Package: mazealls (via r-universe)

October 19, 2026

Maintainer Steven E. Pav <shabbychef@gmail.com>

Version 0.2.1

Date 2025-11-18

License LGPL-3

Title Generate Recursive Mazes

BugReports https://github.com/shabbychef/mazealls/issues

Description Supports the generation of parallelogram, equilateral triangle, regular hexagon, isosceles trapezoid, Koch snowflake, 'hexaflake', Sierpinski triangle, Sierpinski carpet and Sierpinski trapezoid mazes via 'TurtleGraphics'. Mazes are generated by the recursive method: the domain is divided into sub-domains in which mazes are generated, then dividing lines with holes are drawn between them, see J. Buck, Recursive Division, <http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm>.

Depends R (>= 3.0.2)

Imports TurtleGraphics

Suggests testthat, knitr

URL https://github.com/shabbychef/mazealls

Collate 'decagon_maze.r' 'dodecagon_maze.r' 'eq_triangle_maze.r' 'hexaflake_maze.r' 'hexagon_maze.r' 'holey_line.r' 'holey_path.r' 'iso_trapezoid_maze.r' 'koch_maze.r' 'mazealls.r' 'octagon_maze.r' 'parallelogram_maze.r' 'sierpinski_carpet_maze.r' 'sierpinski_hexagon_maze.r' 'sierpinski_maze.r' 'sierpinski_trapezoid_maze.r' 'utils.r'

RoxygenNote 7.3.2

Repository https://shabbychef.r-universe.dev

Date/Publication 2025-11-26 23:48:08 UTC

RemoteUrl https://github.com/shabbychef/mazealls

RemoteRef HEAD

RemoteSha 1025a8248cf2dce78b5ae756754ad443fedcefb9

Contents

decagon_maze 2
dodecagon_maze 4
eq_triangle_maze 7
hexaflake_maze 13
hexagon_maze 16
holey_line 21
holey_path 22
iso_trapezoid_maze 24
koch_maze 27
mazealls 29
mazealls-NEWS 32
octagon_maze 32
parallelogram_maze 34
sierpinski_carpet_maze 39
sierpinski_hexagon_maze 42
sierpinski_maze 45
sierpinski_trapezoid_maze 48

Index **52**

`decagon_maze` *decagon_maze .*

Description

Draw a regular decagon maze, with each side consisting of of 2^{depth} pieces of length `unit_len`.

Usage

```
decagon_maze(
  depth,
  unit_len = 4L,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  method = c("five_flower"),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`method` there are a few ways to recursively draw an decagon. The following values are acceptable:

five_flower Dissects the decagon as 'flower' of five rhombuses in the center, and another five surrounding them.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

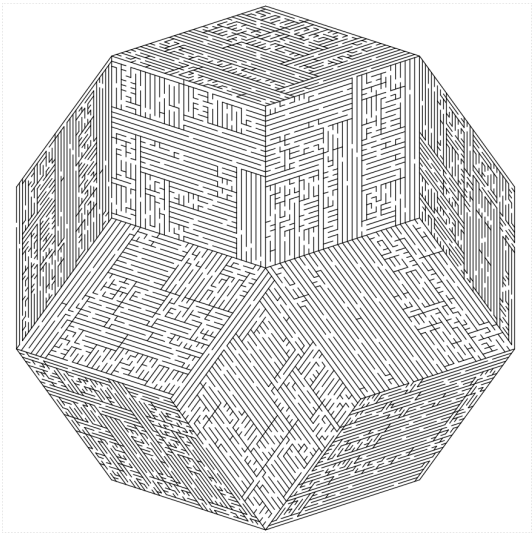
`boundary_hole_locations` the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows` a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in a regular decagon. Dissects the decagon into rhombuses.



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
## Not run:
turtle_init(2200,2200,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
  turtle_setpos(25,1100)
  turtle_setangle(0)
 decagon_maze(5,21,draw_boundary=TRUE,boundary_holes=c(1,6))
})

## End(Not run)
```

---

dodecagon_maze *dodecagon_maze* .

Description

Draw a regular dodecagon maze, with each side consisting of of 2^{depth} pieces of length unit_len.

Usage

```
dodecagon_maze(
  depth,
  unit_len = 4L,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  method = c("hex_ring"),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`method` there are a few ways to recursively draw an decagon. The following values are acceptable:

hex_ring A regular hexagon maze in the center is drawn, with a ring of alternating squares and equilateral triangle mazes around it.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

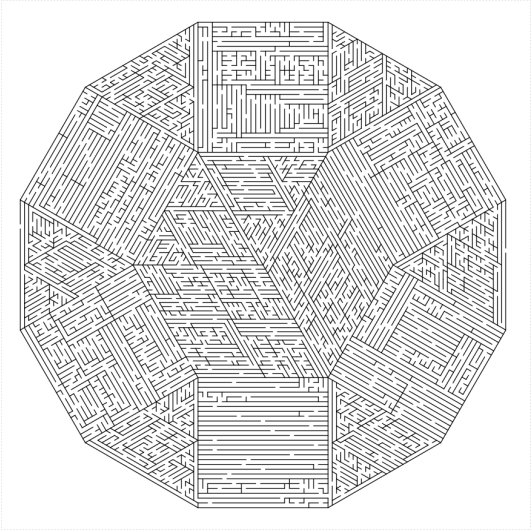
`boundary_hole_locations`
: the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows`
: a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side`
: the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in a regular dodecagon. Currently dissects the maze into a hexagon and a ring of squares and equilateral triangles.



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
## Not run:
```

```
turtle_init(2200,2200,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
  turtle_setpos(25,1100)
  turtle_setangle(0)
  dodecagon_maze(5,21,draw_boundary=TRUE,boundary_holes=c(1,6))
})

## End(Not run)
```

---

eq_triangle_maze *eq_triangle_maze* .

Description

Recursively draw an equilateral triangle maze, with sides consisting of 2^{depth} pieces of length `unit_len`.

