Hexagonal Grids

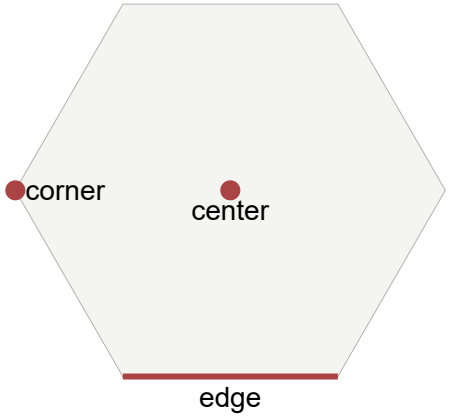
from Red Blob Games

Mar 2013, updated in Mar 2015

Hexagonal grids are used in some games but aren't quite as straightforward or common as square grids. I've been collecting hex grid resources for nearly 20 years, and wrote this guide to the most elegant approaches that lead to the simplest code, largely based on the guides by Charles Fu and Clark Verbrugge. I'll describe the various ways to make hex grids, the relationships between them, as well as some common algorithms. Many parts of this page are interactive; choosing a type of grid will update diagrams, code, and text to match.

The code samples on this page are written in pseudo-code; they're meant to be easy to read and understand. The implementation guide has code in C++, Javascript, C#, Python, Java, Typescript, and more.

Geometry



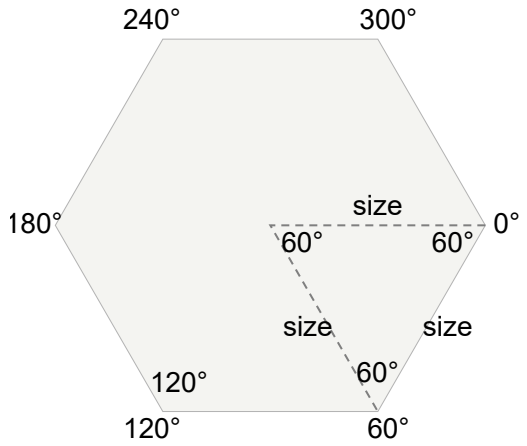
Hexagons are 6-sided polygons. *Regular* hexagons have all the sides (edges) the same length. I'll assume all the hexagons we're working with here are regular. The typical orientations for hex grids are horizontal (**pointy topped**) and vertical (**flat topped**).

Hexagons have 6 edges. Each edge is shared by 2 hexagons. Hexagons have 6 corners. Each corner is shared by 3 hexagons. For more about centers, edges, and corners, see my article on grid parts (squares, hexagons, and triangles).

Angles

In a regular hexagon the interior angles are 120°. There are six "wedges", each an equilateral triangle with 60° angles inside. Corner *i* is at `(60° * i)`, `size` units away from the `center`. In code:

```
function hex_corner(center, size, i):
    var angle_deg = 60 * i
    var angle_rad = PI / 180 * angle_deg
    return Point(center.x + size * cos(angle_rad),
                 center.y + size * sin(angle_rad))
```

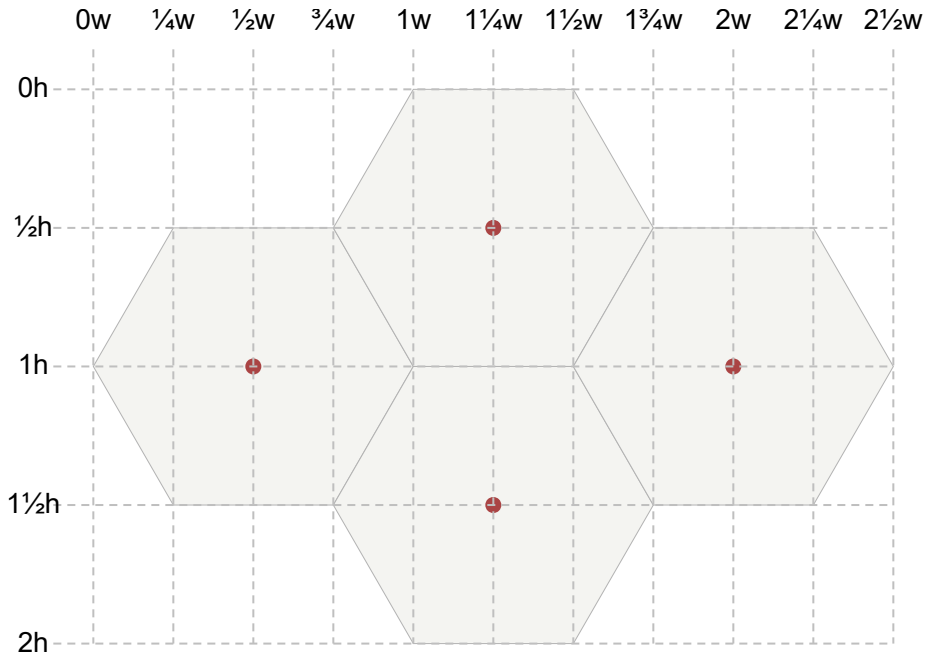


To fill a hexagon, gather the polygon vertices at `hex_corner(…, 0)` through `hex_corner(…, 5)`. To draw a hexagon outline, use those vertices, and then draw a line back to `hex_corner(…, 0)`.

The difference between the two orientations is that x and y are swapped, and that causes the angles to change: **flat topped** angles are 0°, 60°, 120°, 180°, 240°, 300° and **pointy topped** angles are 30°, 90°, 150°, 210°, 270°, 330°.

Size and Spacing

Next we want to put several hexagons together. In the vertical orientation, the width of a hexagon is `width = size * 2`. The horizontal distance between adjacent hexes is `horiz = width * 3/4`.



Switch to **flat topped** or back to **pointy topped** hexagons.

The height of a hexagon is `height = sqrt(3)/2 * width`. The vertical distance between adjacent hexes is `vert = height`.

Some games use pixel art for hexagons that does not match an exactly regular polygon. The angles and spacing formulas I describe in this section won't match the sizes of your hexagons. The rest of the article, describing algorithms on hex grids, will work even if your hexagons are stretched or shrunk a bit.

Coordinate Systems

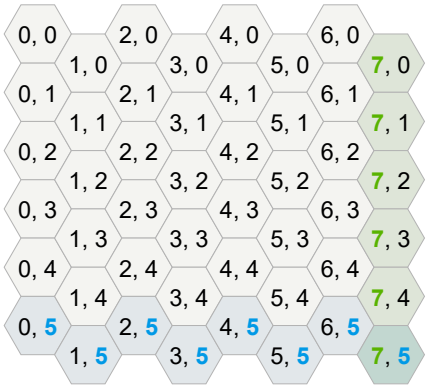
Now let's assemble hexagons into a grid. With square grids, there's one obvious way to do it. With hexagons, there are multiple approaches. I recommend using cube coordinates as the primary representation. Use either axial or offset coordinates for map storage, and displaying coordinates to the user.

Offset coordinates

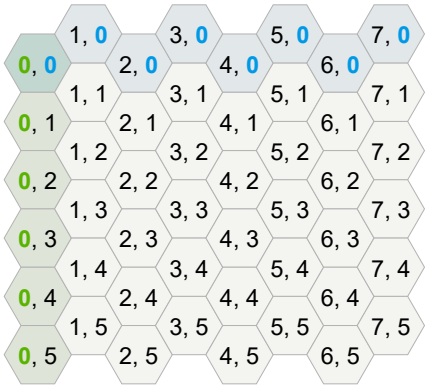
The most common approach is to offset every other column or row. Columns are named `col` or `q`. Rows are named `row` or `r`. You can either offset the odd or the even column/rows, so the horizontal and vertical hexagons each have two variants.

"odd-r" horizontal layout

"even-r" horizontal layout



"odd-q" vertical layout



"even-q" vertical layout

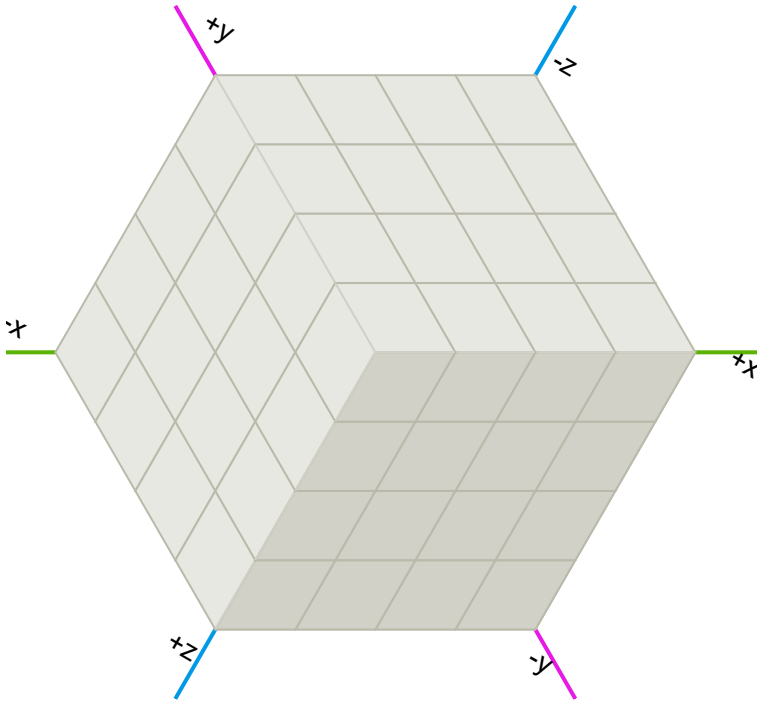
Cube coordinates

Another way to look at hexagonal grids is to see that there are *three* primary axes, unlike the *two* we have for square grids. There's an elegant symmetry with these.

Let's take a cube grid and **slice** out a diagonal plane at `x + y + z = 0`. This is a *weird* idea but it helps us make hex grid algorithms simpler. In particular, we can reuse standard operations from cartesian coordinates: adding coordinates, subtracting coordinates, multiplying or dividing by a scalar, and distances.

Notice the three primary axes on the cube grid, and how they correspond to six hex grid *diagonal* directions; the diagonal grid axes corresponds to a primary hex grid direction.

Because we already have algorithms for square and cube grids, using cube coordinates allows us to adapt those algorithms to hex grids. I will be using this system for most of the algorithms on the page. To use the algorithms with another coordinate system, I'll convert to cube coordinates, run the algorithm, and convert back.

