concrete underlying geometric shape, instead of each shape coming with its own operations.

While there are certainly more points one could wish for in a generic tree layout, we deem these the most important for our generative layout approach, which is introduced in the following section.

3 A Tree Layout Generation Approach

The above requirements state what a generic layout approach should be able to do, but not how to do it. Taking a closer look at existing tree layouts, we ground our approach on two fundamental observations – the first concerning the tree drawings, the second concerning the layout processes that generate them:

- In many cases, there exists a duality between implicit space-filling layouts and explicit node-link layouts. This duality is already exploited by individual tree layout algorithms, but has so far not been used to bring together these two layout styles in one common layout procedure.
- Despite the large variety of tree layouts, many of them follow a similar overall layout process that can be captured in six individual layout stages.

The following section shortly discusses these two observations and derives first design decisions from them, before we describe our approach in detail, as it is built on these decisions.

3.1 Fundamental Design Considerations

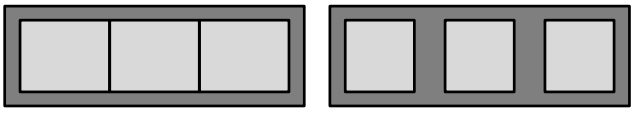
The **duality between implicit and explicit tree layouts** has been observed before [24]. In short, it means that implicit layouts can be transformed into explicit ones and vice versa. For implicit layouts, this transformation is done by simply rendering the nodes as dots inside the constructed areas and connecting them with edges for an explicit look and feel. Tree layouts that make use of this transformation are, for example, the Space-optimized tree [20] or RINGS [28]. They internally perform a space-filling subdivision approach, but render the result as explicit node-link diagrams. Whereas for explicit layouts, the transformation can be performed by merely rendering the areas around the laid out nodes for an implicit look and feel. Tree layouts that make use of this transformation are, for example, the Voronoi Treemap [2] or the Contour Map [16]. They internally perform a node placement and then construct areas around the placed nodes to get an implicit space-filling layout as result.

To fulfill requirement LR1, our generic layout approach uses the first of these two transformations: Internally, it perceives all tree layouts as implicit and assigns drawing areas rather than point positions to the nodes of a tree. This choice makes sense, as the transformation from implicit to explicit is not only computationally less expensive, but this way our approach can also build upon the existing work on generic implicit tree layouts.

The **six stages of the layout process** permit a high-level differentiation between the intent with which different operations are carried out during layout generation:

0. **initialization** for supplying the initial drawing space;
1. **traversal** for moving up or down in the tree;
2. **preprocess** for preparing the nodes to be laid out;
3. **prelayout** for preparing the drawing area for layout;
4. **allocation** for assigning drawing space to the nodes;
5. **postlayout** for final beautification of the layout result.

The very same layout operation performed at different stages of the layout will yield very different results, as it is exemplified in Figure 2 for a small nested Treemap-like layout.



(a) Scaling during Prelayout (b) Scaling during Postlayout

Figure 2: Scaling on different stages of an inclusion-type layout. (a) scales down the dark parent rectangle first and then subdivides it, while (b) performs the subdivision first and scales each of the three resulting rectangles afterwards. This yields different results, with (a) emphasizing the sibling relation much more than (b) does.

Our generic layout approach uses this workflow of six stages as a fixed *layout pipeline*, in which *Stage 0* is only invoked once, whereas *Stages 1* through *5* are repeatedly passed through for each level of the tree. This helps to fulfill requirement SR1 by hiding most of the housekeeping functionality, such as data management and tracking the layout dimensions, and leaving only these stages exposed for customization with a few *layout operators*. The layout pipeline and its six stages are described in the following section.

3.2 The Tree Layout Pipeline

Tree layout procedures differ in whether they traverse the tree **top-down** or **bottom-up**. Since we consider all layout generation as implicit, we can make use of the observation from [25] that states the main distinction between the two is a *subdivision* layout for top-down traversals vs. a *packing* approach for bottom-up traversals. To be able to use the proposed layout pipeline of six stages for both types of traversal, as required by SR2, we make use of a particular data structure that is able to accommodate both.

We partition the tree in its individual levels L_d, where d denotes the depth of a level. Each level consists of a set of tuples of the general form $(\{s_i\},\{n_i\})$. The first element of these tuples contains geometric shapes $s_i \subset \mathbb{R}^{\dim}$ with $\dim \in \{2,3\}$ – e.g., rectangles or circles in 2D, or cuboids or spheres in 3D. The second element contains the nodes of the tree that are associated with the geometric shapes. Shapes and nodes can be thought of as objects that internally hold a number of properties. Shape properties are their position, their extent, and their orientation. Nodes contain information about their parent, as well as a number of numerical attributes, such as the number of children and siblings, the depth, and the Strahler number [1]. The tuples can occur in three different variants:

- **1 shape, n nodes:** Such a tuple holds the initial state of a top-down, partitioning layout. The multiple nodes are siblings. The singular shape encloses the drawing space assigned to the parent of the multiple nodes. For such tuples, the layout algorithm should distribute that space among the nodes.
- **m shapes, 1 node:** Such a tuple holds the initial state of a bottom-up, packing layout. The multiple shapes belong to the children of the singular node. For these tuples, the layout algorithm should tightly pack the shapes and assign the bounding shape of the packing result to the parent node.
- **1 shape, 1 node:** Such a tuple holds the end result of a successful layout. Whether it was generated top-down by partitioning a single into multiples shapes or bottom-up by packing multiple into a single shape, in the end each node is assigned its individual shape.