Usage

```
eq_triangle_maze(
  depth,
  unit_len,
  clockwise = TRUE,
  method = c("stack_trapezoids", "triangles", "uniform", "two_ears", "random",
    "hex_and_three", "shave_all", "shave"),
  start_from = c("midpoint", "corner"),
  boustro = c(1, 1),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth`	the depth of recursion. This controls the side length.
`unit_len`	the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
`clockwise`	whether to draw clockwise.
`method`	there are many ways to recursive draw a triangle. The following values are acceptable:

stack_trapezoids Isosceles trapezoids are stacked on top of each other, with the long sides aligned to the first side.

triangles The triangle maze is recursively drawn as four equilateral triangle mazes of half size, each connected to their neighbors.

uniform The triangle maze is recursively drawn as four equilateral triangle uniform mazes of half size, each connected to their neighbors.

two_ears The triangle maze is recursively drawn as a large parallelogram maze connected to two two half size equilateral triangle mazes, which are 'ears'.

random A method is randomly selected from the available methods.

hex_and_three When 2^{depth} is a power of three, the triangle is drawn as a hexagonal maze of one third size connected to three equilateral triangular mazes, each one third size, at the corners.

shave Here 2^{depth} can be arbitrary. A single line is 'shaved' off the triangle, connected to another equilateral triangle of length one less is drawn next to it. This sub triangle will either be drawn using a 'hex_and_three', 'random', or 'shave' methods, in decreasing order of preference, depending on the side length.

shave_all Here 2^{depth} can be arbitrary. A single line is 'shaved' off the triangle, connected to another equilateral triangle of length one less is drawn next to it. This sub triangle will also be drawn using the 'shave_all' method. These mazes tend to look boring, and are not recommended.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`boustro` an array of two values, which help determine the location of holes in internal lines of length `height`. The default value, `c(1,1)` results in uniform selection. Otherwise the location of holes are chosen with probability proportional to a beta density with the ordered elements of `boustro` set as `shape1` and `shape2`. In sub mazes, this parameter is reversed, which can lead to 'boustrophedonic' mazes. It is suggested that the sum of values not exceed 40, as otherwise the location of internal holes may be not widely dispersed from the mean value.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

`boundary_hole_locations`
: the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

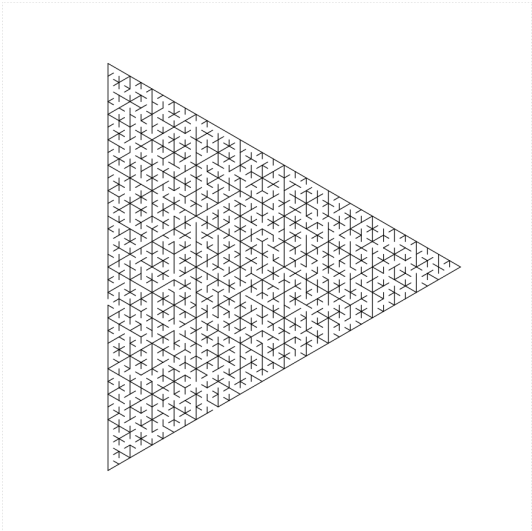
`boundary_hole_arrows`
: a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side`
: the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

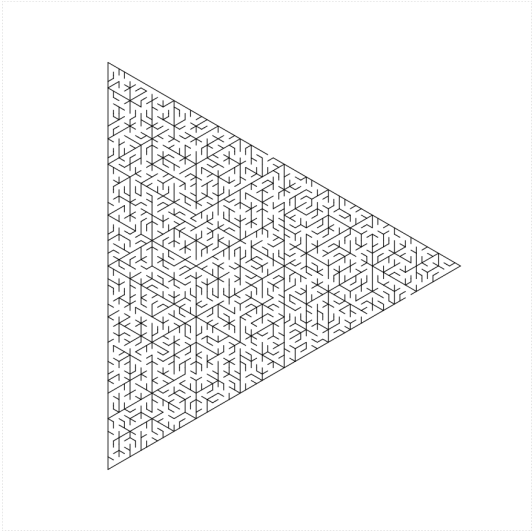
Details

Draws a maze in an equilateral triangle, starting from the midpoint of the first side (or the corner before the first side via the `start_from` option). A number of different recursive methods are supported, dividing the triangle into sub-triangles, or hexagons, parallelogram and triangles, and so on. Optionally draws boundaries around the triangle, with control over which sides have lines and holes. Side length of triangles consists of 2^{depth} segments of length `unit_len`, though `depth` may be non-integral. A number of different methods are supported.

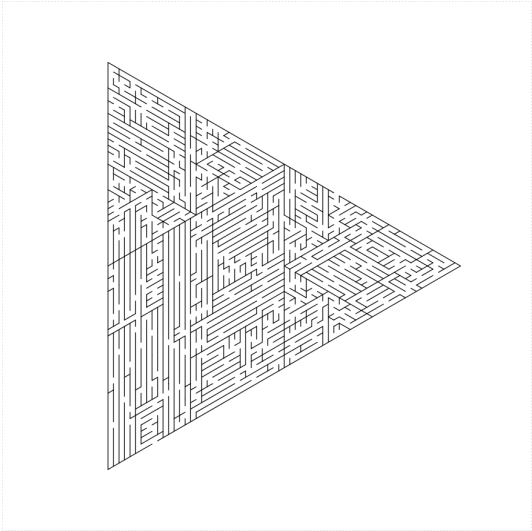
For `method='uniform'`:



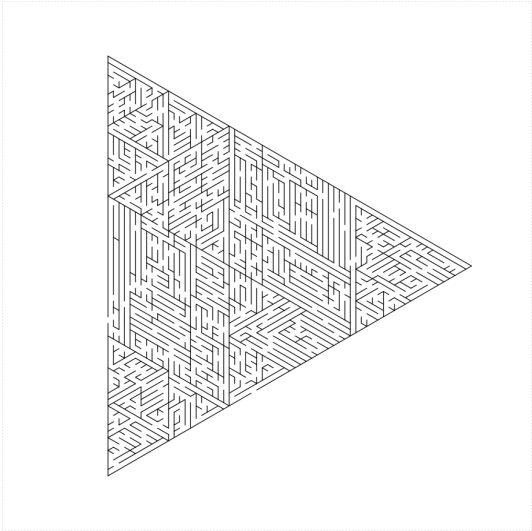
For `method='triangles'`:



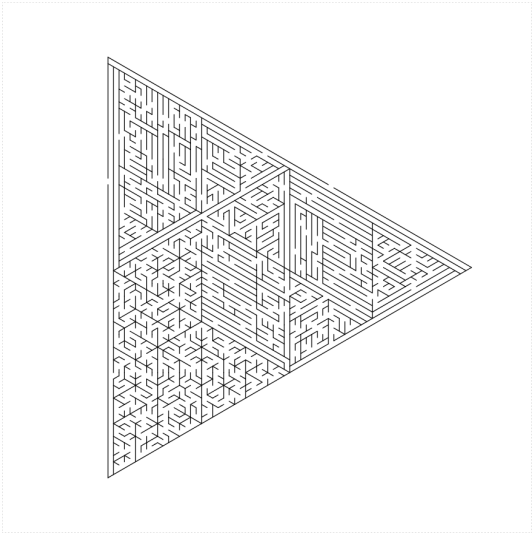
For `method='two_ears'`:



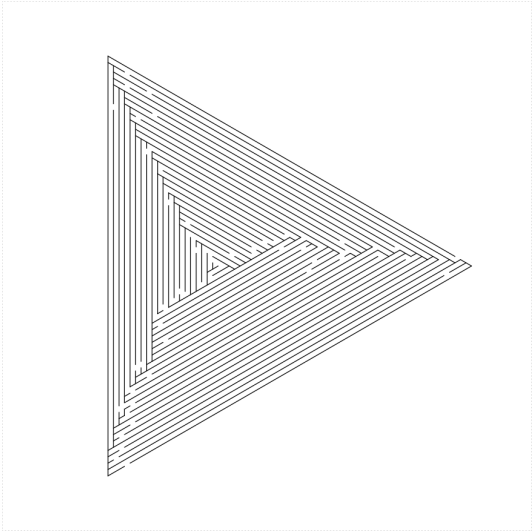
For `method='hex_and_three'`:



For `method='shave'`:



For `method='shave_all'`:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
library(TurtleGraphics)
turtle_init(2500,2500)
turtle_hide()
turtle_up()
turtle_do({
  turtle_left(90)
  turtle_forward(40)
  turtle_right(90)
  eq_triangle_maze(depth=3,12,clockwise=FALSE,method='two_ears',draw_boundary=TRUE)
})

turtle_init(2500,2500)
turtle_hide()
turtle_up()
turtle_do({
  turtle_left(90)
  turtle_forward(40)
  turtle_right(90)
  eq_triangle_maze(depth=3,12,clockwise=FALSE,method='random',draw_boundary=TRUE)
})

# join two together, with green holes on opposite sides
```

```
turtle_init(2500,2500)
turtle_hide()
turtle_up()
turtle_do({
  turtle_left(90)
  turtle_forward(40)
  turtle_right(90)
  eq_triangle_maze(depth=3,12,clockwise=TRUE,method='two_ears',draw_boundary=TRUE,
    boundary_holes=c(1,3),boundary_hole_color=c('clear','clear','green'))
  eq_triangle_maze(depth=3,12,clockwise=FALSE,method='uniform',draw_boundary=TRUE,
    boundary_lines=c(2,3),boundary_holes=c(2),boundary_hole_color='green')
})

# non integral depths also possible:
turtle_init(2500,2500)
turtle_hide()
turtle_up()
turtle_do({
  turtle_left(90)
  turtle_forward(40)
  turtle_right(90)
 eq_triangle_maze(depth=log2(27),12,clockwise=TRUE,method='hex_and_three',draw_boundary=TRUE,
    boundary_holes=c(1,3),boundary_hole_color=c('clear','clear','green'))
  eq_triangle_maze(depth=log2(27),12,clockwise=FALSE,method='shave',draw_boundary=TRUE,
    boundary_lines=c(2,3),boundary_holes=c(2),boundary_hole_color='green')
})
```

`hexaflake_maze` *hexaflake_maze* .