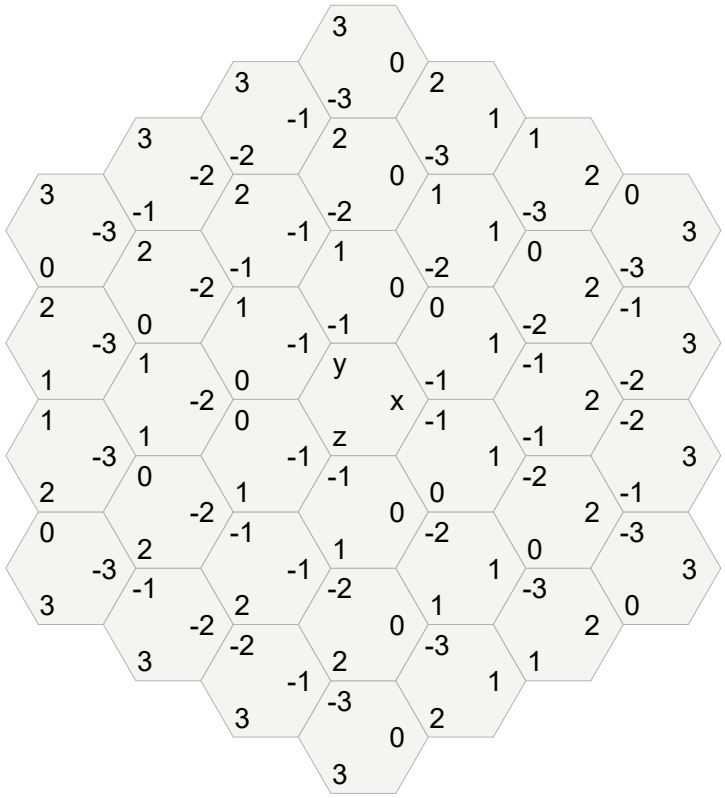


Try: **switch to hexagons** and: **switch back to cubes** .

Study how the cube coordinates work on the hex grid. Selecting the hexes will highlight the cube coordinates corresponding to the three axes.

The cube coordinates are a reasonable choice for a hex grid coordinate system. The constraint is that `x + y + z = 0` so the algorithms must preserve that. The constraint also ensures that there's a canonical coordinate for each hex.

There are *many* different valid cube hex coordinate systems. Some of them have constraints other than `x + y + z = 0`. I've shown only one of the many systems. You can also construct cube coordinates with `x-y`, `y-z`, `z-x`, and that has its



Switch to **flat topped** or back to **pointy topped** hexagons.

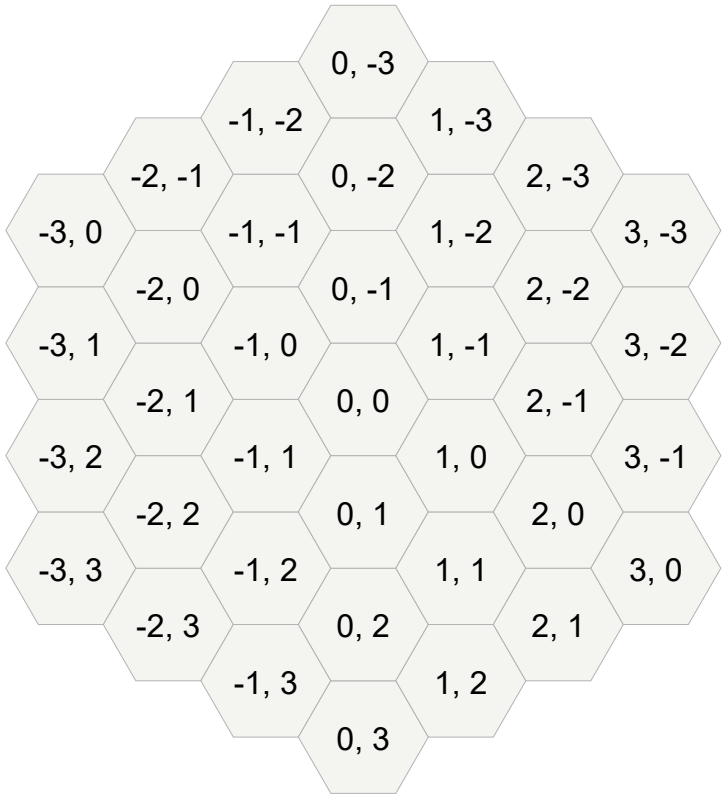
1. Each direction on the cube grid corresponds to a *line* on the hex grid. Try highlighting a hex with `z` at 0, 1, 2, 3 to see how these are related. The row is marked in blue. Try the same for `x` (green) and `y` (purple).
2. Each direction on the hex grid is a combination of *two* directions on the cube grid. For example, north on the hex grid lies between the `+y` and `-z`, so every step north involves adding 1 to `y` and subtracting 1 from `z`.

own set of interesting properties, which I don't explore here.

“But Amit!” you say, “I don’t want to store 3 numbers for coordinates. I don’t know how to store a map that way.”

Axial coordinates

The axial coordinate system, sometimes called “trapezoidal”, is built by taking two of the three coordinates from a cube coordinate system. Since we have a constraint such as `x + y + z = 0`, the third coordinate is redundant. Axial coordinates are useful for map storage and for displaying coordinates to the user. Like cube coordinates, you can use the standard *add, subtract, multiply, divide* operations from cartesian coordinates.



Switch to **flat topped** or back to **pointy topped** hexagons.

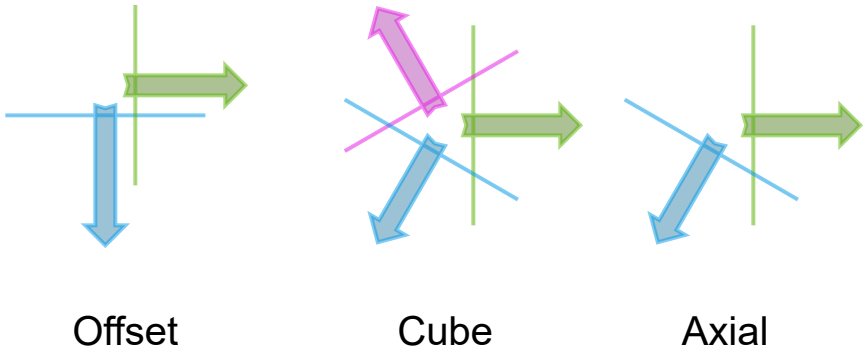
There are many cube coordinate systems, and many axial coordinate systems. I'm not going to show all of the combinations in this guide. I'm going to pick two variables, `q` for "column" and `r` as "row". In the diagrams on this page, `q` is aligned with `x` and `r` is aligned with `z` but this alignment is arbitary, as you can rotate and flip the diagrams to make many different alignments.

The advantage of this system over offset grids is that the algorithms are cleaner. The disadvantage of this system is that storing a rectangular map is a little weird;

see the map storage section for ways to handle that. There are some algorithms that are even cleaner with cube coordinates, but for those, since we have the constraint `x + y + z = 0`, we can calculate the third implicit coordinate and use that for those algorithms. In my projects, I name the axes `q`, `r`, `s` so that I have the constraint `q + r + s = 0`, and then I can calculate `s = -q - r` when needed.

Axes

Offset coordinates are the first thing people think of, because they match the standard cartesian coordinates used with square grids. Unfortunately one of the two axes has to go "against the grain", and ends up complicating things. The cube and axial systems go "with the grain" and have simpler algorithms, but slightly more complicated map storage. There's also another system, called interlaced or doubled, that I haven't explored here; some people say it's easier to work with than cube/axial.



The *axis* is the direction in which that coordinate increases. Perpendicular to the axis is a line where that coordinate stays constant. The grid diagrams above show the perpendicular line.

Coordinate conversion

It is likely that you will use axial or offset coordinates in your project, but many algorithms are simpler to express in cube coordinates. Therefore you need to be able to convert back and forth.

Axial coordinates

Axial coordinates are closely connected to cube coordinates. Axial discards the third coordinate and cube calculates the third coordinate from the other two:

```
function cube_to_axial(cube):
    var q = cube.x
    var r = cube.z
    return Hex(q, r)

function axial_to_cube(hex):
    var x = hex.q
    var z = hex.r
    var y = -x-z
    return Cube(x, y, z)
```

Offset coordinates

Determine which type of offset system you use; ***-r** are pointy top; ***-q** are flat top:

- ○ **odd-r** shoves odd rows by +½ column
- ○ **even-r** shoves even rows by +½ column
- ◉ **odd-q** shoves odd columns by +½ row
- ○ **even-q** shoves even columns by +½ row

```
function cube_to_oddq(cube):
      col = cube.x
      row = cube.z + (cube.x - (cube.x&1)) / 2
      return Hex(col, row)

function oddq_to_cube(hex):
      x = hex.col
      z = hex.row - (hex.col - (hex.col&1)) / 2
      y = -x-z
      return Cube(x, y, z)
```

Implementation note: I use `a&1` (bitwise and) instead of `a%2` (modulo) to detect whether something is even (0) or odd (1), because it works with negative numbers too. See a longer explanation on my implementation notes page.

Neighbors

Given a hex, which 6 hexes are neighboring it? As you might expect, the answer is simplest with cube coordinates, still pretty simple with axial coordinates, and slightly trickier with offset coordinates. We might also want to calculate the 6 "diagonal" hexes.

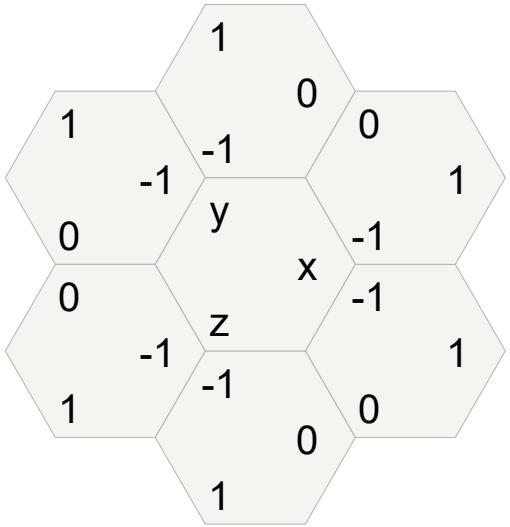
Cube coordinates

Moving one space in hex coordinates involves changing one of the 3 cube coordinates by +1 and changing another one by -1 (the sum must remain 0). There are 3 possible coordinates to change by +1, and 2 remaining that could be changed by -1. This results in 6 possible changes. Each corresponds to one of the hexagonal directions. The simplest and fastest approach is to precompute the permutations and put them into a table of `Cube(dx, dy, dz)` at compile time:

```
var cube_directions = [
   Cube(+1, -1,  0), Cube(+1,  0, -1), Cube( 0, +1, -1),
   Cube(-1, +1,  0), Cube(-1,  0, +1), Cube( 0, -1, +1)
]

function cube_direction(direction):
    return cube_directions[direction]

function cube_neighbor(cube, direction):
    return cube_add(cube, cube_direction(direction))
```



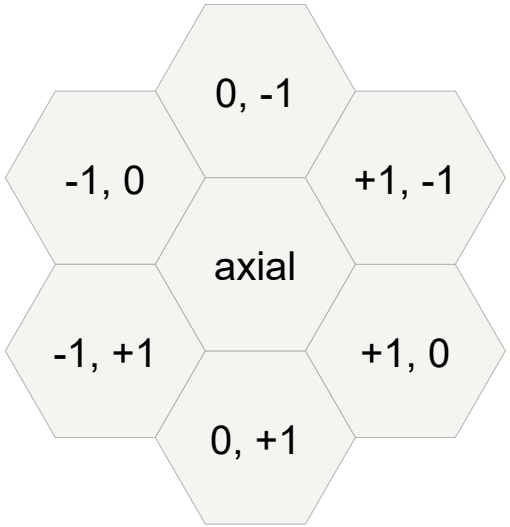
Axial coordinates

As before, we'll use the cube system as a starting point. Take the table of `Cube(dx, dy, dz)` and convert into a table of `Hex(dq, dr)`:

```
var axial_directions = [
   Hex(+1,  0), Hex(+1, -1), Hex( 0, -1),
   Hex(-1,  0), Hex(-1, +1), Hex( 0, +1)
]

function hex_direction(direction):
    return axial_directions[direction]

function hex_neighbor(hex, direction):
    var dir = hex_direction(direction)
    return Hex(hex.q + dir.q, hex.r + dir.r)
```



