Hexagonal Grids

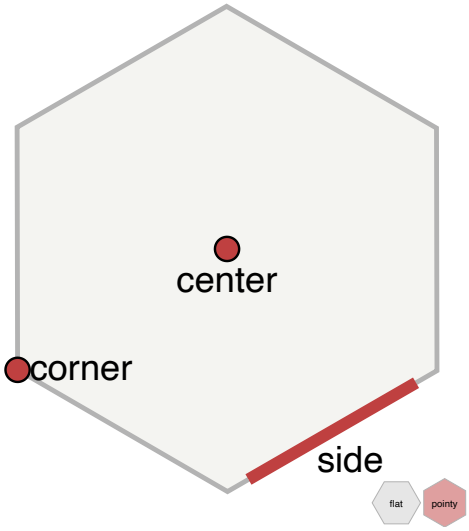
from Red Blob Games

Mar 2013, updated in Mar 2015, Apr 2018, Feb 2019, May 2020, Oct 2021

This guide will cover various ways to make hexagonal grids, the relationships between different approaches, and common formulas and algorithms. I've been collecting hex grid resources[1] for over 25 years. I wrote this guide to the most elegant approaches that lead to the simplest code, starting from the guides by Charles Fu[2] and Clark Verbrugge[3]. Most parts of this page are interactive.

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



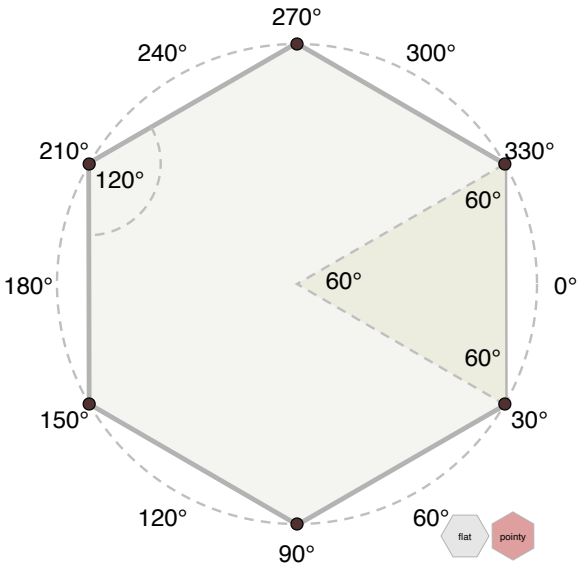
Hexagons are 6-sided polygons. *Regular* hexagons have all the sides the same length. I'll assume all the hexagons we're working with here are regular. The typical orientations for hex grids are vertical columns (**flat topped**) and horizontal rows (**pointy topped**).

Hexagons have 6 sides and 6 corners. Each side is shared by 2 hexagons. Each corner is shared by 3 hexagons. For more about centers, sides, and corners, see my article on grid parts[4] (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. Each corner is `size` units away from the `center`. In code:

```
function pointy_hex_corner(center, size, i):
    var angle_deg = 60 * i - 30°
    var angle_rad = PI / 180 * angle_deg
    return Point(center.x + size * cos(angle_rad),
                 center.y + size * sin(angle_rad))
```



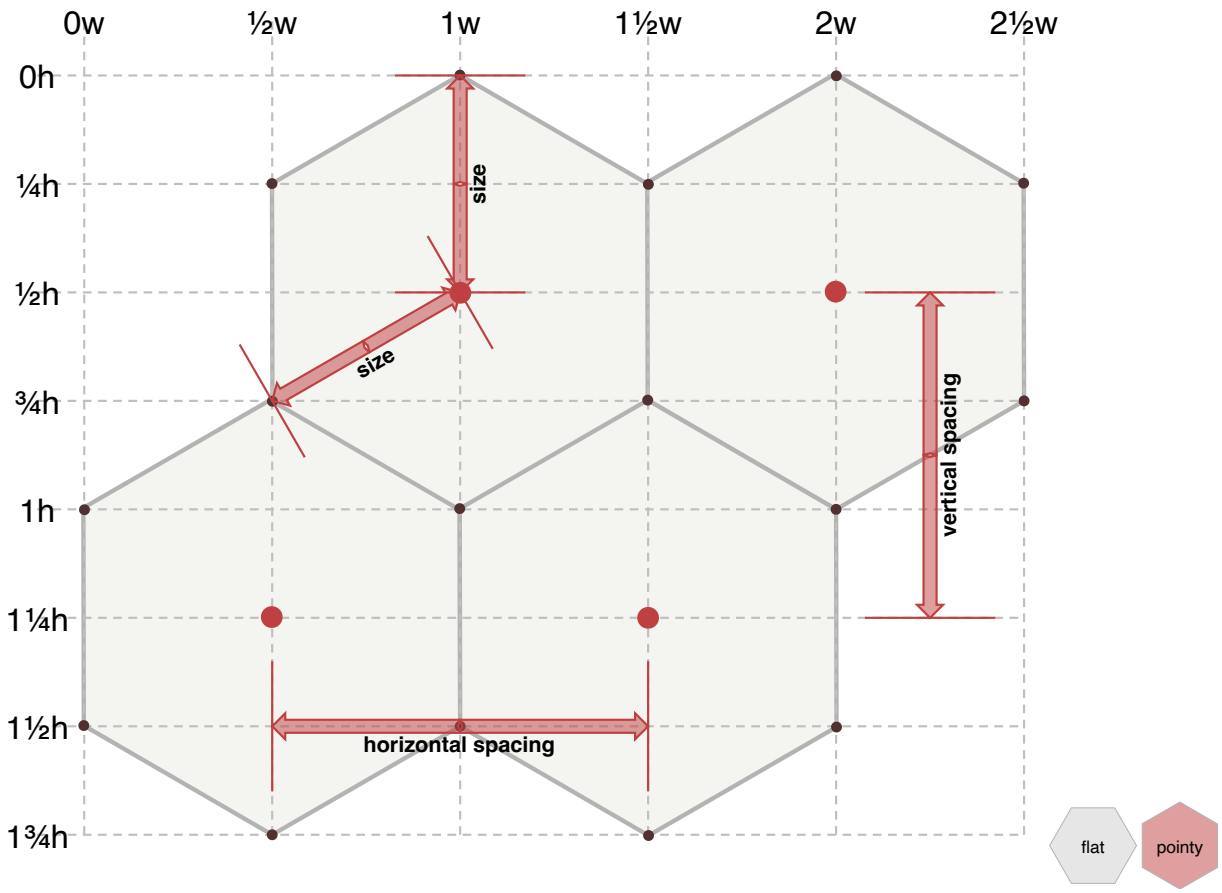
To fill a hexagon, gather the polygon vertices at `hex_corner(…, 0)` through `hex_corner(…, 5)`. To draw a hexagon outline, use those vertices, and then draw a line back to `hex_corner(…, 0)`.

The difference between the two orientations is a rotation, 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°. Note that the diagrams on this page use the y axis pointing *down* (angles increase clockwise); you may have to make some adjustments if your y axis points up (angles increase counterclockwise).

In math, the "circumradius" is the distance from the center to a corner (I call this `size`); the "inradius" is the distance from the center to the middle of an edge. The "maximal diameter" is twice the circumradius; the "minimal diameter" is twice the inradius. Wikipedia[5] has more.

Size and Spacing

Next we want to put several hexagons together. The `size` is the distance from the center to any corner. In the pointy orientation, a hexagon has width `w = sqrt(3) * size` and height `h = 2 * size`. The sqrt(3) comes from sin(60°).



The horizontal distance between adjacent hexagon centers is `w`. The vertical distance between adjacent hexagon centers is `h * 3/4`.

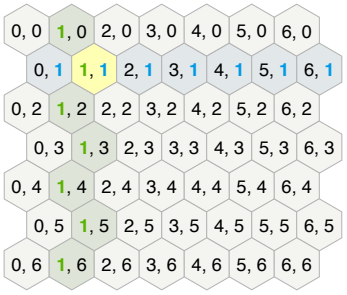
Some games use pixel art for hexagons that does not match an exactly regular polygon. The angles and spacing formulas I describe in this section won't match the sizes of your hexagons. The rest of the article, describing algorithms on hex grids, will work even if your hexagons are stretched or shrunk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

Coordinate Systems

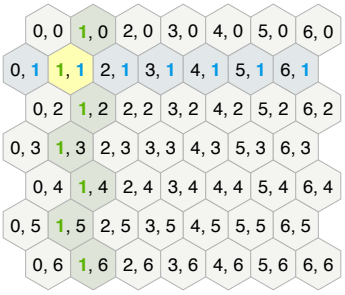
Now let's assemble hexagons into a grid. With square grids, there's one obvious way to do it. With hexagons, there are multiple approaches. I like cube coordinates for algorithms and axial or doubled for storage.

Offset coordinates

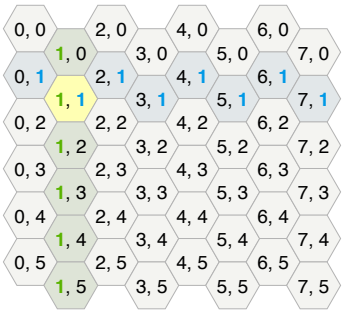
The most common approach is to offset every other column or row. Columns are named `col` (`q`). Rows are named `row` (`r`). You can either offset the odd or the even column/rows, so the horizontal and vertical hexagons each have two variants.



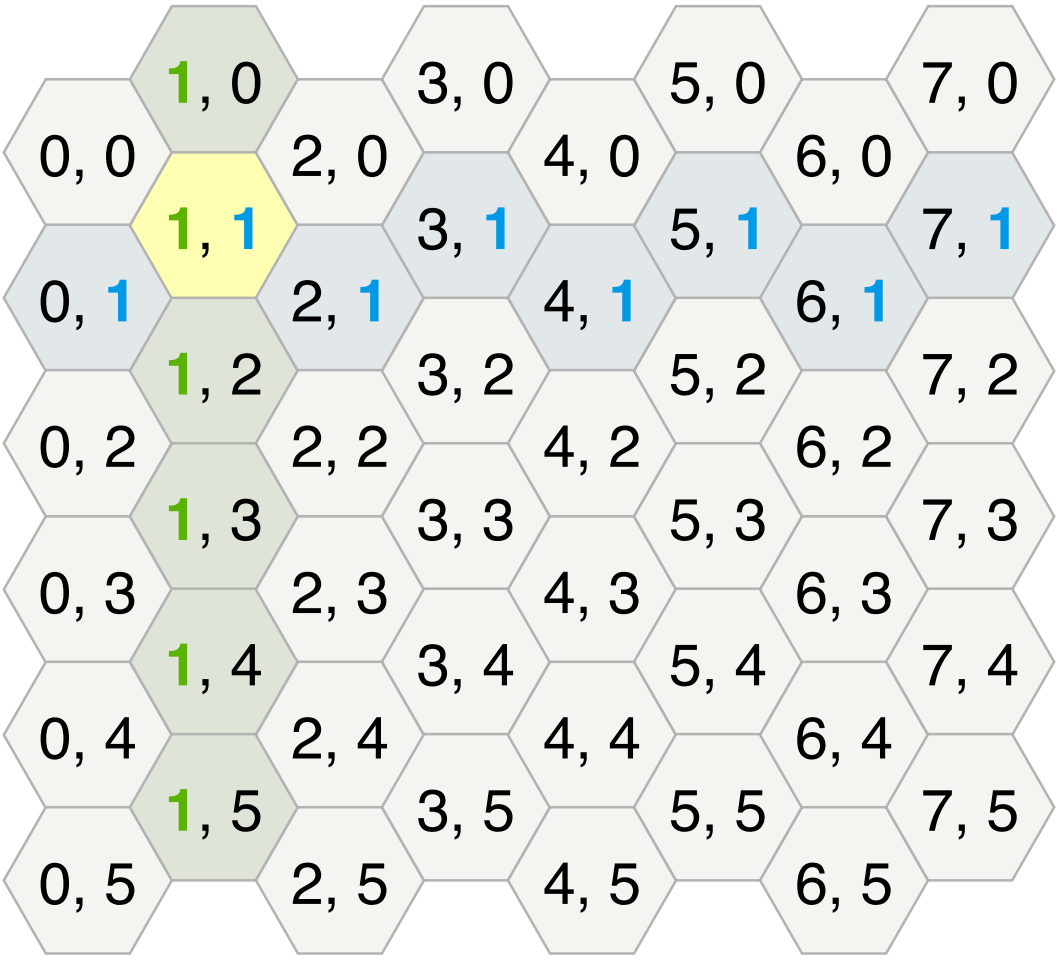
"odd-r" horizontal layout
shoves odd rows right