This transition from $(1,n)/(m,1)$-tuples into $(1,1)$ tuples is performed along the different stages of our layout pipeline. Each layout stage can be viewed as an iterator over all tuples t in L_d, which applies a set of layout operations to t. The layout pipeline is iteratively passed through until all nodes have been assigned their individual shape. For both, top-down and bottom-up traversal, we describe a full pass through the layout pipeline in the following, detailing the changes that each stage makes to L_d. A schematic overview of the entire process is given in Figure 3.

3.2.1 Top-Down Traversal

Stage 0: `initialization` is a preparatory stage that defines the shape of the root node for its subsequent subdivision in the layout process. It can be customized to transform the usually rectangular initial drawing space into an initial shape as it is expected by the following layout. Common uses are radial layouts with angular subdivision that expect a circular space, or layouts that grow outwards and thus require a down-scaled initial space, so that they do not exceed the available overall space during layout.

The Principal State forms the defined starting point for the layout of each level L_d. It consists of a set of $(1,1)$-tuples. This is by definition true for the root level L_0 after initialization and it must be true for the result of *Stage 4* that assigns each node its own shape. The just laid out child nodes are now considered parents themselves and passed as an input to the following traversal to retrieve their children for laying out the next level.

Stage 1: `traversal` is fixed to a top-down DESCEND. It takes the current level L_d and advances it to L_{d+1} by composing a new tuple for each existing one. First, the new tuples contain a copy $\widehat{s_p}$ of the parent shape s_p. This makes sure that all subsequent steps do no longer manipulate the parent shape s_p itself, but the one in which the children are to be laid out. Second, the newly created tuples contain the set of children $\{n_{c1},n_{c2},\ldots\}$ of the respective parent node $\{n_p\}$. If $L_{d+1}=\emptyset$, the layout process terminates.

Stage 2: `preprocess` adapts the set of nodes of each tuple for its subsequent layout. This can be, for example, a sorting operator or a weighting operator. The latter multiplies a numerical attribute of a given node with a weight. Depending on this weight, the size of the later assigned space will be either smaller or larger than it would otherwise have been. It can thus be seen as a scaling on data level. This is particularly important for space-filling layouts in which scaling up a node in view space after the space allocation would result in overlap and thus overplotting.

Stage 3: `prelayout` adapts each tuple's drawing space. This is done, if not all of the given space shall be distributed among the children, e.g., shrinking the space as maybe to maintain a border or reconfiguring the space entirely. The latter is used, for example, to realize parent-centric radial layouts, as it is required by LR2: Instead of further subdividing a circle section resulting from a previous subdivision and thus making just another subdivision w.r.t. the same circle center, one can simply embed a new full circle into the circle segment. This circle will then be subdivided w.r.t. to its own center and thus produce a parent-centric layout. An example for this is given in Section 4.2 in Figure 4g.

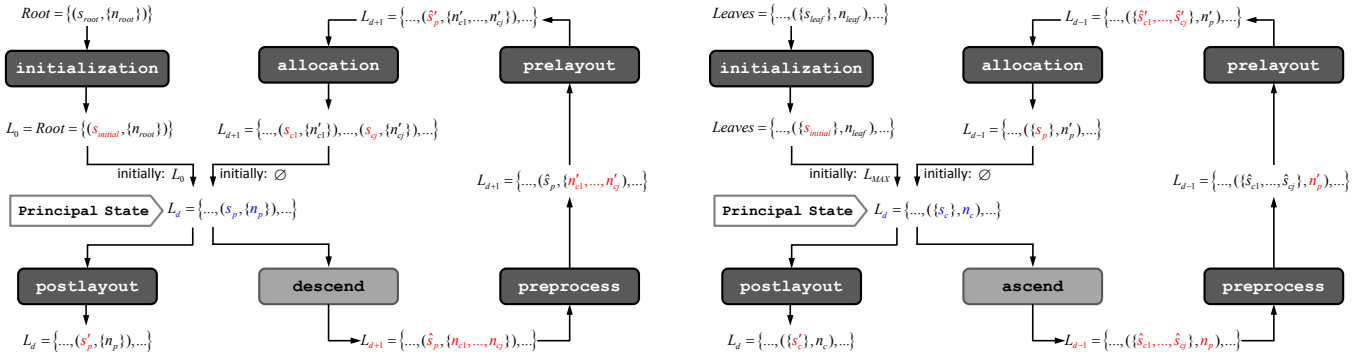
Stage 4: `allocation` assigns each node of a tuple's node set a portion of the tuple's space. These portions are not required to be overlap-free, even though most allocation strategies adhere to a strictly exclusive subdivision. After the assignment of individual drawing space to each node, additional steps can be undertaken to further optimize a possibly crude first space allocation. The end result is again a set of tuples that can be mapped onto the *Principal State* and thus be used as a starting point for the next level's layout.

Stage 5: `postlayout` is performed after *Stage 1* has made its copy of the resulting space and starts off with the next level's layout on an independent drawing space. Then, this stage can perform any final adjustments mainly regarding the appearance, such as reshaping it into a dot and selecting a connector style to produce an explicit node-link rendering. If none is made, the shapes will be drawn as they are – simply as rectangles, circle segments, etc.

