Treemaps
in OCaml
version 0.1

Yoann Padioleau
pad@facebook.com

September 24, 2010

Contents

1 Introduction 1
1.1 Motivations . . . 1
1.2 Getting started . . . 3
1.2.1 Requirements . . . 3
1.2.2 Compiling . . . 4
1.2.3 Quick example of use . . . 4
1.3 Copyright . . . 4
1.4 About this document . . . 4

2 Examples of use 5

3 Seminal Algorithm, Slice and Dice 5
3.1 Treemap data structure . . . 5
3.2 The algorithm . . . 6
3.3 Screen and viewport . . . 9

4 Other Algorithms 11
4.1 Tiling rectangles . . . 12
4.2 Clustered treemaps . . . 15
4.3 Squarified treemaps . . . 15
4.4 Ordered treemaps . . . 23
4.5 Cushion treemaps . . . 33

5 Extra features 33
5.1 Nesting 33
5.2 Labeling 33
5.3 Interactivity 33

6 JSON reader 35

7 Applications 40
7.1 Disk statistics 40
7.2 Source code architecture visualization 41
7.3 Code coverage (tests, deadcode, etc) 42
7.4 Version-control visualization 42

8 Conclusion 42

A Extra Code 42
A.1 treemap.mli 42
A.2 treemap.ml 44
A.3 treemap_json.mli 61
A.4 treemap_json.ml 61
A.5 treemap_graphics.mli 63
A.6 treemap_graphics.ml 64
A.7 main_treemap.ml 66

B Changelog 70

Index 70

References 70

1 Introduction

1.1 Motivations

```
% see code base, many files, like kernel, or even my own code.
% SeeSoft good, thumbnails, but does not scale to thousands of files.
% enter treemaps, space filling!

% size important
% can play intensitiy, ... ex of treemap where size, modulated,
%  and intensitiy, and commit, and semantic, and speedbar!

% ex of pfff treemap, or linux!

% why reinvent ? related soft ?
% where better than fekete ?
```

[5]
[7]
DiskStat.

```
 * Advantages of my solution compared to using kdirstat on ~/www ?
 *  - can customize color for files, eg colors for css/php/js/...
 *  - can focus only on certain files, eg .php
 *
 *  - can access info from my other ocaml libs, eg
 *    pfff_db, and git. To do that with kdirstat would
 *    force me to hack a complex codebase, and dependencies (kde ...)
 *  - can combine static analysis or dynamic analyzis result with treemaps
 *    (but kprof does that too ?)
```

More applications: [6]

```
(*
 * Basic references:
 *  http://en.wikipedia.org/wiki/Treemapping
 *  http://www.cs.umd.edu/hcil/treemap-history/index.shtml
 *
 * Seminal: http://hcil.cs.umd.edu/trs/91-03/91-03.html
 *
 * http://www.smartmoney.com/map-of-the-market/
 * (need jave plugin)
 *
 * Treemaps are cool. They can show multiple attributes at the same time:
 *  - size (size of rectangle)
 *  - depth (if nested, especially when use borders or cushion)
 *  - kind (color)
 *  - intensity (degrade de couleur)
 *  - extra info by for instance drawing points (des petits pois) inside it
 *    can also use filling pattern as in xfig to convey additional info.
 *
 * Does the position (x,y) mean something ? if sort alphabetically, then
 * yes can also give spatial indication. If use squarified then it's kind
 * of sorted by size which also give good spatial indication wether some
 * modules are important or not.
 *
 * More references:
 *  - seminal paper http://hcil.cs.umd.edu/trs/91-03/91-03.html
 *  - cushion so better see structure
 *    (solve pb of having lots of similar small rectangles which forbid to
 *    visually see the bigger picture, that is their enclosing rectangles)
 *  - squarified so can more easily compare two items
 *    (solve pb of elongated rectangle)
 *
 *
 * ***** other ocaml libs
 *
 * 3d stuff: lmntal style, with physics (not that needed)
 * http://ubietylab.net/ubigraph/content/Demos/Networkx.html
 * not free, they have a binding for ocaml
 *
 * **** other perl/python/ruby libs
 *
 * python seems quite good and fresh with latest research :)
 * semi:
 * http://www.machine-envy.com/blog/2006/07/29/a-treemap-viewer-for-python/
 * semi:
 * http://www.scipy.org/Cookbook/Matplotlib/TreeMap?action=show&redirect=TreeMap
 * (but does not have the cushion :( )
 *
 * http://rubytreemap.rubyforge.org/
 *
 * **** other java libraries ...
 *
 * treemap by bouthier (ex maryland)
 * perfuse
 *
 * **** misc
 *
 * http://kdirstat.sourceforge.net/kdirstat/
 * use apparently qtreemap
 *
 * http://kprof.sourceforge.net/
 * also use treemap
 *
 * *** list of libs
 * http://en.wikipedia.org/wiki/List_of_treemapping_software
 *
 *)
```

```
% size, labels, anamorphic (c smaller :) ), git info.
% could add semantic analysis, so if called often, coefficient rectifier
```

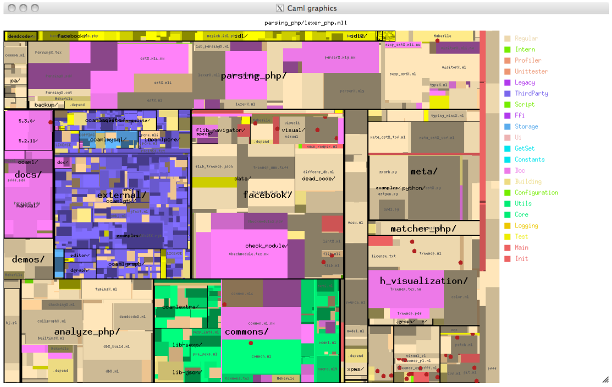

Figure 1: Treemap of source code

1.2 Getting started

1.2.1 Requirements

```
% commons
% json if want json reader
% recommended h_program-visual/
```

1.2.2 Compiling

1.2.3 Quick example of use

```
$ ./treemap_viewer  examples/treemap/ex.json
```

```
$ ./treemap_viewer -algorithm squarified  examples/treemap/ex.json
```

1.3 Copyright

The source code of OCamlTreemap is governed by the following copyright:

4 ⟨*Facebook copyright* 4⟩≡ (44e 61b 64)

```
(* Yoann Padioleau
 *
 * Copyright (C) 2010 Facebook
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
```

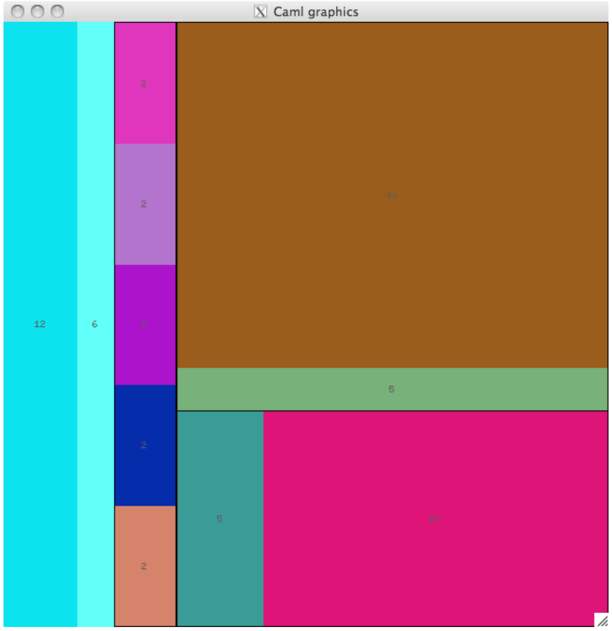

Figure 2: Slice and dice treemap

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the file
 * license.txt for more details.
 *)
```

1.4 About this document

This document is a literate program [1]. It is generated from a set of files that can be processed by tools (Noweb [2] and syncweb [3]) to generate either this manual or the actual source code of the program. So, the code and its documentation are strongly connected.

2 Examples of use

3 Seminal Algorithm, Slice and Dice

3.1 Treemap data structure

```
(*
 * We use the directory/file metaphor for the nodes/leafs,
 * because dirs/files are one of the best example of use of treemaps,
 * and because it was the one chosen by Schneiderman in his original paper.
 *
 * The type is polymorphic because we want the interactive treemap visualizer
 * to offer hooks to display certain information about the dir/file under
 * the cursor.
 *)
```

5 ⟨*type treemap* 5⟩≡ (42 44e)

```
  type ('dir, 'file) treemap =
   (treemap_rect * 'dir, treemap_rect * 'file) Common.tree
      and treemap_rect = {
        size : int;
        color : Simple_color.color;
        label: string;
      }
```

6a ⟨*signature tree and treemap examples* 6a⟩≡ (42)

```
  val treemap_rectangles_ex:
     ((float * float) list * (float * float) list * (float * float * float)) list

  val tree_ex_shneiderman_1991 : (unit, int) Common.tree
  val tree_ex_wijk_1999: (unit, int) Common.tree
  val treemap_ex_ordered_2001: (unit, unit) treemap
```

6b ⟨*variable tree_ex_shneiderman_1991* 6b⟩≡ (44e)

```
  let tree_ex_shneiderman_1991 =
    let ninfo = () in
    Node (ninfo,  [
      Leaf 12;
      Leaf 6;
      Node (ninfo,  [
        Leaf 2;
        Leaf 2;
        Leaf 2;
        Leaf 2;
        Leaf 2;
      ]);
      Node(ninfo,  [
        Node(ninfo,  [
          Leaf 5;
          Leaf 20;
        ]);
        Node(ninfo,  [
          Leaf 5;
        ]);
        Leaf 40;
```

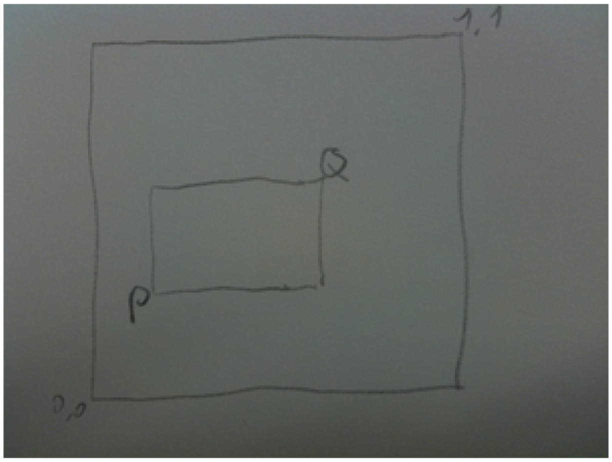

Figure 3: P and Q

```
      ]);
    ])
```

3.2 The algorithm

[8]

6c ⟨*signature display_treemap* 6c⟩≡ (63)

```
  val display_treemap :
    ('dir, 'file) treemap -> int * int -> 'file option Common.matrix
```

7a ⟨*type rectangle1* 7a⟩≡ (44e)

```
  (* The array has 2 elements, for x, y. I use an array because that's how
   * the seminal algorithm on treemap was written. It allows to pass
   * as an int the current split and do x.(axis_split) and do a 1-axis_split
   * in recursive calls to go from a x-split to a y-split.
   *
   * A rectangle is represented by 2 variables called P and Q in the seminal
   * algorithm.
   *)
  type rectangle1 =
    float array (* lower left  coord, P *) *
    float array (* upper right coord, Q *)
```

7b ⟨*function display_treemap* 7b⟩≡ (64)

```
  (*
   * ref: http://hcil.cs.umd.edu/trs/91-03/91-03.html, page 6
```

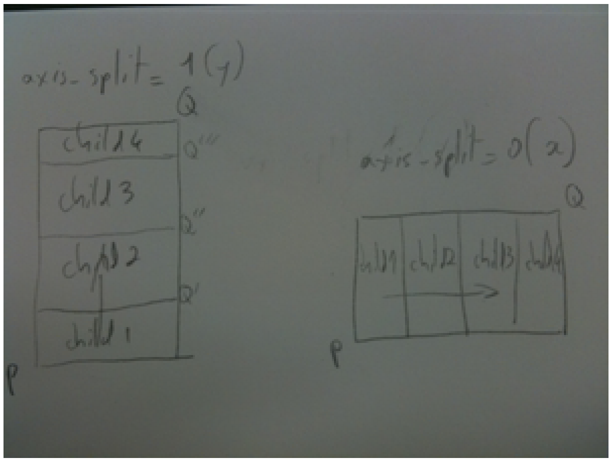

Figure 4: Slicing and dicing

```
 *
 * The algorithm is very simple. Look at the paper. I’ve just added
 * the depth argument.
 *
 * axis_split is 0 when split enclosing rectangle vertically, and 1
 * when doing it horizontally. We alternate hence the (1 - axis_split) below.
 *
 * still? look if python port look the same
 *)
let display_treemap (treemap: (’dir,’file) treemap) (w, h) =

  let mat = Array.make_matrix w h None in

  (* p and q are the coords of the current rectangle being laid out *)
  let rec aux_treemap root p q axis_split ~depth =