"even-r" horizontal layout
shoves even rows right



"odd-q" vertical layout
shoves odd columns down



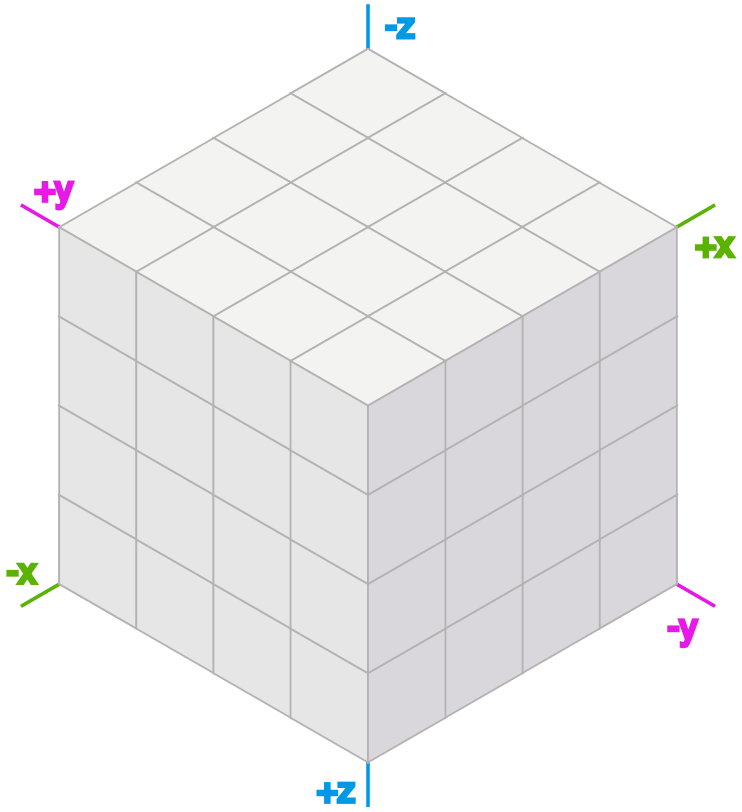
“even-q” vertical layout
shoves even columns down

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.

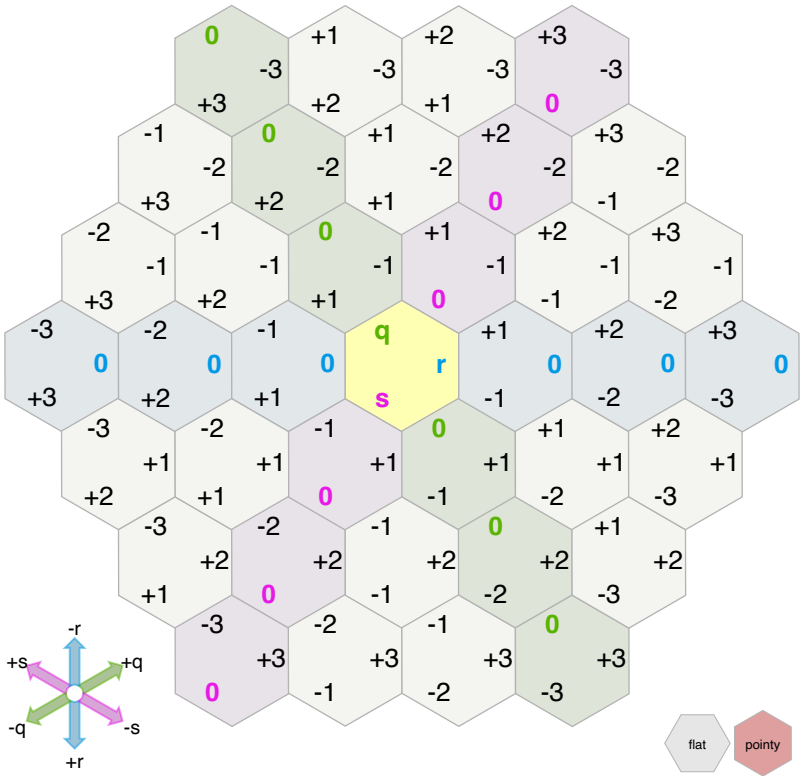
Let's take a cube grid and **slice** out a diagonal plane at `x + y + z = 0`. This is a *weird* idea but it helps us with hex grid algorithms:

1. 3d cartesian coordinates follow standard vector operations: we can add/subtract coordinates, multiply/divide by a scalar, etc. We can reuse these operations with hexagonal grids. Offset coordinates do not support these operations.
2. 3d cartesian coordinates have existing algorithms like distances, rotation, reflection, line drawing, conversion to/from screen coordinates, etc. We can adapt these algorithms to work on hexagonal grids.



Switch to hexagons

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

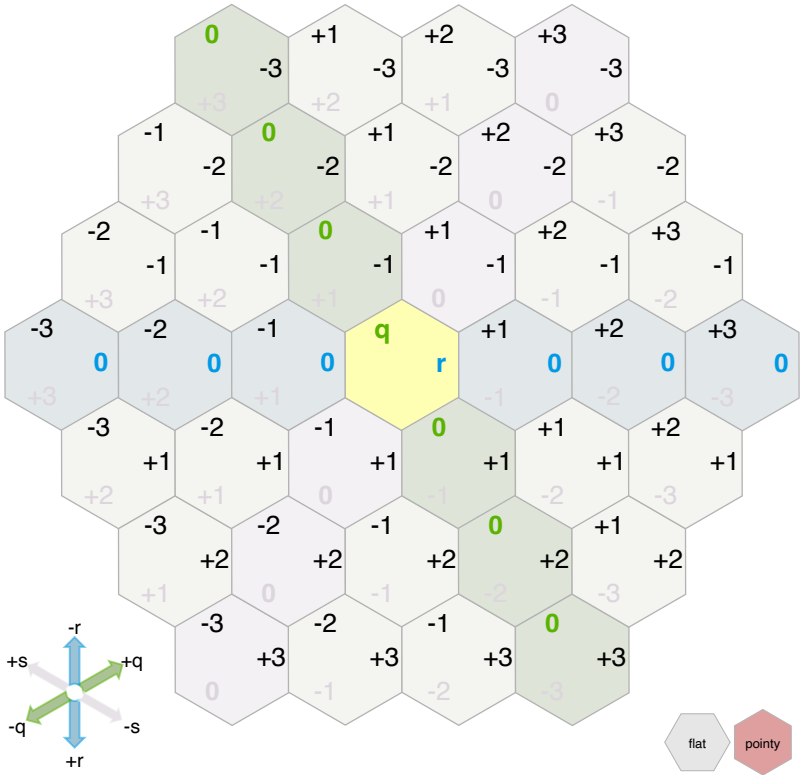


1. Each direction on the cube grid corresponds to a *line* on the hex grid. Try highlighting a hex with `r` at 0, 1, 2, 3 to see how these are related. The row is marked in blue. Try the same for `q` (green) and `s` (purple).
2. Each direction on the hex grid is a combination of *two* directions on the cube grid. For example, northwest on the hex grid lies between the `+s` and `-r`, so every step northwest involves adding 1 to `s` and subtracting 1 from `r`. We'll use this property in the neighbors section.

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

Axial coordinates

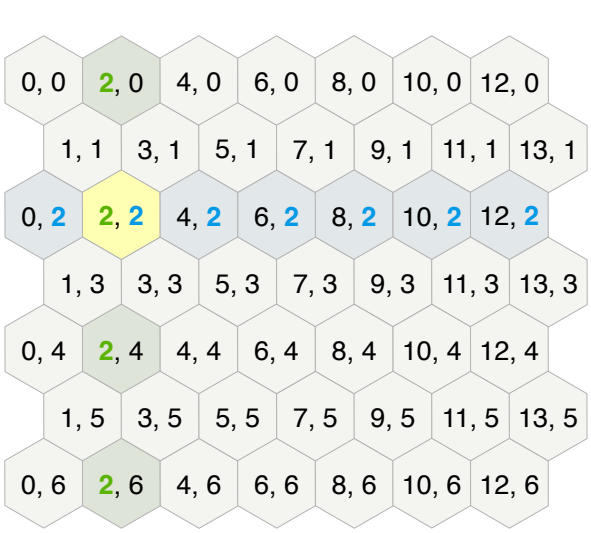
The axial coordinate system, sometimes called “trapezoidal” or “oblique” or “skewed”, is **the same as the cube system** except we don't store the `s` coordinate. Since we have a constraint `q + r + s = 0`, we can calculate `s = -q-r` when we need it.



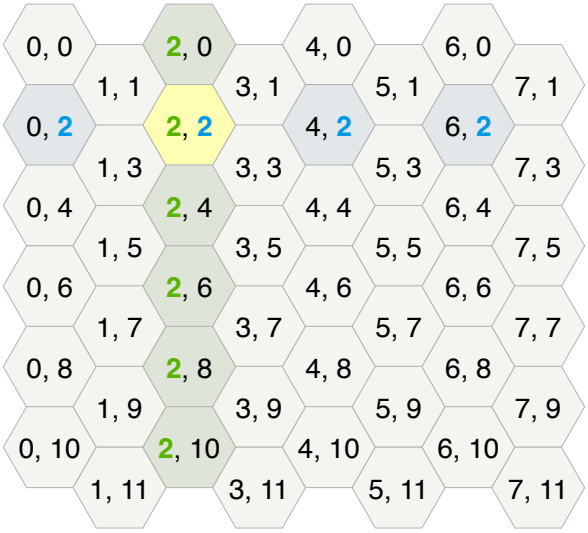
The axial/cube system allows us to add, subtract, multiply, and divide with hex coordinates. The offset coordinate systems do not allow this, and that's part of what makes algorithms simpler with axial/cube coordinates.

Doubled coordinates

Although I recommend axial/cube coordinates, if you are sticking to offset coordinates, consider the doubled variant. It makes many of the algorithms easier to implement. Instead of alternation, the doubled coordinates *double* either the horizontal or vertical step size. It has a constraint `(col + row) % 2 == 0`. In the horizontal (pointy top hex) layout it increases the column by 2 each hex; in the vertical (flat top hex) layout it increases the row by 2 each hex. This allows the in-between values for the hexes that are halfway in between:



"double-width" horizontal layout
doubles column values



"double-height" horizontal layout
doubles row values

I haven't found much information about this system — tri-bit.com called it interlaced[6], rot.js calls it double width[7], and this paper[8] calls it rectangular. Other possible names: brick or checkerboard. I'm not sure what to call it. Tamás Kenéz sent me the core algorithms (neighbors, distances, etc.). If you have any references, please send them to me.

Others

In previous versions of this document, I used x z y for hexagonal coordinates and *also* for cartesian coordinates, and then I also used q r s for hexagonal coordinates. To avoid confusion in this document, I'll use the names q r s for hexagonal coordinates (with the constraint `q + r + s = 0`), and I'll use the names x y z for cartesian coordinates.

There are *many* different valid cube hex coordinate systems. Some of them have constraints other than `q + r + s = 0`. I've shown only one of the many systems. There are also *many* different valid axial hex coordinate systems, found by using reflections and rotations. Some have the 120° axis separation as shown here and some have a 60° axis separation.

There are also cube systems that use `q-r, r-s, s-q`. One of the interesting properties of that system is that it reveals hexagonal directions.

There are spiral coordinate systems I haven't explored. See this question[9] or this question[10] on stackoverflow, or this paper about machine vision[11], or this diagram about "generalized balanced ternary" coordinates[12], or this math paper[13], or this discussion on reddit[14]. They seem potentially useful for fixed sized hexagonal shaped maps.

Recommendations

What do I recommend?