Description

Recursively draw a hexaflake maze, a cross between a Koch snowflake and a Sierpinski triangle. The outer part of the flake consists of a hexagon of side length 3^{depth} pieces of length `unit_len`. The 'inner' and 'outer' pieces of the flake are mazes drawn in different colors.

Usage

```
hexaflake_maze(
  depth,
  unit_len,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  color1 = "black",
  color2 = "gray40",
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
```

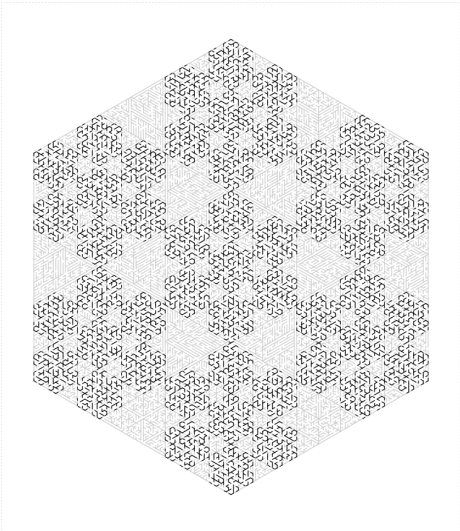
```
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

- `depth` — the depth of recursion. This controls the side length. Should be an integer.
- `unit_len` — the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
- `clockwise` — whether to draw clockwise.
- `start_from` — whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.
- `color1` — The dominant color of the maze.
- `color2` — The negative color of the maze.
- `draw_boundary` — a boolean indicating whether a final boundary shall be drawn around the maze.
- `num_boundary_holes` — the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.
- `boundary_lines` — indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.
- `boundary_holes` — an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.
- `boundary_hole_color` — the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.
- `boundary_hole_locations` — the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.
- `boundary_hole_arrows` — a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.
- `end_side` — the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in an Hexflake. Relies on generation of hexagonal and triangular mazes for the internals. An internal hexagon and six surrounding hexagons are recursively drawn as hexaflakes, connected by 12 equilateral triangles, drawn in the secondary color:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

See Also

sierpinski_trapezoid_maze.

Examples

```
library(TurtleGraphics)
turtle_init(1000,1000,mode='clip')
turtle_hide()
turtle_do({
  turtle_setpos(50,500)
  turtle_setangle(0)
  hexaflake_maze(depth=3,unit_len=10,draw_boundary=TRUE,color2='green')
})
```

hexagon_maze *hexagon_maze .*

Description

Recursively draw a regular hexagon, with sides consisting of 2^{depth} pieces of length `unit_len`.

Usage

```
hexagon_maze(
  depth,
  unit_len,
  clockwise = TRUE,
 method = c("two_trapezoids", "six_triangles", "three_parallelograms", "random"),
  start_from = c("midpoint", "corner"),
  boustro = c(1, 1),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length. If an integer then nice recursive mazes are possible, but non-integral values corresponding to log base 2 of integers are also acceptable.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`method` there are many ways to recursive draw an isosceles trapezoid. The following values are acceptable:

- **two_trapezoids** Two isosceles trapezoids are placed next to each other, with a holey line between them.
- **size_triangles** Six equilateral triangles are packed together, with five holey lines and one solid line.
- **three_parallelograms** Three parallelograms are packed together, with two holey lines and one solid line between them.
- **random** A method is chosen uniformly at random.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`boustro` an array of two values, which help determine the location of holes in internal lines of length `height`. The default value, `c(1,1)` results in uniform selection. Otherwise the location of holes are chosen with probability proportional to a beta density with the ordered elements of `boustro` set as `shape1` and `shape2`. In sub mazes, this parameter is reversed, which can lead to 'boustrophedonic' mazes. It is suggested that the sum of values not exceed 40, as otherwise the location of internal holes may be not widely dispersed from the mean value.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

`boundary_hole_locations` the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

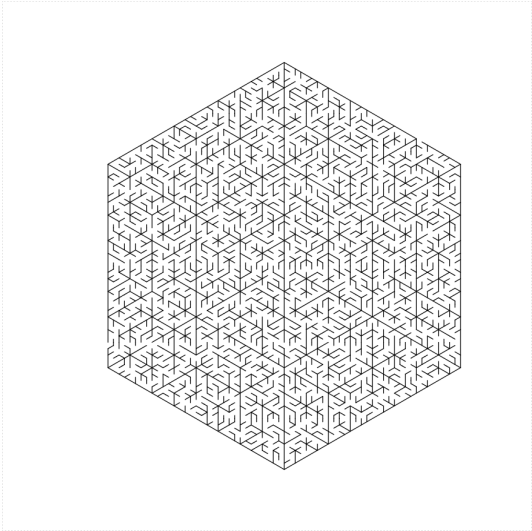
`boundary_hole_arrows` a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

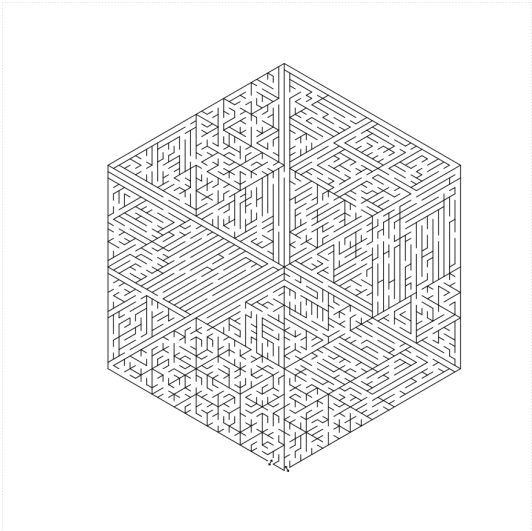
Details

Draws a maze in a regular hexagon, starting from the midpoint of the first side (or the corner before the first side via the `start_from` option). A number of different recursive methods are supported, dividing the triangle into trapezoids, triangles or parallelograms. Optionally draws boundaries around the hexagon, with control over which sides have lines and holes. Sides of the hexagon consist of 2^{depth} segments of length `unit_len`, though `depth` may be non-integral. A number of different methods are supported.

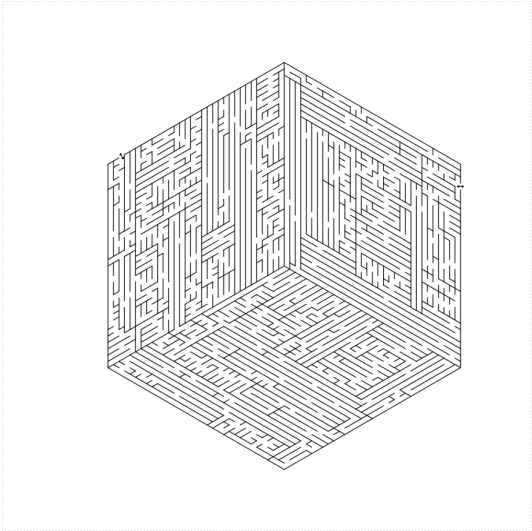
For `method='two_trapezoids'`:



For `method='six_trapezoids'`:



For `method='three_trapezoids'`:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
library(TurtleGraphics)
turtle_init(2000,2000)
turtle_hide()
turtle_do({
 turtle_up()
 turtle_backward(250)
 turtle_right(90)
 turtle_forward(150)
 turtle_left(90)

 turtle_right(60)
 hexagon_maze(depth=3,12,clockwise=FALSE,method='six_triangles',
   draw_boundary=TRUE,boundary_holes=c(1,4),boundary_hole_color='green')
})


turtle_init(2000,2000)
turtle_hide()
turtle_do({
 turtle_up()
 turtle_backward(250)
 turtle_right(90)
```