    (* todo? join the 2 match in a single one ? *)
    (match root with
    | Leaf (tnode, fileinfo) ->
        let color = color_of_treemap_node root in

        let rect_opt =
          draw_rect_treemap_float_ortho
            ((p.(0), p.(1)),
             (q.(0), q.(1)))
            color
            (w, h)
        in
        rect_opt +> Common.do_option (update_mat_with_fileinfo fileinfo mat)

    | Node (tnode, dirinfo) ->
        ()
    );
    let size_root = size_of_treemap_node root in
    let width = q.(axis_split) -. p.(axis_split) in
    match root with
    | Node (mode, children) ->
        children +> List.iter (fun child ->
          (* if want margin, then maybe can increment slightly p and decrement
           * q ? like 1% of its width ?
           *)
          q.(axis_split) <-
            p.(axis_split) +.
            (float_of_int (size_of_treemap_node child) /.
             float_of_int (size_root)) *. width;
          aux_treemap child (Array.copy p) (Array.copy q) (1 - axis_split)
            ~depth:(depth + 1)
          ;
          p.(axis_split) <- q.(axis_split);
        )
    | Leaf _ -> ()
  in
  aux_treemap treemap [|0.0;0.0|] [|1.0;1.0|] 0  ~depth:1;
  mat
```

3.3 Screen and viewport

```
(* Need information such as total width to draw to the right place, outside
 * the viewport, in the status area or legend area.
 *)
```

9a ⟨*type screen_dim* 9a⟩≡ (42 44e)

```
  type screen_dim = {
    (* total width/height *)
    w: int;
    h: int;
    (* the viewport *)
    w_view: int;
    h_view: int;
    (* extra information *)
    h_status: int;
    w_legend: int;
```

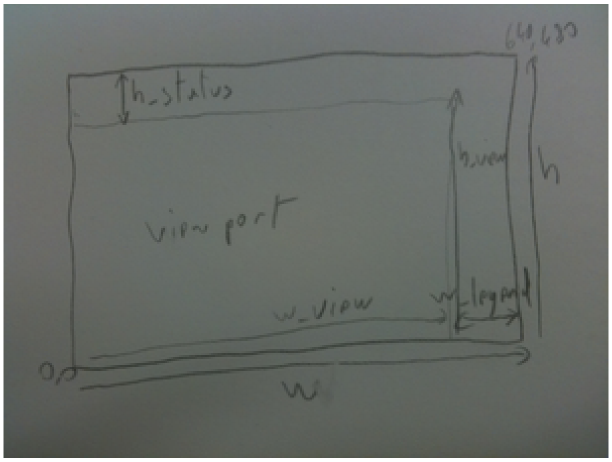

Figure 5: Screen and viewport

```
}
```

9b ⟨*signature graphic helpers* 9b⟩≡ (63) 44a ▷

```
val draw_rect_treemap_float_ortho :
  (float * float) * (float * float) ->
  Graphics.color -> int * int -> ((int * int) * (int * int)) option
```

11a ⟨*function draw_rect_treemap_float_ortho* 11a⟩≡ (64)

```
(*
 * The treemap algorithms assume an ortho? space from 0,0 to 1.1 but
 * our current screen have pixels and goes from 0,0 to 1024,168 for
 * instance. Those functions are here to make the translation
 * (it can produce some aliasing effects).

 * TODO: pass a converter function from ortho space to regular ?
 * as in opengl?
 *)

let draw_rect_treemap_float_ortho ((x1, y1),(x2, y2)) color (w, h) =

  let w = float_of_int w in
  let h = float_of_int h in

  let x1, y1 = int_of_float (x1 *. w), int_of_float (y1 *. h) in
  let x2, y2 = int_of_float (x2 *. w), int_of_float (y2 *. h) in
  let w = (x2 - x1) in
  let h = (y2 - y1) in
```

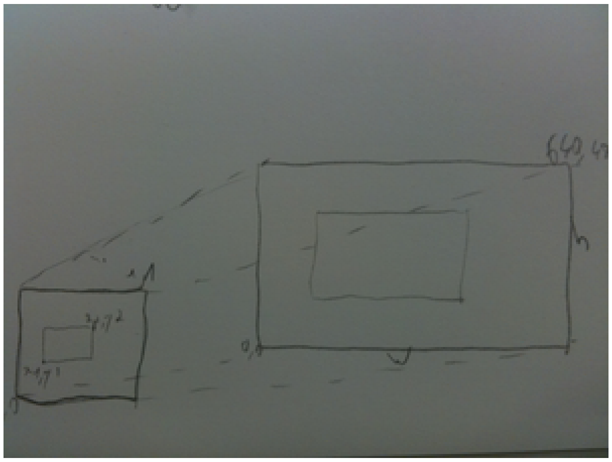

Figure 6: Scaling the ortho plan
fig:treemap-ex

```
    Graphics.set_color color;

    if w <= 0 || h <= 0
    then None
    else begin
      Graphics.fill_rect
        x1 y1 w h;
      Some ((x1,y1), (x2,y2))
    end
```

4 Other Algorithms

11b ⟨*type algorithm* 11b⟩≡ (42 44e)

```
  type algorithm =
    | Classic
    | Squarified
    | SquarifiedNoSort
    | Ordered of pivot

    and pivot =
      | PivotBySize
      | PivotByMiddle
```

11c ⟨*signature algos* 11c⟩≡ (42)

```
  val algos: algorithm list
```

```
val layoutf_of_algo: algorithm -> ('a, 'b) layout_func
```

12a ⟨*variable algos* 12a⟩≡ (44e)

```
let algos = [Classic; Squarified; SquarifiedNoSort;
             Ordered PivotBySize; Ordered PivotByMiddle]
```

12b ⟨*signature display_treemap_algo* 12b⟩≡ (63)

```
val display_treemap_algo :
  ?algo:algorithm ->
  ?drawing_file_hook:
    (Figures.rect_pixel -> 'file -> 'file option Common.matrix -> unit) ->
  ('dir, 'file) treemap ->
  int * int ->
  'file option Common.matrix
```

4.1 Tiling rectangles

12c ⟨*type layout_func* 12c⟩≡ (42 44e)

```
type ('a, 'b) layout_func =
  (float * ('a, 'b) treemap) list ->
  int ->
  rectangle ->
  (float * ('a, 'b) treemap * rectangle) list
```

12d ⟨*function display_treemap_generic* 12d⟩≡ (64)

```
let display_treemap_generic
    ?(drawing_file_hook=(fun _rect _file _mat -> ()))
    (treemap: ('dir,'file) treemap)
    (w, h)
    flayout
 =

  let mat = Array.make_matrix w h None in

  let rec aux_treemap root rect ~depth =
    let (p,q) = rect.p, rect.q in

    if not (valid_rect rect)
    then () (* TODO ? warning ? *)
    else

    (match root with
    | Leaf (tnode, fileinfo) ->
        let color = color_of_treemap_node root in
        let rect_opt =
          draw_rect_treemap_float_ortho
            ((p.x, p.y),
             (q.x, q.y))
            color
            (w, h)
        in
        let info = fileinfo in

        (match rect_opt with
        | None -> ()
        | Some ((x1,y1), (x2,y2)) ->

            for i = x1 to x2 - 1 do
              for j = y1 to y2 - 1 do
                mat.(i).(j) <- Some info;
              done
            done;

            drawing_file_hook {
              F.lower_left =   { F.x = x1; F.y = y1 };
              F.upper_right =  { F.x = x2; F.y = y2 };
            }
              fileinfo
              mat

        );
        draw_label rect  (w, h) depth (tnode).label ~is_dir:false


    | Node (mode, children) ->

        (* let's draw some borders. Far better to see the structure. *)
        let _rect_opt =
          draw_rect_treemap_float_ortho
            ((p.x, p.y),
            (q.x, q.y))
            Graphics.black
            (w, h)

        in
        (* does not work, weird *)
        let border =
          match depth with
          | 1 -> 0.0
```

```
       | 2 -> 0.002
       | 3 -> 0.001
       | 4 -> 0.0005
       | 5 -> 0.0002
       | _ -> 0.0
     in
     let p = {
       x = p.x +. border;
       y = p.y +. border;
     }
     in
     let q = {
       x = q.x -. border;
       y = q.y -. border;
     }
     in
     (* todo? can overflow ... check still inside previous rect *)
     let rect = { p = p; q = q } in

     let children' =
       children +> List.map (fun child ->
         float_of_int (size_of_treemap_node child),
         child
       )
     in

     let rects_with_info =
       (* generic call *)
       flayout children' depth rect
     in
     (* less: assert rects_with_info are inside rect ? *)

     rects_with_info +> List.iter (fun (x, child, rect) ->
       aux_treemap child rect ~depth:(depth + 1)
     );

     draw_label rect (w, h) depth (fst mode).label ~is_dir:true
   )
  in
  aux_treemap treemap rect_ortho ~depth:1;
  mat
```

14 ⟨*function display_treemap_algo* 14⟩≡ (64)

```
let display_treemap_algo ?(algo=Classic) ?drawing_file_hook
 treemap (w, h) =
  (* old: display_treemap            treemap (w, h) *)
  let layoutf = layoutf_of_algo algo in
  display_treemap_generic ?drawing_file_hook
    treemap (w, h) layoutf
```

15a ⟨*layout slice and dice* 15a⟩≡ (44e)

```
let (slice_and_dicing_layout: ('a, 'b) layout_func) =
 fun children depth rect ->

  let p = [| rect.p.x; rect.p.y |] in
  let q = [| rect.q.x; rect.q.y |] in

  let axis_split = (depth + 1) mod 2 in

  let stotal = children +> List.map fst +> Common.sum_float in

  let width = q.(axis_split) -. p.(axis_split) in

  children +> List.map (fun (size, child) ->

    q.(axis_split) <-
      p.(axis_split) +.
      ((size) /. stotal) *. width;

    let rect_here = {
      p = {  x = p.(0); y = p.(1); };
      q = {  x = q.(0); y = q.(1); }
    }
    in
    p.(axis_split) <- q.(axis_split);
    size, child, rect_here
  )
```

4.2 Clustered treemaps

4.3 Squarified treemaps

[10]

15b ⟨*variable tree_ex_wijk_1999* 15b⟩≡ (44e)

```
let tree_ex_wijk_1999 =
  let ninfo = () in
  Node (ninfo,  [
    Leaf 6;
    Leaf 6;
    Leaf 4;
    Leaf 3;
```

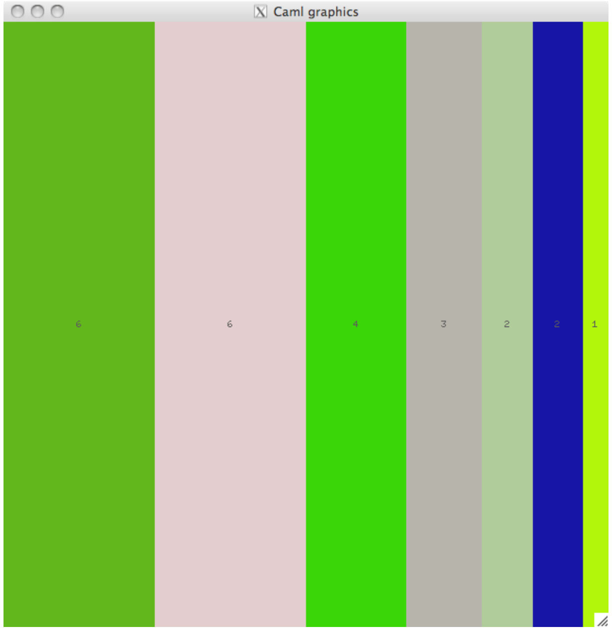

Figure 7: Slice and dice limitations

```
    Leaf 2;
    Leaf 2;
    Leaf 1;
  ])
```

16 ⟨*squarified examples* 16⟩≡ (44e)

```
(* ref: www.win.tue.nl/~vanwijk/stm.pdf
 *
 * In the following I use some of the examples in the paper so you'll need
 * the paper to follow what I say.
 *)


(*
 * A few examples.
 *
 * the total sum in squarified_list_area_ex is 24, just like the area
 * of rect_orig below. This simplifies discussions.
 *
 * I've added the string later as we want squarify to also return
 * information related to the node with its size (that is the full treemap
 * node, with its descendant)
```

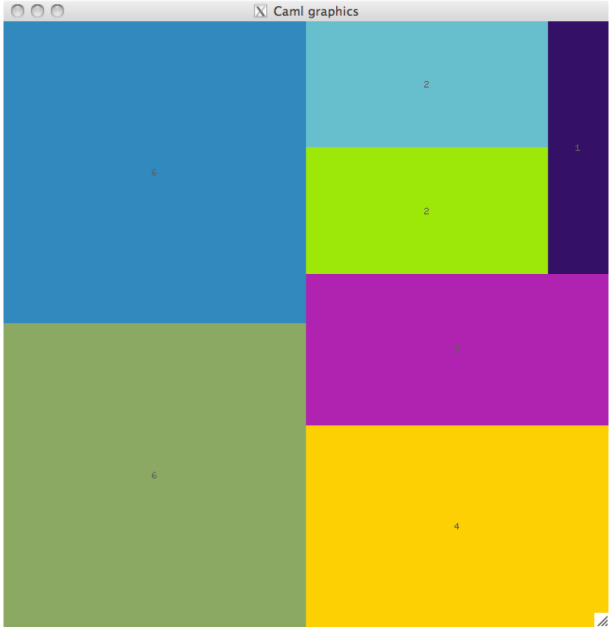

Figure 8: Squarified treemap

```
*)
let squarified_list_area_ex =
  [6; 6; 4; 3; 2; 2; 1] +> List.map (fun x -> float_of_int x, spf "info: %d" x)