	Offset	Doubled	Axial	Cube
Pointy rotation	evenr, oddr	doublewidth	axial	cube
Flat rotation	evenq, oddq	doubleheight		
Other rotations	no		yes	
Vector operations (add, subtract, scale)	no	yes	yes	yes
Array storage	rectangular	no*	rhombus*	no*
Hash storage	any shape		any shape	

	Offset	Doubled	Axial	Cube
Hexagonal symmetry	no	no	no	yes
Easy algorithms	few	some	most	most

* rectangular maps require an adapter, shown in the map storage section

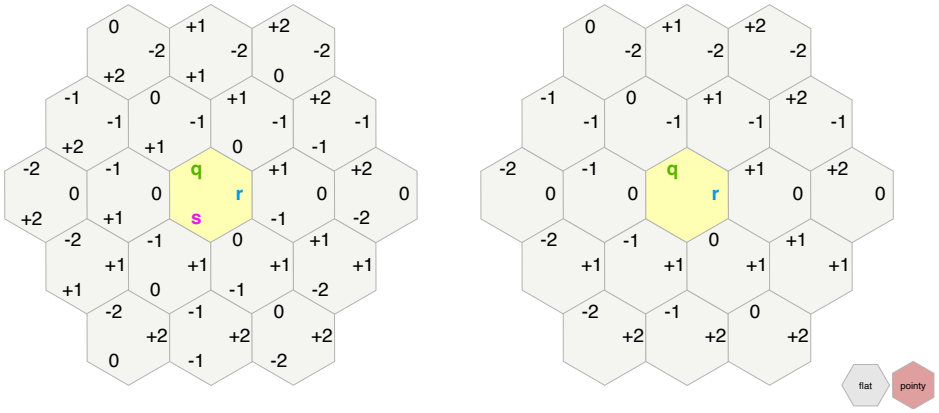
My recommendation: if you are only going to use non-rotated rectangular maps, consider the doubled or offset system that matches your map orientation. For maps with rotation, or non-rectangularly shaped maps, use axial/cube. Either choose to store the `s` coordinate (cube), or calculate it when needed as `-q-r` (axial).

Coordinate conversion

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

Axial coordinates

Axial and Cube coordinates are essentially the same. In the Cube system, we store the third coordinate, `s`. In the Axial system, we don't store it, but instead have to calculate it as needed, `s = -q-r`.




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

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

Converting between the systems like this is probably overkill. If you're using Cube and need Axial, ignore the `s` coordinate. If you're using Axial and need Cube, calculate the `s` coordinate in the algorithms that need it.

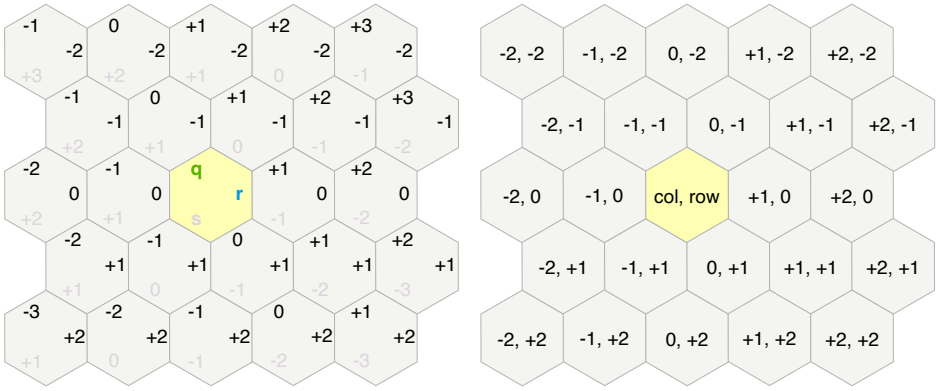
Offset coordinates

Determine which type of offset system you use; ***-r** are pointy top; ***-q** are flat top. The conversion is different for each.

```
function axial_to_oddr(hex):
    var col = hex.q + (hex.r - (hex.r&1)) / 2
    var row = hex.r
    return OffsetCoord(col, row)

function oddr_to_axial(hex):
    var q = hex.col - (hex.row - (hex.row&1)) / 2
```

```
    var r = hex.row
    return Hex(q, r)
```

◉ **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



Convert to/from ◉ **axial** or ○ **cube**.

Implementation note: I use `a&1` (bitwise and[15]) instead of `a%2` (modulo[16]) 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.

Doubled coordinates

```
function doubleheight_to_axial(hex):
    var q = hex.col
    var r = (hex.row - hex.col) / 2
    return Hex(q, r)

function axial_to_doubleheight(hex):
    var col = hex.q
```

```
    var row = 2 * hex.r + hex.q
    return DoubledCoord(col, row)

function doublewidth_to_axial(hex):
    var q = (hex.col - hex.row) / 2
    var r = hex.row
    return Hex(q, r)

function axial_to_doublewidth(hex):
    var col = 2 * hex.q + hex.r
    var row = hex.r
    return DoubledCoord(col, row)
```

If you need Cube coordinates, use `Cube(q, r, -q-r)` instead of `Hex(q, r)`.

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.

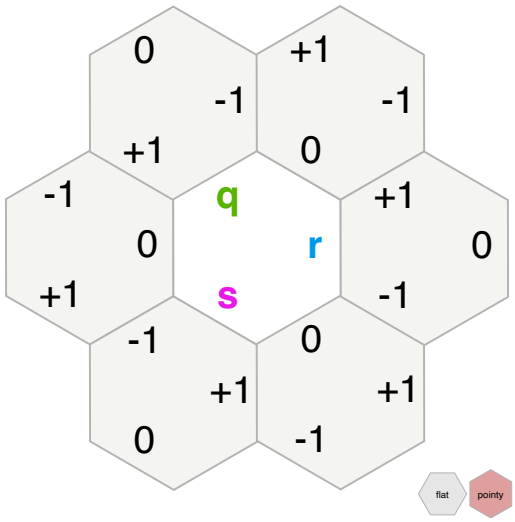
Cube coordinates

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(dq, dr, ds)`:

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

function cube_direction(direction):
```

```
    return cube_direction_vectors[direction]

function cube_add(hex, vec):
    return Cube(hex.q + vec.q, hex.r + vec.r, hex.s + vec.s)

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

With the Cube coordinate systems, we can store *differences* between two coordinates (a "vector"), and then add those differences back to a coordinate to get another coordinate. That's what the `cube_add` function does. Axial and Doubled coordinates also support this, but the Offset coordinates do not.

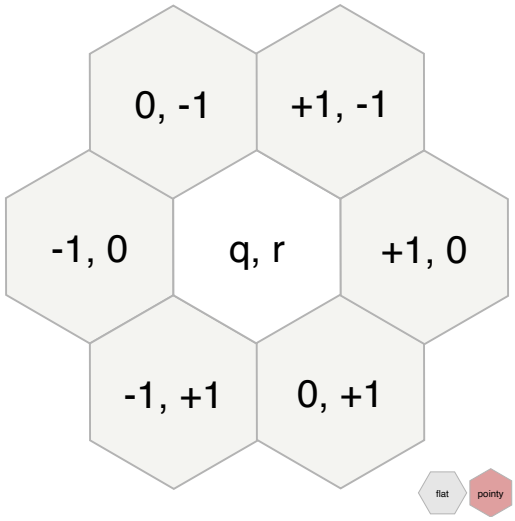


Axial coordinates

Since axial is the same as cube except not storing the third coordinate, the code is the same as the previous section except we won't write out the third coordinate:

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

```
function axial_direction(direction):
    return axial_direction_vectors[direction]

function axial_add(hex, vec):
    return Hex(hex.q + vec.q, hex.r + vec.r)

function axial_neighbor(hex, direction):
    return axial_add(hex, axial_direction(direction))
```



Offset coordinates

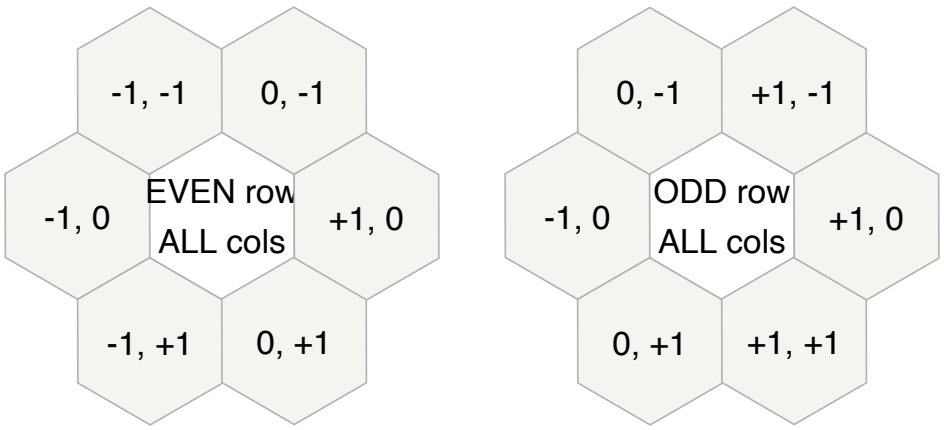
As with cube and axial coordinates, we'll build a table of the numbers we need to add to `col` and `row`. However **offset coordinates can't be safely subtracted and added**. For example, moving southeast from (0, 0) takes us to (0, +1), so we might put (0, +1) into the table for moving southeast. But moving southeast from (0, +1) takes us to (+1, +2), so we would need to put (+1, +1) into that table. *The amount we need to add depends on where in the grid we are*.

Since the movement vector is different for odd and even columns/rows, we will need two separate lists of neighbors. **Pick a grid type** to see the corresponding code.

```
var oddr_direction_differences = [
    // even rows
```

```
    [[+1,  0], [ 0, -1], [-1, -1],
     [-1,  0], [-1, +1], [ 0, +1]],
    // odd rows
    [[+1,  0], [+1, -1], [ 0, -1],
     [-1,  0], [ 0, +1], [+1, +1]],
]

function oddr_offset_neighbor(hex, direction):
    var parity = hex.row & 1
    var diff = oddr_direction_differences[parity][direction]
    return OffsetCoord(hex.col + diff[0], hex.row + diff[1])
```

Pick a grid type: ◉ odd-r ○ even-r ○ odd-q ○ even-q



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.

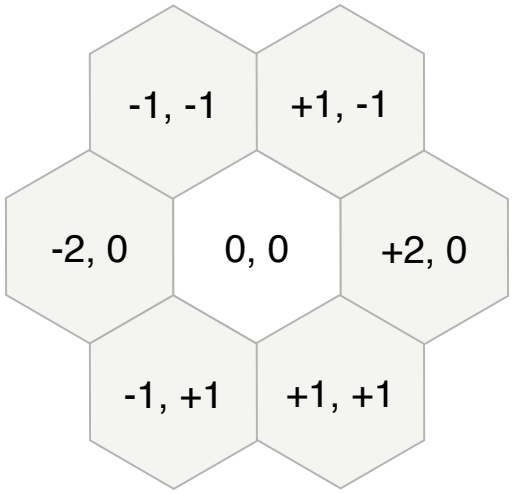
Doubled coordinates

Unlike offset coordinates, the neighbors for doubled coordinates do *not* depend on which column/row we're on. They're the same everywhere, like axial/cube coordinates. Also unlike offset coordinates, we can safely subtract and add doubled coordinates, which makes them easier to work with than offset coordinates.

```
var doublewidth_direction_vectors = [
    DoubledCoord(+2,  0), DoubledCoord(+1, -1),
```

```
    DoubledCoord(-1, -1), DoubledCoord(-2,  0),
    DoubledCoord(-1, +1), DoubledCoord(+1, +1),
]

function doublewidth_add(hex, diff):
    return DoubleCoord(hex.col + diff.col, hex.row + diff.row)

function doublewidth_neighbor(hex, direction):
    var vec = doublewidth_direction_vectors[direction]
    return doublewidth_add(hex, vec)
```

Pick a grid type: ◉ double width ○ double height

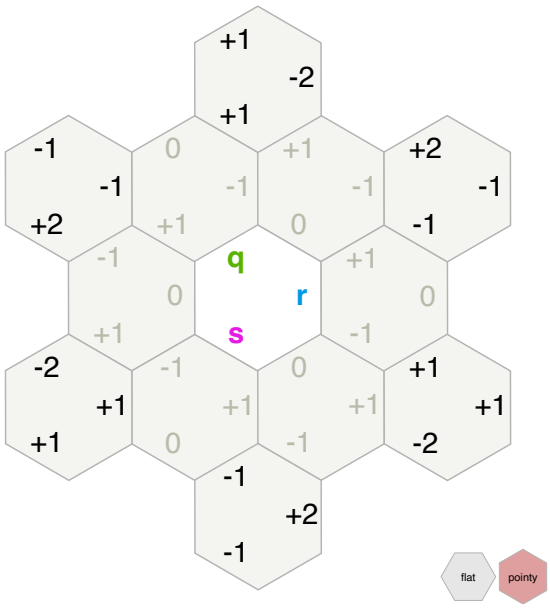


Diagonals

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_diagonal_vectors = [
    Cube(+2, -1, -1), Cube(+1, -2, +1), Cube(-1, -1, +2),
    Cube(-2, +1, +1), Cube(-1, +2, -1), Cube(+1, +1, -2),
]