```
 turtle_forward(150)
 turtle_left(90)

 turtle_right(60)
 hexagon_maze(depth=log2(20),12,clockwise=FALSE,method='six_triangles',
   draw_boundary=TRUE,boundary_holes=c(1,4),boundary_hole_color='green')
})


turtle_init(1000,1000)
turtle_hide()
turtle_do({
 turtle_up()
 turtle_backward(250)
 turtle_right(90)
 turtle_forward(150)
 turtle_left(90)

 turtle_right(60)
 hexagon_maze(depth=3,12,clockwise=FALSE,method='three_parallelograms',
   draw_boundary=TRUE,boundary_holes=c(1,4),boundary_hole_color='green')
})

turtle_init(1000,1000)
turtle_hide()
turtle_do({
 hexagon_maze(depth=3,15,clockwise=TRUE,method='two_trapezoids',
   draw_boundary=TRUE,boundary_holes=c(1,4))
 hexagon_maze(depth=3,15,clockwise=FALSE,method='two_trapezoids',
   draw_boundary=TRUE,boundary_lines=c(2,3,4,5,6),boundary_holes=c(1,4))
})

turtle_init(1000,1000)
turtle_hide()
turtle_do({
 depth <- 3
 num_segs <- 2^depth
 unit_len <- 8
 multiplier <- -1
 hexagon_maze(depth=depth,unit_len,clockwise=FALSE,method='two_trapezoids',
   draw_boundary=FALSE)
 for (iii in c(1:6)) {
  if (iii %in% c(1,4)) {
   holes <- c(1,4)
  } else {
   holes <- c(1)
  }
  hexagon_maze(depth=depth,unit_len,clockwise=TRUE,method='two_trapezoids',
    draw_boundary=TRUE,boundary_holes=holes)
  turtle_forward(distance=unit_len * num_segs/2)
  turtle_right((multiplier * 60) %% 360)
  turtle_forward(distance=unit_len * num_segs/2)
 }
```

```
})
```

---

holey_line *holey_line .*

Description

Draws a line with a randomly selected 'hole' in it.

Usage

```
holey_line(
  unit_len,
  num_segs,
  which_seg = NULL,
  go_back = FALSE,
  hole_color = NULL,
  hole_arrow = FALSE
)
```

Arguments

unit_len	the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
num_segs	the total number of segments. All but one of these, of length unit_len will be drawn. The other, randomly chosen, will be a hole. If num_segs is one, only a hole is made, and no line drawn. If zero or less, no action taken.
which_seg	optional numeric indicating which segment should have the hole. If NULL, the hole segment is chosen uniformly at random.
go_back	whether to return the turtle to starting position when the line has been drawn.
hole_color	the color to plot the 'hole'. A NULL value corresponds to no drawn hole. See the colors function for acceptable values.
hole_arrow	a boolean or indicating whether to draw a perpendicular arrow at a hole.

Details

This function is the workhorse of drawing mazes, as it creates a maze wall with a single hole in it.

Value

Returns the which_seg variable, the location of the hole, though typically the function is called for side effects only.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
library(TurtleGraphics)
turtle_init(1000,1000,mode='clip')
turtle_hide()
y <- holey_line(unit_len=20, num_segs=15)

turtle_right(90)
y <- holey_line(unit_len=20, num_segs=10,hole_arrow=TRUE)
```

`holey_path` *holey_path .*

Description

Make the turtle move multiple units, making turns, and possibly drawing line segments possibly with holes in them.

Usage

```
holey_path(
  unit_len,
  lengths,
  angles,
  draw_line = TRUE,
  has_hole = FALSE,
  hole_color = NULL,
  hole_locations = NULL,
  hole_arrows = FALSE
)
```

Arguments

`unit_len`	the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
`lengths`	an array of the number of units each part of the path. An array of length `n`.
`angles`	after each part of the path is drawn, the turtle turns right by the given angle.
`draw_line`	a boolean array telling whether each part of the path is drawn at all, or whether the turtle simply moves through that path.
`has_hole`	a boolean array telling whether, conditional on the path being drawn, it has a one unit hole.
`hole_color`	the color to plot the 'hole'. A value `NULL` or `'clear'` corresponds to no drawn hole, the latter being useful for mixing drawn colored holes with no hole drawn at all (for which `'white'` would be an acceptable choice if the background were white). Filled holes are often useful for indicating the entry and exit points of a maze. See the `colors` function for acceptable values.

`hole_locations` an optional array of 'locations' of the holes. These affect the `which_seg` of any holey lines which are drawn. If an array of numeric values, a value of zero corresponds to allowing the code to randomly choose the location of a hole; negative values are 'inverted' by adding `length + 1`, so that if the same segment is drawn twice, in different directions, only the sign of the hole location needs to be flipped to have aligned holes. `NA` values will throw an error for now, though this may change in the future.

`hole_arrows` a boolean or boolean array telling whether to draw a perpendicular arrow at a hole.

Details

Causes the turtle to move through a path of connected line segments, possibly drawing lines, possibly drawing holes in those lines. All arguments are recycled to the length of the longest argument via `mapply`, which simplifies the path description.

Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

See Also

`holey_line`.

Examples

```
library(TurtleGraphics)
# draw a triangle with holes on the boundaries
turtle_init(1000,1000)
holey_path(unit_len=20, lengths=rep(10,3), angles=c(120), draw_line=TRUE, has_hole=TRUE)

# draw a square with holes on the boundaries
turtle_init(1000,1000)
turtle_hide()
holey_path(unit_len=20, lengths=rep(10,4), angles=c(90), draw_line=TRUE, has_hole=TRUE,
  hole_color=c('red','green'))

# draw a square spiral
turtle_init(1000,1000)
turtle_hide()
holey_path(unit_len=20, lengths=sort(rep(1:10,2),decreasing=TRUE), angles=c(90),
  draw_line=TRUE, has_hole=FALSE)
```

`iso_trapezoid_maze` *iso_trapezoid_maze .*

Description

Recursively draw a isosceles trapezoid maze, with three sides consisting of 2^{depth} pieces of length `unit_len`, and one long side of length $2^{depth+1}$ pieces, starting from the long side.

Usage

```
iso_trapezoid_maze(
  depth,
  unit_len = 4L,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  method = c("four_trapezoids", "one_ear", "random"),
  boustro = c(1, 1),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length: three sides have `round(2^depth)` segments of length `unit_len`, while the long side is twice as long. `depth` need not be integral.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`method` there are many ways to recursive draw an isosceles trapezoid. The following values are acceptable:

four_trapezoids Four isosceles trapezoids are packed around each other with a 'bone' between them.

one_ear A parallelogram is placed next to an equilateral triangle (an 'ear'). Note this method is acceptable when depth is not an integer.

random A method is chosen uniformly at random.

`boustro` an array of two values, which help determine the location of holes in internal lines of length `height`. The default value, `c(1,1)` results in uniform selection. Otherwise the location of holes are chosen with probability proportional to a beta density with the ordered elements of `boustro` set as `shape1` and `shape2`. In sub mazes, this parameter is reversed, which can lead to 'boustrophedonic' mazes. It is suggested that the sum of values not exceed 40, as otherwise the location of internal holes may be not widely dispersed from the mean value.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

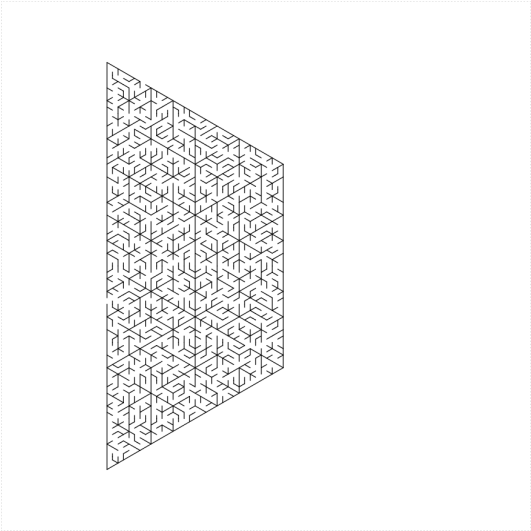
`boundary_hole_locations` the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows` a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

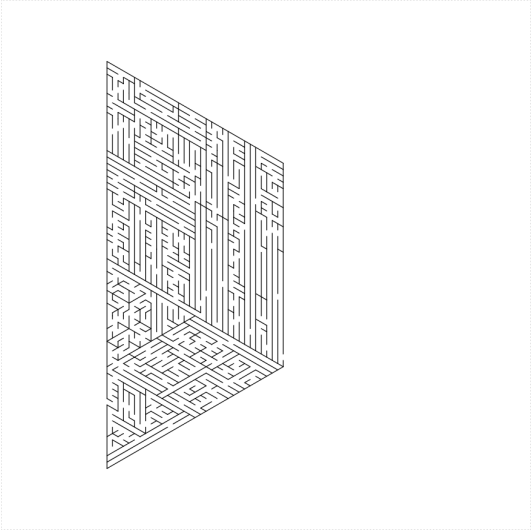
`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in an isoscelese trapezoid with three sides of equal length and one long side of twice that length, starting from the midpoint of the long side (or the corner before the first side via the `start_from` option). A number of different recursive methods are supported. Optionally draws boundaries around the trapezoid, with control over which sides have lines and holes. Three sides of the trapezoid consist of 2^{depth} segments of length `unit_len`, while the longer has 2^{depth}. A number of different methods are supported. For `method='four_trapezoids'`:



For `method='one_ear'`:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
library(TurtleGraphics)
turtle_init(1000,1000)
```

```
turtle_hide()
iso_trapezoid_maze(depth=4,20,clockwise=FALSE,draw_boundary=TRUE)

turtle_init(1000,1000)
turtle_hide()
turtle_do({
iso_trapezoid_maze(depth=3,20,clockwise=TRUE,draw_boundary=TRUE,boundary_holes=3)
})

turtle_init(2000,2000)
turtle_hide()
turtle_up()
turtle_do({
 len <- 22
 iso_trapezoid_maze(depth=log2(len),15,clockwise=TRUE,draw_boundary=TRUE,
   boundary_holes=c(1,3),method='one_ear',
   boundary_hole_color=c('clear','clear','green','clear'))
 iso_trapezoid_maze(depth=log2(len),15,clockwise=FALSE,draw_boundary=TRUE,
   boundary_lines=c(2,3,4),boundary_holes=c(2),method='one_ear',
   boundary_hole_color=c('red'))
})
```