3.2.2 Bottom-Up Traversal

Stage 0: `initialization` defines the shapes of the leaves that are then used for the subsequent packing in the course of the layout process. These are usually small shapes, e.g., small rectangles or small spheres, which can be varied in their size by some chosen attribute of the leaf nodes.

The Principal State forms the defined starting point for the layout of each level in the bottom-up case as well. Initially, it contains only the deepest leaves L_{MAX} that are $(1,1)$-tuples by definition,



(a) Top-Down Traversal

(b) Bottom-Up Traversal

Figure 3: Schema of our tree layout pipeline for realizing (a) top-down layouts and (b) bottom-up layouts. The stages colored dark gray are those that can be configured through operators. The light gray stages are constant as the direction of `traversal` is fixed depending on whether the layout is top-down or bottom-up. The variables s denote geometric shapes, the variables n denote nodes of the tree. The index p marks parent shapes/nodes, the index c marks child shapes/nodes. Changes made at the individual stages to the current level L_d are highlighted in red. Modifications are denoted with a prime symbol, copies are denoted with a hat symbol. Blue indicates a mere renaming of the variables without any change to them, which is done so that each iteration through the layout process starts with a level L_d.

and adds the leaves of the next higher level each time this state is visited with the results of the packing process of lower levels. The just laid out parent nodes are now considered children themselves and passed as an input to the following traversal to retrieve their parents for laying out the next level.

Stage 1: `traversal` is fixed to a bottom-up ASCEND. It takes the current level L_d and advances it to L_{d-1} by composing a new tuple for all sets of sibling nodes among the existing tuples. First, the new tuples contain copies $\widehat{s_{ci}}$ of the child shapes s_{ci} belonging to such a set of siblings. Second, the newly created tuples contain the parent $\{n_p\}$ of the sibling nodes $\{\ldots, n_c, \ldots\}$. If $L_{d-1} = \emptyset$, the layout process terminates.

Stage 2: `preprocess` is of lesser importance for the bottom-up traversal. In some cases, this stage is again used for a weighting of a node, which will lead to a down-scaling or up-scaling of this node and all its contained child nodes in the later allocation stage.

Stage 3: `prelayout` adapts the shapes of the children to be packed. For example, often these shapes are slightly enlarged to ensure that they are not packed border-to-border, but that a small gap in between them remains for better distinction.

Stage 4: `allocation` finally packs the shapes of the siblings and then determines a bounding shape around them, which is assigned to their parent node. Since packing is in principal an NP-hard problem, heuristics are used to yield feasible results in reasonable runtime. As a result, the parent node is assigned its individual shape, which can thus be mapped onto the *Principal State* and be used as a starting point for the next level's layout.

Stage 5: `postlayout` does the same as the postlayout for the top-down case and serves mainly the visual embellishment of the layout and permits to change rendering styles.

3.3 The Tree Layout Operators

Operators are a well-established paradigm in information visualization for configuring such a pipeline [9] and have even been formulated as one of the base design patterns of visualization [12, 29]. With their imperative nature, they capture what to do in which order, which is close to our procedural thinking about layout generation. As the input, as well as the output of all operators are the aforementioned tuples, they can be called in arbitrary order, left out completely (identity operator), or even be called multiple times in a row with no conceptual restriction. Because of this consistent behavior, each pipeline stage does not only admit a single such operator, but also sequences of operators. At each pipeline stage, the operators of such a sequence are applied in order to all tuples of the level L_d, which is currently laid out:

```
foreach t ∈ L_d {
    foreach op ∈ op_sequence {
        op(t, P, c)
    }
}
```

The operators are thereby called with three parameters: t is the tuple it shall be applied to, P is a set of operator-specific parameters that govern the details of its function, and c is a conditional that can be used to select a range of nodes for which this operator is to be applied. The conditional is used to express local tree layouts that apply different operators to different parts of the tree, as it is required by LR3. If the conditional does not hold true for the current tuple t, then t is passed back unchanged. Otherwise, the operator transforms the tuple w.r.t. the given parameters:

$$\textbf{func}\ \ \mathrm{op}(t,P,c)\ \ \{\ \ \textbf{if}\ \ c(t)\ \ \textbf{then}\ \ t \xrightarrow{\mathrm{op}_P} t'\ \ \}$$

Depending on its purpose, each stage changes t only in one aspect – either its geometry, the shape(s), or its data, the node(s). In line with [29], we further discern between two types of operators: *creation operators* and *modification operators* [29]. In combination, the scope of an operator (a tuple's shape or data element) and the type of an operator (creation or modification) yield four different kinds of operators: data creation, shape creation, data modification, and shape modification. These four kinds of operators give additional justification to the observed six stages of the pipeline, as there are exactly four stages (*Stage 1* through *4*) – one to apply each kind of operator, plus one stage each for preparing (*Stage 0*) and finalizing (*Stage 5*) the layout through additional shape modifications. Table 1 lists which types of operators are applicable at each stage and gives some examples for them. With this mapping in the background, the pipeline can check automatically whether a given operator is used correctly at a certain stage and thus aid debugging of the layout. In the following, all four kinds of operators are shortly discussed and some instances of such operators are given.

