Extracted from:

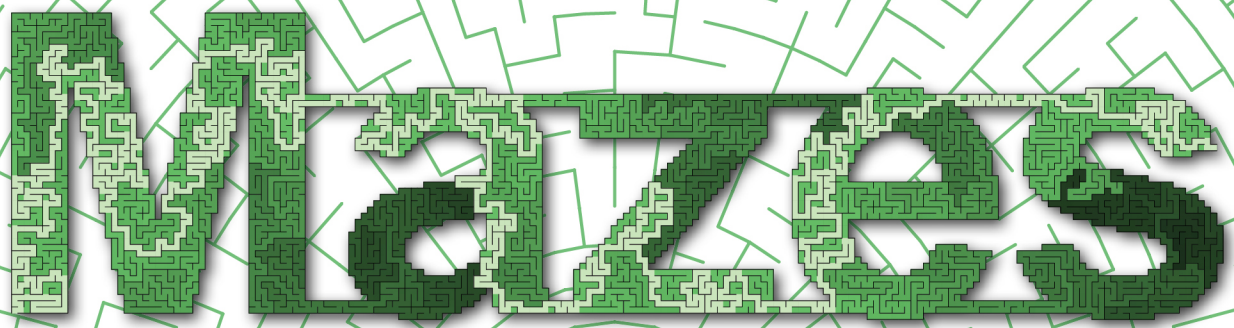
Mazes for Programmers

Code Your Own Twisty Little Passages

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers



for Programmers

Code Your Own Twisty Little Passages

Jamis Buck

Edited by Jacquelyn Carter

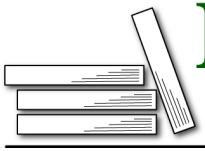
Mazes for Programmers

Code Your Own Twisty Little Passages

Jamis Buck

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

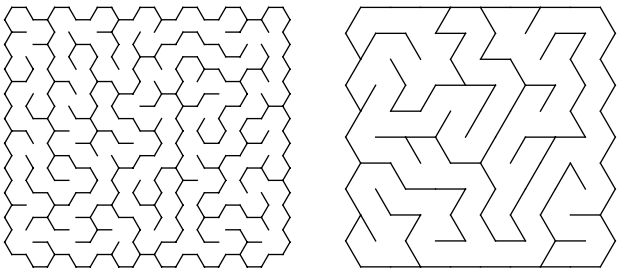
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-055-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2015

When you take a surface and divide it up into different shapes, with no gaps between them and no overlaps, you get what is called a *tessellation* of the surface. Our standard grid is one such tessellation, where we've broken up a flat area, or *plane*, into smaller squares. Another way to say this is that we've *tiled* the plane with squares.

It turns out that squares aren't the only shape that can do this for us. In this chapter, we'll look at two other grids made by tiling other geometric shapes. We'll see how hexagons come together in a honeycomb pattern and triangles form a girder-style lattice. Over the course of the chapter, we'll use these new grids to turn out mazes like the following:

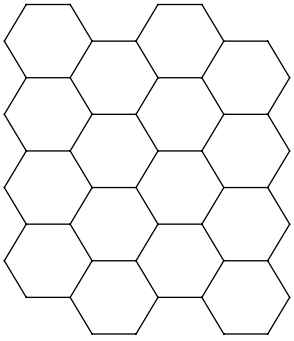


Let's start with the one on the left: a maze on hexagon grid.

Implementing a Hex Grid

So far we've made regular grids and circular grids. Our next goal is to create a grid of hexagons, also called a *hex grid* for short. We'll approach this by first considering a single cell, with an eye to understanding how it relates spatially to its neighbors. From there, we'll take that information and implement the grid itself.

The cells aren't difficult to implement. The trickiest part is just understanding how they all fit together. Let's look at a simple hex grid here:



From that, we can see that each hexagon neighbors up to six other hexagons, one for each of north, south, northwest, northeast, southwest, and southeast. Right off, it's clear that our existing Cell won't cut it; it doesn't include enough neighbors! Let's take care of that now by introducing a new Cell subclass.