(* normally our algorithm should do things proportionnally to the size
 * of the aready. It should not matter that the total sum of area is
 * equal to the size of the rectangle. Indeed later we will always do
 * things in an ortho plan, that is with a rectangle 0x0 to 1x1.
 *)
let squarified_list_area_ex2 =
  squarified_list_area_ex +> List.map (fun (x, info) -> x *. 2.0, info)
let dim_rect_orig =
  { p = {x = 0.0; y = 0.0; }; q = { x = 6.0; y = 4.0} }
```

17 ⟨*type split* 17⟩≡ (44e)

```
type split =
  (* Spread one next to the other, e.g. | | | | | |
   * The split lines will be vertical, but the rectangles
   * would be spreaded horizontally. In the paper they call that horizontal
   * Split but I prefer Spread, because the split lines are actually verticals.
   *)
  | SpreadHorizontally
```

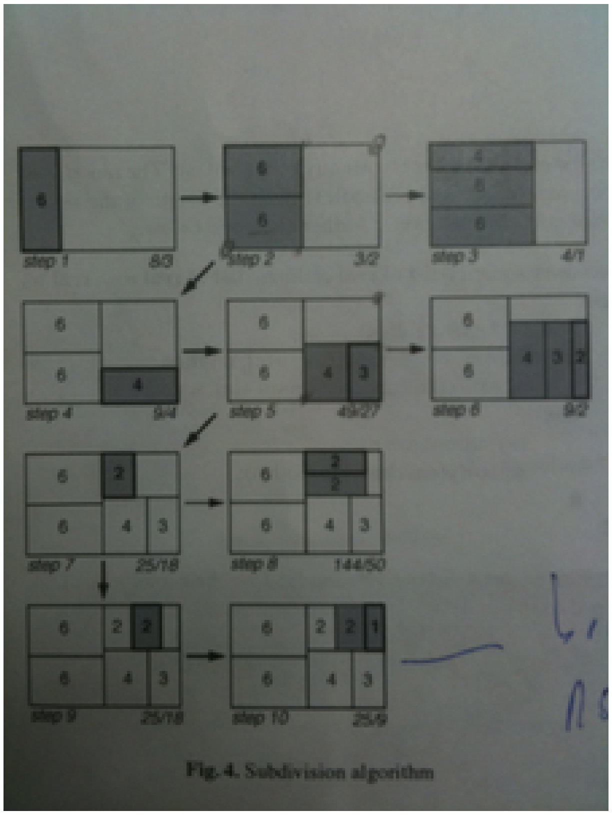

Figure 9: Squarifying algorithm

```
  (* Spread one on top of the other eg _
   *                                   _
   *                                   _
   *)
  | SpreadVertically
```

19a ⟨*function ratio_rect_dim* 19a⟩≡ (44e)

```
(* we want the ratio to be a close to 1 as possible (that is to be a square) *)
let ratio_rect_dim (w,h) =
  let res = max (w /. h) (h /. w) in
  (* assert (res >= 1.0); *)
  res

let _ = example (ratio_rect_dim (6.0, 4.0) = 1.5)
let _ = example (ratio_rect_dim (4.0, 6.0) = 1.5)
```

19b ⟨*function worst* 19b⟩≡ (44e)

```
(* On the running example, at the first step we want to add the rect of
 * size 6 on the left, alone, and its aspect ratio will be 8/3.
 * Indeed its height is fixed (4) and so his width is
 * whatever that must lead to an area of 6, that is 6/4 (1.5)
 * which leads then to an aspect ratio of 4 vs 1.5 = 4 / 1.5 = 8/3.
 * If we add 2 rect of size 6, then their aspect ratio is 1.5 which is
 * better
 *)

let worst elems_in_row  size_side_row =
  let s = Common.sum_float elems_in_row in
  let rplus = Common.maximum elems_in_row in
  let rminus = Common.minimum elems_in_row in

  (* cf formula in paper *)
  max ((Common.square size_side_row *. rplus) /. Common.square s)
      (Common.square s /.  (Common.square size_side_row *. rminus))

let _ = example
  (worst [6.0] 4.0 = 8.0 /. 3.0) (* 2.66667 *)
let _ = example
  (worst [6.0;6.0] 4.0 = 3.0 /. 2.0) (* 1.5, which is close to 1 so better *)
let _ = example
  (worst [6.0;6.0;4.0] 4.0 = 4.0) (* 4.0, we regress *)
```

19c ⟨*function layout* 19c⟩≡ (44e)

```
(* We are given a fixed row which contains a set of elems that we have
 * to spread unoformly, just like in the original algorithm.
 *)
let layout row rect =

  let p = [| rect.p.x; rect.p.y |] in
  let q = [| rect.q.x; rect.q.y |] in

  let children = row in

  let stotal = children +> List.map fst +> Common.sum_float in
  let children = children +> List.map (fun (size, info) ->
    size /. stotal (* percentage *),
    size,
    info
  )
  in

  let res = ref [] in
  let spread =
    if rect_width rect >= rect_height rect
    then SpreadHorizontally
    else SpreadVertically
  in
  let axis_split =
    match spread with
    | SpreadHorizontally -> 0
    | SpreadVertically -> 1
  in
  let width = q.(axis_split) -. p.(axis_split) in

  children +> List.iter (fun (percent_child, size_child, info) ->

    q.(axis_split) <-
      p.(axis_split) +.
      percent_child *. width;
    let rect_here = {
      p = {  x = p.(0); y = p.(1); };
      q = {  x = q.(0); y = q.(1); }
    }
    in
    Common.push2 (size_child, info, rect_here) res;
    p.(axis_split) <- q.(axis_split);
  );
  !res
```

20 ⟨*function squarify_orig* 20⟩≡ (44e)

```
let rec (squarify_orig:
  ?verbose:bool ->
  (float * 'a) list -> (float * 'a) list -> rectangle ->
  (float * 'a * rectangle) list
  ) =
fun ?(verbose=false) children current_row rect ->
 (* does not work well because of float approximation.
  * assert(Common.sum_float (children ++ current_row) = rect_area rect);
  *)
 let (p, q) = rect.p, rect.q in

 let floats xs = List.map fst xs in

 (* First heuristic in the squarified paper *)
 let spread =
   if rect_width rect >= rect_height rect (* e.g. 6 x 4 rectangle *)
   then SpreadHorizontally
   else SpreadVertically
 in

 (* We now know what kind of row we want. If spread horizontally then
  * we will have a row on the left to fill and the size of the side of
  * this row is known and is the height of the rectangle (in our ex 4).
  * In the paper they call this variable 'width' but it's misleading.
  * Note that because we are in Horizontal mode, inside this left row,
  * things will be spreaded this time vertically.
  *)
 let size_side_row =
   match spread with
   | SpreadHorizontally -> rect_height rect
   | SpreadVertically -> rect_width rect
 in
 match children with
 | c::cs ->
     if null current_row ||
        (worst (floats (current_row ++ [c])) size_side_row)
         <=
         (worst (floats current_row)         size_side_row)
     then
       (* not yet optimal row, let's recurse *)
       squarify_orig cs (current_row ++ [c]) rect
     else begin
       (* optimal layout for the left row. We can fix it. *)
       let srow = Common.sum_float (floats current_row) in
       let stotal = Common.sum_float (floats (current_row ++ children)) in
       let portion_for_row = srow /. stotal in
   let row_rect, remaining_rect =
    match spread with
    | SpreadHorizontally ->
      let middle_x =
       (q.x -. p.x) *. portion_for_row
        +. p.x
      in
      {
       p = p;
       q = { x = middle_x; y = q.y };
      },
      {
       p = { x = middle_x; y = p.y};
       q = q;
      }

    | SpreadVertically ->
      let middle_y =
       (q.y -. p.y) *. portion_for_row
        +. p.y in
      {
       p = p;
       q = { x = q.x; y = middle_y;};
      },
      {
       p = { x = p.x; y = middle_y};
       q = q;
      }


   in
   if verbose then begin
    pr2 "layoutrow:";
    pr2_gen current_row;
    pr2 "row rect";
    pr2 (s_of_rectangle row_rect);
   end;

   let rects_row = layout current_row row_rect in
   let rects_remain = squarify_orig children [] remaining_rect in
   rects_row ++ rects_remain
  end
| [] ->
  if verbose then begin
   pr2 "layoutrow:";
   pr2_gen current_row;
```

```
        pr2 "row rect";
        pr2 (s_of_rectangle rect);
      end;

      layout current_row rect
```

23a ⟨*function squarify* 23a⟩≡ (44e)

```
  let squarify children rect =
    (* squarify_orig assume the sum of children = area rect *)
    let area = rect_area rect in
    let total = Common.sum_float (List.map fst children) in
    let children' = children +> List.map (fun (x, info) ->
      (x /. total) *. area,
      info
    )
    in
    squarify_orig children' [] rect
```

23b ⟨*function test_squarify* 23b⟩≡ (44e)

```
  let test_squarify () =
      pr2_gen (worst [6.0] 4.0);
      pr2_gen (worst [6.0;6.0] 4.0);
      pr2_gen (worst [6.0;6.0;4.0] 4.0);
    pr2_xxxxxxxxxxxxxxxxx ();
    squarify squarified_list_area_ex dim_rect_orig +> ignore;
    pr2_xxxxxxxxxxxxxxxxx ();
    squarify squarified_list_area_ex2 rect_ortho +> ignore;
    ()
```

23c ⟨*layout squarify* 23c⟩≡ (44e)

```
  let (squarify_layout: ('a, 'b) layout_func) =
   fun children _depth rect ->
    let children' = children +> Common.sort_by_key_highfirst in
    squarify children' rect

  let (squarify_layout_no_sort_size: ('a, 'b) layout_func) =
   fun children _depth rect ->
    squarify children rect
```

4.4 Ordered treemaps

[11]

23d ⟨*variable treemap_ex_ordered_2001* 23d⟩≡ (44e)

```
  let (treemap_ex_ordered_2001: (unit, unit) treemap) =
    let children = children_ex_ordered_2001 in
```

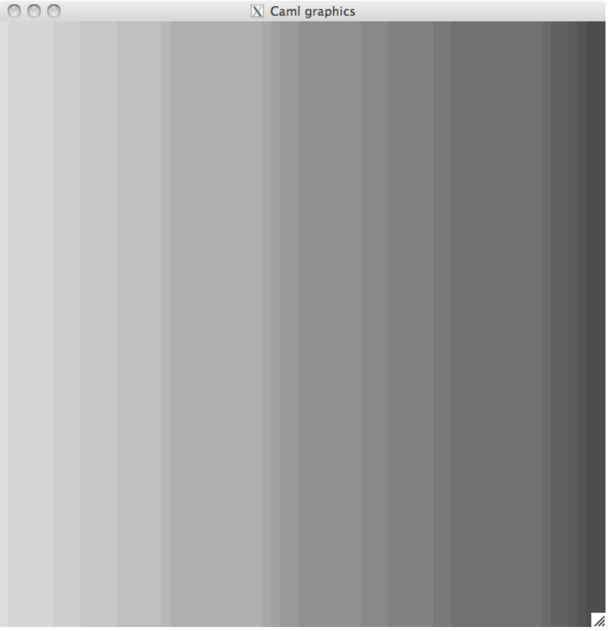

Figure 10: Orders in slice and dice

```
let children_treemap =
  children +> Common.index_list_1 +> List.map (fun (size, i) ->

    Leaf ({
      size = size;
      color = Color.color_of_string (spf "grey%d" (90 - (i * 3)));
      label = spf "size = %d" size;
    }, ())
  )
in
let total_size = Common.sum children in
Node (({
  size = total_size;
  color = Color.black;
  label = "";
}, ()), children_treemap
)
```

24 ⟨*ordered examples* 24⟩≡ (44e)