Table 1: Applicable operators at each stage of the layout process

Stage	Type	Scope	Examples
Stage 0: `initialization`	modification	shape	RESHAPE, SCALE,...
Stage 1: `traversal`	creation	data	DESCEND, ASCEND
Stage 2: `preprocess`	modification	data	ORDER, WEIGHT,...
Stage 3: `prelayout`	modification	shape	SCALE, ROTATE, TRANSLATE,...
Stage 4: `allocation`	creation	shape	SQUARIFY, SLICE, STRIP, PACK,...
Stage 5: `postlayout`	modification	shape	RESHAPE, TRANSLATE, FILL,...

Data Creation Operators construct a tuple's node (set) from existing tuples. In the top-down case, this is done through the DESCEND operator, which takes a node and retrieves its children as a new node set. In the bottom-up case, this is done through the ASCEND operator, which takes a set of sibling nodes and retrieves their parent as a new node. Both operators can be used as an interface to a variety of data sources, e.g., not only given trees that are stored on disk, but also to tree generating algorithms that merely produce a new level when called. Data creation operators are used exclusively in *Stage 1* of the layout process.

Shape Creation Operators have to visually reproduce the effects of the used data creation operator. If in *Stage 1* a DESCEND was used to "split" a parent node into its children, the same has to be done to its geometry – the shape has to be subdivided into a number of shapes for the children. This can be done by using operators, such as SLICE for a slice/dice subdivision, STRIP for a Strip Treemap-like subdivision, or SQUARIFY for a subdivision as it is used in Squarified Treemaps. Yet, if an ASCEND operator was used in *Stage 1* to "merge" a number of child nodes into their parent node, this has to be reflected here as well and the child shapes have to be packed with a PACK operator into a parent shape. Shape creation operators are only used in *Stage 4* of the layout process.

Since SR3 requires our operators to be agnostic of the actual geometric shape they are called upon, they must be defined in a generic way that permits to do so. For example, while the squarified layout was only introduced for Treemaps, a generic SQUARIFY operator must also be able to yield sensible results when called for a circular drawing space. In this case, we use the circle's or circle segment's radius and angle for subdivision, and we approximate its "aspect ratio" through the ratio of its radius and outer arc length. An example is given in Section 4.2 in Figure 4c.

Data Modification Operators prepare the nodes for subdivision or packing. An example is the ORDER operator to sort a set of siblings, as it is required by some subdivision operators, such as SQUARIFY. Another possibility is to scale a node's attribute value through the WEIGHT operator to influence the shape creation. If a node is assigned a weight of 0, this is equivalent to a pruning of the tree at this node. Data modification operators can only be used in *Stage 2* of the layout process.

Shape Modification Operators adapt the visual appearance of shapes. This includes three different aspects: shape transformation, shape alteration, and shape representation. Operators that transform the shape are the common geometric transformations SCALE, ROTATE, TRANSLATE, etc. Yet, these operators cannot, for example, alter a rectangular shape into a circular one. This is, what the RESHAPE operator does. Shape alteration is commonly used in *Stage 0* to yield a circular drawing space for radial layouts, but also in *Stage 5* to alter the shape into a dot to create a node-link diagram. Furthermore, we use the RESHAPE operator to switch from a root-centric to a parent-centric layout approach simply by reshaping, for example, a circle segment from a previous subdivision step into a new full circle. This realizes our requirement LR2. While transforming or altering a shape modifies its geometry, shape representation operators, such as FILL, SET_STROKE_WIDTH, etc., customize its appearance. This also includes operators to configure a connector line in case the displayed shapes require an edge to make the parent-child-relationship explicit. Operators of this kind are used in *Stages 0*, *3*, and *5*.

4 Implementation and Examples

Along the lines of our conceptual approach for generating tree layouts, we implemented a web-based software prototype for the subset of 2D top-down layouts. In contrast to existing web-based frameworks with tree drawing capabilities, such as Protovis [6] or D^3 [7] that permit to choose from a set of predefined layouts, our prototype allows users to assemble their own personal layouts by means of plugging different operator sequences into the stages of the pipeline. We present our implementation in the following section, before giving two examples of how to use it for generating tree drawings, and finally discussing its limitations.

4.1 Implementing a Generic Layout Framework

We chose to implement our operator-based layout framework using JavaScript and SVG, so that it is independent of specific platforms and readily available as an interactive demo over the web.[1] Our implementation consists of:

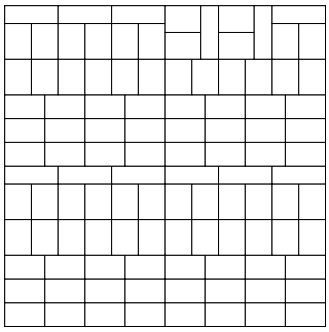
- a **loader** for TreeML files that also computes the node attributes, such as weight of the subtree and Strahler numbers;
- the **pipeline** that parses the operator sequence for each stage and carries it out on each level of the loaded tree;
- the **operators** that are applied to the nodes and their shapes;
- a **renderer** for producing the SVG code from the layout.