Put the following in hex_cell.rb.

hex_cell.rb

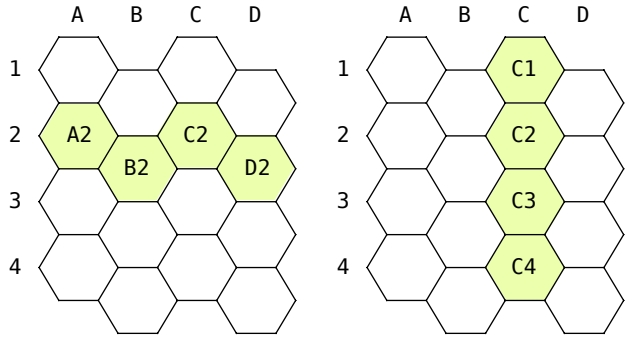
```
require 'cell'

class HexCell < Cell
  attr_accessor :northeast, :northwest
  attr_accessor :southeast, :southwest

  def neighbors
    list = []
    list << northwest if northwest
    list << north if north
    list << northeast if northeast
    list << southwest if southwest
    list << south if south
    list << southeast if southeast
    list
  end
end
```

This simply extends the Cell class, adding new accessors for northwest, northeast, southwest, and southeast, and then updates the neighbors method to return those new directions. Note that the west and east accessors inherited from the Cell class are unused. With the hexagons in our grid oriented the way they are (flat-topped), they will never have neighbors to the east or west.

The next step is to figure out how these cells are arranged in the grid. Although they aren't necessarily laid out in clear rows, it's not too hard to find an arrangement that works well.

Looking at the previous diagram again, we can see that although vertical columns are clearly present in the grid, horizontal rows are less obvious. It's probably no surprise that there are lots of different ways to approach representing hexagons in a program, nor should it be surprising that each will have different pros and cons. We're going to choose a way to do it that maps most closely to our underlying two-dimensional array, simply because it requires the fewest changes on our end to implement.

The following figure shows how it will work, treating each row as a zig-zag path from one side of the grid to the other, whereas columns simply drop vertically through the grid.



This choice has consequences for how we set up the grid. Most immediately, it means that when we set up the adjacency information for cells in some columns (for example, *B* and *D* in the previous illustration), the northwest and northeast diagonals point to the cell's *same row*, whereas southwest and southeast point to the row below. Conversely, for the other columns (for example, *A* and *C*) northwest and northeast are the ones that point to a different row.

Tricky, but not insurmountable!

Let's make a new `Grid` subclass. We'll override the `prepare_grid` method so that it instantiates our new `HexCell` class for cells, and we'll override `configure_cells` to set up the correct adjacency information for each cell.

Put the following in `hex_grid.rb`.

hex_grid.rb

```
Line 1  require 'grid'
-       require 'hex_cell'
-
-       class HexGrid < Grid
5         def prepare_grid
-           Array.new(rows) do |row|
-             Array.new(columns) do |column|
-               HexCell.new(row, column)
-             end
10          end
-         end
-
-         def configure_cells
-           each_cell do |cell|
15           row, col = cell.row, cell.column
-
-           if col.even?
-             north_diagonal = row - 1
-             south_diagonal = row
20           else
-             north_diagonal = row
-             south_diagonal = row + 1
-           end
-
25           cell.northwest = self[north_diagonal, col - 1]
-           cell.north     = self[row - 1, col]
-           cell.northeast = self[north_diagonal, col + 1]
-           cell.southwest = self[south_diagonal, col - 1]
-           cell.south     = self[row + 1, col]
30           cell.southeast = self[south_diagonal, col + 1]
-          end
-         end
-       end
```

Lines 17–23 set up some variables to help us deal with those zig-zagging rows. When the column is even-numbered (`col.even?`), we make it so the northern diagonals point to the preceding row, whereas the southern diagonals point to the current row. When the column is odd-numbered, we swap it so the northern diagonals point to the current row, and the southern ones point to the following row.

With those variables, lines 25–30 compute the cells that abut the current cell.

That will suffice to set up our grid, with all the cells appropriately cozy with their neighbors. All we lack now is a way to display it, since the `to_png` method on `Grid` itself can't handle anything but squares.

Displaying a Hex Grid

To display a hex grid, we need to be able to compute the coordinates of each of its corners, or vertices. We'll see how to compute those coordinates relative to the center of each hexagon, as well as how to compute the overall dimensions of a hex grid, and then we'll plug that all into a new `to_png` implementation.

