A generic algorithm for sequential, rectangular, space filling layouts

Thomas Baudel*
IBM ILOG Center for Advanced Studies

Bertjan Broeksema†
IBM ILOG Center for Advanced Studies

ABSTRACT

We present a generic algorithm to sequentially pave a rectangular area with smaller, fixed-surface, rectangles. The parameters of this algorithm are functional, and it covers the full design space considered. This class of layouts is interesting, because it includes all kinds of treemaps involving the placement of rectangles. For instance, Slice and dice, Squarified, Strip and Pivot layouts are obtained through various formulations of two simple chunking and phrasing functions. Some new and potentially interesting layouts which can be generated using our algorithm are introduced, such as spiral treemaps and data dependent variations of known layout strategies.

1 INTRODUCTION

There have been many attempts to describe the design-space of treemaps and related space-filling layouts. [2] is one such early but important attempt. A recent and broad survey of this design space is provided in [3]. Focused on rectangular layouts, Slingsby et al [4] propose a general technique to configuring such space filling layouts, from which we draw our structuring method. In all this earlier work, though, the *layout* function that arranges the objects on the plane is left as a black box method. Inspired by our earlier work, Discovery [1], we propose here a finer-grained approach for an important subclass of layout algorithms, whose design space is covered by several simpler pure functions, assembled as the parameters of a generic, design-space-covering, algorithm.

The rectangular space filling problem. Our problem can be expressed, formally, as follows:

- taking as input an ordered list L of n positive real values: $\{a,b,c,...\}$, whose sum is equal to a number S.
- find a paving of the unit square $[0,0,1,1]$ with n non-overlapping, orthogonal, rectangles of surfaces $\{a/S,b/S,c/S,...\}$.
- while maximizing a given objective function.

The output is a representation of L, expressed as a list of graphic instructions for each element in L. It is produced through a succession of calls to a rendering function $drawRect(x_i,y_i,w_i,h_i)$. Hence, our algorithm can be characterized as a function that takes a list L and a renderer R: $layout\_and\_draw : L \times R$. The objective function defines, for a large part, the algorithm to apply. It is generally a weighted sum of objectives involving criteria such as the aspect ratio of the rectangles, the preservation of the input order or the stability to resizing.

Sequential methods. Rather than tackling the full design space of the algorithms that solve this problem, we focus on the class of greedy methods, which we call the *sequential layouts*. These methods are not allowed to use backtracking techniques of unbounded depth. They can perform only a fixed number of passes on the input set, partition the input set and apply to each partition element a smaller space-filling layout method (divide and conquer approach). This class of algorithms corresponds to the class of data-linear visualizations [1], augmented with terminal recursion capability, and its maximal worst-case complexity is $O(n^2)$.

There are several reasons to restrict ourselves to this class:

- all widely known algorithms that pave rectangles belong to this class: slice and dice, squarified, pivot layouts, as well as many related visualizations such as mosaic displays and various tree layouts.
- for further improvements, local search techniques provide a simple technique to find local minima.

*e-mail: baudelth@fr.ibm.com
†e-mail: bertjan.broeksema@fr.ibm.com

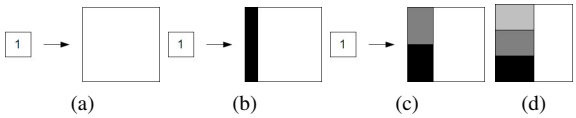


Figure 1: Stacking items in a block.

2 THE BLOCK DATA-STRUCTURE

Because the layout algorithm we consider is sequential, it can only handle the elements in separate chunks containing at least one element. A *block* is a sub-rectangle in the bounding rect that contains one or more elements. In the remainder, B denotes a list of blocks that pave the bounding rect in L. Furthermore, b_i denotes the ith block (where $0 < i <= n$). When $max(i) \equiv n$, all elements are laid out in separate blocks. Complementary, when $max(i) \equiv 1$ all elements are laid out in one block that paves the full bounding rectangle.

Once a block is laid out in the available space, no changes to the aspect ratio or to the location of the block can be made. Consequently, the way a block can be located in the available space rectangle is strongly constrained. There are only four possible locations for a new block: the four sides of the rectangle representing the available space. Additionally, newly created blocks must take either full height and grow horizontally when elements are added or vice-versa. Finally, elements can be stacked in various directions inside a block. In horizontal blocks they can be either stacked from left to right or vice versa. In vertical blocks they can be stacked from top to bottom or vice versa. This brings the total number of possible block configurations to eight.

Figure 1 demonstrates how elements are stacked in a block. The block is placed on the left side of the available space rectangle, and three out of six equally sized elements are added to the block. Elements are stacked from bottom to top. The order of stacking is depicted by color, going from black for the first to light gray for the last element. In figure 1a, no element is added, hence the block height is equal to the available space plane and its width is zero. Next, in figure 1b the first element is added and the block now has a width proportional to the size of the added element with respect to overall size to be laid out. In figure 1c the second element is added. The width of the block again grows proportionally, but the height of the elements is reduced as they are stacked in the block. Finally, figure 1d shows the result of adding yet another element

to the block. Assuming that figure 1d shows the final state of the block, the available space would be reduced to $[x : 0.5, y : 0, w : 0.5, h : 1]$.