Our implementation of the operators $op(t,P,c)$ hides the first argument, the tuple t, from the user as the pipeline takes care of looping through the tuples and carries out the operators on them. Furthermore, the last argument, the conditional c, is optional. If no conditional is given, it is assumed as TRUE and thus the operator is applied to all nodes. The result is an operator signature that looks very much like a procedure call and is thus very familiar to most programmers. A list of the available operators to date can be found online.[2] The following examples show how our prototype is used to generate tree drawings.

4.2 Layout Examples Generated with our Prototype

This section features two examples, first an implicit space-filling tree drawing (shown in Figure 4a-4d) and second an explicit node-link tree drawing (shown in Figure 4e-4h). Other examples can be found online at the framework's demo website. The dataset used throughout all examples is a full tree of height 4, with 4 children on the first level and 3 children on all other levels.

[1] `http://tinyurl.com/operatordemo`
[2] `http://tinyurl.com/operatordocs`



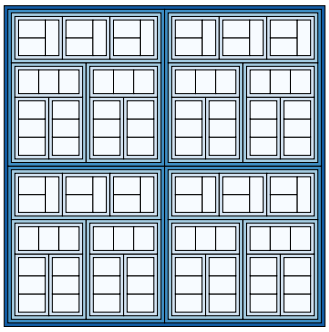
INITIALIZE:

PREPROCESS:
order(DESCENDING,"leaves");
PRELAYOUT:

ALLOCATE:
squarify("leaves");
POSTLAYOUT:

setStrokeWidth(NODES,2);

(a) Squarified Treemap



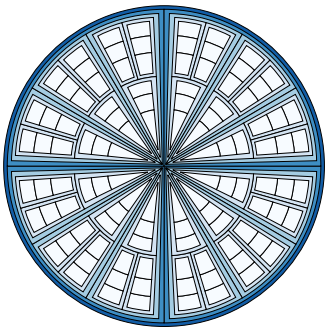
INITIALIZE:

PREPROCESS:
order(DESCENDING,"leaves");
PRELAYOUT:

scale(BY,ALL,-10);
ALLOCATE:
squarify("leaves");
POSTLAYOUT:

setStrokeWidth(NODES,2);
fill("Blues",DARK2LIGHT,
"node.level+1","root.height");

(b) Nested Squarified Treemap



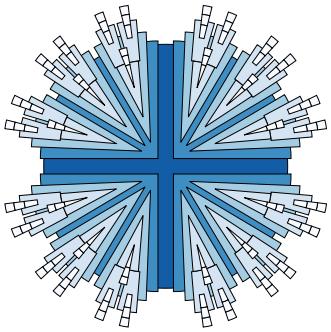
INITIALIZE:

reshape(CIRCLE);
PREPROCESS:
order(DESCENDING,"leaves");
PRELAYOUT:

scale(BY,ALL,-10);
ALLOCATE:
squarify("leaves");
POSTLAYOUT:

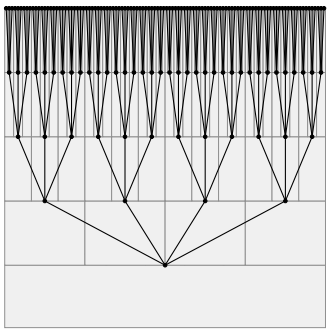
setStrokeWidth(NODES,2);
fill("Blues",DARK2LIGHT,
"node.level+1","root.height");

(c) Nested "Squarified" Pietree



INITIALIZE:
scale(BY,ALL,"-20*root.height");
reshape(CIRCLE);
PREPROCESS:
order(DESCENDING,"leaves");
PRELAYOUT:
translate(TOP,"5*node.level+20",
"!node.isRoot()");
scale(BY,ALL,-10);
ALLOCATE:
squarify("leaves");
POSTLAYOUT:
translate(TOP,"5*node.level+20",
"!node.isRoot()");
setStrokeWidth(NODES,2);
fill("Blues",DARK2LIGHT,
"node.level+1","root.height");

(d) Cascaded Pietree



INITIALIZE:

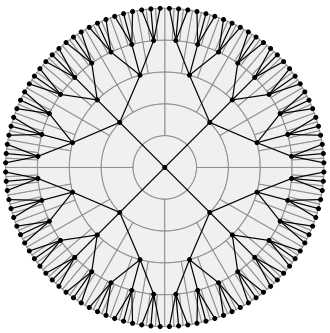
PREPROCESS:

PRELAYOUT:

ALLOCATE:
slice(HORIZONTAL,"leaves");
POSTLAYOUT:

scale(BY,TOP,"-root.dimY*
(1-node.level/root.height)");
reshape(DOT);
connectTo(MIDDLE,TOP);
fill("#000000");
setStrokeWidth(EDGES,3);

(e) Axes-parallel Node-Link Layout



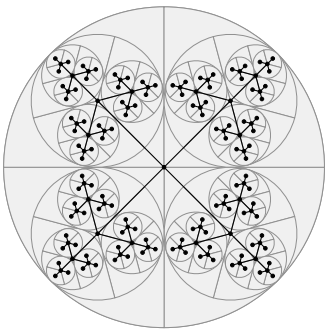
INITIALIZE:
reshape(CIRCLE);
PREPROCESS:

PRELAYOUT:

ALLOCATE:
slice(HORIZONTAL,"leaves");
POSTLAYOUT:

scale(BY,TOP,"-root.dimY*
(1-node.level/root.height)");
reshape(DOT);
connectTo(MIDDLE,TOP);
fill("#000000");
setStrokeWidth(EDGES,3);