---

koch_maze *koch_maze .*

Description

Recursively draw an Koch snowflake maze. The inner part of the snowflake maze consists of an equilateral triangle of side length 3^{depth} pieces of length `unit_len`.

Usage

```
koch_maze(
  depth,
  unit_len,
  clockwise = TRUE,
  draw_boundary = TRUE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length. Should be an integer.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

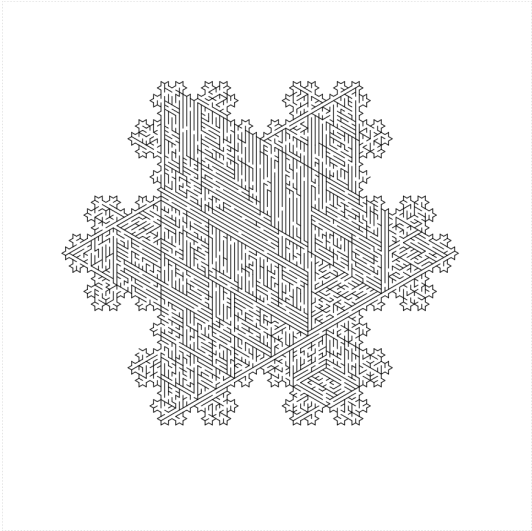
`boundary_hole_locations` the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows` a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in an Koch snowflake, starting from the corner of the first side. Relies on generation of triangular mazes for the internals. The triangular part has sides consisting of `3^depth` segments of length `unit_len`.



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
library(TurtleGraphics)
turtle_init(2000,2000)
turtle_hide()
turtle_up()
set.seed(1234)
turtle_do({
 turtle_backward(distance=400)
 turtle_left(90)
 turtle_forward(650)
 turtle_right(90)
 turtle_right(30)
 koch_maze(depth=3,unit_len=14)
})
```

`mazealls` *generate recursive mazes*

Description

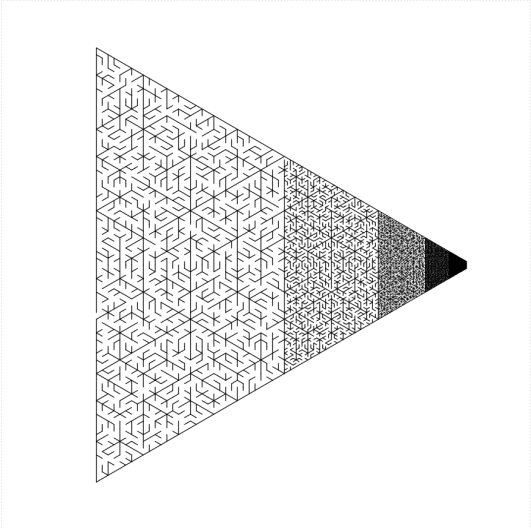
Generate recursive mazes.

Details

Recursive generation of mazes proceeds roughly as follows: subdivide the domain logicall into two or more parts, creating mazes in the sub-parts, then drawing dividing lines between them with some holes. The holes in the dividing lines should be constructed so that the sub-parts form a tree, with exactly one way to get from one of the sub-parts to any one of the others. Then an optional outer boundary with optional holes is drawn to finish the maze.

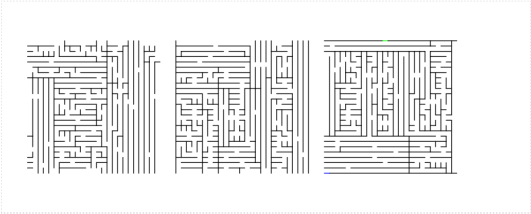
unit length

The `unit_len` parameter controls the graphical length of one 'unit', which is the length of holes between sections of the mazes, and is roughly the width of the 'hallways' of a maze. Here is an example of using different unit lengths in a stack of trapezoids



boundaries

The parameters `draw_boundary`, `boundary_lines`, `boundary_holes`, `num_boundary_holes` and `boundary_hole_color` control the drawing of the final outer boundary of polynomial mazes. Without a boundary the maze can be used in recursive construction. Adding a boundary provides the typical entry and exit points of a maze. The parameter `draw_boundary` is a single Boolean that controls whether the boundary is drawn or not. The parameter `boundary_lines` may be a scalar Boolean, or a numeric array giving the indices of which sides should have drawn boundary lines. The sides are numbered in the order in which they appear, and are controlled by the `clockwise` parameter. The parameter `boundary_holes` is a numeric array giving the indices of the boundary lines that should have holes. If `NULL`, then we uniformly choose `num_boundary_holes` holes at random. Holes can be drawn as colored segments with the `boundary_hole_color`, which is a character array giving the color of each hole. The value 'clear' stands in for clear holes. Arrows can optionally be drawn at the boundary holes via the `boundary_hole_arrows` parameter, which is either a logical array or a numerical array indicating which sides should have boundary hole arrows.



end side

The `end_side` parameter controls which side of the maze the turtle ends on. The default value of 1 essentially causes the turtle to end where it started. The sides are numbered in the order in which the boundary would be drawn. Along with the boundary controls, the ending side can be useful to join together polygons into more complex mazes.

Legal Mumbo Jumbo

mazealls is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

Note

This package is dedicated to my friend, Abie Flaxman, who gave me the idea, and other ideas.

If you like this package, please endorse the author for 'mazes' on LinkedIn.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Maintainer: Steven E. Pav <shabbychef@gmail.com> (ORCID)

See Also

Useful links:

- https://github.com/shabbychef/mazealls
- Report bugs at https://github.com/shabbychef/mazealls/issues

mazealls-NEWS *News for package 'mazealls':*

Description

News for package 'mazealls'

mazealls Version 0.2.1 (2025-11-26)

- fixing figure width styling for CRAN.
- adding Hilbert space filling parallelogram mazes.
- adding Sierpinski hexagon mazes.

mazealls Version 0.2.0 (2017-12-12)

- adding octagon, decagon and dodecagon mazes.
- adding Sierpinski triangle, carpet and trapezoid mazes.
- adding hexaflake maze.
- adding option to draw arrows at boundary holes.
- adding boustrophedon factor to parallelogram, triangle, trapezoid, hexagon mazes.

mazealls Initial Version 0.1.0 (2017-11-12)

- first CRAN release.

octagon_maze *octagon_maze .*

Description

Draw a regular octagon maze, with each side consisting of of 2^{depth} pieces of length `unit_len`.