Offset coordinates

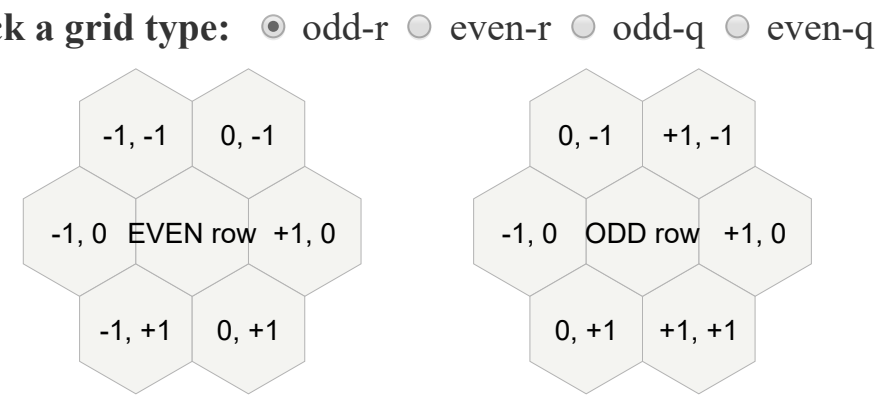
With offset coordinates, the change depends on where in the grid we are. If we're on an offset column/row then the rule is different than if we're on a non-offset column/row.

As above, we'll build a table of the numbers we need to add to `col` and `row`. However this time there will be two arrays, one for odd columns/rows and one for even columns/rows. Look at `(1,1)` on the grid map above and see how `col` and `row` change as you move in each of the six directions. Then do this again for `(2,2)`. The tables and code are different for each of the four offset grid types, so **pick a grid type** to see the corresponding code.

```
var oddr_directions = [
   [ Hex(+1,  0), Hex( 0, -1), Hex(-1, -1),
```

```
      Hex(-1,  0), Hex(-1, +1), Hex( 0, +1) ],
    [ Hex(+1,  0), Hex(+1, -1), Hex( 0, -1),
      Hex(-1,  0), Hex( 0, +1), Hex(+1, +1) ]
]

function oddr_offset_neighbor(hex, direction):
    var parity = hex.row & 1
    var dir = oddr_directions[parity][direction]
    return Hex(hex.col + dir.col, hex.row + dir.row)
```



Using the above lookup tables is the easiest way to to calculate neighbors. It's also possible to derive these numbers, for those of you who are curious.

Diagonals

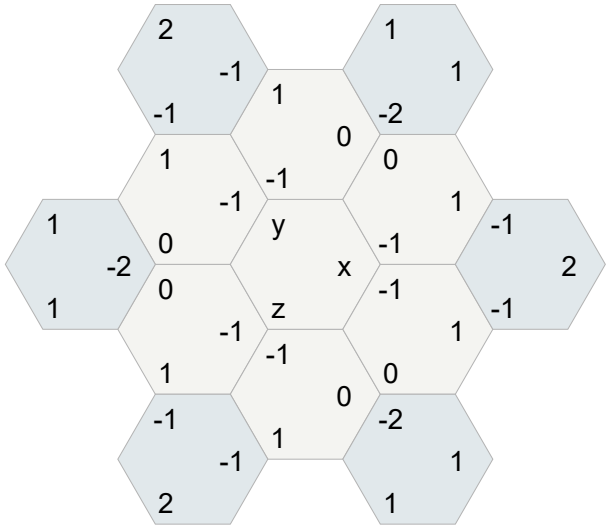
Moving to a "diagonal" space in hex coordinates changes one of the 3 cube coordinates by ±2 and the other two by ∓1 (the sum must remain 0).

```
var cube_diagonals = [
    Cube(+2, -1, -1), Cube(+1, +1, -2), Cube(-1, +2, -1),
    Cube(-2, +1, +1), Cube(-1, -1, +2), Cube(+1, -2, +1)
```

```
]

function cube_diagonal_neighbor(cube, direction):
    return cube_add(cube, cube_diagonals[direction])
```

As before, you can convert these into axial by dropping one of the three coordinates, or convert to offset by precalculating the results.



Distances

Cube coordinates

In the cube coordinate system, each hexagon is a cube in 3d space. Adjacent hexagons are distance 1 apart in the hex grid but distance 2 apart in the cube grid.

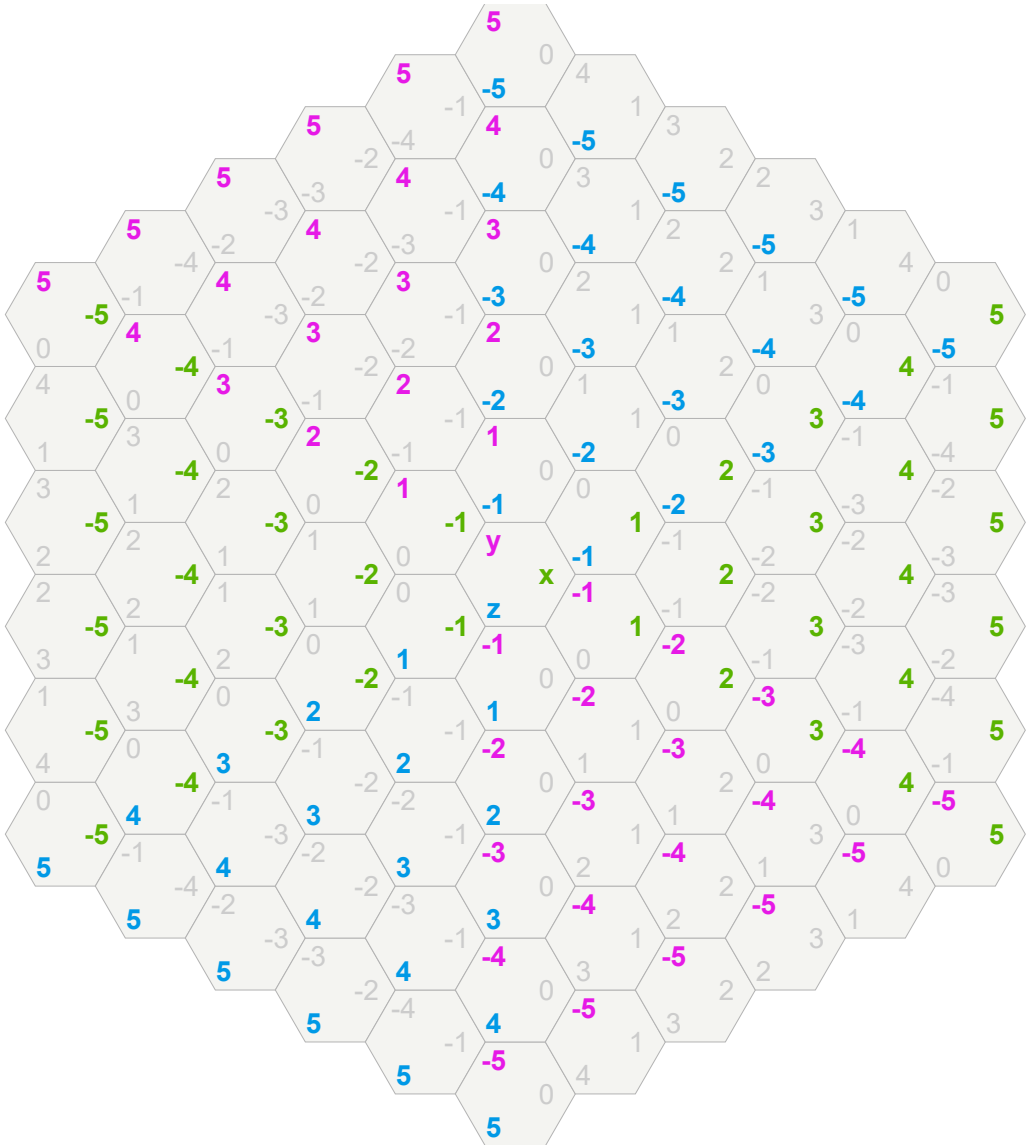
This makes distances simple. In a square grid, Manhattan distances are `abs(dx) + abs(dy)`. In a cube grid, Manhattan distances are `abs(dx) + abs(dy) + abs(dz)`. The distance on a hex grid is half that:

```
function cube_distance(a, b):
    return (abs(a.x - b.x) + abs(a.y - b.y) + abs(a.z - b.z)) / 2
```

An equivalent way to write this is by noting that one of the three coordinates must be the sum of the other two, then picking that one as the distance. You may prefer the “divide by two” form above, or the “max” form here, but they give the same result:

```
function cube_distance(a, b):
    return max(abs(a.x - b.x), abs(a.y - b.y), abs(a.z - b.z))
```

In the diagram, the max is highlighted. Also notice that each color indicates one of the 6 “diagonal” directions.



Axial coordinates

In the axial system, the third coordinate is implicit. Let's convert axial to cube to calculate distance:

```
function hex_distance(a, b):
    var ac = axial_to_cube(a)
    var bc = axial_to_cube(b)
    return cube_distance(ac, bc)
```

If your compiler inlines `axial_to_cube` and `cube_distance`, it will generate this code:

```
function hex_distance(a, b):
    return (abs(a.q - b.q)
          + abs(a.q + a.r - b.q - b.r)
          + abs(a.r - b.r)) / 2
```

There are lots of different ways to write hex distance in axial coordinates, but no matter which way you write it, *axial hex distance is derived from the Mahattan distance on cubes*. For example, the "difference of differences" described here results from writing `a.q + a.r - b.q - b.r` as `a.q - b.q + a.r - b.r`, and using "max" form instead of the "divide by two" form of `cube_distance`. They're all equivalent once you see the connection to cube coordinates.