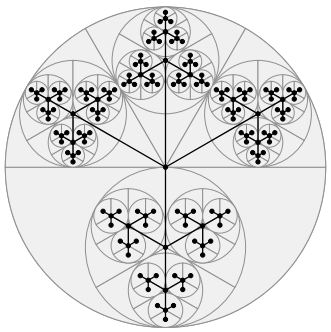
(f) Radial Node-Link Layout



INITIALIZE:
reshape(CIRCLE);
PREPROCESS:

PRELAYOUT:
reshape(CIRCLE);
ALLOCATE:
slice(HORIZONTAL,"leaves");
POSTLAYOUT:
reshape(CIRCLE);
scale(BY,TOP,"-root.dimY*
(1-node.level/root.height)");
reshape(DOT);
connectTo(MIDDLE,TOP);
fill("#000000");
setStrokeWidth(EDGES,3);

(g) Bubble Tree



INITIALIZE:
reshape(CIRCLE);
PREPROCESS:
weight(3,"node.id==1");
PRELAYOUT:
reshape(CIRCLE);
ALLOCATE:
slice(HORIZONTAL,"leaves");
POSTLAYOUT:
reshape(CIRCLE);
scale(BY,TOP,"-root.dimY*
(1-node.level/root.height)");
reshape(DOT);
connectTo(MIDDLE,TOP);
fill("#000000");
setStrokeWidth(EDGES,3);

(h) Weighted Bubble Tree

Figure 4: An example for the step-wise creation of a space-filling tree drawing (a)-(d) and a node-link tree drawing (e)-(h). The black layout operators are those that were added from one step to the next to yield the change in appearance. The gray shapes in the background of the node-link layouts in (e)-(h) are not part of the drawings, but only overlaid to illustrate the internally performed subdivision.

Generating an **implicit space-filling tree drawing** is rather simple, since our approach is built around the notion of distributing space. In the following, we build such a layout in four steps:

The *Squarified Treemap* (Figure 4a) is generated with only three lines of code in our operator-based representation: The PREPROCESS must perform the ordering of the siblings w.r.t. the node attribute that is later used for the actual squarified subdivision in the ALLOCATE stage. This is important to yield good results, as the squarified layout performs best if applied greedily [8]. In our case, this is the number of leaves. To make the lines of the thus generated shapes somewhat thicker, we additionally assign all shapes a stroke width of 2 in the POSTLAYOUT stage.

The *Nested Squarified Treemap* (Figure 4b) adds two additional lines of code to the previous layout, the most important being the one that is responsible for the actual nesting: The scaling of shapes prior to their subdivision in the PRELAYOUT stage BY -10 pixels in ALL directions. This creates the border on each side of the rectangular shapes and thus effectively generates a nested Treemap [13]. To make the different levels of inclusion even more obvious, the levels are additionally color-coded in the POSTLAYOUT stage. For this, the "Blues" color scheme from ColorBrewer [11] is applied from dark to light. The color scale ranges from 0 (dark blue) to a maximal value equaling the height of the entire tree (light blue). A node is then colored according to its level.

The *Nested Squarified Pietree* (Figure 4c) is an example for an entirely new layout generated out of the previous one by simply adding one operator: Reshaping the originally rectangular drawing space into a circular one in the INITIALIZE stage. Pietrees were originally only defined for Slice&Dice subdivisions [21]. Yet, with our shape-agnostic layout operators, we can just as easily produce a radial "squarified" subdivision.

The *Cascaded Squarified Pietree* (Figure 4d) produces the final layout by adding a small offset to the position of each individual shape – very much in the spirit of Cascaded Treemaps [19]. To achieve this, we add the TRANSLATE operator in two places: in the PRELAYOUT stage to perform the shifting on the shape's copy before the next subdivision, and in the POSTLAYOUT stage to also show this shifting for the original shape. Since this layout has the property of growing outwards, we furthermore have to scale down the root shape proportionally to the number of levels during INITIALIZATION.

To produce an **explicit node-link tree drawing**, one has to adhere to the space-centric thinking and compute an implicit layout first, before substituting the generated spaces with dots and connecting them via edges during POSTLAYOUT. We produce such a layout in the following four steps:

To generate a *Bottom-to-Top Layout* (Figure 4e) in its implicit form, a horizontal slicing w.r.t. the number of leaves is performed in the ALLOCATE stage. The resulting space is then scaled down inversely proportional to the current level in the POSTLAYOUT stage. This yields an Icicle Plot-like subdivision [15], which is then transformed into its explicit appearance through the remaining layout operators in the POSTLAYOUT stage: At first, the rectangle is reshaped into a dot. The position of this dot is then adjusted to be the middle-top of each rectangle, the dot is filled solid black, and the stroke width of the edges is set to 3.

A *Radial Layout* (Figure 4f) is generated from the previous layout by reshaping the root into a circle in the INITIALIZE stage. This transforms the underlying Icicle Plot into a Sunburst [27] and thus the axes-parallel node-link layout into a radial one.