Usage

```
octagon_maze(
  depth,
  unit_len = 4L,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  method = c("ammann_beenker"),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
```

```
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`method` there are a few ways to recursively draw an octagon. The following values are acceptable:

ammann_beenker Decompose into 4 45-degree rhombuses and two squares.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

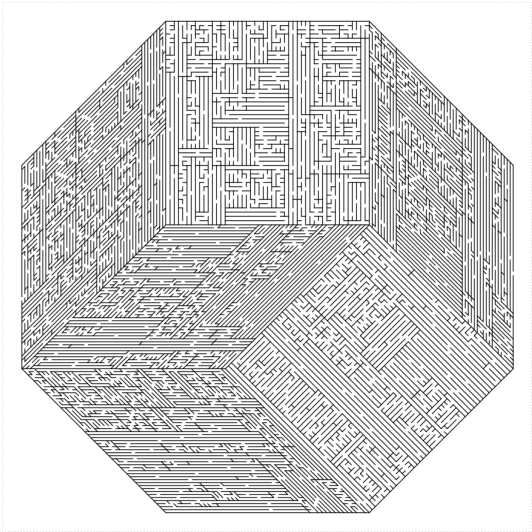
`boundary_hole_locations` the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows` a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in a regular octagon via dissection into rhombuses.



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
## Not run:
turtle_init(2000,2000,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
  turtle_setpos(75,1000)
  turtle_setangle(0)
 octagon_maze(6,12,draw_boundary=TRUE)
})

## End(Not run)
```

---

parallelogram_maze *parallelogram_maze .*

Description

Recursively draw a parallelogram maze, with the first side consisting of `height` segments of length `unit_len`, and the second side `width` segments of length `unit_len`. The angle between the first and second side may be set.

Usage

```
parallelogram_maze(
  unit_len,
  height,
  width = height,
  angle = 90,
  clockwise = TRUE,
 method = c("two_parallelograms", "four_parallelograms", "uniform", "random", "hilbert"),
  start_from = c("midpoint", "corner"),
  hilbert_balance = 1,
  balance = 0,
  height_boustro = c(1, 1),
  width_boustro = c(1, 1),
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`height` the length of the first side in numbers of `unit_len` segments.

`width` the length of the second side in numbers of `unit_len` segments.

`angle` the angle (in degrees) between the first and second sides. Note that this is the angle that the Turtle turns when rounding the first corner, so it is the internal angle at the starting point (if starting from a corner), and the external angle at the second corner.

`clockwise` whether to draw clockwise.

`method` there are many ways to recursive draw an isosceles trapezoid. The following values are acceptable:

two_parallelograms The parallelogram maze is built as two parallelogram mazes with a holey line between them.

four_parallelograms The parallelogram maze is built as four parallelogram mazes with three holey lines and one solid line between them.

uniform The parallelogram maze is built as four parallelogram mazes with three holey lines and one solid line between them. Sub-mazes are chosen to be nearly equal in size.

hilbert The parallelogram maze is built as for parallelogram mazes with a U-shaped connection between them.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`hilbert_balance`
this is essentially the balance parameter but for the `hilbert` method, controlling horizontal and vertical splits.

`balance` for the `two_parallelograms` method, we choose whether to split on height or width based on a balance condition. The log odds of choosing height over width is the factor `balance` times the sign of the difference `height - width`. When balance takes the default value of 0, you have equal odds of selecting to split on height or width. Note that balance is positive and large, you tend to generate nearly uniform splits. When balance is negative and large, you tend to have imbalanced mazes, and the imbalance propagates.

`height_boustro` an array of two values, which help determine the location of holes in internal lines of length `height`. The default value, `c(1,1)` results in uniform selection. Otherwise the location of holes are chosen with probability proportional to a beta density with `shape1` and `shape2` the two elements of `height_boustro` in order. In sub mazes, this parameter is reversed, which can lead to 'boustrophedonic' mazes. The sum of values should probably not exceed 30, as otherwise the location of internal holes is forced.

`width_boustro` an array of two values, which help determine the location of any split along lines which are length `width`.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes`
the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color`
the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

`boundary_hole_locations`
the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the

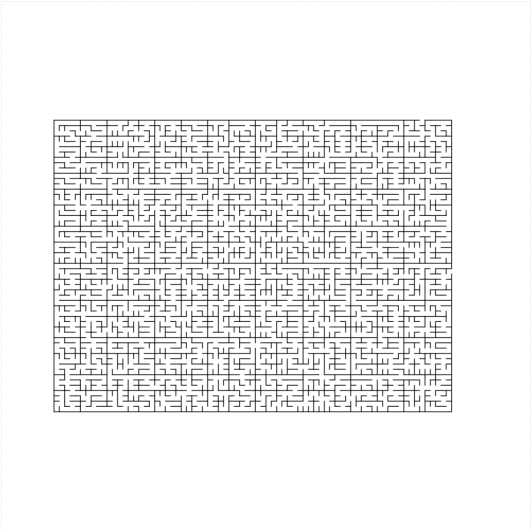
location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows`
: a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

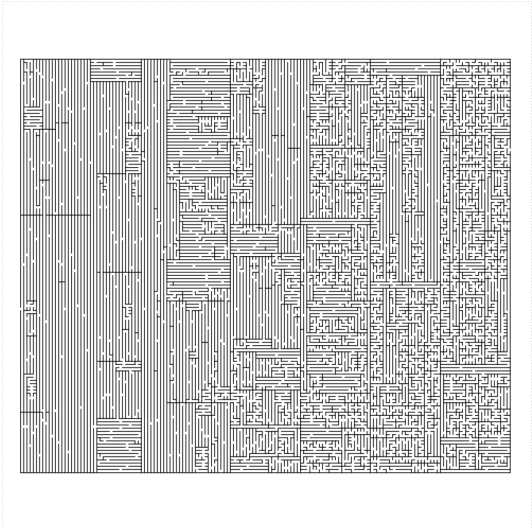
`end_side`
: the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in an parallelogram, starting from the midpoint of the first side (or the corner before the first side via the `start_from` option). Can recursively subdivide into two or four parallelograms. The first (and third) side shall consist of `height` segments of length `unit_len`. The second and fourth side consist of `width` segments of length `unit_len`. The angle between them is `angle`. Here is an example maze:



This function admits a `balance` parameter which controls how the maze should be recursively subdivided. A negative value creates imbalanced mazes, while positive values create more uniform mazes. Here are create seven mazes created side by side with an increasing balance parameter:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

```
library(TurtleGraphics)

turtle_init(500,300,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
 turtle_setpos(15,15)
 turtle_setangle(0)
 parallelogram_maze(angle=90,unit_len=10,width=45,height=25,method='uniform',
  start_from='corner',draw_boundary=TRUE)
})

# testing imbalance condition
turtle_init(400,500,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
 turtle_setpos(15,250)
 turtle_setangle(0)
 parallelogram_maze(angle=90,unit_len=10,width=30,height=40,
   method='two_parallelograms',draw_boundary=TRUE,balance=-1.0)
})
```

```
# a bunch of imbalanced mazes, fading into each other
turtle_init(850,400,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
  turtle_setpos(15,200)
  turtle_setangle(0)
  valseq <- seq(from=-1.5,to=1.5,length.out=4)
  blines <- c(1,2,3,4)
  bholes <- c(1,3)
  set.seed(12354)
  for (iii in seq_along(valseq)) {
     parallelogram_maze(angle=90,unit_len=10,width=20,height=25,
      method='two_parallelograms',draw_boundary=TRUE,balance=valseq[iii],
       end_side=3,boundary_lines=blines,boundary_holes=bholes)
     turtle_right(180)
     blines <- c(2,3,4)
     bholes <- c(3)
  }
})