```
(* ref:
*)
```

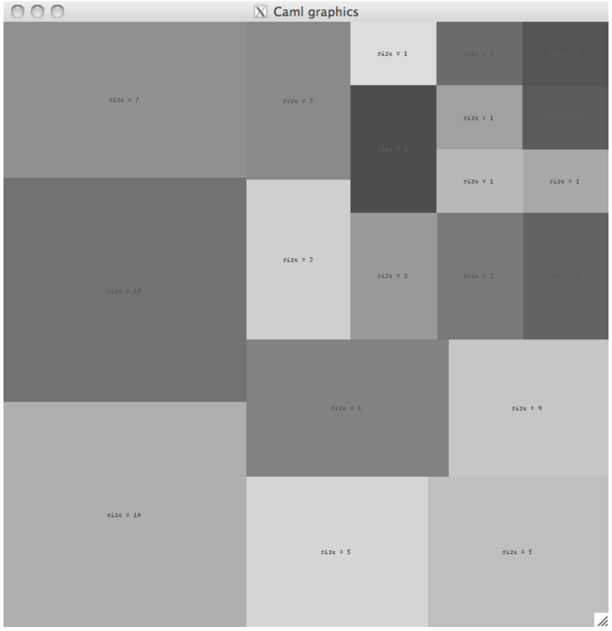

Figure 11: Orders in squarified

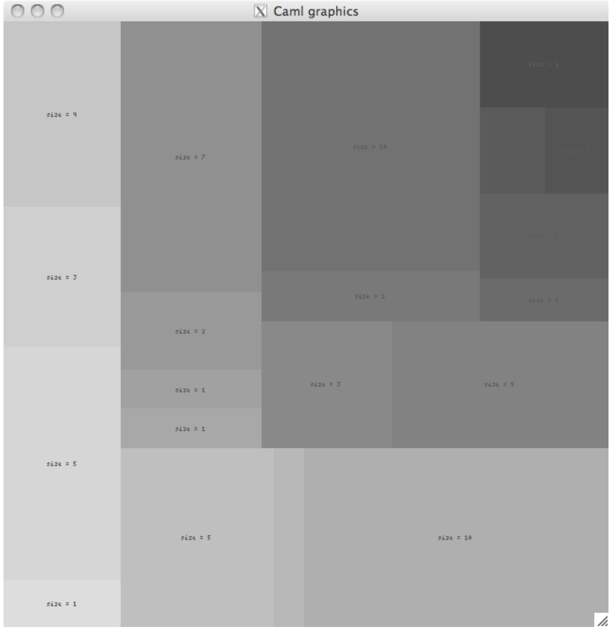

Figure 12: Orders in squarified no sort

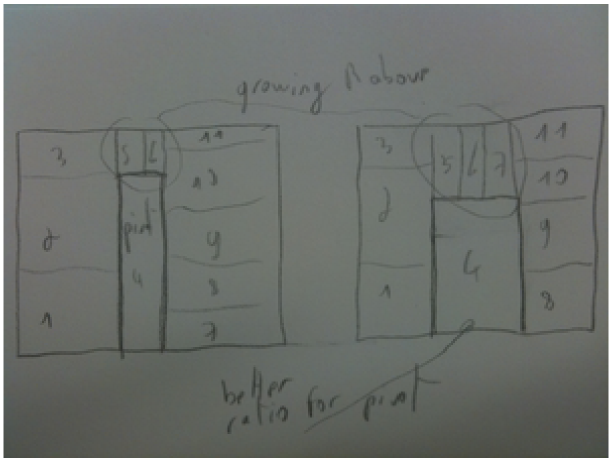

Figure 13: Finding a good split point

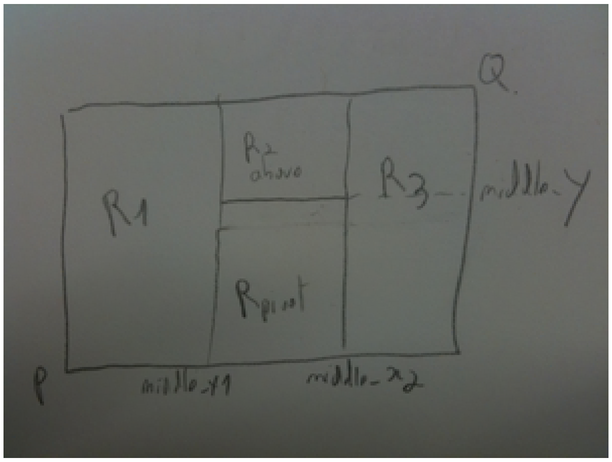

Figure 14: Pivot coordinates part1

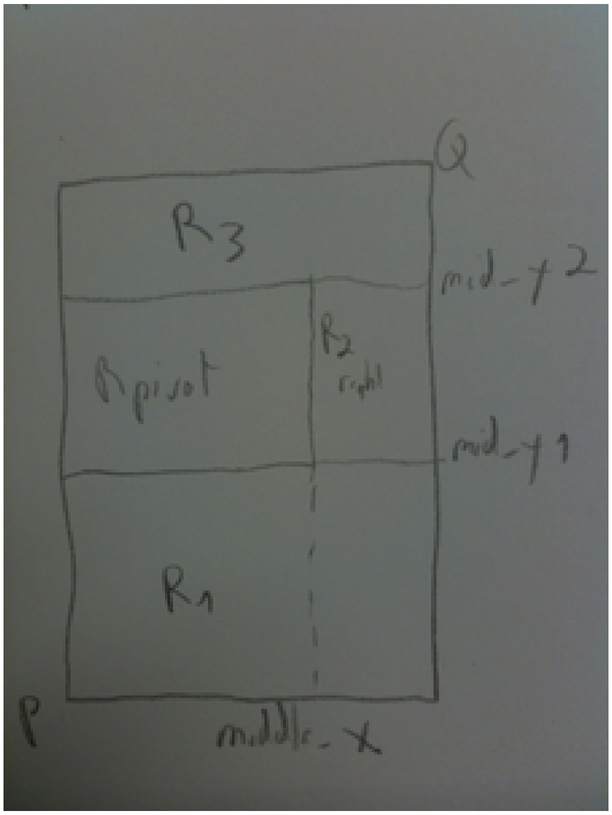

Figure 15: Pivot coordinates part2

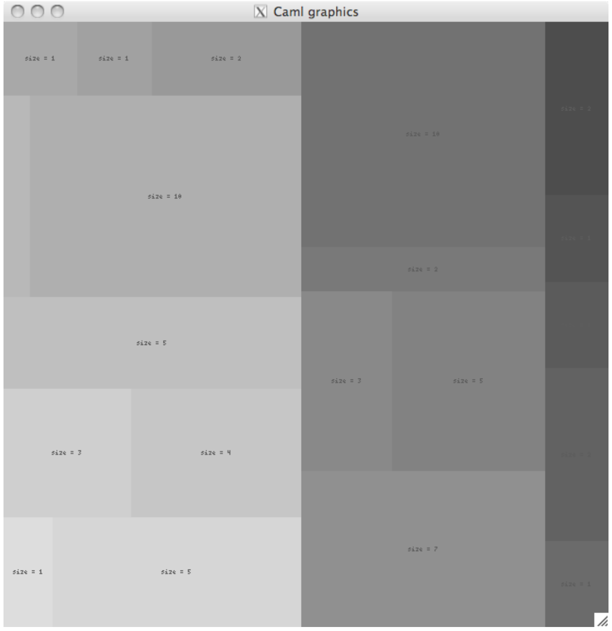

Figure 16: Ordered by middle treemap

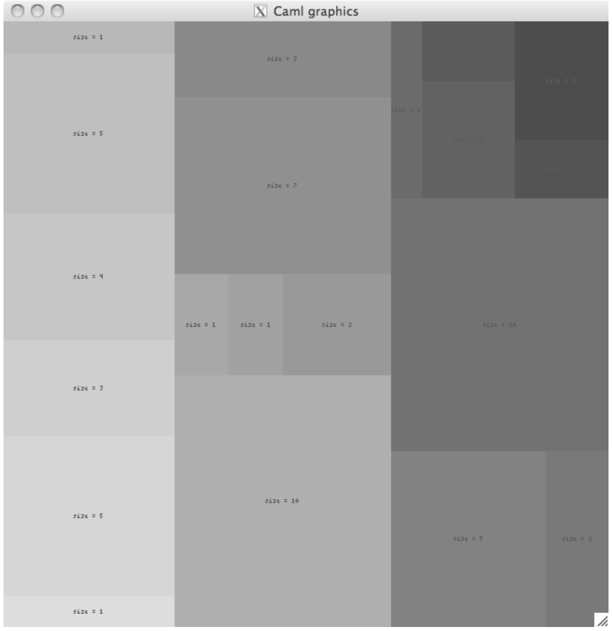

Figure 17: Ordered by size treemap

```
let children_ex_ordered_2001 = [
    1; 5; 3; 4; 5; 1;
    10; 1; 1; 2; 7; 3;
    5; 2; 10; 1; 2; 1;
    1; 2;
  ]
```

29a ⟨*type pivotized* 29a⟩≡ (44e)

```
type 'a pivotized = {
  left: 'a;
  right: 'a;
  pivot: 'a; (* this one should be singleton and the other a list *)
  above_pivot: 'a;
}
```

29b ⟨*function compute_rects_pivotized* 29b⟩≡ (44e)

```
let compute_rects_pivotized childs_pivotized rect spread =
  let (p, q) = rect.p, rect.q in

  let x = childs_pivotized in
  let size = {
    left = Common.sum_float (Common.map fst x.left);
    right = Common.sum_float (Common.map fst x.right);
    pivot = Common.sum_float (Common.map fst x.pivot);
    above_pivot = Common.sum_float (Common.map fst x.above_pivot);
  }
  in

  let total_size = size.left +. size.right +. size.pivot +. size.above_pivot in

  let portion_for_left = size.left /. total_size in
  let portion_for_right = size.right /. total_size in

  let portion_for_pivot_vs_above =
    (size.pivot ) /. (size.pivot +. size.above_pivot)
  in

  (* computing the rectangle of the left and right is easy as the
   * height is fixed (when we spread horizontally)
   *)
  match spread with
  | SpreadHorizontally ->
      (* TODO do something that adapt to rect ? lourd que rect
       * commence pas 0,0, ca fait faire des calculs en plus. *)
      let middle_x1 =
        p.x +. ((rect_width rect) *. portion_for_left)
      in
      let middle_x2 =
        q.x -.  ((rect_width rect) *. portion_for_right)
      in
      let middle_y =
        p.y +. ((rect_height rect) *. portion_for_pivot_vs_above)
      in
      { left = {
            p = p;
            q = { x = middle_x1; y = q.y } };
        right = {
            p = { x = middle_x2; y = p.y };
            q = q; };
        pivot = {
            p = { x = middle_x1; y = p.y};
            q = { x = middle_x2; y = middle_y}; };
        above_pivot = {
            p = { x = middle_x1; y = middle_y };
            q = { x = middle_x2; y = q.y; } };
      }

  | SpreadVertically ->
      (* just the reverse of previous code, x become y and vice versa *)
      let middle_y1 =
        p.y +. ((rect_height rect) *. portion_for_left)
      in
      let middle_y2 =
        q.y -. ((rect_height rect) *. portion_for_right)
      in

      let middle_x =
        p.x +. ((rect_width rect) *. portion_for_pivot_vs_above)
      in
      { left = {
          p = p;
          q = { x = q.x; y = middle_y1; } };
        right = {
          p = { x = p.x; y = middle_y2; };
          q = q; };
        pivot = {
          p = { x = p.x;  y = middle_y1; };
          q = { x = middle_x; y = middle_y2; } };
        above_pivot = {
          p = { x = middle_x; y = middle_y1; };
          q = { x = q.x; y = middle_y2; } }
      }
```

31a ⟨*function balayer_right_wrong* 31a⟩≡ (44e)

```
let rec balayer_right_wrong xs =
  match xs with
  | [] -> []
  | x::xs ->
      let first =
        [], x::xs
      in
      let last =
        x::xs, []
      in
      let rest = balayer_right_wrong xs in
      let rest’ = rest +> List.map (fun (start, theend) -> x::start, theend) in
      [first] ++ rest’ ++ [last]

let balayer_right xs =
  let n = List.length xs in
  let res = ref [] in
  for i = 0 to n do
    Common.push2 (take i xs, drop i xs) res;
  done;
  List.rev !res
let _ = example (balayer_right [1;2;3;2] =
    [
      [], [1;2;3;2];
      [1], [2;3;2];
      [1;2], [3;2];
      [1;2;3], [2];
      [1;2;3;2], [];
    ])
```

31b ⟨*function orderify_children* 31b⟩≡ (44e)

```
let rec orderify_children ?(pivotf=PivotBySize) xs rect =

  let rec aux xs rect =
    match xs with
    | [] -> []
    | [size, x] ->
        [size, x, rect]

    | x::y::ys ->

        let left, pivot, right =
          match pivotf with
          | PivotBySize ->
              let pivot_max = Common.maximum (xs +> List.map fst) in
```

```
      Common.split_when
       (fun x -> fst x = pivot_max) xs
    | PivotByMiddle ->
      let nmiddle = List.length xs / 2 in
      let start, thend = Common.splitAt nmiddle xs in

      start, List.hd thend, List.tl thend
   in

   let spread =
    if rect_width rect >= rect_height rect (* e.g. 6 x 4 rectangle *)
    then SpreadHorizontally
    else SpreadVertically
   in

   let right_combinations = balayer_right right in

   let scores_and_rects =
    right_combinations +> List.map (fun (above_pivot, right) ->

     let childs_pivotized =
      { left = left;
       pivot = [pivot];
       right = right;
       above_pivot = above_pivot;
      }
     in
     let rects = compute_rects_pivotized childs_pivotized rect spread in
     ratio_rect_dim (rect_width rects.pivot, rect_height rects.pivot),
     (rects,
     childs_pivotized)
    )
   in
   let best = Common.sort_by_key_lowfirst scores_and_rects +> List.hd in
   let (_score, (rects, childs_pivotized)) = best in

   (* pr2_gen rects; *)
   aux childs_pivotized.left rects.left ++
   aux childs_pivotized.pivot rects.pivot ++
   aux childs_pivotized.above_pivot rects.above_pivot ++
   aux childs_pivotized.right rects.right ++
   []
in
aux xs rect
```

32 ⟨*function test_orderify* 32⟩≡ (44e)

```
let test_orderify () =
  let xs = children_ex_ordered_2001 +> List.map float_of_int in
  let rect = rect_ortho in

  let fake_treemap = () in
  let children = xs +> List.map (fun size -> size, fake_treemap) in

  let layout = orderify_children children rect in
  pr2_gen layout
```

33a ⟨*layout ordered* 33a⟩≡ (44e)