The *Bubble Tree* [4] of the next step (Figure 4g) is a parent-centric layout where the children are distributed around their parent as a center, as opposed to the previous root-centric layouts, in which the root remains the center of the layout for all nodes. This property is added to the previous layout in the PRELAYOUT stage by reshaping the gained circle segment at each level into a new full circle, which will be embedded into the generated segment. The reshaping has also to take place in the POSTLAYOUT stage, as otherwise the dot will be placed in the middle of the circle segment and not in the middle of the inscribed circle, which gives a slightly skewed placement. Note, that the scaling in the POSTLAYOUT is no longer necessary for this layout and can optionally be omitted.

Finally, the last step shows a *Weighted Bubble Tree* (Figure 4h) that illustrates the effect of weighting a particular node (and thus also the entire subtree beneath it). In this case, a node was selected by its ID and assigned three times as much weight for the subdivision, as it would normally have had. This is done by a simple call of the WEIGHT operator during the PREPROCESS stage.

4.3 Limitations

Our operator-based framework aims to be a powerful tool that can generate a wealth of different tree drawings (LR1-LR3) with only limited coding effort for the user (SR1-SR3). To achieve this, the complexity of the layout algorithms has either been hidden in the individual operators (increasing the coding effort for the implementer) or stripped away (decreasing the number of producible layouts by the user). This is a trade-off that has to be decided on by the implementer. In our prototype, we chose to make the cut at 2D tree drawings with rectangular and circular shapes. Therefore, layouts relying on 3D or polygonal subdivision cannot be reproduced with it. Yet, most common layouts can be generated with this set of geometric shapes, while the involved computational geometry, e.g., for the RESHAPE or the shape-agnostic SQUARIFY operators, is still manageable and has satisfactory runtimes even for larger trees. An extension to 3D along the lines of [25] certainly lies within reach. While these issues concern the implementer, they say nothing about the actual handling of our operator-based approach by the user, on which we report in the next section.

5 Evaluation

We conducted a user study with 8 participants from the University of Calgary's Computer Science and Environmental Design departments (3 female, 5 male / 3 PhD students, 3 Master students, 2 undergraduates). The participants had no prior knowledge of our prototype and they were given a 15 minutes introduction. Then they had 45 minutes to complete a number of tree layout tasks using the operator-based framework in a simple setup consisting of a text editor to code the operator sequences and an internet browser to test the outcome. Throughout the study, participants had access to documentation of all available operators. The study was subdivided in two parts of increasing difficulty. In the first part, the participants were given a predefined operator-based tree layout in which they had to make small changes to adapt it – for example, re-order the nodes or rotate the layout. As this first part was for familiarizing the participants with the actual coding, they were given hints if they experienced difficulties. In the second part, the participants were handed a printout of the Nested Squarified Treemap drawing from Figure4b and they were asked to build it from scratch. The study was conducted individually with each participant and data was gathered through think-aloud protocol [5], screen capturing, and semi-structured interviews [18] afterwards.

Overall, most participants liked to draw trees using the generative approach for its simplicity and immediateness. The simplicity was highlighted by many participants, as they enjoyed not having to deal with the complexity hidden behind the operators. One participant mentioned that "*with few instructions, I can generate complex visualization, which I really like very much.*" Another participant said "*I just need to understand what effects [an operator] has on the visualization, but I do not need to understand the inner workings.*" The observed simplicity is directly tied to the immediateness, as a code change means to move an operator from one place to an-

other one, without having to care about any surrounding code. It was remarked by one of the participants, that "*when you have very simple commands like this, it makes it easier for you to explore and try out different things.*" These statements reflect the two properties of operator-based design: the imperative nature of the operators, which only specifies *what* shall be done, but leaves the *how* to the software, as well as their consistency, which permits to experiment by shuffling them around freely.

The latter was particularly helpful for the second part of the study, in which the participants had most problems with finding the right stages for the operators and the recursive nature of the layout process. One participant commented that "*the most difficult things for me were to know where to put everything.*" Yet, since the operators can be arranged in almost any order and will in the worst case generate unwanted, impractical tree drawings, 5 out of 8 participants were able to complete the second part just by using a trial and error approach and playing around with different operator orderings. Since this playful approach in turn also gave them insight into the layout process, each trial was valuable as a hands-on learning experience about tree layouts. While we never had in mind that our approach could be a teaching aid about tree layouts, the participants' feedback suggests to explore this option among other aspects of future work, as they are sketched in the following section.

6 Conclusion and Future Work

The user feedback shows that the operator-based approach represents a nice balance between simplicity and flexibility for generating drawings of rooted trees. The participants of our study liked this flexibility, even if it meant that they first had to learn about the tree layout process running in the background. Yet, this learning phase is fast to master with a playful, trial-and-error mindset. To validate the findings from our evaluation, we are are currently in the planning phase of a larger user study that will compare our framework with other existing APIs for generating tree layouts.

In future work, we want to investigate how to utilize layout operator sequences for common interaction events, such as ONCLICK, ONDRAG, etc. For example, a double-click can then be used for interactive folding by carrying out a "reshape(NONE)" operator for the clicked node and all its descendants. Or a zooming interaction can be tied to the WEIGHT operator illustrated in Figure 4h to enlarge subtrees of interest, while at the same time automatically scaling down other parts of the layouts. This would make the operators useful even beyond the pure layout generation.