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

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



Distances

Cube coordinates

Since cube hexagonal coordinates are based on 3d cube coordinates, we can *adapt* the distance calculation to work on hexagonal grids. Each hexagon corresponds to a cube in 3d space. Adjacent hexagons are distance 1 apart in the hex grid but distance 2 apart in the cube grid. For every 2 steps in the cube grid, we need only 1 step in the hex grid. In the 3d cube grid, Manhattan distances are `abs(dx) + abs(dy) + abs(dz)`. The distance on a hex grid is half that:

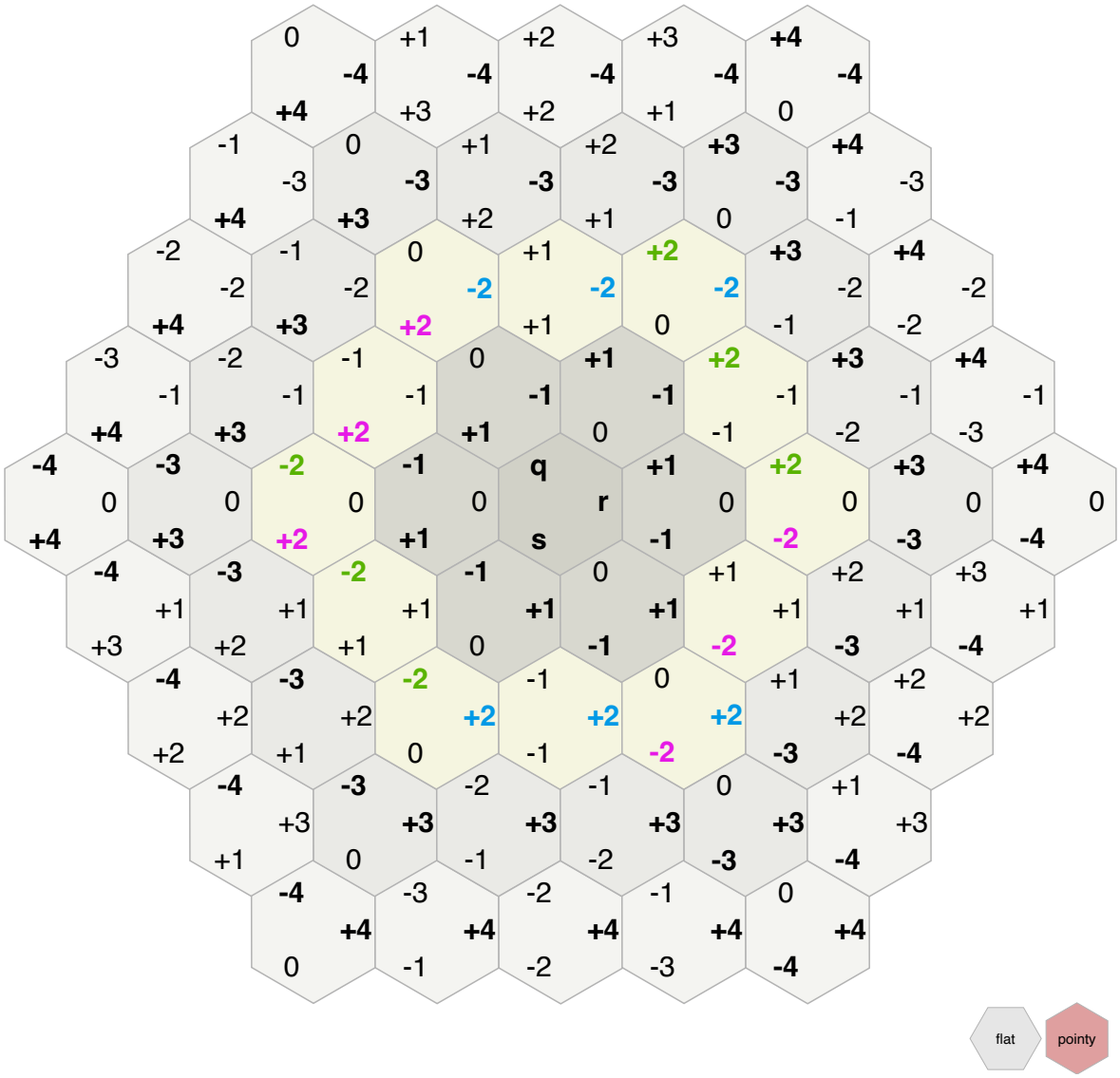
```
function cube_subtract(a, b):
    return Cube(a.q - b.q, a.r - b.r, a.s - b.s)

function cube_distance(a, b):
    var vec = cube_subtract(a, b)
```

```
    return (abs(vec.q) + abs(vec.r) + abs(vec.s)) / 2
    // or: (abs(a.q - b.q) + abs(a.r - b.r) + abs(a.s - b.s)) / 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_subtract(a, b):
    return Cube(a.q - b.q, a.r - b.r, a.s - b.s)

function cube_distance(a, b):
    var vec = cube_subtract(a, b)
    return max(abs(vec.q), abs(vec.r), abs(vec.s))
    // or: max(abs(a.q - b.q), abs(a.r - b.r), abs(a.s - b.s))
```

The maximum of the three coordinates is the distance. You can also use the max of `abs(vec.q-vec.r), abs(vec.r-vec.s), abs(vec.s-vec.q)` to figure out which of the 6 "wedges" a hex is in; see diagrams here.



Xiangguo Li's paper *Storage and addressing scheme for practical hexagonal image processing.*[17] (DOI[18]) gives a formula for Euclidean distance, which can be adapted to axial coordinates: `sqrt(dq² + dr² + dq*dr)`.

Axial coordinates

In the axial system, the third coordinate is implicit. We can always convert axial to cube to calculate distance:

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

Once we inline those functions it ends up as:

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

which can also be written:

```
function axial_subtract(a, b):
    return Hex(a.q - b.q, a.r - b.r)

function axial_distance(a, b):
    var vec = axial_subtract(a, b)
    return (abs(vec.q)
          + abs(vec.q + vec.r)
          + abs(vec.r)) / 2
```

There are lots of different ways to write hex distance in axial coordinates. 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[19] 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 axial/cube coordinates, then use axial/cube distance.

```
function offset_distance(a, b):
    var ac = offset_to_axial(a)
    var bc = offset_to_axial(b)
    return axial_distance(ac, bc)
```

We'll use the same pattern for many of the algorithms: convert offset to axial/cube, run the axial/cube version of the algorithm, and convert any axial/cube results back to offset coordinates. There are also more direct formulas for distances; see the rot.js manual[20] for a formula in the "Odd shift" section.

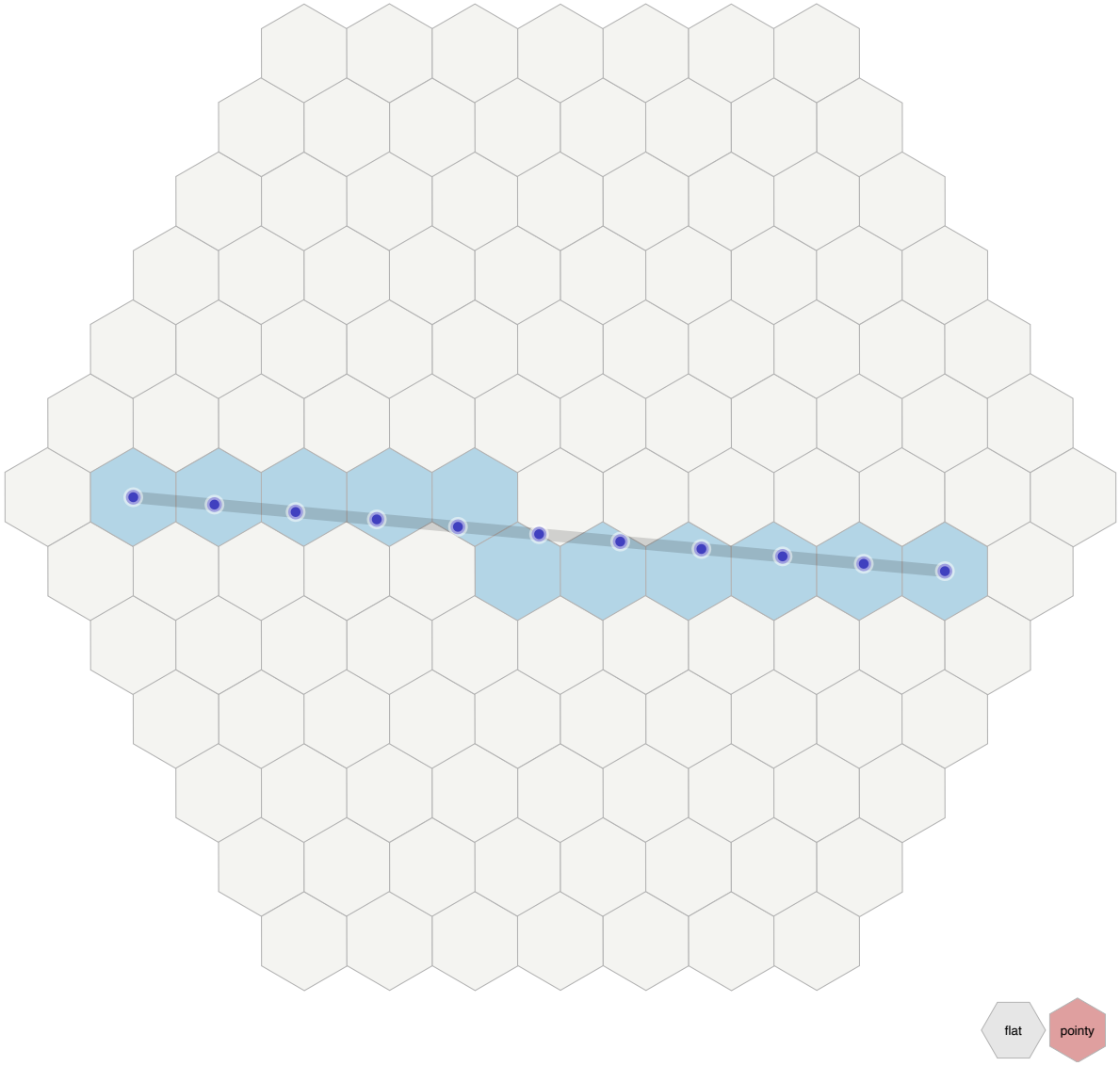
Doubled coordinates

Although converting doubled coordinates to axial/cube coordinates works, there's also a direct formula for distances, from the rot.js manual[21]:

```
function doublewidth_distance(a, b):
    var dcol = abs(a.col - b.col)
    var drow = abs(a.row - b.row)
    return drow + max(0, (dcol-drow)/2)

function doubleheight_distance(a, b):
    var dcol = abs(a.col - b.col)
    var drow = abs(a.row - b.row)
    return dcol + max(0, (drow-dcol)/2)
```

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.



1. First we calculate `N`=10 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.

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

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

function cube_lerp(a, b, t): # for hexes
    return Cube(lerp(a.q, b.q, t),
                lerp(a.r, b.r, t),
                lerp(a.s, b.s, 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 exactly on the side 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, 2e-6, -3e-6)` to one or both of the endpoints before starting the loop. This will "nudge" the line in one direction to avoid landing on side boundaries.
- The DDA Algorithm[22] 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.
- There are times when this algorithm slightly goes outside the marked hexagons (example). I haven't come up with an easy fix for this.
- 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.q-A.q`, `B.r-A.r`, `B.s-A.s`, and `1.0/N` outside the loop. Multiplication can be turned into repeated addition. You'll end up with something like the DDA algorithm.
- This code can be adapted to work with axial coordinates — define `axial_lerp` and then use `axial_distance`, `axial_round` in `axial_linedraw`. It is likely it can work with doubled coordinates as well.
- I use axial or cube coordinates for line drawing, but if you want something for offset coordinates, take a look at this article[23].
- There are many variants of line drawing. Sometimes you'll want "super cover"[24]. Someone sent me hex super cover line drawing code but I haven't studied it yet.
- A paper from Yong-Kui, Liu, *The Generation of Straight Lines on Hexagonal Grids*, Computer Graphics Forum 12-1 (Feb 1993) (DOI[25]), describes a variant of Bresenham's line drawing algorithm for hexagonal grids.