Offset coordinates

As with axial coordinates, we'll convert offset coordinates to cube coordinates, then use cube distance.

```
function offset_distance(a, b):
    var ac = offset_to_cube(a)
```

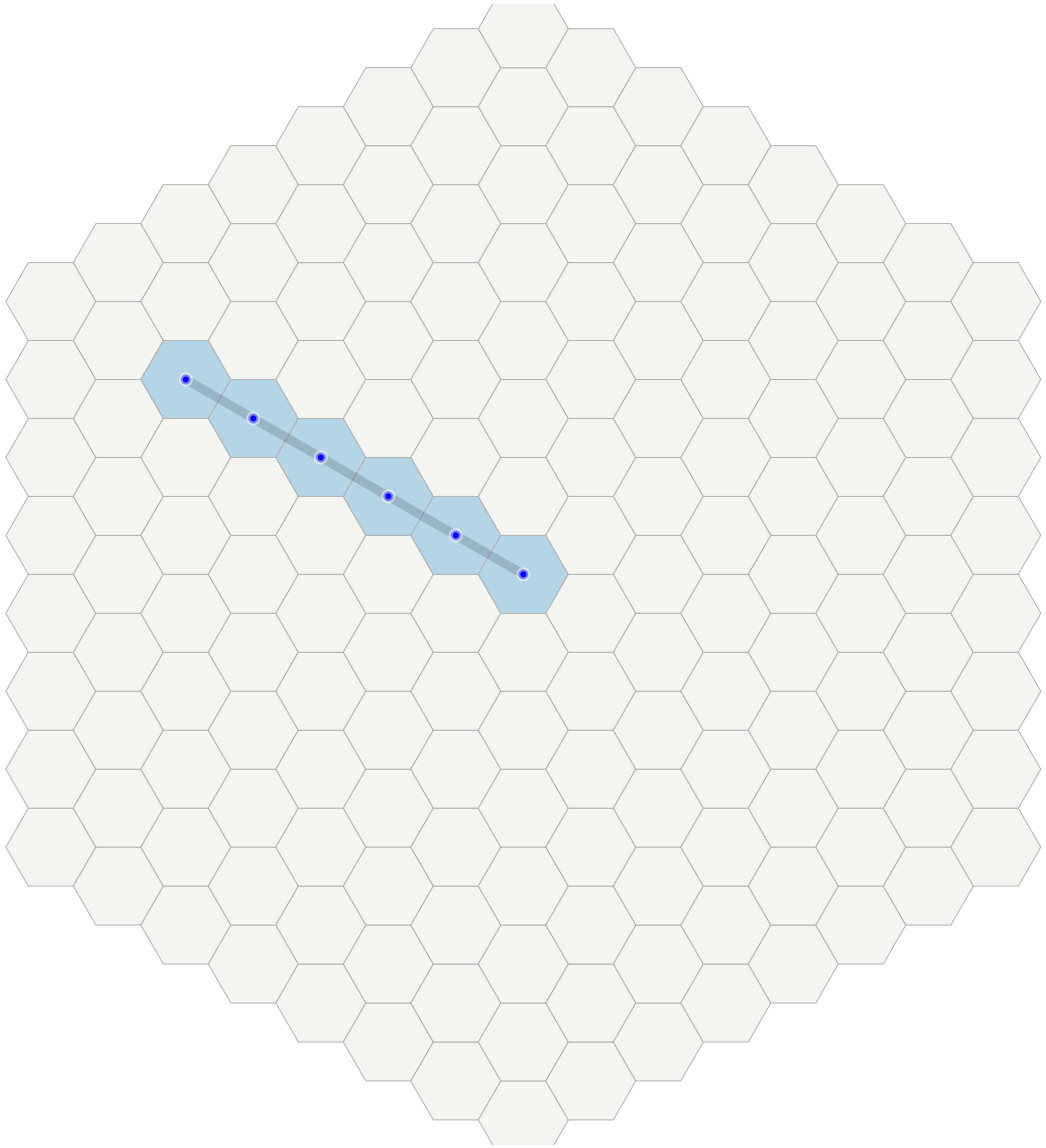
```
    var bc = offset_to_cube(b)
    return cube_distance(ac, bc)
```

We'll use the same pattern for many of the algorithms: convert hex to cube, run the cube version of the algorithm, and convert any cube results back to hex coordinates (whether axial or offset).

Line drawing

How do we draw a line from one hex to another? I use linear interpolation for line drawing. Evenly *sample* the line at `N+1` points, and figure out which hexes those samples are in.

Putting these together to draw a line from `A` to `B`:



1. First we calculate `N` to be the hex distance between the endpoints.
2. Then evenly sample `N+1` points between point `A` and point `B`. Using linear interpolation, each point will be `A + (B - A) * 1.0/N * i`, for values of `i` from `0` to `N`, inclusive. In the diagram these sample points are the dark blue

dots. This results in floating point coordinates.

3. Convert each sample point (float) back into a hex (int). The algorithm is called cube_round.

```
function lerp(a, b, t): // for floats
    return a + (b - a) * t

function cube_lerp(a, b, t): // for hexes
    return Cube(lerp(a.x, b.x, t),
                lerp(a.y, b.y, t),
                lerp(a.z, b.z, t))

function cube_linedraw(a, b):
    var N = cube_distance(a, b)
    var results = []
    for each 0 ≤ i ≤ N:
        results.append(cube_round(cube_lerp(a, b, 1.0/N * i)))
    return results
```

More notes:

- There are times when `cube_lerp` will return a point that's just on the edge between two hexes. Then `cube_round` will push it one way or the other. The *lines will look better* if it's always pushed in the same direction. You can do this by adding an "epsilon" hex `Cube(1e-6, 1e-6, -2e-6)` to one or both of the endpoints before starting the loop. This will "nudge" the line in one direction to avoid landing on edge boundaries.

- The DDA Algorithm on square grids sets `N` to the max of the distance along each axis. We do the same in cube space, which happens to be the same as the hex grid distance.
- The `cube_lerp` function needs to return a cube with float coordinates. If you're working in a statically typed language, you won't be able to use the `Cube` type but instead could define `FloatCube`, or inline the function into the line drawing code if you want to avoid defining another type.
- You can optimize the code by inlining `cube_lerp`, and then calculating `B.x-A.x`, `B.x-A.y`, and `1.0/N` outside the loop. Multiplication can be turned into repeated addition. You'll end up with something like the DDA algorithm.
- I use axial or cube coordinates for line drawing, but if you want something for offset coordinates, take a look at this article.
- There are many variants of line drawing. Sometimes you'll want "super cover". Someone sent me hex super cover line drawing code but I haven't studied it yet.

Movement Range

Coordinate range

Given a hex `center` and a range N, which hexes are within N steps from it?

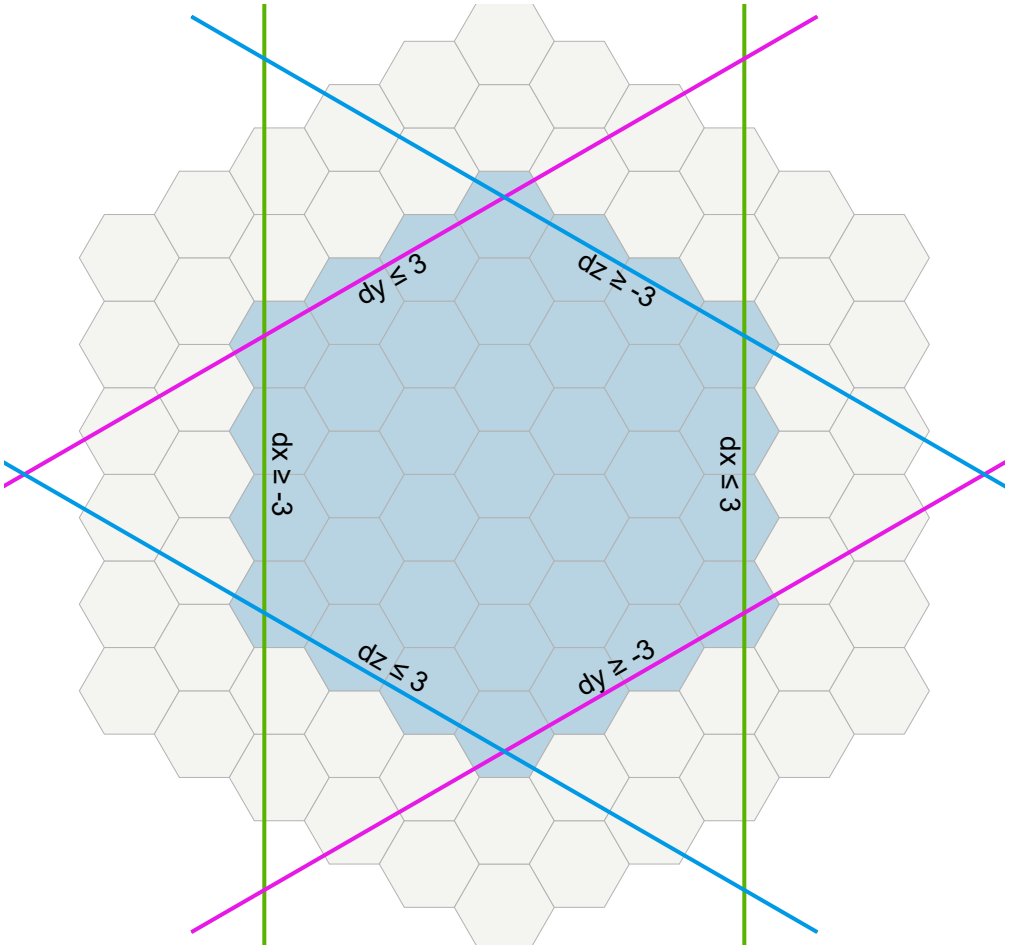
We can work backwards from the hex distance formula, `distance = max(abs(dx), abs(dy), abs(dz))`. To find all hexes within N steps, we need `max(abs(dx), abs(dy), abs(dz)) ≤ N`. This means we need all three of: `abs(dx) ≤ N` and `abs(dy) ≤ N` and `abs(dz) ≤ N`. Removing absolute value, we get `-N ≤ dx ≤ N` and `-N ≤ dy ≤ N` and `-N ≤ dz ≤ N`. In code it's a nested loop:

```
var results = []
for each -N ≤ dx ≤ N:
    for each -N ≤ dy ≤ N:
        for each -N ≤ dz ≤ N:
            if dx + dy + dz = 0:
                results.append(cube_add(center, Cube(dx, dy, dz)))
```

This loop will work but it's somewhat inefficient. Of all the values of `dz` we loop over, only one of them actually satisfies the `dx + dy + dz = 0` constraint on cubes. Instead, let's directly calculate the value of `dz` that satisfies the constraint:

```
var results = []
for each -N ≤ dx ≤ N:
    for each max(-N, -dx-N) ≤ dy ≤ min(N, -dx+N):
        var dz = -dx-dy
        results.append(cube_add(center, Cube(dx, dy, dz)))
```

This loop iterates over exactly the needed coordinates. In the diagram, each range is a pair of lines. Each line is an inequality. We pick all the hexes that satisfy all six inequalities.



Intersecting ranges

If you need to find hexes that are in more than one range, you can intersect the ranges before generating a list of hexes.