On the conceptual side, we are eager to investigate the prospect of not only using our layout pipeline for either top-down or bottom-up layouts, but to actually combine the two in the same hybrid manner used for other local layout decisions. This can be done by using the *Principal State* as a junction point between the two, because it contains the same $(1,1)$-tuples for both traversal directions. As a result, it would for example be possible to pack bottom-up a number of subtrees that have before been laid out top-down. This would further broaden the scope of our generic approach to an even larger variety of producible tree layouts.

Acknowledgements

Work on this research was funded by the *NSERC SurfNet Strategic Network*, as well as by the *German Research Foundation (DFG)*. The authors thank Christian Tominski, Steffen Hadlak, and Martin Luboschik for their helpful comments on this paper.

References

[1] D. Auber, M. Delest, J.-P. Domenger, P. Duchon, and J.-M. Fédou. New Strahler numbers for rooted plane trees. In *Proc. of CMCSA'04*, pages 203–215. Birkhäuser, 2004.

[2] M. Balzer and O. Deussen. Voronoi treemaps. In *Proc. of IEEE InfoVis'05*, pages 49–56. IEEE Computer Society, 2005.

[3] T. Baudel and B. Broeksema. Capturing the design space of sequential space-filling layouts. *IEEE TVCG*, 18(12):2593–2602, 2012.

[4] R. Boardman. Bubble Trees – the visualization of hierarchical information structures. In *Extended abstracts of CHI'00*, pages 315–316. ACM Press, 2000.

[5] M. T. Boren and J. Ramey. Thinking aloud: reconciling theory and practice. *IEEE Transactions on Professional Communication*, 43(3):261–278, 2000.

[6] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE TVCG*, 15(6):1121–1128, 2009.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D^3: Data driven documents. *IEEE TVCG*, 17(12):2301–2309, 2011.

[8] M. Bruls, K. Huizing, and J. van Wijk. Squarified Treemaps. In *Proc. of IEEE/TCVG VisSym'00*, pages 33–42. Eurographics Assoc., 2000.

[9] E. H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proc. of IEEE InfoVis'00*, pages 69–75. IEEE Computer Society, 2000.

[10] G. Griffin, S. Li, C. Gramazio, and R. Chang. An analytical approach for the creative design of new visualizations. In *IEEE InfoVis'11 Poster Session*, 2011.

[11] M. A. Harrower and C. A. Brewer. ColorBrewer.org: An online tool for selecting color schemes for maps. *The Cartographic Journal*, 40(1):27–37, 2003.

[12] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE TVCG*, 12(5):853–860, 2006.

[13] B. Johnson and B. Shneiderman. Tree-Maps: A space-filling approach to the visualization of hierarchical information structures. In *Proc. of IEEE Vis'91*, pages 284–291. IEEE Computer Society, 1991.

[14] S. Jürgensmann and H.-J. Schulz. A visual survey of tree visualization. In *IEEE InfoVis'10 Poster Session*, 2010.

[15] J. B. Kruskal and J. M. Landwehr. Icicle plot: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[16] H. Kubota, T. Nishida, and Y. Sumi. Visualization of contents archive by contour map representation. In *New Frontiers in Artificial Intelligence*, pages 19–32. Springer, 2006.

[17] S. Leipert. Drawing trees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 45. CRC Press, 2004.

[18] T. R. Lindlof and B. C. Taylor. *Qualitative Communication Research Methods*. SAGE Publications, 3rd edition, 2011.

[19] H. R. Lü and J. Fogarty. Cascaded Treemaps: Examining the visibility and stability of structure in Treemaps. In *Proc. of GI'08*, pages 259–266. Canadian Information Processing Society, 2008.

[20] Q. V. Nguyen and M. L. Huang. Space-optimized tree: A connection+enclosure approach for the visualization of large hierarchies. *Palgrave Information Visualization*, 2(1):3–15, 2003.

[21] R. O'Donnell, A. Dix, and L. J. Ball. Exploring the PieTree for representing numerical hierarchical data. In *Proc. of HCI'06*, pages 239–254. Springer, 2006.

[22] A. Rusu. Tree drawing algorithms. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 5. CRC Press, 2013.

[23] H.-J. Schulz. Treevis.net: A tree visualization reference. *IEEE Computer Graphics and Applications*, 31(6):11–15, 2011.

[24] H.-J. Schulz, S. Hadlak, and H. Schumann. Point-based tree representation: A new approach for large hierarchies. In *Proc. of IEEE PacificVis'09*, pages 81–88. IEEE Computer Society, 2009.

[25] H.-J. Schulz, S. Hadlak, and H. Schumann. The design space of implicit hierarchy visualization: A survey. *IEEE TVCG*, 17(4):393–411, 2011.

[26] A. Slingsby, J. Dykes, and J. Wood. Configuring hierarchical layouts to address research questions. *IEEE TVCG*, 15(6):977–984, 2009.

[27] J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald. An evaluation of space-filling information visualizations for depiction hierarchical structures. *International Journal of Human-Computer Studies*, 53(5):663–694, 2000.

[28] S. T. Teoh and K.-L. Ma. RINGS: A technique for visualizing large hierarchies. In *Proc. of GD'02*, pages 51–73. Springer, 2002.

[29] Z. Xie, Z. Guo, M. O. Ward, and E. A. Rundensteiner. Operator-centric design patterns for information visualization software. In *Proc. of VDA'10*, pages 7530–0J. SPIE, 2010.