We're going to assume that our grid is composed of *regular* hexagons—hexagons whose sides are all the same length. With that assumption, there is a lovely little derivation involving *equilateral* triangles (triangles whose sides are all equal) that lets us get the measurements we need. For the sake of brevity, we'll skip the derivation itself here, but if you're into geometry at all it's pleasantly straightforward.

Essentially, what we want are the lengths of a_1, a_2, and b in the following diagram:

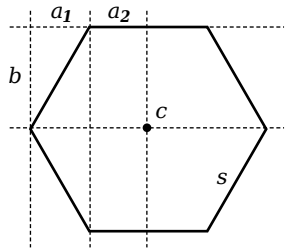


Figure 3—A Dissected Hexagon

If c is the center of our hexagon, and s is the length of a side, then it turns out that a_1 and a_2 are identical. (We'll just keep it simple, then, and call them both a.) We also find that a is half of s, and the length of b is $s\sqrt{3}/2$. From that, it follows that the width of our hexagon (from western point to eastern point) is exactly $2s$, and the height is $2b$. In code:

```
a_size = s / 2.0
b_size = s * Math.sqrt(3) / 2.0
width  = s * 2.0
height = b_size * 2.0
```

With these numbers we can compute the x- and y-coordinates of all six of the hexagon's vertices. If `cx` and `cy` represent the coordinates of the center point for some cell, and we call those vertices "far" that are farther from the center and "near" those that are nearer, we get:

```
x_far_west  = cx - s
x_near_west = cx - a_size
x_near_east = cx + a_size
x_far_east  = cx + s

y_north     = cy - b_size
y_mid       = cy
y_south     = cy + b_size
```

Now that we have those named coordinates we can say (for instance) that the vertex at the 3 o'clock position is at (x_far_east,y_mid).

The last bit we need before implementing our new drawing code is a way to calculate the dimensions of our canvas. It's not as straightforward as a regular grid, because adjacent cells are offset from each other. In cases like this, it's helpful to take our grid and overlay a regular grid on top of it, like the following figure.

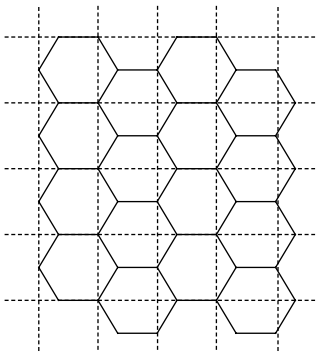


Figure 4—Measuring a Hex Grid

We know how to compute the dimensions of a regular grid, so if we can figure out how wide each cell of the overlay is, we can determine the size of our original grid.

In this case, we've got a 4×4 grid of hexagons. If you recall Figure 3, *A Dissected Hexagon*, on page 9, you can see that each of the rectangles here are three a lengths wide, measuring from the western point of one hex to the western point of the next neighbor. Counting the squares, then, this 4×4 canvas as a whole is as wide as four of those squares, plus one more a length. In code, it might come together like this:

```
canvas_width = 3 * columns * a_size + a_size
```

The height is much more straightforward. We can easily see that each square is as tall as one hexagon, and that the canvas as a whole is as tall as four of those hexes, plus another half-hex (a single *b* length). In other words:

```
canvas_height = rows * height + b_size
```

Putting this all together, we can finally write our new to_png method! Put the following in hex_grid.rb, somewhere inside the HexGrid class.