```
let (ordered_layout: ?pivotf:pivot -> ('a, 'b) layout_func) =
 fun ?pivotf children depth rect ->
  orderify_children ?pivotf children rect
```

4.5 Cushion treemaps

[9]

5 Extra features

5.1 Nesting

5.2 Labeling

5.3 Interactivity

33b ⟨*signature display_treemap_interactive* 33b⟩≡ (63)

```
val display_treemap_interactive :
  ?algo:algorithm ->
  ?drawing_file_hook:
    (Figures.rect_pixel -> 'file -> 'file option Common.matrix -> unit) ->
  (* used to display file information in the status area *)
  ?info_of_file_under_cursor:(Graphics.status -> 'file -> string) ->
  ('dir, 'file) treemap ->
  screen_dim ->
  unit
```

33c ⟨*function update_mat_with_fileinfo* 33c⟩≡ (64)

```
let update_mat_with_fileinfo fileinfo mat rect =

  let ((x1,y1), (x2,y2)) = rect in

  for i = x1 to x2 - 1 do
    for j = y1 to y2 - 1 do
      mat.(i).(j) <- Some fileinfo;
    done
  done
```

34 ⟨*function display_treemap_interactive* 34⟩≡ (64)

```
let display_treemap_interactive
 ?algo
 ?drawing_file_hook
 ?(info_of_file_under_cursor=(fun _ _ -> ""))
 treemap
 dim
  =
  let dim = ref dim in
  let matrix_info = ref (
    display_treemap_algo
      ?algo
      ?drawing_file_hook
      treemap
      (!dim.w_view, !dim.h_view)
  )
  in
  while true do
    let status = Graphics.wait_next_event [
        Graphics.Mouse_motion;
        Graphics.Key_pressed;
        Graphics.Button_down;
        Graphics.Button_up;
      ]
    in
    let (x,y) = status.Graphics.mouse_x, status.Graphics.mouse_y in

    if x >= 0 && y >= 0 && x < !dim.w_view && y < !dim.h_view
    then begin

      (* clear the status area *)
      Graphics.set_color Graphics.white;
      Graphics.fill_rect 0 (!dim.h - !dim.h_status) !dim.w (!dim.h);

      Graphics.set_color Graphics.black;
      Graphics.moveto (0 + !dim.w / 2) (!dim.h - (!dim.h_status / 2));

      let info =
        try
          !matrix_info.(x).(y)
        with Invalid_argument(s) ->
          pr2 (spf "pb with coord (%d,%d).  %s" x y s);
          raise (Invalid_argument(s))
```

```
   in
   match info with
   | None -> pr2 "Impossible";
   | Some file ->
     let s = info_of_file_under_cursor status file in
     (* draw_string_centered (spf "x = %03d, y = %03d; info = %s" x y s); *)
     Graphics.set_font "-misc-*-*-*-*-12-*-*-*-*-*-*";
     draw_string_centered (spf "%s" s);
  end;

  (* a resize has taken place *)
  let w, h = Graphics.size_x (), Graphics.size_y () in
  if w <> !dim.w || h <> !dim.h
  then begin
   dim := current_dim ~w_legend:!dim.w_legend ~h_status:!dim.h_status;
   Graphics.clear_graph ();
   matrix_info :=
    display_treemap_algo
     ?algo
     ?drawing_file_hook
     treemap
     (!dim.w_view, !dim.h_view);
   (* draw_legend_hook !dim ? *)
  end
 done
```

35 ⟨*function info_of_file_under_cursor_default* 35⟩≡ (64)

```
let info_of_file_under_cursor_default = fun status (f, _) ->
 let s = f in
 if status.Graphics.button
 then begin
  pr2 (spf "%s" f);
  (* Sys.command (spf "/home/pad/packages/Linux/bin/emacsclient -n %s" f) +> ignore; *)
 end;
 if status.Graphics.keypressed (* Graphics.key_pressed () *)
 then raise (UnixExit 0);
 s
```

6 JSON reader

```
$ find .
.
./a
```

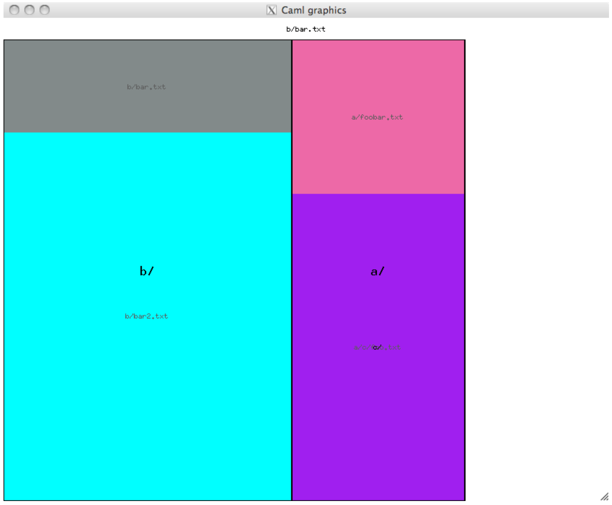

Figure 18: Treemap from `ex.json`

```
./a/c
./a/c/foo.txt
./a/foobar.txt
./b
./b/bar.txt
./b/bar2.txt
```

```
$ ./treemap_viewer -algorithm squarified  examples/treemap/ex.json
```

36 ⟨*ex.json* 36⟩≡

```
{
  "kind": "Node",     "label": ".",
  "children": [
    {
      "kind": "Node",    "label": "a/",
      "children": [
        {
          "kind": "Node",  "label": "c/",
          "children": [
            {
              "kind": "Leaf", "size": 2, "color": "purple",
              "label": "a/c/foo.txt"
            }
          ]
        },
        {
          "kind": "Leaf", "size": 1, "color": "HotPink2",
          "label": "a/foobar.txt"
        }
      ]
    },
    {
      "kind": "Node", "label": "b/",
      "children": [
        {
          "kind": "Leaf", "size": 1, "color": "azure4",
          "label": "b/bar.txt"
        },
        {
          "kind": "Leaf", "size": 4, "color": "cyan",
          "label": "b/bar2.txt"
        }
] } ] }
```

37a ⟨*signature treemap_of_json* 37a⟩≡ (61a)

```
val treemap_of_json:
  Json_type.json_type ->
  (Common.dirname, Common.filename * int) Treemap.treemap
```

37b ⟨*signature json_of_treemap* 37b⟩≡ (61a)

```
val json_of_treemap:
  ('dir, 'file) Treemap.treemap -> Json_type.json_type
```

37c ⟨*function treemap_of_json* 37c⟩≡ (61b)

```
(* cf json_of_treemap_basic below. Just do reverse operation *)
let rec treemap_of_json j =
  match j with
  | J.Object [
      "kind", J.String "Node";
      "label", J.String s;
      "children", J.Array xs;
    ] ->
      let children = xs +> List.map treemap_of_json in

      let sizes = children +> List.map Treemap.size_of_treemap_node in
      let size = Common.sum sizes in

      let rect = {
        label = s;
        color = Color.black;
        size = size;
      }
```

```
        in
        Node ((rect, s), children)

    | J.Object [
        "kind", J.String "Leaf";
        "size", J.Int size;
        "color", J.String scolor;
        "label", J.String lbl;
      ] ->
        let rect = {
          label = lbl;
          color = Color.color_of_string scolor;
          size = size;
        }
        in
        Leaf (rect, (lbl, size))

    | _ ->
        failwith "wrong format"
```

38a ⟨*function json_of_color* 38a⟩≡ (61b)

```
let json_of_color c = J.String (Color.string_of_color c)
```

38b ⟨*function json_of_treemap* 38b⟩≡ (61b)

```
(* I was first using ocamltarzan to auto generate the json_of, but it
 * leds to verbosity, so I ended up manually coding it.
 *)
let rec (json_of_treemap: ('a, 'b) Treemap.treemap -> J.json_type)
 = function
  | Node (((rect, _a), xs)) ->
      let { size = v_size; color = v_color; label = v_label } = rect in

      let bnds = [] in

      let children =
        J.Array (List.map json_of_treemap xs)
      in
      let bnd = ("children", children) in
      let bnds = bnd :: bnds in

      let arg = J.String v_label in
      let bnd = ("label", arg) in
      let bnds = bnd :: bnds in

      let arg = J.String "Node" in
      let bnd = ("kind", arg) in
      let bnds = bnd :: bnds in

      J.Object bnds

  | Leaf (rect, _b) ->
      let { size = v_size; color = v_color; label = v_label } = rect in

      let bnds = [] in
      let arg = J.String v_label in
      let bnd = ("label", arg) in
      let bnds = bnd :: bnds in
      let arg = json_of_color v_color in
      let bnd = ("color", arg) in
      let bnds = bnd :: bnds in
      let arg = J.Int v_size in
      let bnd = ("size", arg) in
      let bnds = bnd :: bnds in

      let arg = J.String "Leaf" in
      let bnd = ("kind", arg) in
      let bnds = bnd :: bnds in
      J.Object bnds
```

39 ⟨*function test_json_of* 39⟩≡ (61b)

```
let test_json_of dir =
  let maxc = 256 in
  let tree = tree_of_dir ~file_hook:(fun file -> Common.filesize file) dir in
  let treemap = treemap_of_tree
    ~size_of_leaf:(fun (f, intleaf) -> intleaf)
    ~color_of_leaf:(fun (f, intleaf) ->
      Color.rgb (Random.int maxc) (Random.int maxc) (Random.int maxc)
    )
    ~label_of_dir:(fun dir -> basename dir)
    ~label_of_file:(fun (f, intleaf) -> f)
    tree
  in
  let json =
    json_of_treemap
      (*
      (fun _ -> J.Null)
      (fun _ -> J.Null)
      *)
      treemap in
  let s = Json_out.string_of_json json in
  pr s
```

40a ⟨*function test_of_json* 40a⟩≡ (61b)

```
  let test_of_json file =
      let json = Json_in.load_json file in
      let treemap = treemap_of_json json in

      let json2 = json_of_treemap treemap in
      let s = Json_out.string_of_json json2 in
      pr s
```

40b ⟨*treemap_json actions* 40b⟩≡ (61b)

```
    "-test_json_of", "<dir>",
    Common.mk_action_1_arg test_json_of;
    "-test_of_json", "<file>",
    Common.mk_action_1_arg test_of_json;
```

7 Applications

7.1 Disk statistics

KDirStat WindowsStat MacosStat

40c ⟨*signature tree_of_dir* 40c⟩≡ (42)

```
  type directory_sort =
    | NoSort
    | SortDirThenFiles
    | SortDirAndFiles
    | SortDirAndFilesCaseInsensitive

  val tree_of_dir:
    ?filter_file:(Common.filename -> bool) ->
    ?filter_dir:(Common.dirname -> bool) ->
    ?sort:directory_sort ->
    file_hook:(Common.filename -> 'a) ->
    Common.dirname ->
    (Common.dirname, Common.filename * 'a) Common.tree
```

40d ⟨*function tree_of_dir* 40d⟩≡ (44e)

```
  let tree_of_dir2
    ?(filter_file=(fun _ -> true))
    ?(filter_dir=(fun _ -> true))
    ?(sort=SortDirAndFilesCaseInsensitive)
    ~file_hook
    dir
   =
    let rec aux dir =
      let subdirs =
        Common.readdir_to_dir_list dir +> List.map (Filename.concat dir) in
      let files =
        Common.readdir_to_file_list dir +> List.map (Filename.concat dir) in

      let subdirs =
        subdirs +> Common.map_filter (fun dir ->
          if filter_dir dir
          then Some (dir, aux dir)
          else None
        )
      in
      let files =
        files +> Common.map_filter (fun file ->
          if filter_file file
          then Some (file, (Leaf (file, file_hook file)))
          else None
        )
      in

      let agglomerated =
        match sort with
        | NoSort -> subdirs ++ files
        | SortDirThenFiles ->
            Common.sort_by_key_lowfirst subdirs ++
            Common.sort_by_key_lowfirst files
        | SortDirAndFiles ->
            Common.sort_by_key_lowfirst (subdirs ++ files)
        | SortDirAndFilesCaseInsensitive ->
            let xs = (subdirs ++ files) +> List.map (fun (s, x) ->
              lowercase s, x
            )
            in
            Common.sort_by_key_lowfirst xs
      in
      let children = List.map snd agglomerated in
      Node(dir, children)
    in
    aux dir
```

7.2 Source code architecture visualization

archi

linux fekete.

sgrep/slayer plugin, slayer :)

7.3 Code coverage (tests, deadcode, etc)

7.4 Version-control visualization

git

SeeSoft. Work by UIUC on cvs and visualization. Also video of evolution of java code.

8 Conclusion

Hope you like it.

[12]

A Extra Code

A.1 treemap.mli

42 ⟨*treemap.mli* 42⟩≡

```
  open Figures

  ⟨type treemap 5⟩

  val xy_ratio : float

  val rect_ortho: rectangle

  type treemap_rendering = treemap_rectangle list
   and treemap_rectangle = {
     tr_rect: rectangle;
     tr_color: int (* Simple_color.color *);
     tr_label: string;
     tr_depth: int;
     tr_is_node: bool;
   }