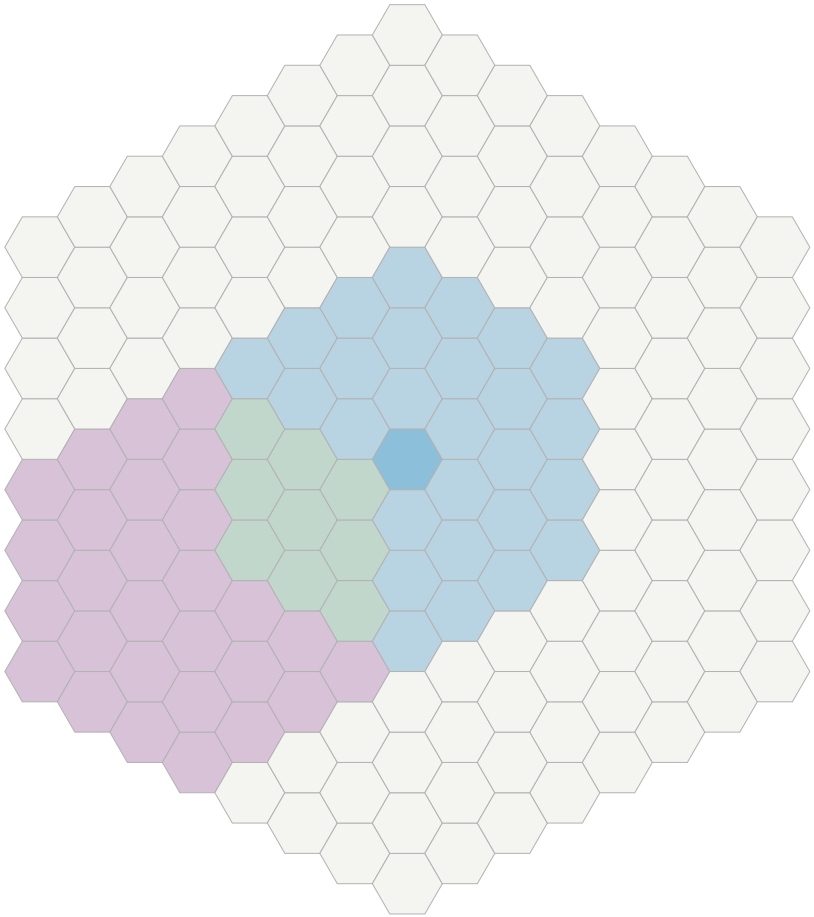
You can either think of this problem algebraically or geometrically. Algebraically, each region is expressed as inequality constraints of the form `-N ≤ dx ≤ N`, and we're going to solve for the intersection of those constraints. Geometrically, each

region is a cube in 3D space, and we're going to intersect two cubes in 3D space to form a cuboid in 3D space, then project back to the x + y + z = 0 plane to get hexes. I'm going to solve it algebraically:

First, we rewrite constraint $-N \le dx \le N$ into a more general form, $x_{min} \le x \le x_{max}$, and set $x_{min} = center.x - N$ and $x_{max} = center.x + N$. We'll do the same for y and z, and end up with this generalization of the code from the previous section:

```
var results = []
for each x_min ≤ x ≤ x_max:
    for each max(y_min, -x-z_max) ≤ y ≤ min(y_max, -x-z_min):
        var z = -x-y
        results.append(Cube(x, y, z))
```

The intersection of two ranges $a \le x \le b$ and $c \le x \le d$ is $max(a, c) \le x \le min(b, d)$. Since a hex region is expressed as ranges over x, y, z, we can separately intersect each of the x, y, z ranges then use the nested loop to generate a list of hexes in the intersection. For one hex region we set $x_{min} = H.x - N$ and $x_{max} = H.x + N$ and likewise for y and z. For intersecting two hex regions we set $x_{min} = max(H_1.x - N, H_2.x - N)$ and $x_{max} = min(H_1.x + N, H_2.x + N)$, and likewise for y and z. The same pattern works for intersecting three or more regions.



Obstacles

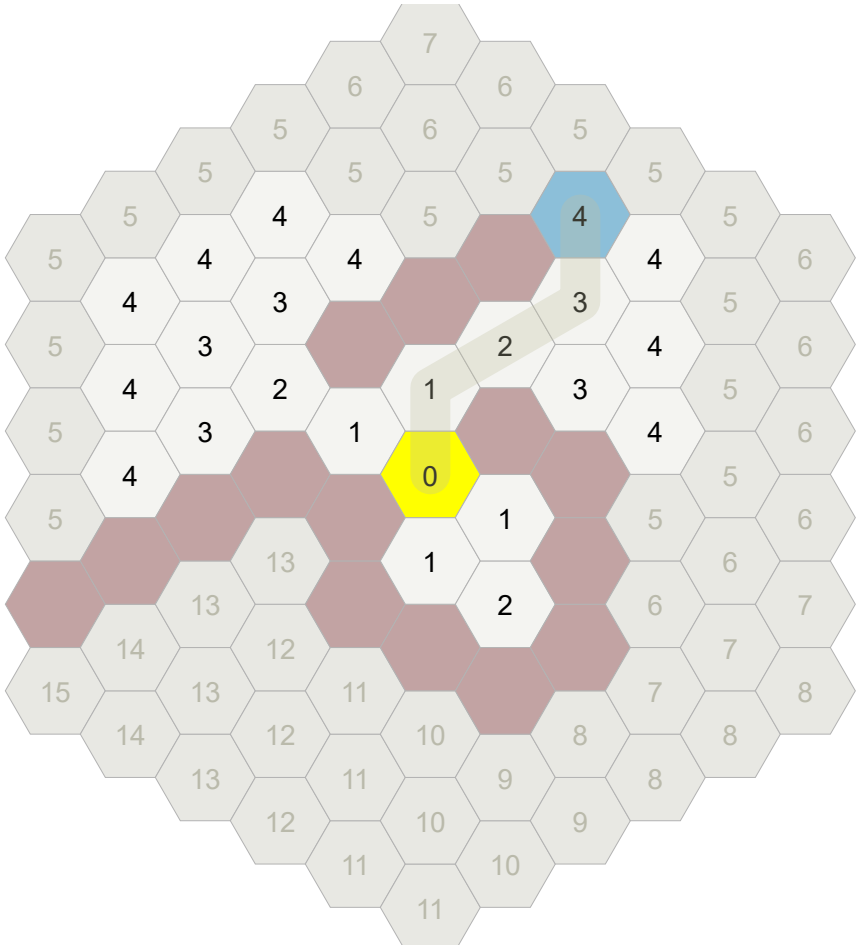
If there are obstacles, the simplest thing to do is a distance-limited flood fill (breadth first search). In this diagram, the limit is set to 4 moves. In the code, `fringes[k]` is an array of all hexes that can be reached in `k` steps. Each time through the main loop, we expand level `k-1` into level `k`.

```
function cube_reachable(start, movement):
    var visited = set()
```

```
    add start to visited
    var fringes = []
    fringes.append([start])

    for each 1 < k ≤ movement:
        fringes.append([])
        for each cube in fringes[k-1]:
            for each 0 ≤ dir < 6:
                var neighbor = cube_neighbor(cube, dir)
                if neighbor not in visited, not blocked:
                    add neighbor to visited
                    fringes[k].append(neighbor)

    return visited
```



Limit `movement = 4` :

Rotation

Given a hex vector (difference between one hex and another), we might want to rotate it to point to a different hex. This is simple with cube coordinates if we

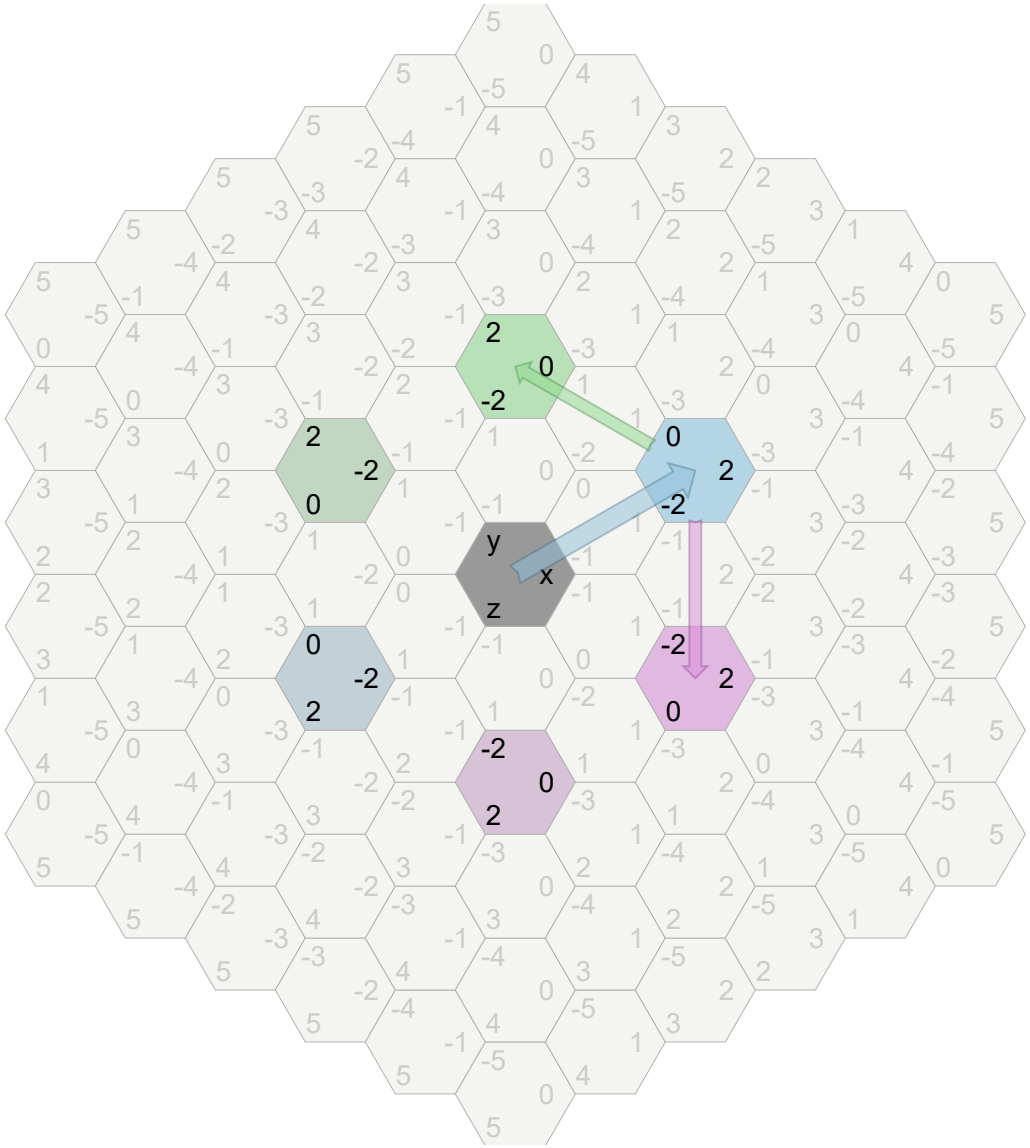
stick with rotations of 1/6th of a circle.

A rotation 60° right shoves each coordinate one slot to the right:

```
        [ x,  y,  z]
to  [-z, -x, -y]
```

A rotation 60° left shoves each coordinate one slot to the left:

```
      [ x,  y,  z]
to       [-y, -z, -x]
```



As you play with diagram, notice that each 60° rotation *flips* the signs and also physically "rotates" the coordinates. After a 120° rotation the signs are flipped back to where they were. A 180° rotation flips the signs but the coordinates have rotated back to where they originally were.

Here's the full recipe for rotating a position P around a center position C to result in a new position R:

1. Convert positions P and C to cube coordinates.
2. Calculate a vector by subtracting the center: `P_from_C = P - C = Cube(P.x - C.x, P.y - C.y, P.z - C.z)`.
3. Rotate the vector `P_from_C` as described above, and call the resulting vector `R_from_C`.
4. Convert the vector back to a position by adding the center: `R = R_from_C + C = Cube(R_from_C.x + C.x, R_from_C.y + C.y, R_from_C.z + C.z)`.
5. Convert the cube position R back to to your preferred coordinate system.