3 Parameterizing functions

Next we describe the functions that parametrize the algorithm. Before going into more detail on these functions we first introduce the Context. The context stores local state information. In order to keep the pseudo code readable, it is presented as an interface with the functions that are required in our algorithm (i.e. score, phrase and recurse) which can be implemented in different ways. In practice functors are used to implement the various variants of these functions, which are set as function objects on the Context. For the same reason the bookkeeping calls to the context (e.g. to notify that a new block is started or that a level is fully laid out) are left out. Calls to the functions in the context interface are passed on to these functors. We now describe the three functions in more detail.

Chunking Given the list L, chunking is the process of deciding whether the ith element gets added to block b_k or to block b_{k+1}. This decision is made using a simple scoring function which takes a block b_k, a size $L[i]$ and returns a score s. Or more formally: $score : b_k \times L[i] \rightarrow s$. When $score(b_k \times L[i] <= score(b_k \times L[i-1])$ then adding $L[i+1]$ to block b_k is an improvement of the layout.

Deciding whether or not an element must be added to the current block can be based on various variables such as the ratio between the number of elements in block b_k with respect to the total number of elements. An example is the MinAspectRatioScore implementation. It determines whether adding the next element either improves or degrades the aspect ratio of the smallest element currently put in the block.

Phrasing When a new block is started, the algorithm must decide how the block will be laid out in the available space. Recall, that a block must be placed along one of the four sides of the available space, and that a stack direction must be given as well. Those two characteristics, taken together, form a block configuration. We call *phrasing* the process of picking a configuration for each successive block. Phrasing is again a simple function of the context that takes the previous block b_{i-1} and returns the block configuration for block b_i. That is, $phrasing : b_{i-1} \rightarrow BlockConfiguration$. The first block is phrased according to the initial configuration set in the context, which is one of the earlier mentioned eight possibilities. Depending on the strategy, the next configuration can be determined in a number of ways. We distinguish data dependent and data independent phrasing strategies. The data independent strategies are depicted in figure 2. The data dependent strategies choose the next configuration based on the current situation.

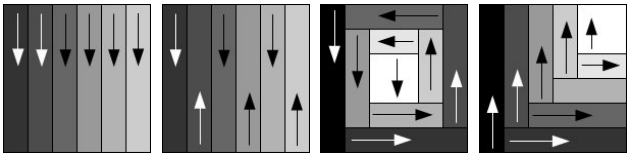
Recurse After having isolated a block, the algorithm may decide to recurse into the block, reapplying itself to further improve the aspect ratio or other optimization goals. This feature allows implementing the various types of pivot layouts.

4 Algorithm

Finally, we present the full algorithm in pseudo code as listed in listing 1.

Listing 1: Complete algorithm to layout and draw a tree with usage example

```
1  layout(L, C, BR) {
2    b = new Block(C. initialConfiguration , BR)
3    B = new List(b)
4    int prevScore = inf
5    for (int i = 0; i < L.size(); ++i) {
6      int curScore = C.score(b, L[i])
7      if (curScore > prevScore) { // Not an improvement:
8        if(C.recurse(b))
9          B.append(layout(b.items, C, b.rect))
10       else
11         B.append(b) // Accumulate block in result,
12       b.reduce(BR) // Reduce the bounding rect such that b is excluded,
13       int nextConfiguration = C.phrase(b) // Start a new block.
14       b = new Block(nextConfiguration, BR);
15     }
16     b.add(L[i])
17     prevScore = curScore
18   }
19   return B;
20 }
21
22 draw(C, L, R, BR) {
23   B=layout(L, C, BR)
24   foreach(b : B)
25     foreach( e : b )
26       R.drawRect(t, b.rectangle(e))
27 }
28
29 layout_and_draw(L) { // Configuration for fig 2c
30         C = new Context()
31         C. initialConfiguration = LEFT_TOP_TO_BOTTOM
32         C.chunking = new MinAspectRatioScore()
33         C.phrasing = new Spiral()
34         BR = new Rectangle(0,0,1,1)
35         R = new Renderer()
36         draw(C, L, R, BR)
37 }
```



(a) (b) (c) (d)

Figure 2: Four data independent phrasing strategies to create a space filling layout: (a) Strip; (b) Zigzag; (c) Spiral; and (d) Spikes.

5 Conclusion

We presented a generic algorithm that sequentially paves a rectangular area with smaller, fixed-surface, rectangles. This class of layouts is interesting, because, beyond encompassing simple grids, tables and trees, it also includes all kinds of treemaps. Our algorithm is parametrized by two functions, score and phrase. These can be implemented in various ways in order to reproduce well known existing layouts and a few new and potentially interesting layouts such as spiral treemaps and data dependent variations of known treemap layouts. As future work, we propose to show that this algorithm covers the full design space of sequential, rectangular, space-filling layouts.

References

[1] T. Baudel. Browsing through an information visualization design space. In *CHI '04 extended abstracts on Human factors in computing systems*, CHI EA '04, pages 765–766, New York, NY, USA, 2004. ACM.

[2] B. B. Bederson, B. Shneiderman, and M. Wattenberg. Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, Oct. 2002.

[3] H.-J. Schulz, S. Hadlak, and H. Schumann. The Design Space of Implicit Hierarchy Visualization: A Survey. *IEEE transactions on visualization and computer graphics*, 17(4):393–411, May 2010.

[4] A. Slingsby, J. Dykes, and J. Wood. Configuring hierarchical layouts to address research questions. *IEEE transactions on visualization and computer graphics*, 15(6):977–84, 2009.