  ⟨type screen_dim 9a⟩

  ⟨type algorithm 11b⟩

  ⟨type layout_func 12c⟩

  ⟨signature algos 11c⟩

  val render_treemap_algo:
```

```
  ?algo:algorithm -> ('dir, 'file) treemap -> treemap_rendering



(* treemap maker, see also treemap_json.ml *)
⟨signature treemap_of_tree 43⟩

(* tree maker, see also Common.tree2_of_files *)
⟨signature tree_of_dir 40c⟩

val tree_of_dir_or_file:
  ?filter_file:(Common.filename -> bool) ->
  ?filter_dir:(Common.dirname -> bool) ->
  ?sort:directory_sort ->
  file_hook:(Common.filename -> 'a) ->
  Common.path ->
  (Common.dirname, Common.filename * 'a) Common.tree

val tree_of_dirs_or_files:
  ?filter_file:(Common.filename -> bool) ->
  ?filter_dir:(Common.dirname -> bool) ->
  ?sort:directory_sort ->
  file_hook:(Common.filename -> 'a) ->
  Common.path list ->
  (Common.dirname, Common.filename * 'a) Common.tree


(* internal functions *)
⟨signature treemap accessors 44b⟩

⟨signature algorithm accessors 44c⟩

(* tests *)
⟨signature tree and treemap examples 6a⟩

val actions : unit -> Common.cmdline_actions
```

43 ⟨*signature treemap_of_tree* 43⟩≡ (42)

```
val treemap_of_tree :
  size_of_leaf:('file -> int) ->
  color_of_leaf:('file -> Simple_color.color) ->
  ?label_of_file:('file -> string) ->
  ?label_of_dir:('dir -> string) ->
  ('dir, 'file) Common.tree ->
```

```
    ('dir, 'file) treemap
```

44a ⟨*signature graphic helpers* 9b⟩+≡ (63) ◁9b

```
  val info_of_file_under_cursor_default :
    Graphics.status -> (Common.filename * 'a) -> string

  val current_dim:
    w_legend:int -> h_status:int -> screen_dim
```

44b ⟨*signature treemap accessors* 44b⟩≡ (42)

```
  val color_of_treemap_node :
    ('a, 'b) treemap -> Simple_color.color
  val size_of_treemap_node :
    ('a, 'b) treemap  -> int
```

44c ⟨*signature algorithm accessors* 44c⟩≡ (42)

```
  val s_of_algo: algorithm -> string
  val algo_of_s: string -> algorithm
```

44d ⟨*signature test treemap functions* 44d⟩≡ (63)

```
  val test_treemap_manual : unit -> unit
  val test_treemap_tree : algorithm -> int -> unit
  val test_treemap_dir : string -> algorithm -> unit
```

A.2 treemap.ml

44e ⟨*treemap.ml* 44e⟩≡

```
  ⟨Facebook copyright 4⟩

  open Common

  module F = Figures
  open Figures

  module Color = Simple_color

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  ⟨type treemap 5⟩
    (* with tarzan *)
```

```
⟨type algorithm 11b⟩

⟨variable algos 12a⟩

⟨type screen_dim 9a⟩


⟨type rectangle1 7a⟩

(* A cleaner rectangle type, not tied to the seminal paper design decisions *)

(* Now that my treemap visualizer uses a minimap, it does not completely
 * use the full width.
 * old: was 16/9 = 1.777777
 *)
let xy_ratio = 1.6

(* The dimentions are in a  [0.0-1.0] range for y and [0.0-xyratio] for x,
 * where xyratio is used to cope with most 16/9 screens.
  *)
let rect_ortho =
  { p = {x = 0.0; y = 0.0; }; q = { x = xy_ratio; y = 1.0} }

(* the dimentions are in a  [0.0-1.0] range *)
type treemap_rendering = treemap_rectangle list
 and treemap_rectangle = {
   tr_rect: rectangle;
   tr_color: int (* Simple_color.color *);
   tr_label: string;
   tr_depth: int;
   tr_is_node: bool;
 }
 (* with tarzan *)

⟨type layout_func 12c⟩


(*****************************************************************************)
(* Accessors *)
(*****************************************************************************)

⟨function treemap accessors 54a⟩

⟨function algorithm accessors 54c⟩
```

```
(*****************************************************************************)
(* Treemap Helpers *)
(*****************************************************************************)

⟨function treemap_of_tree 56⟩

let treemap_of_tree ~size_of_leaf  ~color_of_leaf
    ?label_of_file ?label_of_dir tree =
 Common.profile_code "Treemap.treemap_of_tree" (fun () ->
   treemap_of_tree2 ~size_of_leaf  ~color_of_leaf
     ?label_of_file ?label_of_dir tree)

(*****************************************************************************)
(* Treemap algorithms *)
(*****************************************************************************)

(*---------------------------------------------------------------------------*)
(* basic algorithm *)
(*---------------------------------------------------------------------------*)

(* display_treemap and display_treemap_generic are now in
 * in treemap_graphics.ml, because of Graphics dependency.
 *)

(*---------------------------------------------------------------------------*)
(* slice and dice algorithm layout *)
(*---------------------------------------------------------------------------*)

⟨layout slice and dice 15a⟩

(*---------------------------------------------------------------------------*)
(* squarified algorithm *)
(*---------------------------------------------------------------------------*)

⟨squarified examples 16⟩

⟨type split 17⟩

⟨function ratio_rect_dim 19a⟩

⟨function worst 19b⟩

⟨function layout 19c⟩

(* the main algorithmic part of squarifying *)
⟨function squarify_orig 20⟩
```

```
⟨function squarify 23a⟩


⟨function test_squarify 23b⟩

⟨layout squarify 23c⟩


(*---------------------------------------------------------------------------*)
(* Ordered squarified algorithm *)
(*---------------------------------------------------------------------------*)

⟨ordered examples 24⟩

⟨type pivotized 29a⟩

⟨function compute_rects_pivotized 29b⟩

⟨function balayer_right_wrong 31a⟩

⟨function orderify_children 31b⟩

⟨function test_orderify 32⟩


⟨layout ordered 33a⟩

(*---------------------------------------------------------------------------*)
(* cushion algorithm *)
(*---------------------------------------------------------------------------*)

(* TODO *)

(*---------------------------------------------------------------------------*)
(* frontend *)
(*---------------------------------------------------------------------------*)

let layoutf_of_algo algo =
  match algo with
  | Classic -> slice_and_dicing_layout
  | Squarified -> squarify_layout
  | SquarifiedNoSort -> squarify_layout_no_sort_size
  | Ordered pivotf -> ordered_layout ~pivotf
```

```
let (render_treemap_algo2:
      ?algo:algorithm -> ('dir, 'file) treemap -> treemap_rendering) =
 fun ?(algo=Classic) treemap ->
  let flayout = layoutf_of_algo algo in

  let treemap_rects = ref [] in

  let rec aux_treemap root rect ~depth =
    let (p,q) = rect.p, rect.q in

    if not (valid_rect rect)
    then () (* TODO ? warning ? *)
    else

    (match root with
    | Leaf (tnode, fileinfo) ->
        let color = color_of_treemap_node root in

        Common.push2 {
          tr_rect = rect;
          tr_color = color;
          tr_label = tnode.label;
          tr_depth = depth;
          tr_is_node = false;
        } treemap_rects;


    | Node (mode, children) ->

       (* let's draw some borders. Far better to see the structure. *)
        Common.push2 {
          tr_rect = rect;
          tr_color = Color.black;
          tr_label = (fst mode).label;
          tr_depth = depth;
          tr_is_node = true;
        } treemap_rects;

        (* does not work, weird *)
        let border =
          match depth with
          | 1 -> 0.0
          | 2 -> 0.002
          | 3 -> 0.001
          | 4 -> 0.0005
          | 5 -> 0.0002
          | _ -> 0.0
        in
        let p = {
          x = p.x +. border;
          y = p.y +. border;
        }
        in
        let q = {
          x = q.x -. border;
          y = q.y -. border;
        }
        in
        (* todo? can overflow ... check still inside previous rect *)
        let rect = { p = p; q = q } in

        let children' =
          children +> List.map (fun child ->
            float_of_int (size_of_treemap_node child),
            child
          )
        in

        let rects_with_info =
          (* generic call *)
          flayout children' depth rect
        in
        (* less: assert rects_with_info are inside rect ? *)

        rects_with_info +> List.iter (fun (x, child, rect) ->
          aux_treemap child rect ~depth:(depth + 1)
        );


    )
  in
  aux_treemap treemap rect_ortho ~depth:1;

  List.rev !treemap_rects

let render_treemap_algo ?algo x =
  Common.profile_code "Treemap.render_treemap" (fun () ->
    render_treemap_algo2 ?algo x)

(*****************************************************************************)
```

```
(* Main display function  *)
(******************************************************************************)

(* now in treemap_graphics.ml *)

(******************************************************************************)
(* Source converters  *)
(******************************************************************************)

type directory_sort =
  | NoSort
  | SortDirThenFiles
  | SortDirAndFiles
  | SortDirAndFilesCaseInsensitive

⟨function tree_of_dir 40d⟩

(* specialized version *)
let tree_of_dir3
  ?(filter_file=(fun _ -> true))
  ?(filter_dir=(fun _ -> true))
  ?(sort=SortDirAndFilesCaseInsensitive)
  ~file_hook
  dir
 =
  if sort <> SortDirAndFilesCaseInsensitive
  then failwith "Only SortDirAndFilesCaseInsensitive is handled";

  let rec aux dir =

    let children = Sys.readdir dir in
    let children = Array.map (fun x -> Common.lowercase x, x) children in

    Array.fast_sort (fun (a1, b1) (a2, b2) -> compare a1 a2) children;

    let res = ref [] in

    children +> Array.iter (fun (_, f) ->
      let full = Filename.concat dir f in

      let stat = Common.unix_lstat_eff full in

      match stat.Unix.st_kind with
      | Unix.S_REG ->
          if filter_file full
          then Common.push2 (Leaf (full, file_hook full)) res
      | Unix.S_DIR ->
          if filter_dir full
          then Common.push2 (aux full) res
      (* symlink ?? *)
      | _ -> ()
    );
    Node(dir, List.rev !res)
  in
  aux dir


let tree_of_dir ?filter_file ?filter_dir ?sort ~file_hook a =
  Common.profile_code "Treemap.tree_of_dir" (fun () ->
    tree_of_dir3 ?filter_file ?filter_dir ?sort ~file_hook a)

let rec tree_of_dir_or_file
  ?filter_file
  ?filter_dir
  ?sort
  ~file_hook
  path
 =
 if Common.is_directory path
 then
   tree_of_dir ?filter_file ?filter_dir ?sort ~file_hook path
 else Leaf (path, file_hook path)



(* Some nodes may have stuff in common that we should factor.
 * todo: factorize code with Common.tree_of_files
 *)
let add_intermediate_nodes root_path nodes =
  let root = chop_dirsymbol root_path in
  if not (Common.is_absolute root)
  then failwith ("must pass absolute path, not: " ^ root);

  let root = Common.split "/" root in

  (* extract dirs and file from file, e.g. ["home";"pad"], "__flib.php", path *)
  let xs = nodes +> List.map (fun x ->
    match x with
    | Leaf (file, _) -> Common.dirs_and_base_of_file file, x
    | Node (dir, _) -> Common.dirs_and_base_of_file dir, x
  )
  in
  (* remove the root part *)
  let xs = xs +> List.map (fun ((dirs, base), node) ->
    let n = List.length root in
    let (root', rest) =
      Common.take n dirs,
      Common.drop n dirs
    in
    assert(root' =*= root);
    (rest, base), node
  )
  in
  (* now ready to build the tree recursively *)
  let rec aux current_root xs =
    let files_here, rest =
      xs +> List.partition (fun ((dirs, base), _) -> null dirs)
    in
    let groups =
      rest +> group_by_mapped_key (fun ((dirs, base),_) ->
        (* would be a file if null dirs *)
        assert(not (null dirs));
        List.hd dirs
      ) in

    let nodes =
      groups +> List.map (fun (k, xs) ->
        let xs' = xs +> List.map (fun ((dirs, base), node) ->
          (List.tl dirs, base), node
        )
        in
        let dirname = Filename.concat current_root k in
        Node (dirname, aux dirname xs')
      )
    in
    let leaves = files_here +> List.map (fun ((_dir, base), node) ->
      node
    ) in
    nodes ++ leaves
  in
  aux root_path xs




let tree_of_dirs_or_files2
  ?filter_file
  ?filter_dir
  ?sort
  ~file_hook
  paths
 =
  match paths with
  | [] -> failwith "tree_of_dirs_or_files: empty list"
  | [x] ->
      tree_of_dir_or_file ?filter_file ?filter_dir ?sort ~file_hook x
  | xs ->
      let nodes =
        xs +> List.map (fun x ->
          tree_of_dir_or_file ?filter_file ?filter_dir ?sort ~file_hook x
        )
      in
      let root = Common.common_prefix_of_files_or_dirs xs in
      let nodes = add_intermediate_nodes root nodes in
      Node (root, nodes)

let tree_of_dirs_or_files ?filter_file ?filter_dir ?sort ~file_hook x =
  Common.profile_code "Treemap.tree_of_dirs_or_files" (fun () ->
    tree_of_dirs_or_files2 ?filter_file ?filter_dir ?sort ~file_hook x
  )
(****************************************************************************)
(* Testing *)
(****************************************************************************)

⟨concrete rectangles example 57⟩



⟨variable tree_ex_shneiderman_1991 6b⟩

⟨variable tree_ex_wijk_1999 15b⟩

⟨variable treemap_ex_ordered_2001 23d⟩







(****************************************************************************)
```