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(q), abs(r), abs(s))`. To find all hexes within N steps, we need `max(abs(q), abs(r), abs(s)) ≤ N`. This means we need *all* three to be true: `abs(q) ≤ N` and `abs(r) ≤ N` and `abs(s) ≤ N`. Removing absolute value, we get `-N ≤ q ≤ +N` and `-N ≤ r ≤ +N` and `-N ≤ s ≤ +N`. In code it's a nested loop:

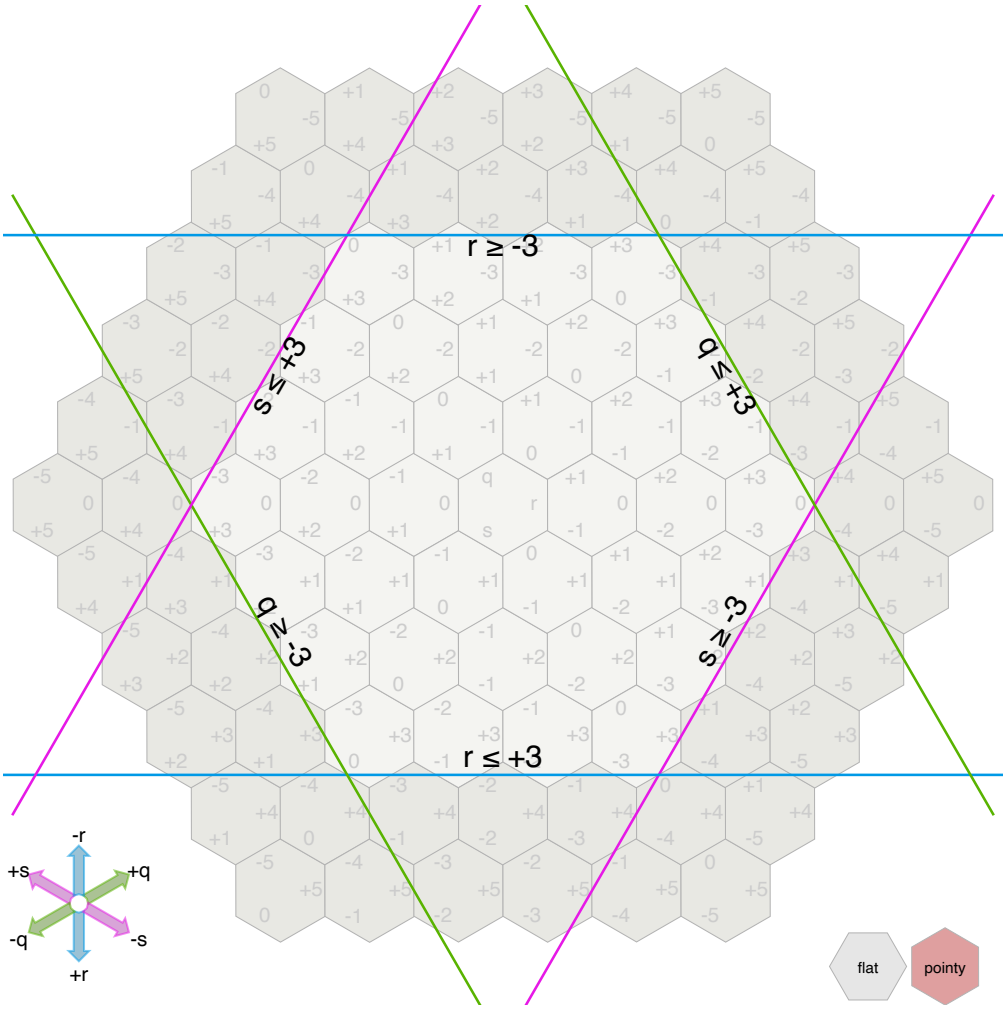
```
var results = []
for each -N ≤ q ≤ +N:
    for each -N ≤ r ≤ +N:
        for each -N ≤ s ≤ +N:
            if q + r + s == 0:
                results.append(cube_add(center, Cube(q, r, s)))
```

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

```
var results = []
for each -N ≤ q ≤ +N:
    for each max(-N, -q-N) ≤ r ≤ min(+N, -q+N):
        var s = -q-r
        results.append(cube_add(center, Cube(q, r, s)))
```

This loop iterates over exactly the needed coordinates. In the diagram, each range is a pair of lines. Each line is an inequality (a half-plane[26]). We pick all the hexes that satisfy all six inequalities. This loop also works nicely with axial coordinates:

```
var results = []
for each -N ≤ q ≤ +N:
    for each max(-N, -q-N) ≤ r ≤ min(+N, -q+N):
        results.append(axial_add(center, Hex(q, r)))
```



Intersecting ranges

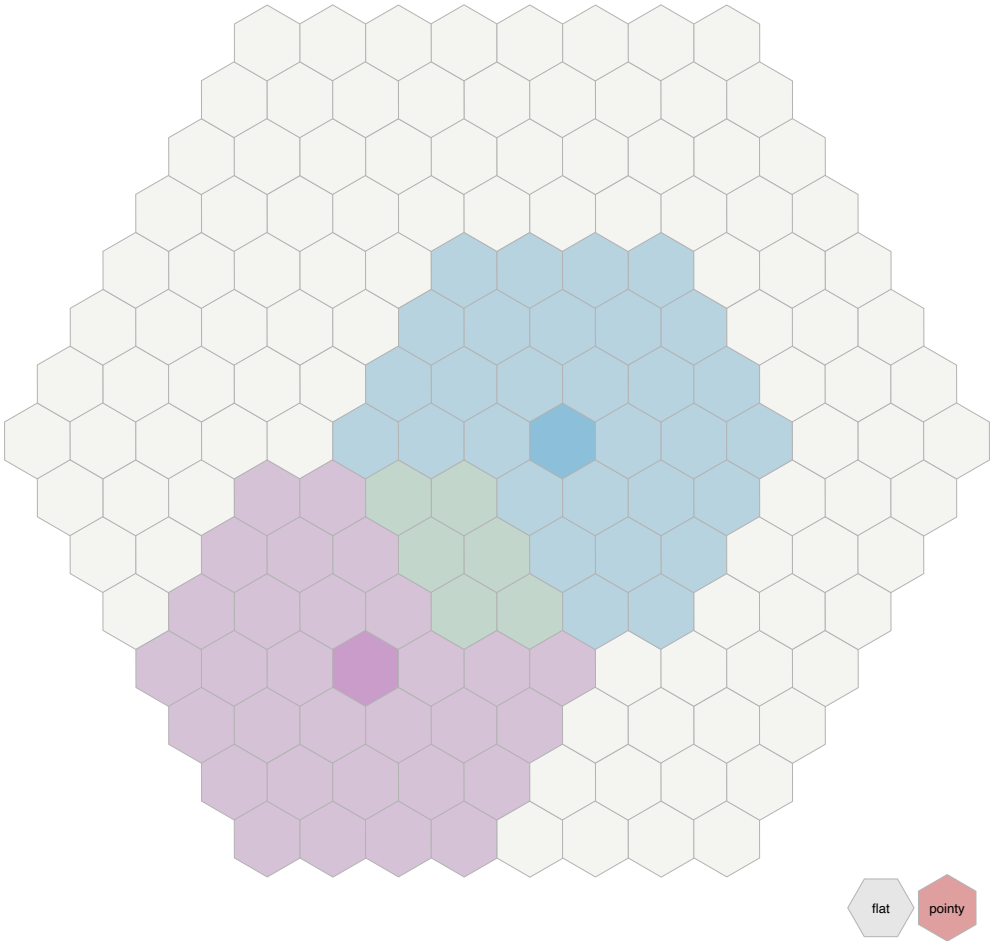
If you need to find hexes that are in more than one range, you can intersect the ranges before generating a list of hexes.

You can either think of this problem algebraically or geometrically. Algebraically, each hexagonally-shaped region is expressed as inequality constraints of the form `-N ≤ dq ≤ +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[27] in 3D space, then project back to the q + r + s = 0 plane to get hexes. I'm going to solve it algebraically:

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

```
var results = []
for each q_min ≤ q ≤ q_max:
    for each max(r_min, -q-s_max) ≤ r ≤ min(r_max, -q-s_min):
        results.append(Hex(q, r))
```

The intersection of two ranges $a \le q \le b$ and $c \le q \le d$ is $max(a, c) \le q \le min(b, d)$. Since a hex region is expressed as ranges over q, r, s, we can separately intersect each of the q, r, s ranges then use the nested loop to generate a list of hexes in the intersection. For one hex region we set $q_{min} = H.q - N$ and $q_{max} = H.q + N$ and likewise for r and s. For intersecting two hex regions we set $q_{min} = max(H_1.q - N, H_2.q - N)$ and $q_{max} = min(H_1.q + N, H_2.q + N)$, and likewise for r and s. The same pattern works for intersecting three or more regions, and can generalize to other shapes[28] (triangles, trapezoids, rhombuses, non-regular hexagons).

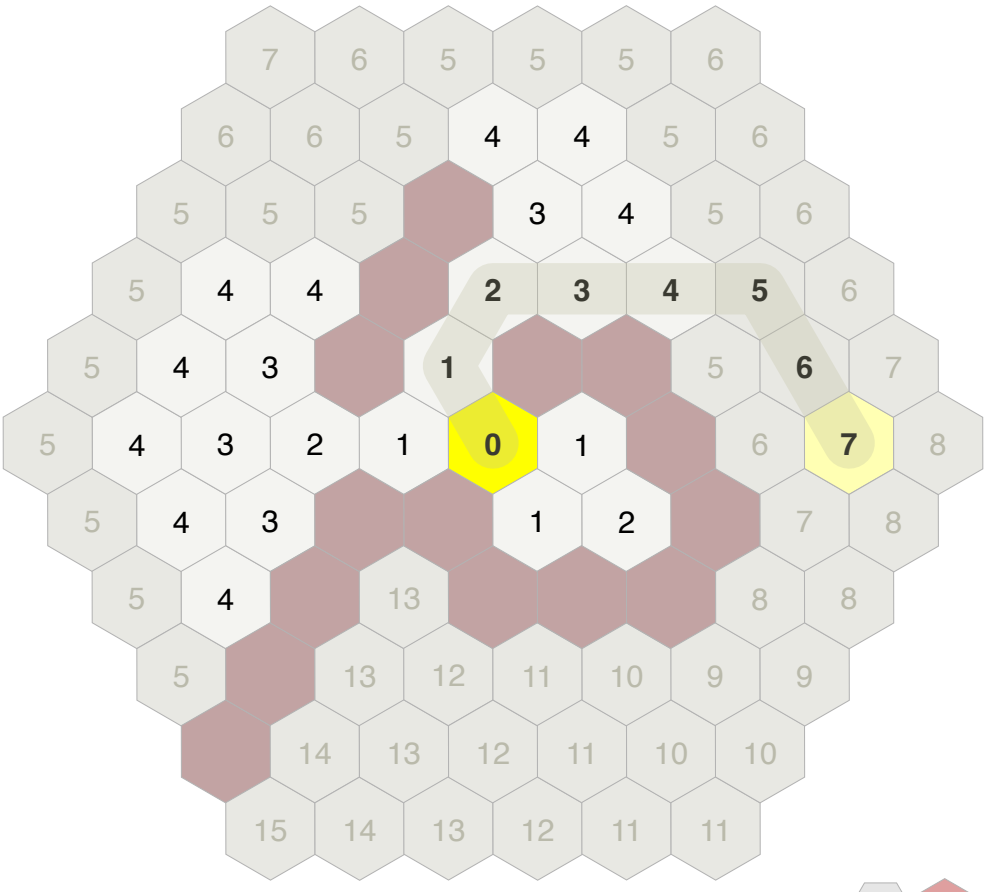


Obstacles

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 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`. This works equally well with any of the hex coordinate systems (cube, axial, offset, doubled).

```
function hex_reachable(start, movement):
    var visited = set() # set of hexes
    add start to visited
    var fringes = [] # array of arrays of hexes
```

```
fringes.append([start])

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

return visited
```