hex_grid.rb

```
def to_png(size: 10)
  a_size = size / 2.0
  b_size = size * Math.sqrt(3) / 2.0
  width = size * 2
  height = b_size * 2

  img_width = (3 * a_size * columns + a_size + 0.5).to_i
  img_height = (height * rows + b_size + 0.5).to_i

  background = ChunkyPNG::Color::WHITE
  wall = ChunkyPNG::Color::BLACK

  img = ChunkyPNG::Image.new(img_width + 1, img_height + 1, background)

  [:backgrounds, :walls].each do |mode|
    each_cell do |cell|
      cx = size + 3 * cell.column * a_size
      cy = b_size + cell.row * height
      cy += b_size if cell.column.odd?

      # f/n = far/near
      # n/s/e/w = north/south/east/west
      x_fw = (cx - size).to_i
      x_nw = (cx - a_size).to_i
      x_ne = (cx + a_size).to_i
      x_fe = (cx + size).to_i

      # m = middle
      y_n = (cy - b_size).to_i
      y_m = cy.to_i
      y_s = (cy + b_size).to_i

      if mode == :backgrounds
        color = background_color_for(cell)
        if color
          points = [[x_fw, y_m], [x_nw, y_n], [x_ne, y_n],
                    [x_fe, y_m], [x_ne, y_s], [x_nw, y_s]]
          img.polygon(points, color, color)
        end
      else
```

```
        img.line(x_fw, y_m, x_nw, y_s, wall) unless cell.southwest
        img.line(x_fw, y_m, x_nw, y_n, wall) unless cell.northwest
        img.line(x_nw, y_n, x_ne, y_n, wall) unless cell.north
        img.line(x_ne, y_n, x_fe, y_m, wall) unless cell.linked?(cell.northeast)
        img.line(x_fe, y_m, x_ne, y_s, wall) unless cell.linked?(cell.southeast)
        img.line(x_ne, y_s, x_nw, y_s, wall) unless cell.linked?(cell.south)
      end
    end
  end

  img
end
```

The named size parameter on line 1 is what we've called *s* previously—the size of a single side of the hexagon. The computations that follow (lines 2–5) use that value to determine the dimensions of our hexagons, as discussed.

The next two lines (7 and 8) compute the total width of our canvas. The extra 0.5 makes sure we always round to the nearest whole number.

The three lines starting at 17 compute the center point of the current cell, since that's what our subsequent calculations will be relative to. It all comes back to Figure 4, *Measuring a Hex Grid*, on page 10, except here we're measuring the distance of that current cell from the center of the first cells in its corresponding row and column.

Once we know the center point, we can compute the coordinates of the corners of the current cell, as we talked about earlier. Lines 23–31 take care of that.

The remaining lines work identically to our original to_png method, drawing the appropriate walls for each cell based on which neighbors exist and have been linked to the current cell.

That's it! With our new to_png, we ought to be able to draw hex grids and—by extension—mazes. Let's do that next!

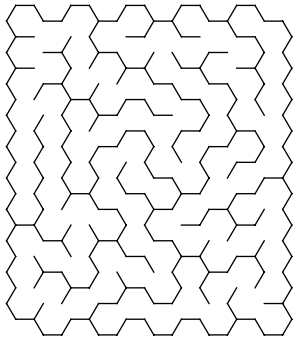
Making Hexagon (Sigma) Mazes

A maze made on a hex grid is, for some perverse reason, properly called a *sigma maze*. Fortunately, it works the same regardless of what you call it: just choose a maze algorithm and let it run its course. Let's use the Recursive Backtracker algorithm to put our new to_png implementation through its paces. Put the following in a file named hex_maze.rb.

hex_maze.rb

```
require 'recursive_backtracker'
require 'hex_grid'

grid = HexGrid.new(10, 10)
RecursiveBacktracker.on(grid)

grid.to_png.save('hex.png')
```

Running it, you ought to get something like this figure. Very nice! It's definitely working as we wanted. Experiment a bit with some other maze algorithms and see what you get, but be careful with Binary Tree and Sidewinder! These two need a bit of babysitting.



To understand why, recall *The Binary Tree Algorithm*, on page ?. For each cell, we choose between north and east to decide which neighbor to link, but in the case of a hexagon grid, *cells have no eastern neighbor*. The best we have are northeast, and southeast. Similarly, Sidewinder wants to choose an eastern neighbor but will also be foiled by our new geometry. So what do we do?

Well, neither of those algorithms actually wants "east." What they really want is "the cell in the same row, in the next column over." On a regular grid, that just happens to be *east.* For a hex grid, that will be either northeast or southeast, depending on the current column, but we don't even need to worry about that. We took care of the row/column assignments when we set up the grid. To get the neighbor in the next column over, we can just use our array accessor, like this:

```
east = grid[cell.row, cell.column+1]
```

Give it a try! Implement the Binary Tree and Sidewinder algorithms on a hex grid, and see what you get. When you're ready, we'll move on to another style of grid: triangles!