```
(* Actions *)
(*****************************************************************************)

let actions () = [
  ⟨treemap actions 60⟩
]
```

54a ⟨*function treemap accessors* 54a⟩≡ (44e)

```
let color_of_treemap_node x =
  match x with
  | Node (({color = c}, _), _) -> c
  | Leaf (({color = c}, _)) -> c

let size_of_treemap_node x =
  match x with
  | Node (({size = s}, _), _) -> s
  | Leaf (({size = s}, _)) -> s
```

54b ⟨*function current_dim* 54b⟩≡ (64)

```
let current_dim ~w_legend ~h_status =

  let w, h = Graphics.size_x (), Graphics.size_y () in

  let w_view, h_view =
    Graphics.size_x () - w_legend,
    Graphics.size_y () - h_status
  in

  {
    w = w;
    h = h;
    w_view = w_view;
    h_view = h_view;
    h_status = h_status;
    w_legend = w_legend;
  }
```

54c ⟨*function algorithm accessors* 54c⟩≡ (44e)

```
let algo_of_s algo =
  match algo with
  | "classic" -> Classic
  | "squarified" -> Squarified
  | "squarified_no_sort" -> SquarifiedNoSort
  | "ordered" -> Ordered PivotBySize
  | "ordered_by_size" -> Ordered PivotBySize
  | "ordered_by_middle" -> Ordered PivotByMiddle
  | "default" -> Ordered PivotByMiddle
  | _ -> failwith "not a valid algorithm"

let s_of_algo algo =
  match algo with
  | Classic -> "classic"
  | Squarified -> "squarified"
  | SquarifiedNoSort -> "squarified_no_sort"
  | Ordered PivotBySize -> "ordered_by_size"
  | Ordered PivotByMiddle -> "ordered_by_middle"
```

55 ⟨*graphic helpers* 55⟩≡ (64)

```
let draw_string_centered str =
  let (w, h) = Graphics.text_size str in
  Graphics.rmoveto (- w / 2) (- h / 2);
  Graphics.draw_string str

let draw_text_center_rect_float_ortho ((x1, y1),(x2, y2)) color (w, h) str =
  let w = float_of_int w in
  let h = float_of_int h in

  let x1, y1 = int_of_float (x1 *. w), int_of_float (y1 *. h) in
  let x2, y2 = int_of_float (x2 *. w), int_of_float (y2 *. h) in

  let w = (x2 - x1) in
  let h = (y2 - y1) in

  Graphics.set_color color;
  Graphics.moveto (x1 + w / 2 ) (y1 + h / 2);
  let (w2, h2) = Graphics.text_size str in
  if str <> "" && w2 < w && h2 < h
  then begin
    (* does not work :( Graphics.set_text_size 40; *)
    draw_string_centered str;
    (*
    pr2 str;
    pr2_gen (x1, y1);
    *)
  end;
  ()


let draw_label rect (w, h) depth label ~is_dir =
  let (p, q) = rect.p, rect.q in
  let font_label_opt =
    if is_dir then
      match depth with
      | 1 -> None
      | 2 -> Some  "-misc-*-*-*-*-20-*-*-*-*-*-*"
      | 3 -> Some  "-misc-*-*-*-*-10-*-*-*-*-*-*"
      | 4 -> Some "-misc-*-*-*-*7-*-*-*-*-*-*"
      | _ -> None
    else
      Some "-misc-*-*-*-*-6-*-*-*-*-*-*"
  in

  font_label_opt +> Common.do_option (fun font ->
    Graphics.set_font font;

    draw_text_center_rect_float_ortho
      ((p.x, p.y),
      (q.x, q.y))
      (if is_dir then Graphics.black else Color.c "grey37")
      (w, h)
      label
  )
```

56 ⟨*function treemap_of_tree* 56⟩≡ (44e)

```
let treemap_of_tree2
    ~size_of_leaf
    ~color_of_leaf
    ?(label_of_file=(fun _ -> ""))
    ?(label_of_dir=(fun _ -> ""))
    tree =
  let rec aux tree =
    match tree with
    | Node (nodeinfo, xs) ->
        let sizeme = ref 0 in

        let child = List.map (fun x ->
          let (res, size) = aux x in
          sizeme := !sizeme + size;
          res
        ) xs
        in
        (* old:
         * let children = xs +> List.map aux in
         * let child = children +> List.map fst in
         * let sizes = children +> List.map snd in
```

```
     * let sizeme = Common.sum sizes in
     *)
    let sizeme = !sizeme in
    Node((
      {
        size = sizeme;
        color = Color.black; (* TODO ? nodes have colors ? *)
        label = label_of_dir nodeinfo;
      }, nodeinfo),
        child), sizeme
  | Leaf leaf ->
    let sizeme = size_of_leaf leaf in
    let nodeinfo = leaf in
    Leaf((
      {
        size = sizeme;
        color = color_of_leaf leaf;
        label = label_of_file leaf;
      }, nodeinfo)
    ), sizeme
 in
 let (tree, _size) = aux tree in
 tree
```

57 ⟨*concrete rectangles example* 57⟩≡ (44e)

```
(* src: python treemap.py
 * lower, upper, rgb
 *)
let treemap_rectangles_ex = [
 [0.0, 0.0], [1.0, 1.0],
 [0.0, 0.0], [0.27659574468085107, 1.0],
 [0.0, 0.0], [0.27659574468085107, 0.38461538461538464],
 [0.0, 0.38461538461538464], [0.27659574468085107, 1.0],
 [0.0, 0.38461538461538464], [0.10372340425531915, 1.0],
 [0.10372340425531915, 0.38461538461538464], [0.27659574468085107, 1.0],
 [0.27659574468085107, 0.0], [0.36170212765957449, 1.0],
 [0.36170212765957449, 0.0], [0.8936170212765957, 1.0],
 [0.36170212765957449, 0.0], [0.8936170212765957, 0.20000000000000001],
 [0.36170212765957449, 0.20000000000000001], [0.8936170212765957, 0.28000000000000003],
 [0.36170212765957449, 0.28000000000000003], [0.8936170212765957, 0.76000000000000001],
 [0.36170212765957449, 0.28000000000000003], [0.45035460992907805, 0.76000000000000001],
 [0.45035460992907805, 0.28000000000000003], [0.58333333333333337, 0.76000000000000001],
 [0.58333333333333337, 0.28000000000000003], [0.8936170212765957, 0.76000000000000001],
 [0.58333333333333337, 0.28000000000000003], [0.8936170212765957, 0.48571428571428577],
 [0.58333333333333337, 0.48571428571428577], [0.8936170212765957, 0.62285714285714289],
 [0.58333333333333337, 0.62285714285714289], [0.8936170212765957, 0.76000000000000001],
  [0.36170212765957449, 0.76000000000000001], [0.8936170212765957, 1.0],                (
  [0.36170212765957449, 0.76000000000000001], [0.62765957446808507, 1.0],               (
  [0.62765957446808507, 0.76000000000000001], [0.8936170212765957, 1.0],                (
  [0.8936170212765957, 0.0], [1.0, 1.0],
  [0.8936170212765957, 0.0], [1.0, 0.59999999999999998],
  [0.8936170212765957, 0.59999999999999998], [1.0, 1.0],                                (
]
```

58a ⟨*function test_treemap_manual* 58a⟩≡ (64)

```
(* test draw_rect_treemap_float_ortho *)
let test_treemap_manual () =
  Graphics.open_graph " 640x640";
  Graphics.set_color (Graphics.rgb 1 1 1);
  let w, h = Graphics.size_x (), Graphics.size_y () in

  treemap_rectangles_ex +> List.iter (fun (upper, lower, (r,g,b)) ->
    match upper, lower with
    | [x1, y1], [x2, y2] ->
        let maxc = float_of_int 256 in
        let (r,g,b) =
          int_of_float (r *. maxc),
          int_of_float (g *. maxc),
          int_of_float (b *. maxc)
        in
        let color = Graphics.rgb (r) (g) (b) in

        draw_rect_treemap_float_ortho ((x1, y1),(x2, y2)) color (w, h)
        +> ignore
    | _ -> failwith "wront format"
  );
  Common.pause();
  ()
```

58b ⟨*function test_treemap* 58b⟩≡ (64)

```
let test_treemap algorithm treemap =
  Graphics.open_graph " 640x640";
  Graphics.set_color (Graphics.rgb 1 1 1);
  let w, h = Graphics.size_x (), Graphics.size_y () in

  Graphics.set_line_width 2;

  display_treemap_algo ~algo:algorithm treemap (w, h) +> ignore;
  while true do
    let status = Graphics.wait_next_event [
        Graphics.Key_pressed;
      ]
    in
    if status.Graphics.keypressed (* Graphics.key_pressed () *)
    then raise (UnixExit 0);
  done;
  (* old: pause (); *)
  ()
```

59a ⟨*function test_treemap_tree* 59a⟩≡ (64)

```
let test_treemap_tree algorithm ex =
  let maxc = 256 in

  let tree =
    match ex with
    | 1 -> tree_ex_shneiderman_1991
    | 2 -> tree_ex_wijk_1999
    | _ -> raise Impossible
  in

  let treemap = treemap_of_tree
    ~size_of_leaf:(fun intleaf -> intleaf)
    ~color_of_leaf:(fun intleaf ->
      Graphics.rgb (Random.int maxc) (Random.int maxc) (Random.int maxc)
    )
    ~label_of_file:(fun intleaf -> i_to_s intleaf)
    tree
  in
  test_treemap algorithm treemap
```

59b ⟨*function test_treemap_dir* 59b⟩≡ (64)

```
let test_treemap_dir dir algo =

  let w_view_hint, h_view_hint = 640, 640 in
  let h_status = 30 in

  Graphics.open_graph (spf " %dx%d" w_view_hint (h_view_hint+ h_status));
  Graphics.set_color (Graphics.rgb 1 1 1);
  let w_view, h_view =
    Graphics.size_x (),
    Graphics.size_y () - h_status
  in
  let w, h = Graphics.size_x (), Graphics.size_y () in

  let maxc = 256 in
  let dim = {
    w = w;
    h = h;
    w_view = w_view;
    h_view = h_view;
    h_status = h_status;
    w_legend = 10;
  }
  in

  (* work ? Graphics.set_line_width 2; *)

  let tree =
    tree_of_dir ~file_hook:(fun file ->
      file, Common.filesize file
    )
      dir
  in

  let treemap = treemap_of_tree
    ~size_of_leaf:(fun (f, intleaf) -> intleaf)
    ~color_of_leaf:(fun (f, intleaf) ->
      Graphics.rgb (Random.int maxc) (Random.int maxc) (Random.int maxc)
    )
    ~label_of_dir:(fun dir -> basename dir)
    tree
  in

  display_treemap_interactive
    ~algo
    treemap
    dim
    ~info_of_file_under_cursor:(fun status (f, size) ->
      let s = f in
      if status.Graphics.button
      then begin
        pr2 (spf "%s" f);
        Sys.command (spf "/home/pad/packages/Linux/bin/emacsclient -n %s" f) +> ignore;
      end;

      if status.Graphics.keypressed (* Graphics.key_pressed () *)
      then raise (UnixExit 0);
      s
    );


  ()
```

60 ⟨*treemap actions* 60⟩≡ (44e)

```
    "-test_squarify", "<>",
    Common.mk_action_0_arg (test_squarify);
    "-test_orderify", "<>",
    Common.mk_action_0_arg (test_orderify);
```

A.3 treemap_json.mli

61a ⟨*treemap_json.mli* 61a⟩≡

```
⟨signature treemap_of_json 37a⟩

⟨signature json_of_treemap 37b⟩

val json_of_treemap_rendering:
  Treemap.treemap_rendering -> Json_type.json_type

val actions : unit -> Common.cmdline_actions
```

A.4 treemap_json.ml

61b ⟨*treemap_json.ml* 61b⟩≡

```
⟨Facebook copyright 4⟩

open Common

module J = Json_type

open Treemap
open Figures

module Color = Simple_color

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Json -> Treemap *)
(*****************************************************************************)

⟨function treemap_of_json 37c⟩

(*****************************************************************************)
(* Treemap -> Json *)
```

```
(*****************************************************************************)

⟨function json_of_color 38a⟩

⟨function json_of_treemap 38b⟩


(*****************************************************************************)
(* Treemap rendering *)
(*****************************************************************************)

let rec vof_rectangle { p = v_p; q = v_q } =
  let bnds = [] in
  let arg = vof_point v_q in
  let bnd = ("q", arg) in
  let bnds = bnd :: bnds in
  let arg = vof_point v_p in
  let bnd = ("p", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds
and vof_point { x = v_x; y = v_y } =
  let bnds = [] in
  let arg = Ocaml.vof_float v_y in
  let bnd = ("y", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_float v_x in
  let bnd = ("x", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds

let rec vof_treemap_rendering v = Ocaml.vof_list vof_treemap_rectangle v
and
  vof_treemap_rectangle {
                          tr_rect = v_tr_rect;
                          tr_color = v_tr_color;
                          tr_label = v_tr_label;
                          tr_depth = v_tr_depth
                        } =
  let bnds = [] in
  let arg = Ocaml.vof_int v_tr_depth in
  let bnd = ("tr_depth", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_string v_tr_label in
  let bnd = ("tr_label", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_int v_tr_color in
  let bnd = ("tr_color", arg) in
  let bnds = bnd :: bnds in
  let arg = vof_rectangle v_tr_rect in
  let bnd = ("tr_rect", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds
let json_of_treemap_rendering rendering =
  let v = vof_treemap_rendering rendering in
  Ocaml.json_of_v v