Rotation

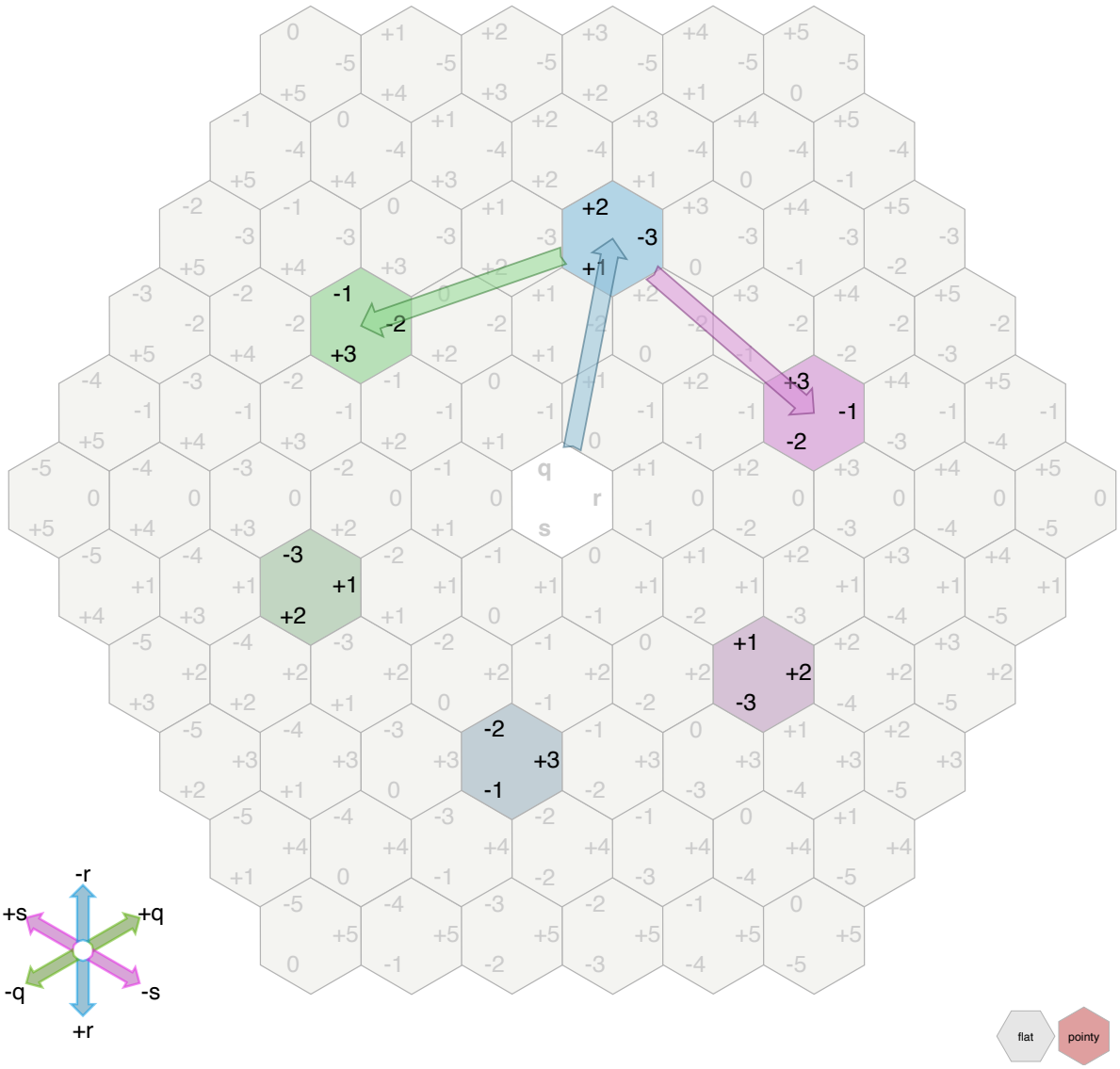
Given a hex vector (difference between one hex and another), we might want to rotate it to point to a different hex. This is simple with cube coordinates if we stick with rotations of 1/6th of a circle.

A rotation 60° right (clockwise ↻) shoves each coordinate one slot to the left ←:

```
      [ q,  r,  s]
to      [-r, -s, -q]
to        [  s,  q,  r]
```

A rotation 60° left (counter-clockwise ↺) shoves each coordinate one slot to the right →:

```
        [ q,  r,  s]
to    [-s, -q, -r]
to [r,  s,  q]
```



As you play with diagram, notice that each 60° rotation *flips* the signs and also physically "rotates" the coordinates. Take a look at the axis legend on the bottom left to see how this works. 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.

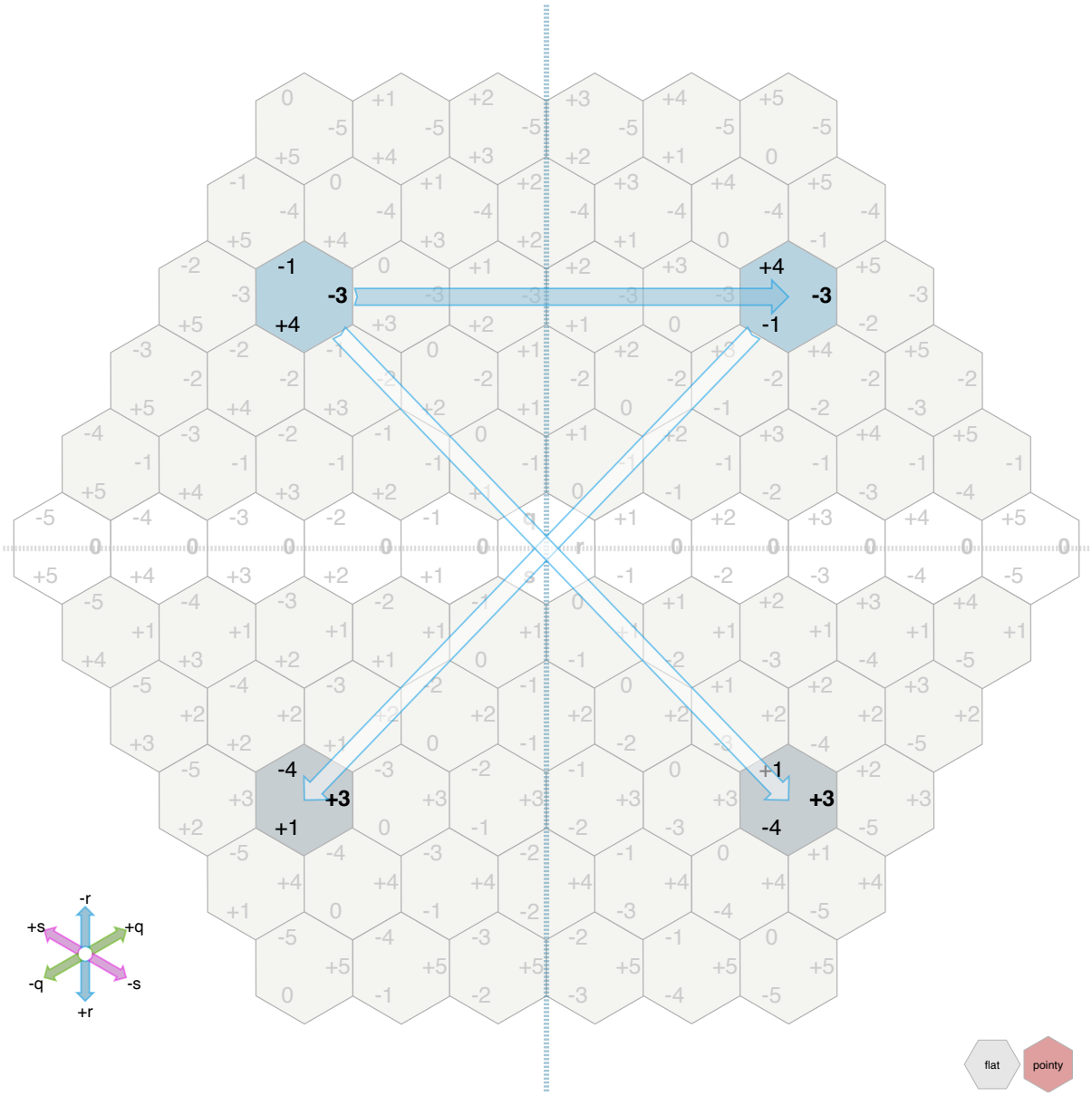
Here's the full recipe for rotating a position `hex` around a center position `center` to result in a new position `rotated`:

1. Convert positions `hex` and `center` to cube coordinates.
2. Calculate a *vector* by subtracting the center: `vec = cube_subtract(hex, center) = Cube(hex.q - center.q, hex.r - center.r, hex.s - center.s)`.
3. Rotate the vector `vec` as described above, and call the resulting vector `rotated_vec`.
4. Convert the vector back to a position by adding the center: `rotated = cube_add(rotated_vec, center) = Cube(rotated_vec.q + center.q, rotated_vec.r + center.r, rotated_vec.s + center.s)`.
5. Convert the cube position `rotated` back to to your preferred coordinate system.

It's several conversion steps but each step is short. 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[29] for other ways to calculate rotation.

Reflection

Given a hex, we might want to reflect it across one of the axes. With cube coordinates, we *swap* the coordinates that *aren't* the axis we're reflecting over. The axis we're reflecting over stays the same.



Reflection axis: ○ q ○ r ○ s

```
function reflectQ(h) { return Cube(h.q, h.s, h.r); }
function reflectR(h) { return Cube(h.s, h.r, h.q); }
function reflectS(h) { return Cube(h.r, h.q, h.s); }
```

To reach the other two reflections, *negate* the coordinates of the original and the first reflection. These are shown as white arrows in the diagram.

To reflect over a line that's not at 0, pick a reference point on that line. Subtract the reference point, perform the reflection, then add the reference point back.

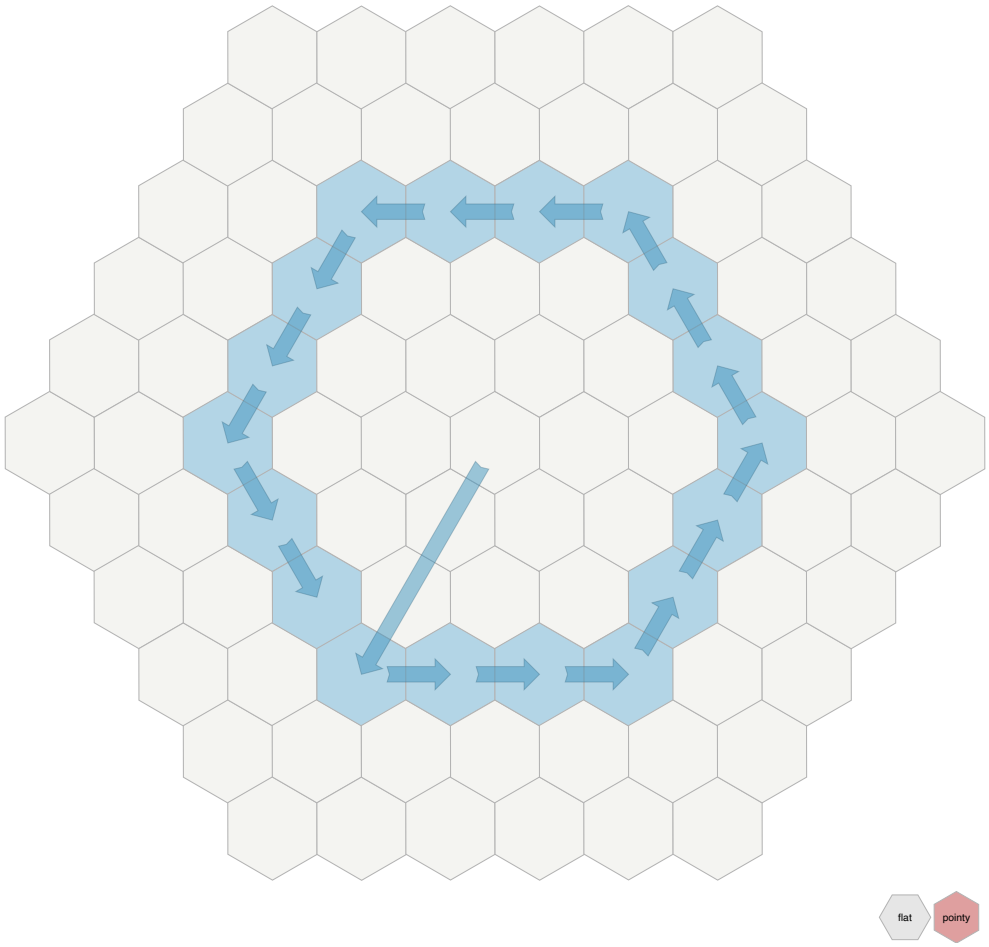
Rings

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_scale(hex, factor):
    return Cube(hex.q * factor, hex.r * factor, hex.s * factor)

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

In this code, `hex` 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, `hex` moves one hex along the ring. After `6 * radius` steps it ends up back where it started.