It's several conversion steps but each step is simple. You can shortcut some of these steps by defining rotation directly on axial coordinates, but hex vectors don't work for offset coordinates and I don't know a shortcut for offset coordinates. Also see this stackexchange discussion for other ways to calculate rotation.

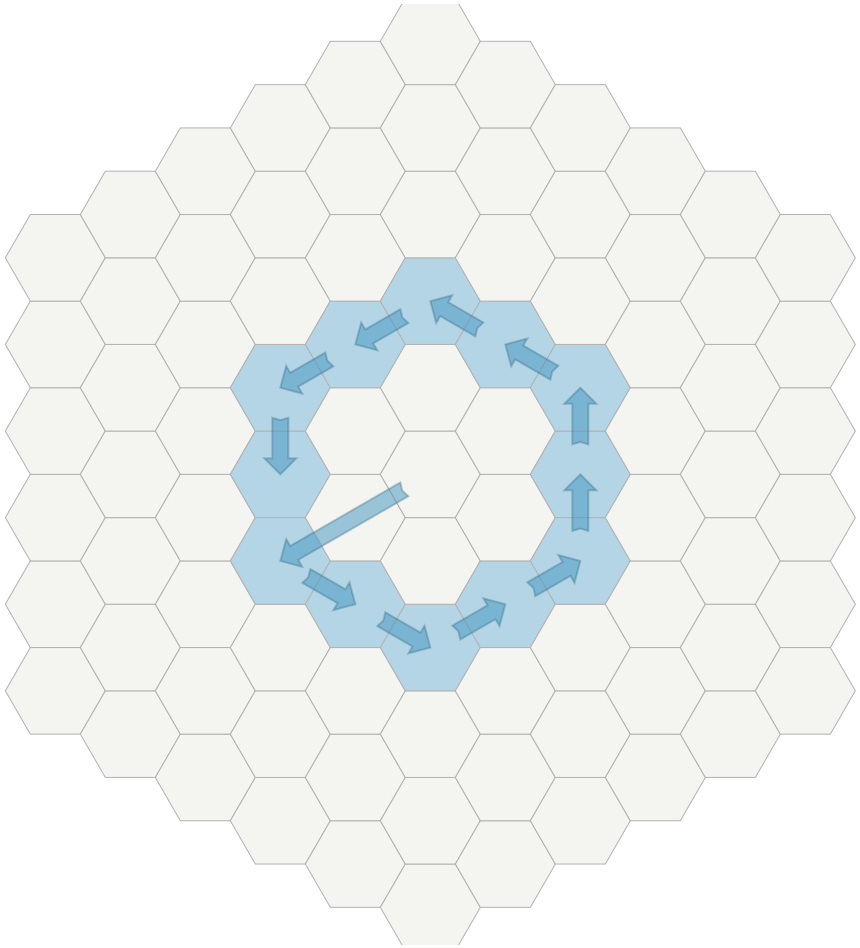
Rings

Single ring

To find out whether a given hex is on a ring of a given `radius`, calculate the distance from that hex to the center and see if it's `radius`. To get a list of all such hexes, take `radius` steps away from the center, then follow the rotated vectors in a path around the ring.

```
function cube_ring(center, radius):
    var results = []
    # this code doesn't work for radius == 0; can you see why?
    var cube = cube_add(center,
                        cube_scale(cube_direction(4), radius))
    for each 0 ≤ i < 6:
        for each 0 ≤ j < radius:
            results.append(cube)
            cube = cube_neighbor(cube, i)
    return results
```

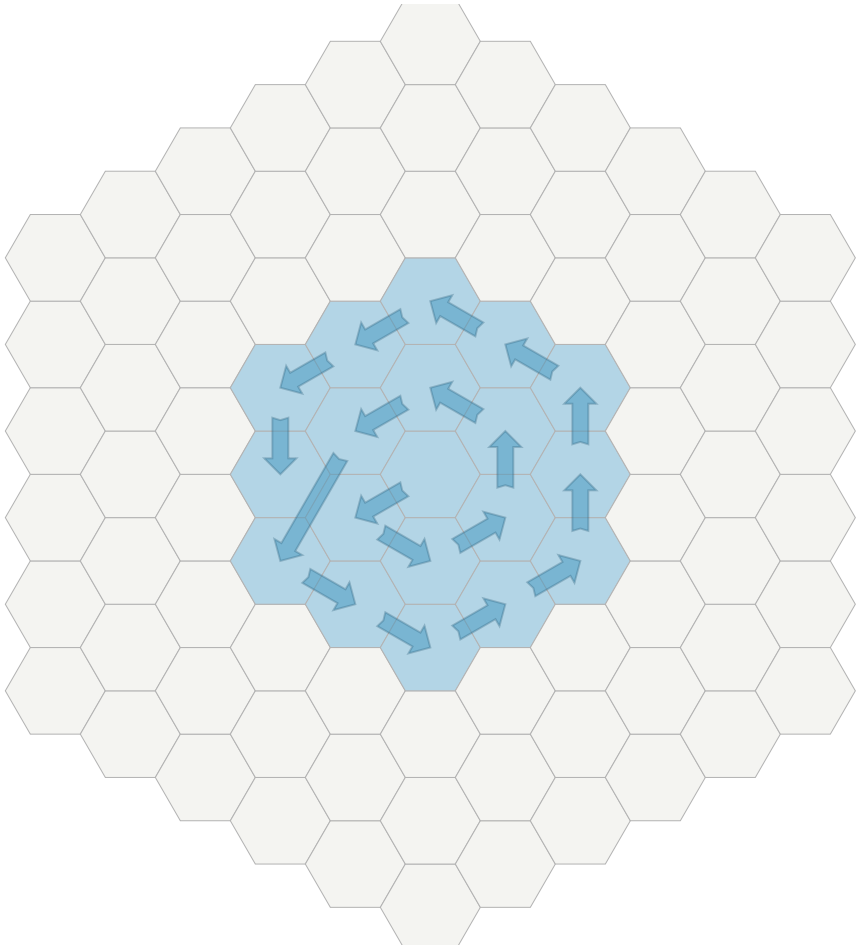
In this code, `cube` starts out on the ring, shown by the large arrow from the center to the corner in the diagram. I chose corner 4 to start with because it lines up the way my direction numbers work but you may need a different starting corner. At each step of the inner loop, `cube` moves one hex along the ring. After `6 * radius` steps it ends up back where it started.



Spiral rings

Traversing the rings one by one in a spiral pattern, we can fill in the interior:

```
function cube_spiral(center, radius):
    var results = [center]
    for each 1 ≤ k ≤ radius:
        results = results + cube_ring(center, k)
    return results
```



The area of the larger hexagon will be the sum of the circumferences, plus 1 for the center; use this formula to help you calculate the area.

Visiting the hexes this way can also be used to calculate movement range.

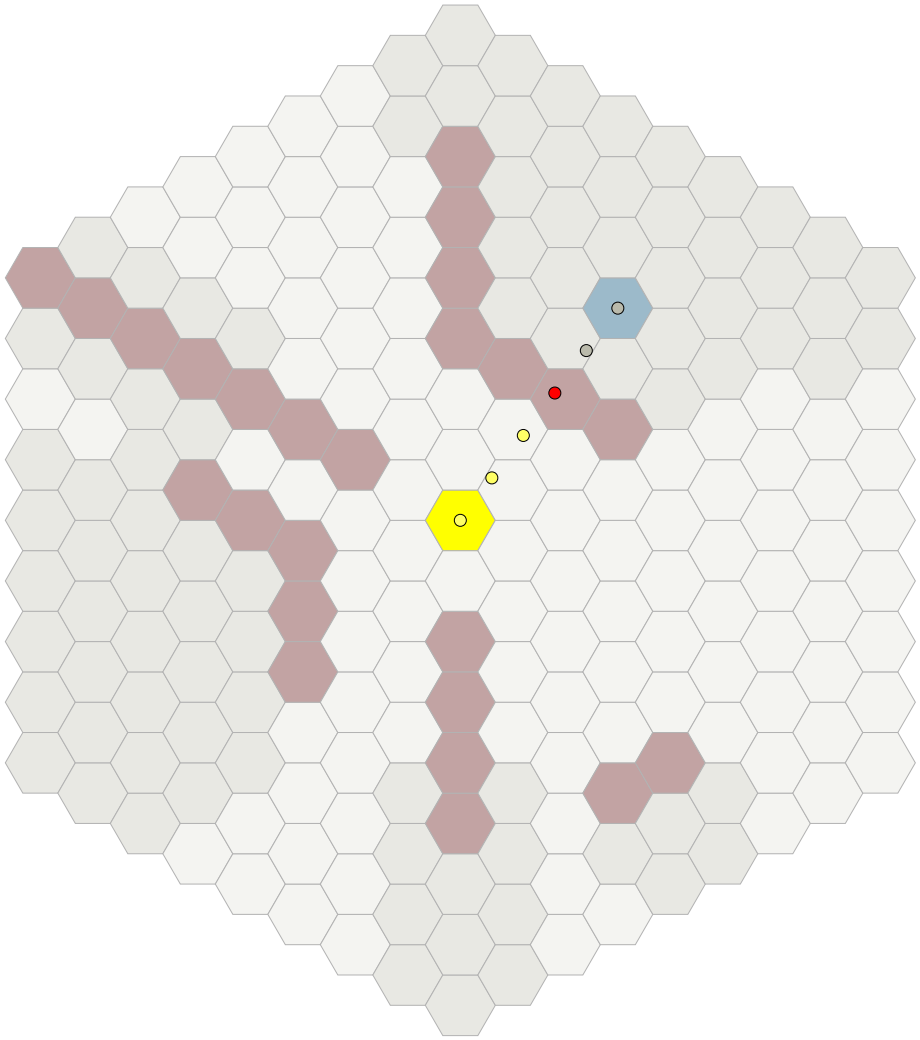
Field of view

Given a location and a distance, what is visible from that location, not blocked by obstacles? The simplest way to do this is to draw a line to every hex that's in range. If the line doesn't hit any walls, then you can see the hex. Mouse over a hex to see the line being drawn to that hex, and which walls it hits.

This algorithm can be slow for large areas but it's so easy to implement that it's what I recommend starting with.

There are many different ways to define what's "visible". Do you want to be able to see the center of the other hex from the center of the starting hex? Do you want to see any part of the other hex from the center of the starting point? Maybe any part of the other hex from any part of the starting point? Are there obstacles that occupy less than a complete hex? Field of view turns out to be trickier and more varied than it might seem at first. Start with the simplest algorithm, but expect that it may not compute exactly the answer you want for your project. There are even situations where the simple algorithm produces results that are illogical.

Clark Verbrugge's guide describes a "start at center and move outwards" algorithm to calculate field of view. Also see the Duelo project, which has an an online demo of directional field of view and code on Github. Also see my article on 2d visibility calculation for an algorithm that works on polygons, including hexagons. For grids, the roguelike community has a nice set of algorithms for



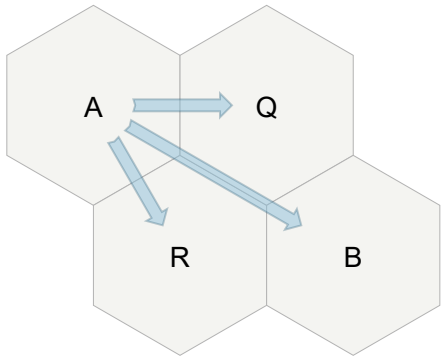
square grids (see this and this and this); some of them might be adapted for hex grids.