# a somewhat 'boustrophedonic' maze
turtle_init(500,300,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
 turtle_setpos(15,15)
 turtle_setangle(0)
 parallelogram_maze(angle=90,unit_len=10,width=47,height=27,
   method='two_parallelograms', height_boustro=c(21,3),width_boustro=c(21,3),balance=-0.25,
    start_from='corner',draw_boundary=TRUE)
})

# a Hilbert Curve maze
turtle_init(510,510,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
 turtle_setpos(15,15)
 turtle_setangle(0)
 parallelogram_maze(angle=90,unit_len=10,width=48,height=48,
    method='hilbert',hilbert_balance=3,
    height_boustro=c(10,2),width_boustro=c(10,2),
   boundary_holes=c(1,4),start_from='corner',boundary_hole_locations=c(1,1,1,1),draw_boundary=TRUE)
})
```

---

sierpinski_carpet_maze

sierpinski_carpet_maze .

Description

Recursively draw a Sierpinski carpet maze in a parallelogram, with the first side consisting of `height` segments of length `unit_len`, and the second side `width` segments of length `unit_len`. The angle between the first and second side may be set.

Usage

```
sierpinski_carpet_maze(
  unit_len,
  height,
  width = height,
  angle = 90,
  clockwise = TRUE,
  method = "random",
  color1 = "black",
  color2 = "gray40",
  start_from = c("midpoint", "corner"),
  balance = 0,
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`unit_len`	the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
`height`	the length of the first side in numbers of `unit_len` segments.
`width`	the length of the second side in numbers of `unit_len` segments.
`angle`	the angle (in degrees) between the first and second sides.
`clockwise`	whether to draw clockwise.
`method`	passed to `parallelogram_maze` to control the method of drawing the sub mazes.
`color1`	The dominant color of the maze.
`color2`	The negative color of the maze.
`start_from`	whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.
`balance`	passed to `parallelogram_maze` to control imbalance of sub mazes.
`draw_boundary`	a boolean indicating whether a final boundary shall be drawn around the maze.
`num_boundary_holes`	the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color`
the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

`boundary_hole_locations`
the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows`
a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a Sierpinski carpet as two-color maze in a parallelogram.

Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

See Also

`parallelogram_maze`, `sierpinski_maze`.

Examples

```
library(TurtleGraphics)
turtle_init(800,900,mode='clip')
turtle_hide()
```

```
turtle_up()
turtle_do({
 turtle_setpos(35,400)
 turtle_setangle(0)
 sierpinski_carpet_maze(angle=80,unit_len=8,width=30,height=30,
   method='two_parallelograms',draw_boundary=TRUE,balance=-1.0,color2='green')
})

## Not run:
library(TurtleGraphics)
turtle_init(2000,2000,mode='clip')
turtle_hide()
turtle_up()
bholes <- list(c(1,2), c(1), c(2))
turtle_do({
 turtle_setpos(1000,1100)
 turtle_setangle(180)
 for (iii in c(1:3)) {
  mybhol <- bholes[[iii]]
  sierpinski_carpet_maze(angle=120,unit_len=12,width=81,height=81,
   draw_boundary=TRUE,boundary_lines=c(1,2,3),num_boundary_holes=0,
   boundary_holes=mybhol,balance=1.0,color2='green',
   start_from='corner')
  turtle_left(120)
 }
})

## End(Not run)
```

---

sierpinski_hexagon_maze

sierpinski_hexagon_maze .

Description

Draws a Sierpinski hexagon maze.

Usage

```
sierpinski_hexagon_maze(
  depth,
  unit_len,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
 style = c("sierpinski", "four_triangles", "hexaflake", "reverse_flake", "outer_flake",
    "dragon_left", "dragon_right", "dragon_leftright", "dragon_rightleft"),
  color1 = "black",
  color2 = "gray40",
  draw_boundary = FALSE,
```

```
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth` the depth of recursion. This controls the side length. Should be an integer.

`unit_len` the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.

`clockwise` whether to draw clockwise.

`start_from` whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.

`style` controls the style of Sierpinski triangle. The following are recognized:

- **sierpinski** The traditional Sierpinski Triangle of four triangles with the center in the minor color, `color2`.
- **hexaflake** Looks more like a hexaflake in a triangle.
- **reverse_flake** Like a hexaflake, but the colors reverse in recursive calls.
- **outer_flake** Like a hexaflake, but only for the outermost ring.
- **dragon_left** Looks like a dragon fractal.
- **dragon_right** Looks like a dragon fractal.
- **dragon_leftright** Looks like a dragon fractal.
- **dragon_rightleft** Looks like a dragon fractal.

`color1` The dominant color of the maze.

`color2` The negative color of the maze.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes` the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color` the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

`boundary_hole_locations`
: the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows`
: a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side`
: the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Recursively draw a Sierpinski hexagon maze. The sides of the hexagon consist of 2^{depth} pieces of length `unit_len`. The hexagon is drawn as a ring of six Sierpinski trapezoids in a ring around a Sierpinski hexagon of a smaller size. The 'inner' and 'outer' pieces of mazes drawn in different colors. The

Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

See Also

`eq_triangle_maze`, `hexaflake_maze`, `sierpinski_carpet_maze`, `sierpinski_trapezoid_maze`.

Examples

```
library(TurtleGraphics)
turtle_init(800,800,mode='clip')
turtle_up()
turtle_hide()
turtle_do({
 turtle_setpos(50,400)
 turtle_setangle(30)
 sierpinski_hexagon_maze(depth=4,unit_len=20,boundary_lines=TRUE,
  draw_boundary=TRUE,boundary_holes=c(1,3),
  start_from='corner',
  color1='black',color2='green',
  style='sierpinski')
})
```

sierpinski_maze *sierpinski_maze .*

Description

Recursively draw a Sierpinski triangle maze. The sides of the triangle consist of 2^{depth} pieces of length `unit_len`. The 'inner' and 'outer' pieces of the flake are mazes drawn in different colors.

Usage

```
sierpinski_maze(
  depth,
  unit_len,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  method = "random",
  style = c("four_triangles", "hexaflake", "dragon_left", "dragon_right"),
  color1 = "black",
  color2 = "gray40",
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth`	the depth of recursion. This controls the side length. Should be an integer.
`unit_len`	the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
`clockwise`	whether to draw clockwise.
`start_from`	whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.
`method`	controls the method to draw the underlying equilateral triangles. See `eq_triangle_maze`.
`style`	controls the style of Sierpinski triangle. The following are recognized: **four_triangles** The traditional Sierpinski Triangle of four triangles with the center in the minor color, `color2`. **hexaflake** Looks more like a hexaflake in a triangle. **dragon_left** Looks like a dragon fractal. **dragon_right** Looks like a dragon fractal.

`color1` The dominant color of the maze.

`color2` The negative color of the maze.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes`
the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color`
the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

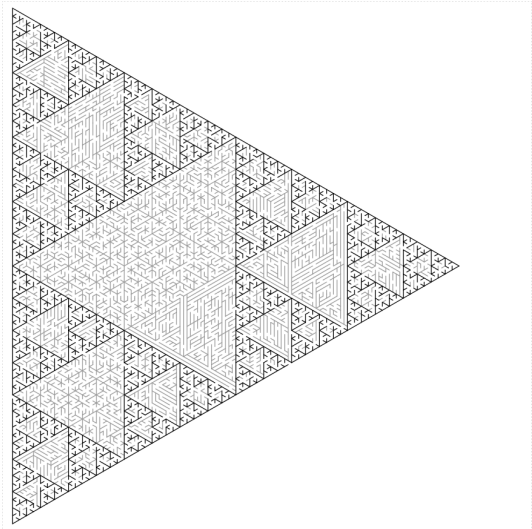
`boundary_hole_locations`
the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

`boundary_hole_arrows`
a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in an Sierpinski equilateral Triangle. The inner quarter is drawn in the secondary color, while the outer three quarters are drawn recursively. This is the traditional Sierpinski Triangle, generated when `style=='four_triangles'`:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

See Also

eq_triangle_maze, hexaflake_maze, sierpinski_carpet_maze, sierpinski_trapezoid_maze,

Examples

```
library(TurtleGraphics)
turtle_init(1000,1000,mode='clip')
turtle_up()
turtle_hide()
turtle_do({
 turtle_setpos(10,500)
 turtle_setangle(0)
 sierpinski_maze(depth=5,unit_len=19,boundary_lines=TRUE,
   boundary_holes=c(1,3),color1='black',color2='gray60')
})
```

sierpinski_trapezoid_maze *sierpinski_trapezoid_maze .*

Description

Recursively draw a Sierpinski isosceles trapezoid maze, with three sides consisting of 2^{depth} pieces of length `unit_len`, and one long side of length $2^{depth+1}$ pieces, starting from the long side.