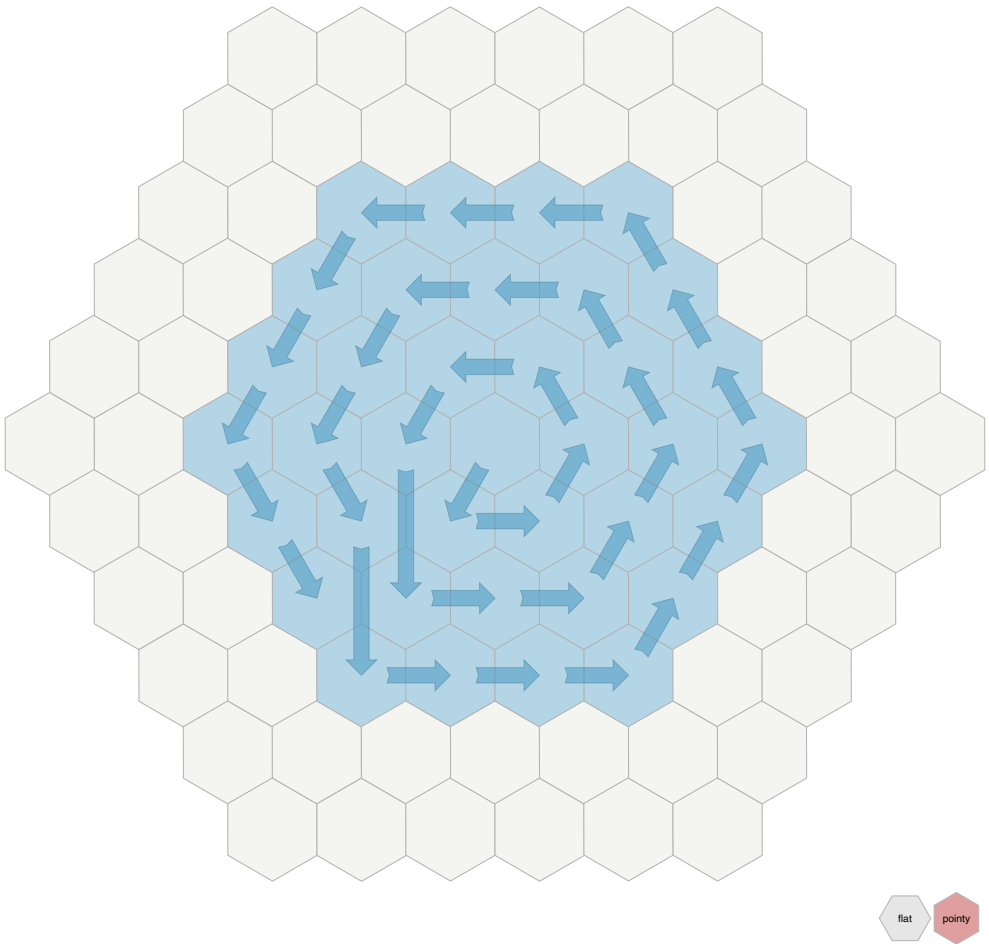
The scale, add, and neighbor operations also work on axial and doubled coordinates, so the same algorithm can be used. For offset coordinates, convert to one of the other formats, generate the ring, and convert back.

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 = list(center)
    for each 1 ≤ k ≤ radius:
```

```
        results = list_append(results, cube_ring(center, k))
    return results
```



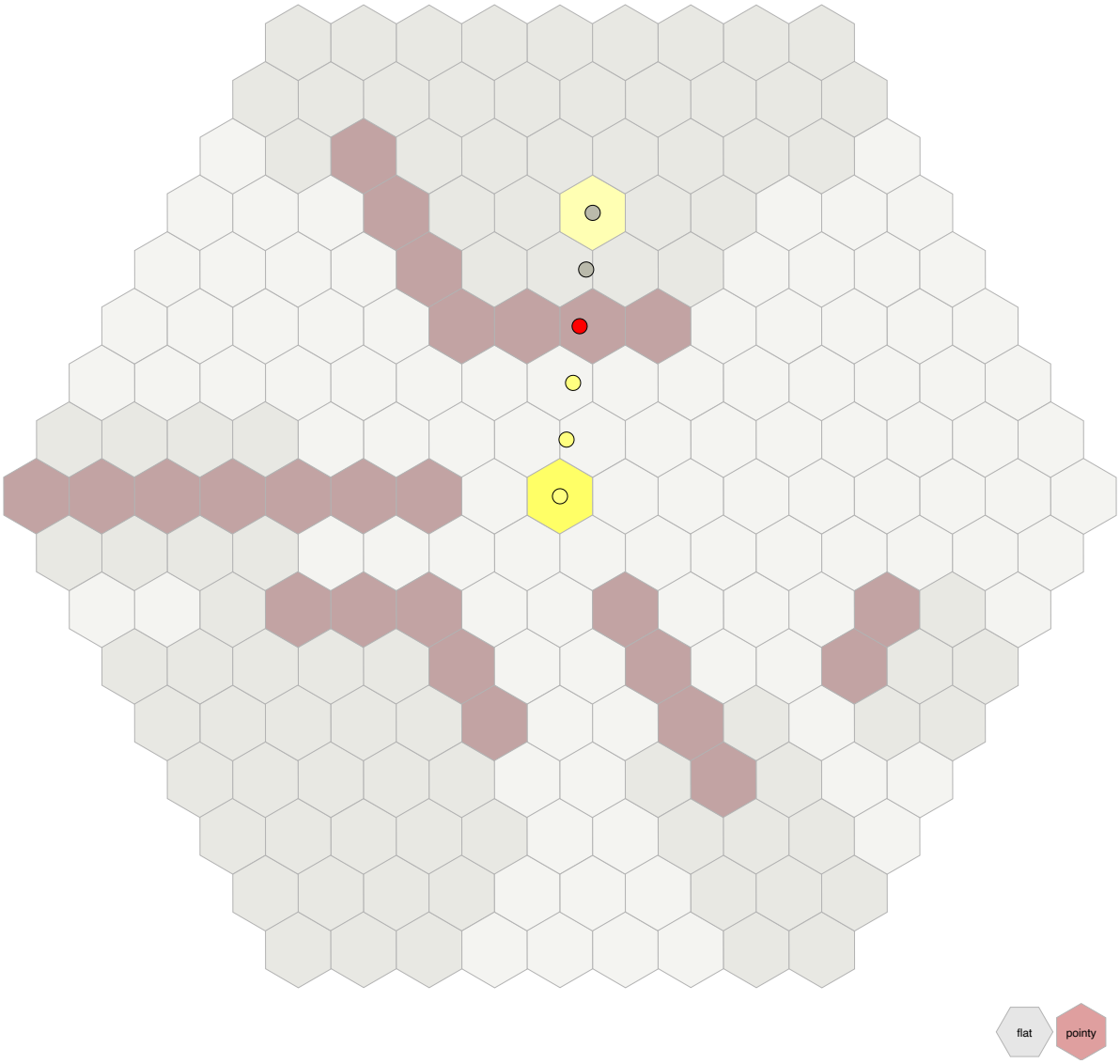
Spirals also give us a way to *count* how many hexagon tiles are in the larger hexagon. The center is 1 hex. Each ring is 6 * `k` hexes. The sum will 1 + 6 * sum(1…radius). Using this formula[30], that simplifies to (1 + 3 * radius * (radius+1)).

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.

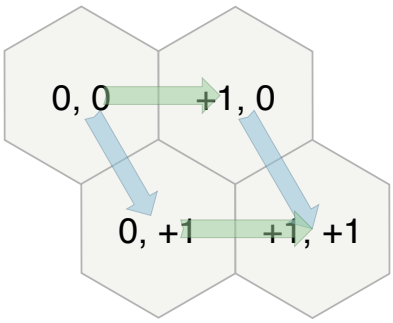
Clark Verbrugge's guide[31] describes a "start at center and move outwards" algorithm to calculate field of view. Also see the Duelo[32] project, which has an an online demo of directional field of view[33] and code on Github. Also see my article on 2d visibility calculation for an algorithm that works on polygons, including hexagons. For grids, the roguelike community has a nice set of algorithms for square grids (see this[34] and this[35] and this[36]); 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*. The arrow (0,0)→(1,0) is the q basis vector (x=sqrt(3), y=0) and (0,0)→(0,1) is the r basis vector (x=sqrt(3)/2, y=3/2). The pixel coordinate is `q_basis * q + r_basis * r`. For example, the hex at (1,1) is the sum of 1 q vector and 1 r vector. A hex at (3,2) would be the sum of 3 q vectors and 2 r vectors.



The code for ○ flat top or ○ pointy top is:

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

This can also be viewed as a matrix multiply, where the basis vectors are the columns of the matrix:

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

The matrix approach will come in handy later when we want to convert pixel coordinates back to hex coordinates. To invert the process of hex-to-pixel into a pixel-to-hex process, we will invert the hex-to-pixel matrix into a pixel-to-hex matrix.

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):
    var x = size * sqrt(3) * (hex.col + 0.5 * (hex.row&1))
```

```
    var 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 axial coordinates, then use the axial to pixel conversion. By inlining the conversion code then optimizing, it will end up being the same as above.

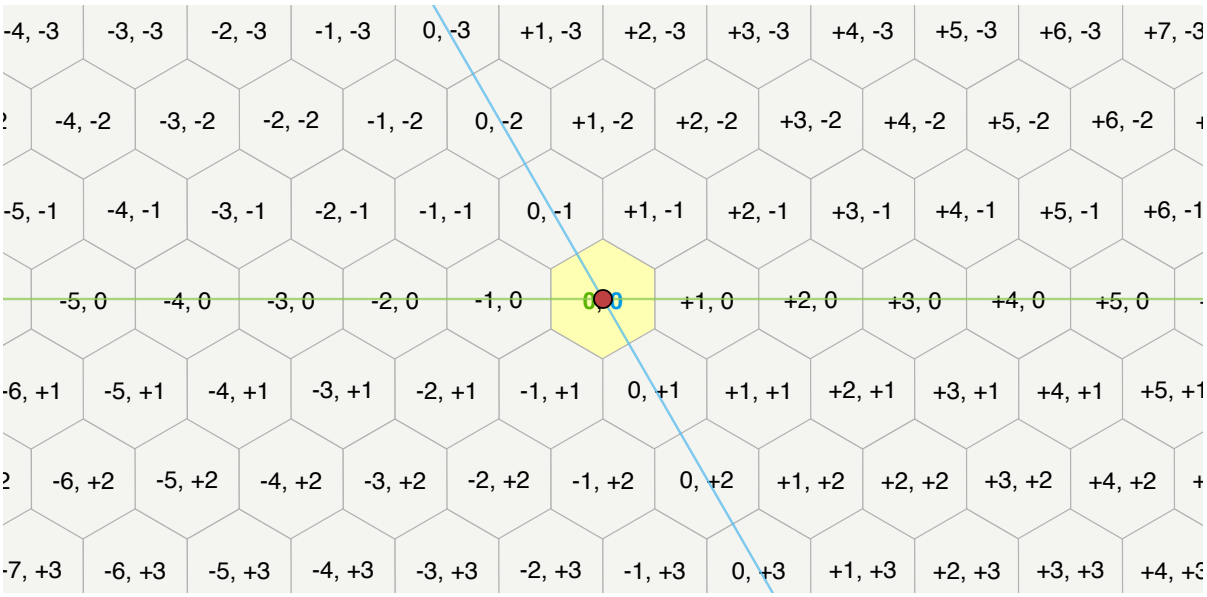
Doubled coordinates

Doubled makes many algorithms simpler than offset.