Hex to pixel

For hex to pixel, it's useful to review the size and spacing diagram at the top of the page.

Axial coordinates

For axial coordinates, the way to think about hex to pixel conversion is to look at the *basis vectors*. In the diagram, the arrow A→Q is the q basis vector and A→R is the r basis vector. The pixel coordinate is `q_basis * q + r_basis * r`. For example, B at (1, 1) is the sum of the q and r basis vectors.



If you have a matrix library, this is a simple matrix multiplication; however I'll write the code out without matrices here. For the `x = q, z = r` axial grid I use in this guide, the conversion is:

```
function hex_to_pixel(hex):
    x = size * sqrt(3) * (hex.q + hex.r/2)
    y = size * 3/2 * hex.r
    return Point(x, y)
```

Axial coordinates: ○ flat top or ◉ pointy top

The matrix approach will come in handy later when we want to convert pixel coordinates back to hex coordinates. All we will have to do is invert the matrix. For cube coordinates, you can either use cube basis vectors (x, y, z), or you can convert to axial first and then use axial basis vectors (q, r).

Offset coordinates

For offset coordinates, we need to offset either the column or row number (it will no longer be an integer).

```
function oddr_offset_to_pixel(hex):
    x = size * sqrt(3) * (hex.col + 0.5 * (hex.row&1))
    y = size * 3/2 * hex.row
    return Point(x, y)
```

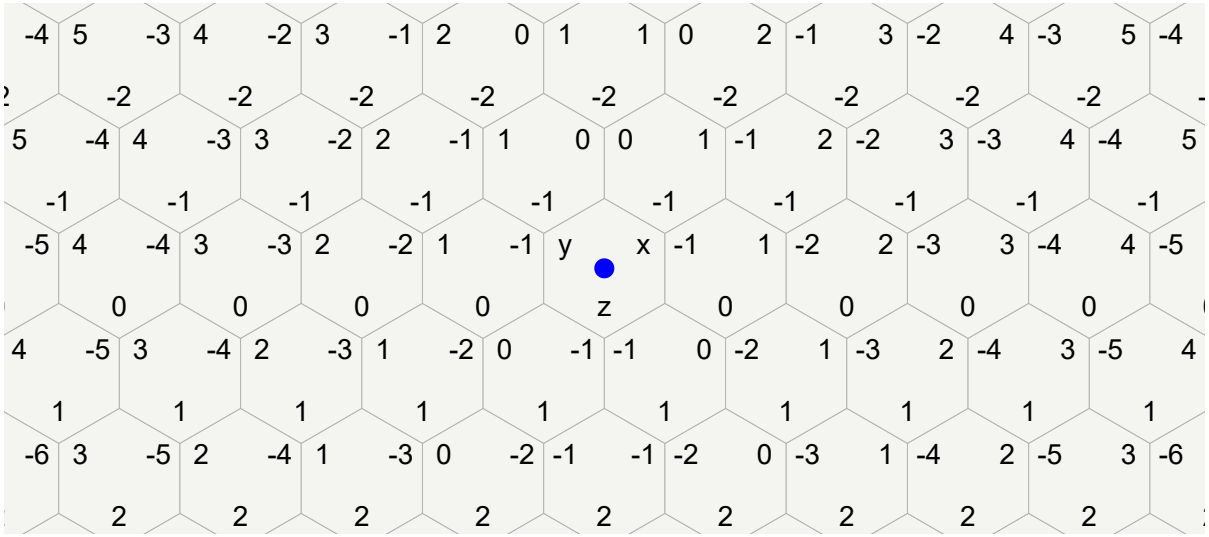
Offset coordinates: ◉ odd-r ○ even-r ○ odd-q ○ even-q

Unfortunately offset coordinates don't have basis vectors that we can use with a matrix. This is one reason pixel-to-hex conversions are harder with offset coordinates.

Another approach is to convert the offset coordinates into cube/axial coordinates, then use the cube/axial to pixel conversion. By inlining the conversion code then optimizing, it will end up being the same as above.

Pixel to Hex

One of the most common questions is, how do I take a pixel location (such as a mouse click) and convert it into a hex grid coordinate? I'll show how to do this for axial or cube coordinates. For offset coordinates, the simplest thing to do is to convert the cube to offset at the end.



1. First we *invert* the hex to pixel conversion. This will give us a *fractional* hex coordinate, shown as a small blue circle in the diagram.
2. Then we find the hex containing the fractional hex coordinate, shown as the highlighted hex in the diagram.

To convert from hex coordinates to pixel coordinates, we multiplied `q, r` by *basis vectors* to get `x, y`. You can think of this as a matrix multiply. Here's the matrix for "pointy top" hexes:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \text{size} \times \begin{bmatrix} \text{sqrt}(3) & \text{sqrt}(3)/2 \\ 0 & 3/2 \end{bmatrix} \times \begin{bmatrix} q \\ r \end{bmatrix}$$

Converting from pixel coordinates back to hex coordinates is straightforward. We can invert the matrix:

$$\begin{bmatrix} q \\ r \end{bmatrix} = \begin{bmatrix} \text{sqrt}(3)/3 & -1/3 \\ 0 & 2/3 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \div \text{size}$$

This calculation will give us fractional axial coordinates (floats) for `q` and `r`. The hex_round() function will convert the fractional axial coordinates into integer axial hex coordinates.

Here's the code for "pointy top" axial hexes:

```
function pixel_to_hex(x, y):
    q = (x * sqrt(3)/3 - y / 3) / size
    r = y * 2/3 / size
    return hex_round(Hex(q, r))
```

And here's the code for "flat top" axial hexes:

```
function pixel_to_hex(x, y):
    q = x * 2/3 / size
    r = (-x / 3 + sqrt(3)/3 * y) / size
    return hex_round(Hex(q, r))
```

That's three lines of code to convert a pixel location into an axial hex coordinate. If you use offset coordinates, use `return cube_to_{odd,even}{r,q}(cube_round(Cube(q, -q-r, r)))`.

There are many other ways to convert pixel to hex; see this page for the ones I know of.

Rounding to nearest hex

Sometimes we'll end up with a *floating-point* cube coordinate `(x, y, z)`, and we'll want to know which hex it should be in. This comes up in line drawing and pixel to hex. Converting a floating point value to an integer value is called *rounding* so I call this algorithm `cube_round`.

With cube coordinates, `x + y + z = 0`, even with floating point cube coordinates. So let's round each component to the nearest integer, `(rx, ry, rz)`. However, although `x + y + z = 0`, after rounding we do *not* have a guarantee that `rx + ry + rz = 0`. So we *reset* the component with the largest change back to what the constraint `rx + ry + rz = 0` requires. For example, if the y-change `abs(ry-y)` is larger than `abs(rx-x)` and `abs(rz-z)`, then we reset `ry = -rx-rz`. This guarantees that `rx + ry + rz = 0`. Here's the algorithm:

```
function cube_round(cube):
    var rx = round(cube.x)
    var ry = round(cube.y)
    var rz = round(cube.z)

    var x_diff = abs(rx - cube.x)
    var y_diff = abs(ry - cube.y)
    var z_diff = abs(rz - cube.z)

    if x_diff > y_diff and x_diff > z_diff:
        rx = -ry-rz
    else if y_diff > z_diff:
        ry = -rx-rz
    else:
        rz = -rx-ry

    return Cube(rx, ry, rz)
```

For non-cube coordinates, the simplest thing to do is to convert to cube coordinates, use the rounding algorithm, then convert back:

```
function hex_round(hex):
    return cube_to_axial(cube_round(axial_to_cube(hex)))
```

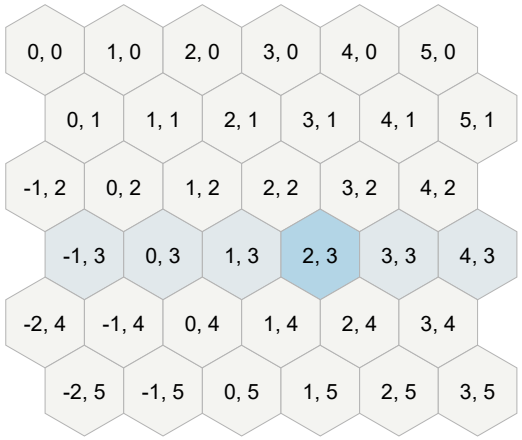
The same would work if you have `oddr`, `evenr`, `oddq`, or `evenq` instead of `axial`.

Implementation note: `cube_round` and `hex_round` take *float* coordinates instead of *int* coordinates. If you've written a Cube and Hex class, they'll work fine in

dynamically languages where you can pass in floats instead of ints, and they'll also work fine in statically typed languages with a unified number type. However, in most statically typed languages, you'll need a separate class/struct type for float coordinates, and `cube_round` will have type `FloatCube → Cube`. If you also need `hex_round`, it will be `FloatHex → Hex`, using helper function `floatcube_to_floathex` instead of `cube_to_hex`. In languages with parameterized types (C++, Haslkell, etc.) you might define `Cube<T>` where `T` is either `int` or `float`. Alternatively, you could write `cube_round` to take three floats as inputs instead of defining a new type just for this function.

Map storage in axial coordinates

One of the common complaints about the axial coordinate system is that it leads to wasted space when using a rectangular map; that's one reason to favor an offset coordinate system. However all the hex coordinate systems lead to wasted space when using a triangular or hexagonal map. We can use the same strategies for storing all of them.



0	-2, 0	-1, 0	0, 0	1, 0	2, 0	3, 0	4, 0	5, 0
0	-2, 1	-1, 1	0, 1	1, 1	2, 1	3, 1	4, 1	5, 1
-1	-2, 2	-1, 2	0, 2	1, 2	2, 2	3, 2	4, 2	5, 2
-1	-2, 3	-1, 3	0, 3	1, 3	2, 3	3, 3	4, 3	5, 3
-2	-2, 4	-1, 4	0, 4	1, 4	2, 4	3, 4	4, 4	5, 4
-2	-2, 5	-1, 5	0, 5	1, 5	2, 5	3, 5	4, 5	5, 5

Shape: ◉ rectangle ○ triangle ○ hexagon ○ rhombus

→ `array[r][q + r/2]`

Notice in the diagram that the wasted space is on the left and right sides of each row (except for rhombus maps) This gives us three strategies for storing the map:

1. **Ignore** the problem. Use a dense array with nulls or some other sentinel at the unused spaces. At most there's a factor of two for these common shapes; it may not be worth using a more complicated solution.
2. Use a **hash table** instead of dense array. This allows arbitrarily shaped maps, including ones with holes. When you want to access the hex at `q,r` you access `hash_table(hash(q,r))` instead. Encapsulate this into the getter/setter in the grid class so that the rest of the game doesn't need to know about it.
3. **Slide** the rows to the left, and use variable sized rows. In many languages a 2D array is an array of arrays; there's no reason the arrays have to be the same

length. This removes the waste on the right. In addition, if you start the arrays at the leftmost column instead of at 0, you remove the waste on the left.