Usage

```
sierpinski_trapezoid_maze(
  depth,
  unit_len = 4L,
  clockwise = TRUE,
  start_from = c("midpoint", "corner"),
  color1 = "black",
  color2 = "gray40",
  flip_color_parts = 1,
  draw_boundary = FALSE,
  num_boundary_holes = 2,
  boundary_lines = TRUE,
  boundary_holes = NULL,
  boundary_hole_color = NULL,
  boundary_hole_locations = NULL,
  boundary_hole_arrows = FALSE,
  end_side = 1
)
```

Arguments

`depth`	the depth of recursion. This controls the side length: three sides have `round(2^depth)` segments of length `unit_len`, while the long side is twice as long. `depth` need not be integral.
`unit_len`	the unit length in graph coordinates. This controls the width of the 'holes' in the boundary lines and generally controls the spacing of mazes.
`clockwise`	whether to draw clockwise.
`start_from`	whether to start from the midpoint of the first side of a maze, or from the corner facing the first side.
`color1`	The dominant color of the maze.
`color2`	The negative color of the maze.
`flip_color_parts`	a numerical array which can contain values 1 through 4. Those parts of the maze, when drawn recursively, have their colors flipped. A value of `3` corresponds to a traditional Sierpinski triangle, while `1` corresponds to a Hexaflake. Values of `2` or `4` look more like dragon mazes.

`draw_boundary` a boolean indicating whether a final boundary shall be drawn around the maze.

`num_boundary_holes`
the number of boundary sides which should be randomly selected to have holes. Note that the `boundary_holes` parameter takes precedence.

`boundary_lines` indicates which of the sides of the maze shall have drawn boundary lines. Can be a logical array indicating which sides shall have lines, or a numeric array, giving the index of sides that shall have lines.

`boundary_holes` an array indicating which of the boundary lines have holes. If `NULL`, then boundary holes are randomly selected by the `num_boundary_holes` parameter. If numeric, indicates which sides of the maze shall have holes. If a boolean array, indicates which of the sides shall have holes. These forms are recycled if needed. See `holey_path`. Note that if no line is drawn, no hole can be drawn either.

`boundary_hole_color`
the color of boundary holes. A value of `NULL` indicates no colored holes. See `holey_path` for more details. Can be an array of colors, or colors and the value `'clear'`, which stands in for `NULL` to indicate no filled hole to be drawn.

`boundary_hole_locations`
the 'locations' of the boundary holes within each boundary segment. A value of `NULL` indicates the code may randomly choose, as is the default. May be a numeric array. A positive value up to the side length is interpreted as the location to place the boundary hole. A negative value is interpreted as counting down from the side length plus 1. A value of zero corresponds to allowing the code to pick the location within a segment. A value of `NA` may cause an error.

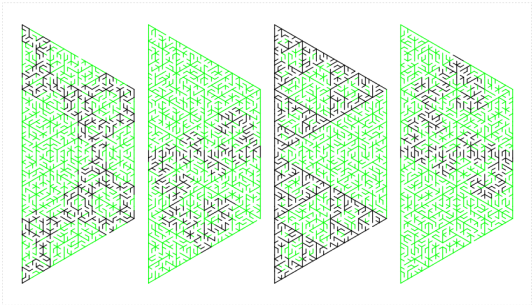
`boundary_hole_arrows`
a boolean or boolean array indicating whether to draw perpendicular double arrows at the boundary holes, as a visual guide. These can be useful for locating the entry and exit points of a maze.

`end_side` the number of the side to end on. A value of 1 corresponds to the starting side, while higher numbers correspond to the drawn side of the figure in the canonical order (that is, the order induced by the `clockwise` parameter).

Details

Draws a maze in an isoscelese trapezoid with three sides of equal length and one long side of twice that length, starting from the midpoint of the long side (or the corner before the first side via the `start_from` option). Differently colors the parts of the maze for a Sierpinski effect.

Here are mazes for different values of `flip_color_parts` ranging from 1 to 4:



Value

nothing; the function is called for side effects only, though in the future this might return information about the drawn boundary of the shape.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

See Also

iso_trapezoid_maze, hexaflake_maze, sierpinski_carpet_maze, sierpinski_maze.

Examples

```
require(TurtleGraphics)
turtle_init(1000,1000,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
  turtle_setpos(500,500)
  turtle_setangle(0)
  sierpinski_trapezoid_maze(unit_len=15,depth=4,color1='black',color2='green',
    clockwise=TRUE,draw_boundary=TRUE,boundary_holes=c(1,3))
  sierpinski_trapezoid_maze(unit_len=15,depth=4,color1='black',color2='green',
    clockwise=FALSE,draw_boundary=TRUE,
    boundary_lines=c(2,3,4),boundary_holes=3)
})

# stack some trapezoids!
require(TurtleGraphics)
turtle_init(750,900,mode='clip')
turtle_hide()
```

```
turtle_up()
turtle_do({
  turtle_setpos(25,450)
  turtle_setangle(0)
  blines <- c(1,2,4)
  for (dep in seq(from=4,to=0)) {
    sierpinski_trapezoid_maze(unit_len=13,depth=dep,color1='black',color2='green',
      flip_color_parts=2,
      clockwise=TRUE,boundary_lines=blines,draw_boundary=TRUE,boundary_holes=c(1,3),
      end_side=3)
    turtle_right(180)
    blines <- c(1,2,4)
  }
})
## Not run:
require(TurtleGraphics)
turtle_init(750,900,mode='clip')
turtle_hide()
turtle_up()
turtle_do({
  turtle_setpos(25,450)
  turtle_setangle(0)
  blines <- c(1,2,4)
  for (dep in seq(from=5,to=0)) {
    sierpinski_trapezoid_maze(unit_len=13,depth=dep,color1='black',color2='green',
      flip_color_parts=3,
      clockwise=TRUE,boundary_lines=blines,draw_boundary=TRUE,boundary_holes=c(1,3),
      end_side=3)
    turtle_right(180)
    blines <- c(1,2,4)
  }
})

## End(Not run)
```

Index

∗ **Sierpinski**
sierpinski_carpet_maze, 39
sierpinski_hexagon_maze, 42
sierpinski_maze, 45
sierpinski_trapezoid_maze, 48
∗ **package**
mazealls, 29
∗ **plotting**
decagon_maze, 2
dodecagon_maze, 4
eq_triangle_maze, 7
hexaflake_maze, 13
hexagon_maze, 16
holey_line, 21
holey_path, 22
iso_trapezoid_maze, 24
koch_maze, 27
octagon_maze, 32
parallelogram_maze, 34
sierpinski_carpet_maze, 39
sierpinski_hexagon_maze, 42
sierpinski_maze, 45
sierpinski_trapezoid_maze, 48

colors, *21, 22*

decagon_maze, 2
dodecagon_maze, 4

eq_triangle_maze, 7, *44, 45*, *47*

hexaflake_maze, 13, *44*, *47*, *50*
hexagon_maze, 16
holey_line, 21, *23*
holey_path, *3*, *5*, *8*, *14*, *17*, 22, *25*, *28*, *33*, *36*, *41*, *43*, *46*, *49*

iso_trapezoid_maze, 24, *50*

koch_maze, 27

mazealls, 29
mazealls-NEWS, 32
mazealls-package *(*mazealls*)*, 29

octagon_maze, 32

parallelogram_maze, 34, *40, 41*

sierpinski_carpet_maze, 39, *44*, *47*, *50*
sierpinski_hexagon_maze, 42
sierpinski_maze, *41*, 45, *50*
sierpinski_trapezoid_maze, *15*, *44*, *47*, 48