(*****************************************************************************)
(* Testing *)
(*****************************************************************************)
⟨function test_json_of 39⟩

⟨function test_of_json 40a⟩


(*****************************************************************************)
(* Actions *)
(*****************************************************************************)

let actions () = [
⟨treemap_json actions 40b⟩

]
```

A.5 treemap_graphics.mli

63 ⟨*treemap_graphics.mli* 63⟩≡

```
open Treemap

(* seminal code and algorithm *)
⟨signature display_treemap 6c⟩

⟨signature display_treemap_algo 12b⟩

(* main entry point *)
⟨signature display_treemap_interactive 33b⟩

⟨signature graphic helpers 9b⟩

⟨signature test treemap functions 44d⟩
```

A.6 treemap_graphics.ml

```
64  ⟨treemap_graphics.ml 64⟩≡
      ⟨Facebook copyright 4⟩

      open Common

      open Treemap

      module Color = Simple_color

      module F = Figures

      (*****************************************************************************)
      (* Graphics Helpers *)
      (*****************************************************************************)

      ⟨function current_dim 54b⟩

      ⟨function draw_rect_treemap_float_ortho 11a⟩

      ⟨graphic helpers 55⟩

      (*****************************************************************************)
      (* Treemap Helpers *)
      (*****************************************************************************)

      ⟨function update_mat_with_fileinfo 33c⟩

      (*****************************************************************************)
      (* Main display function  *)
      (*****************************************************************************)

      ⟨function display_treemap 7b⟩


      (*---------------------------------------------------------------------------*)
      (* generic frontend, taking layout-maker function as a parameter  *)
      (*---------------------------------------------------------------------------*)

      ⟨function display_treemap_generic 12d⟩


      ⟨function display_treemap_algo 14⟩
```

```
⟨function display_treemap_interactive 34⟩

⟨function info_of_file_under_cursor_default 35⟩

(*****************************************************************************)
(* Testing *)
(*****************************************************************************)

⟨function test_treemap_manual 58a⟩

⟨function test_treemap 58b⟩

(* test tree_of_dir *)
⟨function test_treemap_dir 59b⟩

(* test treemap_of_tree, and display_treemap *)
⟨function test_treemap_tree 59a⟩

(*****************************************************************************)
(* Actions *)
(*****************************************************************************)

let actions () = [
⟨treemap_graphics actions 65⟩
]
```

65 ⟨*treemap_graphics actions* 65⟩≡ (64)

```
  "-test_treemap_manual", "<>",
  Common.mk_action_0_arg (test_treemap_manual);

  "-test_treemap", "<algorithm>",
  Common.mk_action_1_arg (fun s ->
    let treemap = treemap_ex_ordered_2001 in
    test_treemap (algo_of_s s) treemap

  );

  "-test_treemap_tree", "<algorithm> <ex>",
  Common.mk_action_2_arg (fun s i ->
    test_treemap_tree (algo_of_s s) (s_to_i i)
  );
  "-test_treemap_dir", "<dir> <algorithm>",
  Common.mk_action_2_arg (fun dir str ->
    test_treemap_dir dir (algo_of_s str)
  );
```

A.7 main_treemap.ml

66a ⟨*function main_action* 66a⟩≡ (66d)

```
let main_action jsonfile =
  let json = Json_in.load_json jsonfile in
  let treemap = Treemap_json.treemap_of_json json in

  let rendering = Treemap.render_treemap_algo treemap in
  let json = Treemap_json.json_of_treemap_rendering rendering in
  let s = Json_out.string_of_json json in
  pr2 s;

  let dim = init_graph !big_screen in

  Treemap_graphics.display_treemap_interactive
    ~algo:!algorithm
    ~info_of_file_under_cursor:Treemap_graphics.info_of_file_under_cursor_default
    treemap dim
  ;
  ()
```

66b ⟨*treemap_viewer cmdline options* 66b⟩≡ (66d)

```
    "-algorithm", Arg.String (fun s ->
      algorithm := Treemap.algo_of_s s;
    ),
    (spf " <algo> (choices are: %s, default = %s"
        (Treemap.algos +> List.map Treemap.s_of_algo +> Common.join ", ")
        (Treemap.s_of_algo !algorithm));

    "-big_screen", Arg.Set big_screen,
    " ";
    "-verbose", Arg.Set verbose,
    " ";
```

66c ⟨*treemap_viewer flags* 66c⟩≡ (66d)

```
let algorithm = ref Treemap.Squarified
let big_screen = ref false

let verbose = ref false
```

66d ⟨*main_treemap.ml* 66d⟩≡

```
open Common

(*****************************************************************************)
```

```
(* Purpose *)
(*****************************************************************************)

(*****************************************************************************)
(* Flags *)
(*****************************************************************************)

⟨treemap_viewer flags 66c⟩

(* action mode *)
let action = ref ""

let version = "0.1"

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

let init_graph big_screen =

  let w_view_hint, h_view_hint =
    if big_screen
    then
      2300, 1500
    else
      640, 640
  in
  let h_status = 30 in
  let w_legend = 200 in

  Graphics.open_graph
    (spf " %dx%d" (w_view_hint + w_legend) (h_view_hint+ h_status));
  Graphics.set_color (Graphics.rgb 1 1 1);
  let w_view, h_view =
    Graphics.size_x () - w_legend,
    Graphics.size_y () - h_status
  in
  let w, h = Graphics.size_x (), Graphics.size_y () in

  {
    Treemap.w = w;
    h = h;
    w_view = w_view;
    h_view = h_view;
    h_status = h_status;
    w_legend = w_legend;
  }


(*****************************************************************************)
(* Main action *)
(*****************************************************************************)

⟨function main_action 66a⟩

(*****************************************************************************)
(* The options *)
(*****************************************************************************)

let all_actions () =
 Treemap.actions () ++
 Treemap_json.actions () ++
 []

let options () =
  [
  ⟨treemap_viewer cmdline options 66b⟩
  ] ++
  Common.options_of_actions action (all_actions()) ++
  Common.cmdline_flags_devel () ++
  Common.cmdline_flags_verbose () ++
  Common.cmdline_flags_other () ++
  [
  "-version",   Arg.Unit (fun () ->
    pr2 (spf "ocamltreemap version: %s" version);
    exit 0;
  ),
    "  guess what";

  (* this can not be factorized in Common *)
  "-date",   Arg.Unit (fun () ->
    pr2 "version: $Date: 2008/10/26 00:44:57 $";
    raise (Common.UnixExit 0)
    ),
  "   guess what";
  ] ++
  []

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```

```
let main () =
  let usage_msg =
    "Usage: " ^ Common.basename Sys.argv.(0) ^
      " [options] <json file> " ^ "\n" ^ "Options are:"
  in
  (* does side effect on many global flags *)
  let args = Common.parse_options (options()) usage_msg Sys.argv in

  (* must be done after Arg.parse, because Common.profile is set by it *)
  Common.profile_code "Main total" (fun () ->

    (match args with

    (* --------------------------------------------------------- *)
    (* actions, useful to debug subpart *)
    (* --------------------------------------------------------- *)
    | xs when List.mem !action (Common.action_list (all_actions())) ->
        Common.do_action !action xs (all_actions())

    | _ when not (Common.null_string !action) ->
        failwith ("unrecognized action or wrong params: " ^ !action)

    (* --------------------------------------------------------- *)
    (* main entry *)
    (* --------------------------------------------------------- *)
    | [x] ->
        main_action x

    (* --------------------------------------------------------- *)
    (* empty entry *)
    (* --------------------------------------------------------- *)
    | [] ->
        Common.usage usage_msg (options());
        failwith "too few arguments"

    | x::y::xs ->
        Common.usage usage_msg (options());
        failwith "too many arguments"
    )
  )

(*****************************************************************************)
let _ =
  Common.main_boilerplate (fun () ->
      main ();
  )
```

B Changelog

Indexes

⟨*concrete rectangles example* 57⟩
⟨*ex.json* 36⟩
⟨*Facebook copyright* 4⟩
⟨*function algorithm accessors* 54c⟩
⟨*function balayer_right_wrong* 31a⟩
⟨*function compute_rects_pivotized* 29b⟩
⟨*function current_dim* 54b⟩
⟨*function display_treemap* 7b⟩
⟨*function display_treemap_algo* 14⟩
⟨*function display_treemap_generic* 12d⟩
⟨*function display_treemap_interactive* 34⟩
⟨*function draw_rect_treemap_float_ortho* 11a⟩
⟨*function info_of_file_under_cursor_default* 35⟩
⟨*function json_of_color* 38a⟩
⟨*function json_of_treemap* 38b⟩
⟨*function layout* 19c⟩
⟨*function main_action* 66a⟩
⟨*function orderify_children* 31b⟩
⟨*function ratio_rect_dim* 19a⟩
⟨*function squarify* 23a⟩
⟨*function squarify_orig* 20⟩
⟨*function test_json_of* 39⟩
⟨*function test_of_json* 40a⟩
⟨*function test_orderify* 32⟩
⟨*function test_squarify* 23b⟩
⟨*function test_treemap* 58b⟩
⟨*function test_treemap_dir* 59b⟩
⟨*function test_treemap_manual* 58a⟩
⟨*function test_treemap_tree* 59a⟩
⟨*function tree_of_dir* 40d⟩
⟨*function treemap accessors* 54a⟩
⟨*function treemap_of_json* 37c⟩
⟨*function treemap_of_tree* 56⟩
⟨*function update_mat_with_fileinfo* 33c⟩
⟨*function worst* 19b⟩
⟨*graphic helpers* 55⟩
⟨*layout ordered* 33a⟩
⟨*layout slice and dice* 15a⟩

⟨layout squarify 23c⟩
⟨main_treemap.ml 66d⟩
⟨ordered examples 24⟩
⟨signature algorithm accessors 44c⟩
⟨signature algos 11c⟩
⟨signature display_treemap 6c⟩
⟨signature display_treemap_algo 12b⟩
⟨signature display_treemap_interactive 33b⟩
⟨signature graphic helpers 9b⟩
⟨signature json_of_treemap 37b⟩
⟨signature test treemap functions 44d⟩
⟨signature tree and treemap examples 6a⟩
⟨signature tree_of_dir 40c⟩
⟨signature treemap accessors 44b⟩
⟨signature treemap_of_json 37a⟩
⟨signature treemap_of_tree 43⟩
⟨squarified examples 16⟩
⟨treemap actions 60⟩
⟨treemap.ml 44e⟩
⟨treemap.mli 42⟩
⟨treemap_graphics actions 65⟩
⟨treemap_graphics.ml 64⟩
⟨treemap_graphics.mli 63⟩
⟨treemap_json actions 40b⟩
⟨treemap_json.ml 61b⟩
⟨treemap_json.mli 61a⟩
⟨treemap_viewer cmdline options 66b⟩
⟨treemap_viewer flags 66c⟩
⟨type algorithm 11b⟩
⟨type layout_func 12c⟩
⟨type pivotized 29a⟩
⟨type rectangle1 7a⟩
⟨type screen_dim 9a⟩
⟨type split 17⟩
⟨type treemap 5⟩
⟨variable algos 12a⟩
⟨variable tree_ex_shneiderman_1991 6b⟩
⟨variable tree_ex_wijk_1999 15b⟩
⟨variable treemap_ex_ordered_2001 23d⟩

References

[1] Donald Knuth,, *Literate Programming*, http://en.wikipedia.org/wiki/Literate_Program cited page(s) 4

[2] Norman Ramsey, *Noweb*, http://www.cs.tufts.edu/~nr/noweb/ cited page(s) 4

[3] Yoann Padioleau, *Syncweb, literate programming meets unison*, http://padator.org/software/project-syncweb/readme.txt cited page(s) 4

[4] Yoann Padioleau, *Commons Pad OCaml Library*, http://padator.org/docs/Commons.pdf cited page(s)

[5] Wikipedia, *Treemapping*, http://en.wikipedia.org/wiki/Treemapping cited page(s) 1

[6] Ben Shneiderman, *History of Treemap Research*, http://www.cs.umd.edu/hcil/treemap-history/index.shtml cited page(s) 2

[7] Martin Wattenberg, *Map of the Market*, 1998, http://www.smartmoney.com/map-of-the-market/ cited page(s) 1

[8] Ben Shneiderman, *Tree visualization with Tree-maps: A 2-d space-filling approach History of Treemap Research*, 1991, http://hcil.cs.umd.edu/trs/91-03/91-03.html cited page(s) 6

[9] Jarke van Wijk, Huub van de Wetering, *Cushion Treemaps: Visualization of Hierarchical Information*, 1999, TODO cited page(s) 33

[10] Mark Bruls, Kees Huizing, and Jarke van Wijk, *Squarified Treemaps*, 2000, www.win.tue.nl/~vanwijk/stm.pdf cited page(s) 15

[11] Ben Shneiderman and Martin Wattenberg, *Ordered Treemap Layouts*, 2001, ftp://ftp.cs.umd.edu/pub/hcil/Reports-Abstracts-Bibliography/2001-06html/2001-06.htm cited page(s) 23

[12] Wikipedia *List of treemapping software*, http://en.wikipedia.org/wiki/List_of_treemapping_software cited page(s) 42