To do this for arbitrary convex shaped maps, you'd keep an additional array of "first columns". When you want to access the hex at `q,r` you access `array[r][q - first_column[r]]` instead. Encapsulate this into the getter/setter in the grid class so that the rest of the game doesn't need to know about it. In the diagram `first_column` is shown on the left side of each row.

If your maps are fixed shapes, the "first columns" can be calculated on the fly instead of being stored in an array.

- For rectangle shaped maps, `first_column[r] == -floor(r/2)`, and you'd end up accessing `array[r][q + r/2]`, which turns out to be equivalent to converting the coordinates into offset grid coordinates.
- For triangle shaped maps as shown here, `first_column[r] == 0`, so you'd access `array[r][q]` — how convenient! For triangle shaped maps that are oriented the other way (not shown in the diagram), it's `array[r][q+r]`.
- For hexagon shaped maps of radius `N`, where `N = max(abs(x), abs(y), abs(z)`, we have `first_column[r] == -N - min(0, r)`. You'd access `array[r][q + N + min(0, r)]`. However, since we're starting with some

values of `r < 0`, we also have to offset the row, and use `array[r + N][q + N + min(0, r)]`.

- For rhombus shaped maps, everything matches nicely, so you can use `array[r][q]`.

Wraparound maps

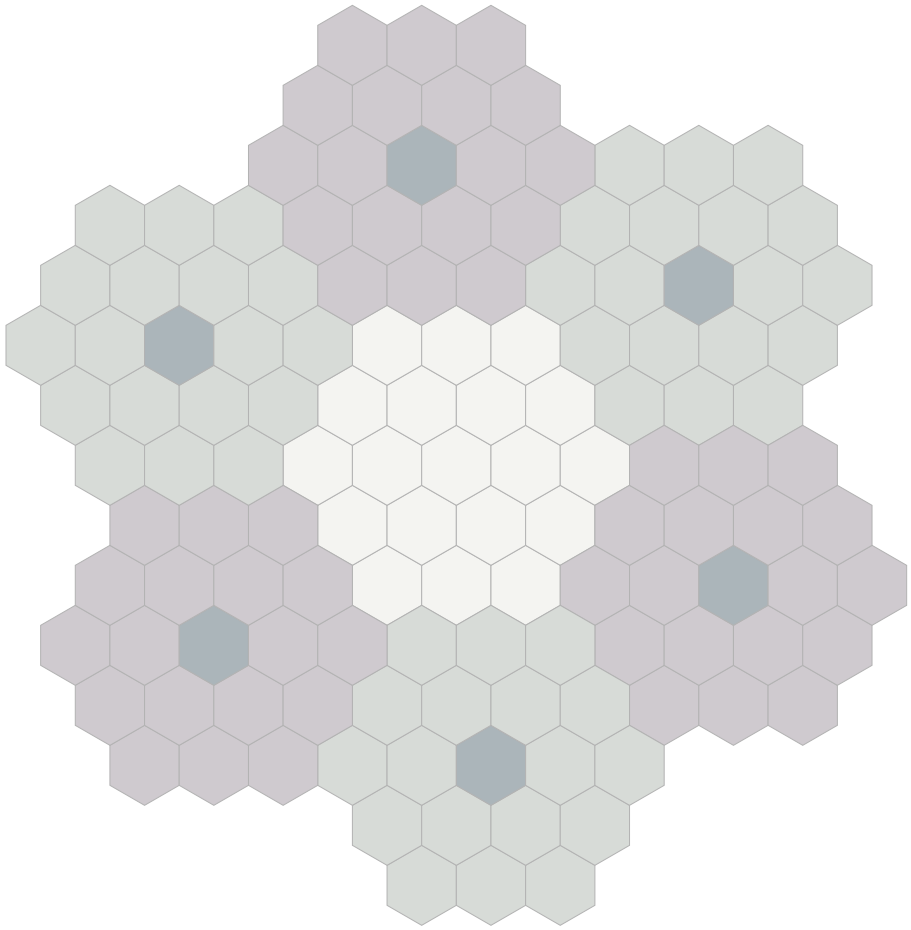
In some games you want the map to "wrap" around the edges. In a square map, you can either wrap around the x-axis only (roughly corresponding to a sphere) or both x- and y-axes (roughly corresponding to a torus). Wraparound depends on the map shape, not the tile shape. To wrap around a rectangular map is easy with offset coordinates. I'll show how to wrap around a hexagon-shaped map with cube coordinates.

Corresponding to the center of the map, there are six "mirror" centers. When you go off the map, you subtract the mirror center closest to you until you are back on the main map. In the diagram, try exiting the center map, and watch one of the mirrors enter the map on the opposite side.

The simplest implementation is to precompute the answers. Make a lookup table storing, for each hex just off the map, the corresponding cube on the other side. For each of the six mirror centers `M`, and each of the locations on the map `L`, store `mirror_table[cube_add(M, L)] = L`. Then any time you calculate a hex that's

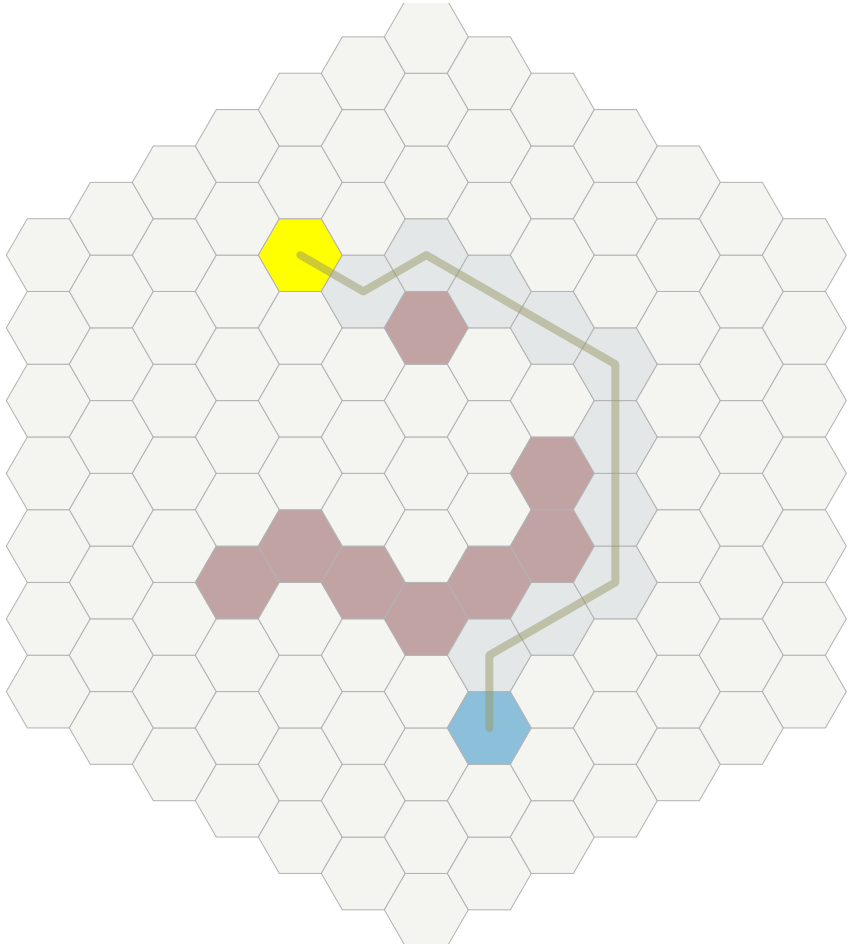
in the mirror table, replace it by the unmirrored version. See stackoverflow for another approach.

For a hexagonal shaped map with radius `N`, the mirror centers will be `Cube(2*N+1, -N, -N-1)` and its six rotations.



Pathfinding

If you're using graph-based pathfinding such as A* or Dijkstra's algorithm or Floyd-Warshall, pathfinding on hex grids isn't different from pathfinding on square grids. The explanations and code from my pathfinding tutorial will work equally well on hexagonal grids.



Mouse over a hex in the diagram to see the path to it. Click or drag to toggle walls.

1. **Neighbors**. The sample code I provide in the pathfinding tutorial calls `graph.neighbors` to get the neighbors of a location. Use the function in the neighbors section for this. Filter out the neighbors that are impassable.
2. **Heuristic**. The sample code for A* uses a `heuristic` function that gives a distance between two locations. Use the distance formula, scaled to match the movement costs. For example if your movement cost is 5 per hex, then multiply the distance by 5.

More

I have an **guide to implementing your own hex grid library**, including sample code in C++, Java, C#, Javascript, Haxe, and Python.

- In my Guide to Grids, I cover axial coordinate systems to address square, triangle, and hexagon edges and corners, and algorithms for the relationships among tiles, edges, and corners. I also show how square and hex grids are related.

- The best early guide I saw to the axial coordinate system was Clark Verbrugge's guide, written in 1996.
- The first time I saw the cube coordinate system was from Charles Fu's posting to rec.games.programmer in 1994.
- DevMag has a nice visual overview of hex math including how to represent areas such as half-planes, triangles, and quadrangles. There's a PDF version here that goes into more detail. **Highly reocmmended**! The GameLogic Grids library implements these and many other grid types in Unity.
- James McNeill has a nice visual explanation of grid transformations.
- Overview of hex coordinate types: staggered (offset), interlaced, 3d (cube), and trapezoidal (axial).
- The Rot.js library has a list of hex coordinate systems: non-orthogonal (axial), odd shift (offset), double width (interlaced), cube.
- Range for cube coordinates: given a distance, which hexagons are that distance from the given one?
- Distances on hex grids using cube coordinates, and reasons to use cube coordinates instead of offset.
- This guide explains the basics of measuring and drawing hexagons, using an offset grid.
- Convert cube hex coordinates to pixel coordinates.
- This thread explains how to generate rings.
- The HexPart system uses both hexes and rectangles to make some of the algorithms easier to work with.
- Are there pros and cons of "pointy topped" and "flat topped" hexagons?

- Line of sight in a hex grid with offset coordinates, splitting hexes into triangles
- Hexnet explains how the correspondence between hexagons and cubes goes much deeper than what I described on this page, generalizing to higher dimensions.
- I used the PDF hex grids from this page while working out some of the algorithms.
- Generalized Balanced Ternary for hex coordinates seems interesting; I haven't studied it.
- Hex-Grid Utilities is a C# library for hex grid math, with neighbors, grids, range finding, path finding, field of view. Open source, MIT license.
- The Reddit discussion and Hacker News discussion and MetaFilter discussion have more comments and links.

The code that powers this page is written in a mixture of Haxe and Javascript: Cube.hx, Hex.hx, Grid.hx, ScreenCoordinate.hx, ui.js, and cubegrid.js (for the cube/hex animation). However, if you are looking to write your own hex grid library, I suggest instead looking at my guide to implementation.

There are more things I want to do for this guide. I'm keeping a list on Trello. Do you have suggestions for things to change or add? Comment below.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

Created 11 Mar 2013 with Haxe and D3.js ; Last modified: 31 Jan 2018