SpaceTree Representation

Bit representation of the underlying SpaceTree (1=node, 0=leaf);
the refinement bits are given in the order defined by a depth-first traversal or the resp. SpaceTree
nodes:

> SPACETREE := [1,0,0,0,0];

\( SPACETREE := [1, 0, 0, 0, 0] \) \hspace{1cm} (1.1)

Pointer to current element in SPACETREE:

> STPTR := 0;

\( STPTR := 0 \) \hspace{1cm} (1.2)

Function to construct a uniformly refined tree of given depth:

> fullTree := proc(depth::integer)
    # parameter depth: depth of the generated spacetree
    if depth = 0
        then return [0]
    else
        # combine four subtrees of depth (depth-1)
        return [ 1, seq( op( fullTree(depth-1) ), k=1..4) ];
    end if;
end proc:

A "Fibonacci-SpaceTree" (the depth decreases for the two last subtrees):

> fibTree := proc(depth::integer)
    # parameter depth: depth of the generated spacetree
    if depth = 0
        then return [0]
    elif depth = 1
        then return [ 1, seq( op( fibTree(depth-1) ), k=1..4) ];
    elif depth = 2
        then return [ 1, seq( op( fibTree(depth-1) ), k=1..2),
            seq( op( fibTree(depth-2) ), k=3..4)];
    else return [ 1, op( fibTree(depth-1) ), seq( op( fibTree(depth-2) ), k=2..3), op( fibTree(depth-3) ) ];
    end if;
end proc:

Examples:

> SPACETREE := fullTree(2);

\( SPACETREE := [1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0] \) \hspace{1cm} (1.3)

> fibTree(3);

\[ [1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0] \] \hspace{1cm} (1.4)

Function to "expand" a specified leaf of a spacetree into a refined node:

> adaptTree := proc(st::list, position::integer)
    # param. list: spacetree in bitstream notation
    # param. position: index of the node to expand
    # if node at given position is already an interior node,

then do nothing:
    if (st[position] = 1) then return st; end if;
    # replace a 0 bit at given position by a uniform tree of
depth 1:
    return [ seq(st[i], i=1..position-1), 1, 0,0,0,0, seq(st
    [i], i=position+1..nops(st) ) ];
end proc:
> SPACETREE := adaptTree(fullTree(2), 15);
  SPACETREE := [1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

\[ \text{Helper Functions for Plotter Graphics} \]

We will plot traversals of bitstream-encoded spacetrees by connecting the centre points of the
respective tree elements.
The resp. plots will by polygonal lines drawn using a "plotter graphics" concept:
our plotter can draw lines of a given stepsize in the four (horizontal and vertical) directions.
The Maple implementation is based on building a list of coordinates to be connected.

attach an element to the end of a list (of points):
> attach := proc(li, elem)
    # eval(li) as li might be call-by-reference
    return [ op(eval(li)),elem];
end proc;

\[ \text{attach} := \text{proc}\left(\text{li, elem}\right) \text{ \text{ return } [ \text{op}(\text{eval(li))},\text{elem}] \text{ \text{ end proc}} \right) \text{ (2.1)} \]

Initialise plotter (start in the center of the root element, i.e., the unit square):
> init_turtle := proc()
    global points, stepsize, TURTLE_x, TURTLE_y;
    points := [];
    stepsize := 1;
    TURTLE_x := 1/2;
    TURTLE_y := 1/2;
end proc:

Mark the current position of the plotter -> will draw a line from the previous marked position to
current position:
> mark := proc()
    #option trace;
    global points, TURTLE_x, TURTLE_y;
    points := attach(points, [ TURTLE_x, TURTLE_y ] );
end proc:

4 functions to move plotter in the four horizontal/vertical directions (up, down, left, right):
> up := proc()
    #option trace;
    global stepsize, TURTLE_x, TURTLE_y;
    TURTLE_y := TURTLE_y + stepsize;
end proc:
> down := proc()
    #option trace;
    global stepsize, TURTLE_x, TURTLE_y;
TURTLE_y := TURTLE_y - stepsize;
end proc:

> left := proc()
    #option trace;
    global stepsize, TURTLE_x, TURTLE_y;
    TURTLE_x := TURTLE_x - stepsize;
end proc:

> right := proc()
    #option trace;
    global stepsize, TURTLE_x, TURTLE_y;
    TURTLE_x := TURTLE_x + stepsize;
end proc:

The 2 functions fine and coarse will implement a step down or up in the spacetree hierarchy, i.e. step from the centre of a child cell in the centre of the parent, or vice versa:

> fine := proc(xshift::integer, yshift::integer)
    # option trace;
    global stepsize, TURTLE_x, TURTLE_y;
    stepsize := stepsize / 2;
    TURTLE_x := TURTLE_x + xshift*stepsize/2;
    TURTLE_y := TURTLE_y + yshift*stepsize/2;
end proc:

> coarse := proc(xshift::integer, yshift::integer)
    # option trace;
    global stepsize, TURTLE_x, TURTLE_y;
    TURTLE_x := TURTLE_x + xshift*stepsize/2;
    TURTLE_y := TURTLE_y + yshift*stepsize/2;
    stepsize := stepsize * 2;
end proc:

Algorithm for the adaptive Hilbert Curve

Corresponding to the respective (context-free) grammer of the Hilbert curve

> H := proc()
    # option trace;
    global SPACETREE, STPTR;

    STPTR := STPTR + 1;
    if SPACETREE[STPTR] = 0
    then
        mark();
    else
        # recursive calls to children
        fine(-1,-1);
        A(); up();
        H(); right();
        H(); down();
B();
    coarse(-1,1);
end if;
end proc:
> A := proc()
    # option trace;
    global SPACETREE, STPTR;

    STPTR := STPTR + 1;
    if SPACETREE[STPTR] = 0
    then
        mark();
    else
        fine(-1, -1);
        H(); right();
        A(); up();
        A(); left();
        C();
        coarse(1,-1);
    end if;
end proc:
> B := proc()
    # option trace;
    global SPACETREE, STPTR;

    STPTR := STPTR + 1;
    if SPACETREE[STPTR] = 0
    then
        mark();
    else
        fine(1, 1);
        C(); left();
        B(); down();
        B(); right();
        H();
        coarse(-1, 1);
    end if;
end proc:
> C := proc()
    # option trace;
    global SPACETREE, STPTR;

    STPTR := STPTR + 1;
    if SPACETREE[STPTR] = 0
    then
mark();
else
    fine(1, 1);
    B(); down();
    C(); left();
    C(); up();
    A();
    coarse(1, -1);
end if;
end proc:

> Hilbert := proc(st::list)
    global points, SPACETREE, STPTR;
    SPACETREE := st;
    STPTR := 0;
    init_turtle();
    H();
    return eval(points);
end proc;

Hilbert := proc(st::list)
    global points, SPACETREE, STPTR;
    SPACETREE := st;
    STPTR := 0;
    init_turtle();
    H();
    return eval(points);
end proc;

Hilbert( [1,0,0,0,0] );

> Hilbert( [1,0,0,0,0] );

\[
\begin{bmatrix}
\frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{3}{4}
\end{bmatrix}
\begin{bmatrix}
\frac{1}{4} & \frac{3}{4} \\ \frac{3}{4} & \frac{1}{4}
\end{bmatrix}
\]

> fibTree(2);

[1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

> plot( Hilbert( [1,0,0,0,1,0,1,0,0,0,0,0,0,0,0] ),
    scaling=CONSTRAINED, thickness=3, view=[0..1, 0..1]);
> plot( Hilbert( fibTree(9) ),
    scaling=CONSTRAINED, thickness=3, view=[0..1, 0..1]);
Partitioning of the Hilbert Curve

> colours := [black, red, green, yellow, brown, magenta, cyan, navy, pink, grey, blue, khaki, coral];
> colours := [black, red, green, yellow, brown, magenta, cyan, navy, pink, grey, blue, khaki, coral]  

(4.1)

partition splits the given list pts into number partitions (each partition is again a list)
> partition := proc(pts::list, number::posint)
local parts, i;
parts := [ pts[ (number-1)*floor(nops(pts)/number)..-1 ] ];
for i from number-1 by -1 to 2 do
    parts := [ pts[ (i-1)*floor(nops(pts)/number)..i*floor(nops(pts)/number) ],
               op(parts) ];
end do;
parts := [ pts[ 1..floor(nops(pts)/number) ], op(parts) ];
return parts;
end proc:

pts := Hilbert(fibTree(9));parts := partition(pts,11):
plot(parts, axes=BOXED, scaling=CONSTRAINED, thickness=3, color=colours);