```
function doublewidth_to_pixel(hex):
    var x = size * sqrt(3)/2 * hex.col
    var y = size * 3/2 * hex.row
    return Point(x, y)

function doubleheight_to_pixel(hex):
    var x = size * 3/2 * hex.col
    var y = size * sqrt(3)/2 * hex.row
    return Point(x, y)
```

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 red 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`. This was a matrix multiply:

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

Matrix for: ○ flat top or ○ pointy top

To invert the hex-to-pixel process into a pixel-to-hex process we invert the pointy-top hex-to-pixel matrix[37] into a pixel-to-hex matrix:

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

This calculation will give us fractional axial coordinates (floats) for `q` and `r`. The axial_round() function will convert the fractional axial coordinates into integer axial hex coordinates. Here's the code:

```
function pixel_to_pointy_hex(point):
    var q = (sqrt(3)/3 * point.x  -  1./3 * point.y) / size
    var r = (                        2./3 * point.y) / size
    return axial_round(Hex(q, r))
```

Code for: ○ flat top or ○ pointy top

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

Just as with integer cube coordinates, `frac.q + frac.r + frac.s = 0` with fractional (floating point) cube coordinates. We can round each component to the nearest integer, `q = round(frac.q); r = round(frac.r); s = round(frac.s)`. However, after rounding we do *not* have a guarantee that `q + r + s = 0`. We do have a way to correct the problem: *reset* the component with the largest change back to what the constraint `q + r + s = 0` requires. For example, if the r-change `abs(r-frac.r)` is larger than `abs(q-frac.q)` and `abs(s-frac.s)`, then we reset `r = -q-s`. This guarantees that `q + r + s = 0`. Here's the algorithm:

```
function cube_round(frac):
    var q = round(frac.q)
    var r = round(frac.r)
    var s = round(frac.s)

    var q_diff = abs(q - frac.q)
    var r_diff = abs(r - frac.r)
    var s_diff = abs(s - frac.s)

    if q_diff > r_diff and q_diff > s_diff:
        q = -r-s
    else if r_diff > s_diff:
        r = -q-s
    else:
        s = -q-r

    return Cube(q, r, s)
```

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

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

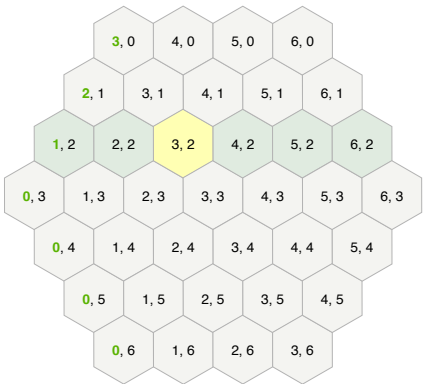
The same would work if you have `oddr`, `evenr`, `oddq`, or `evenq` instead of `axial`. Jacob Rus has a direct implementation of axial_round[39] without converting to cube first.

Implementation note: `cube_round` and `axial_round` take *float* coordinates instead of *int* coordinates. If you've written a Cube and Hex class, they'll work fine in dynamically typed 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 `axial_round`, it will be `FloatHex → Hex`, using helper function `floatcube_to_floathex` instead of `cube_to_hex`. In languages with parameterized types (C++, Haskell, 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.

Patrick Surry has a visualization showing why the rounding algorithm works[40].

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.



	q = 0	q = 1	q = 2	q = 3	q = 4	q = 5	q = 6
r = 0	(null)	(null)	(null)	3, 0	4, 0	5, 0	6, 0
r = 1	(null)	(null)	2, 1	3, 1	4, 1	5, 1	6, 1
r = 2	(null)	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2
r = 3	0, 3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3
r = 4	0, 4	1, 4	2, 4	3, 4	4, 4	5, 4	(null)
r = 5	0, 5	1, 5	2, 5	3, 5	4, 5	(null)	(null)
r = 6	0, 6	1, 6	2, 6	3, 6	(null)	(null)	(null)

Shape: ○ rectangle ○ hexagon ○ rhombus ○ down-triangle ○ up-triangle

Switch to **array of arrays**

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. Use a **2D Array**. Use nulls or some other sentinel at the unused spaces. Store `Hex(q, r)` at `array[r][q]`. 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. Store `Hex(q, r)` in `hash_table(hash(q,r))`.

3. Use an **array of arrays** by sliding row to the left, and shrinking the rows to the minimum size. For pointy-topped hexes, store `Hex(q, r)` in `array[r - first_row][q - first_column(r)]`. Some examples for the map shapes above:
   - **Rectangle**. Store `Hex(q, r)` at `array[r][q + floor(r/2)]`. Each row has the same length. This is equivalent to odd-r offset.
   - **Hexagon**. Store `Hex(q, r)` at `array[r][q - max(0, N-r)]`. Row `r` size is `2*N+1 - abs(N-r)`.
   - **Rhombus**. Conveniently, `first_row` and `first_column(r)` are both 0. Store `Hex(q, r)` at `array[r][q]`. All rows are the same length.
   - **Down-triangle**. Store `Hex(q, r)` at `array[r][q]`. Row `r` has size `N+1-r`.
   - **Up-triangle**. Store `Hex(q, r)` at `array[r][q - N+1+r]`. Row `r` has size `1+r`.

   For flat-topped hexes, swap the roles of the rows and columns, and use `array[q - first_column][r - first_row(q)]`.

Encapsulate access into the getter/setter in a map class so that the rest of the game doesn't need to know about the map storage. Your maps may not look exactly like these, so you will have to adapt one of these approaches.

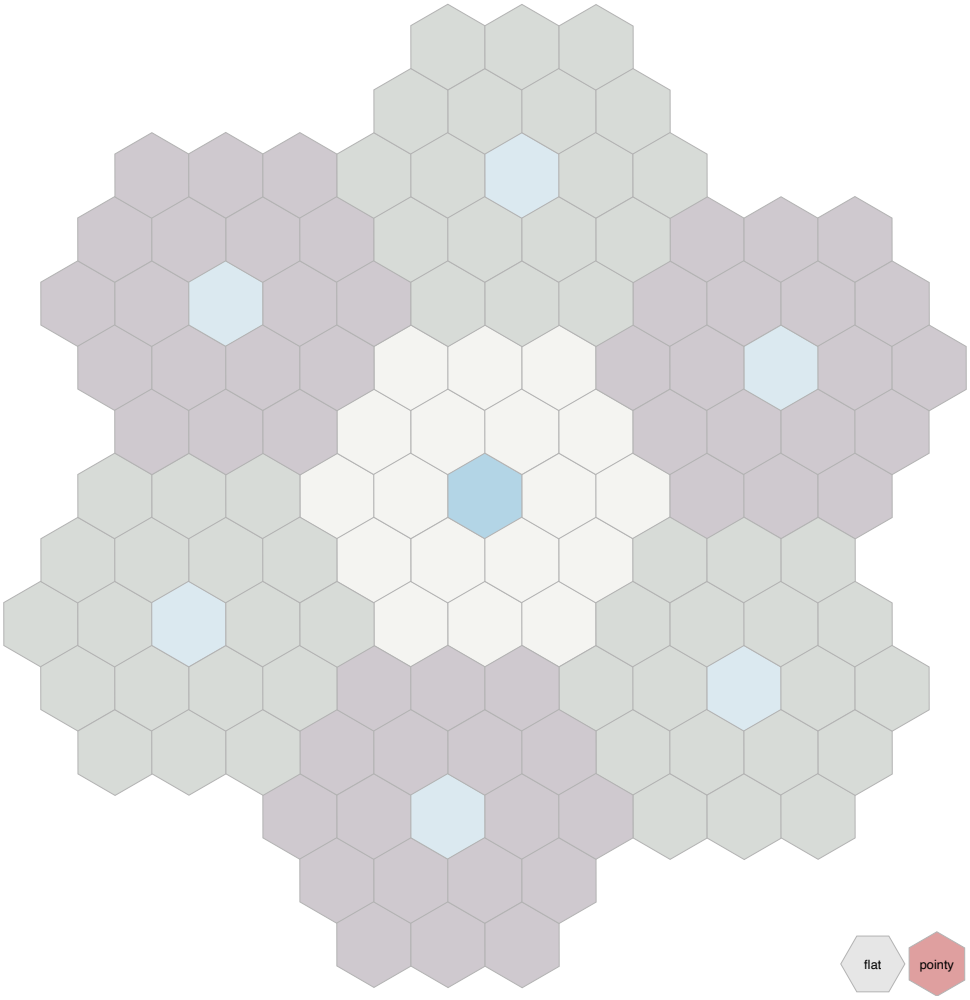
Wraparound maps

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[41] for another approach.

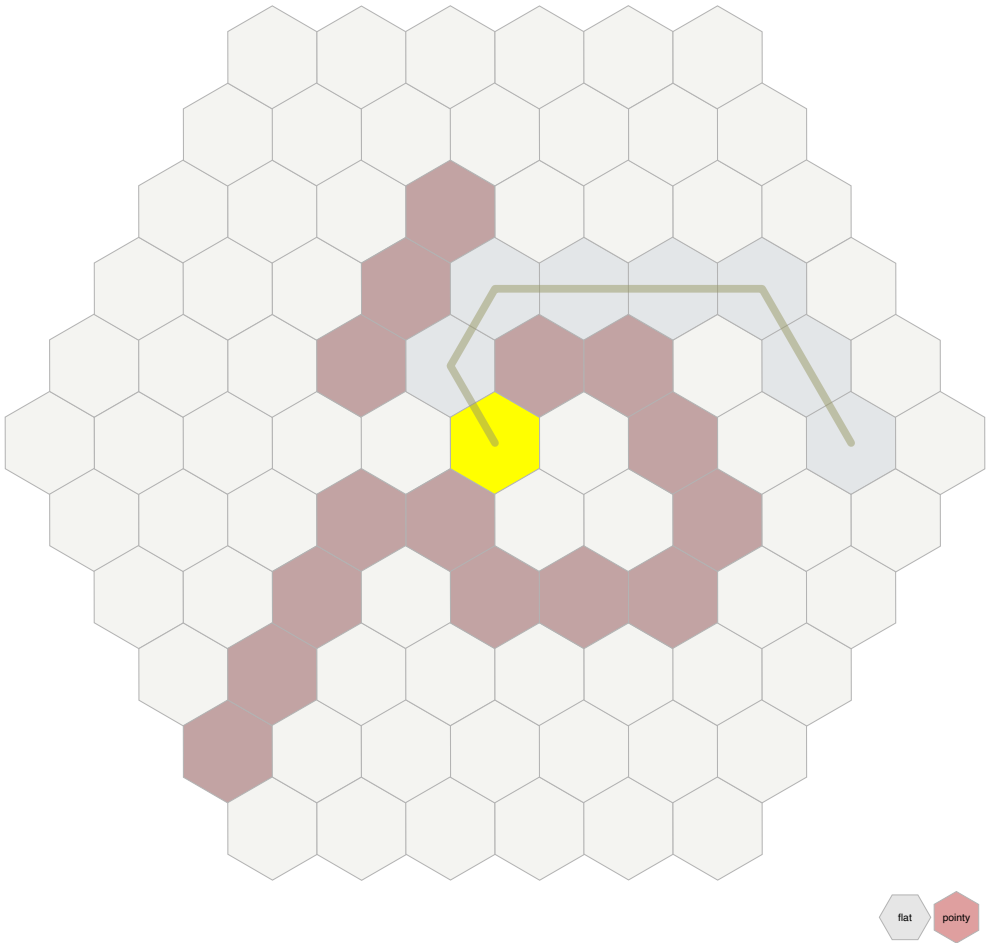
For a hexagonal shaped map with radius `N`, the mirror centers will be `Cube(2*N+1, -N, -N-1)` and its six rotations.



Related: Sander Evers has a nice explanation of how to combine small hexagons into a grid of large hexagons[42] and also a coordinate system to represent small hexagons within a larger one[43].

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[44] 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.

- The best early guide I saw to the axial coordinate system was Clark Verbrugge's guide[45], written in 1996.
- The first time I saw the cube coordinate system was from Charles Fu's posting to rec.games.programmer[46] in 1994.
- DevMag has a nice visual overview of hex math[47] including how to represent areas such as half-planes, triangles, and quadrangles. There's a PDF version here[48] that goes into more detail. **Highly recommended**! The GameLogic Grids[49] library implements these and many other grid types in Unity.
- In my Guide to Grids, I cover axial coordinate systems to address square, triangle, and hexagon sides and corners, and algorithms for the relationships among tiles, sides, and corners. I also show how square and hex grids are related.
- James McNeill has a nice visual explanation of grid transformations[50].
- Overview of hex coordinate types[51]: staggered (offset), interlaced, 3d (cube), and trapezoidal (axial).
- The Rot.js library[52] has a list of hex coordinate systems: non-orthogonal (axial), odd shift (offset), double width (interlaced), cube.
- Range for cube coordinates[53]: given a distance, which hexagons are that distance from the given one?
- Distances on hex grids[54] using cube coordinates, and reasons to use cube coordinates instead of offset.
- Convert cube hex coordinates to pixel coordinates[55].
- This thread[56] explains how to generate rings.
- The HexPart[57] 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[58]?
- Line of sight in a hex grid[59] with offset coordinates, splitting hexes into triangles

- Hexnet explains how the correspondence between hexagons and cubes[60] goes much deeper than what I described on this page, generalizing to higher dimensions.
- I used the PDF hex grids from this page[61] while working out some of the algorithms.
- Hexagonal Image Processing[62] (DOI[63]) is an entire book that uses a hierarchical hexagonal coordinate system.
- Hex-Grid Utilities[64] is a C# library for hex grid math, with neighbors, grids, range finding, path finding, field of view. Open source, MIT license.
- This is the oldest reference I can find for axial grids: Luczak, E. and Rosenfeld, A., *Distance on a Hexagonal Grid*. IEEE Transactions on Computers (1976) (DOI[65]) It calls the axial system *oblique coordinates* and the offset systems *pseudohexagonal grids*.
- Snyder, Qi, Sander's paper *Coordinate system for hexagonal pixels* (DOI[66]) describes gradients, diffusion, and map storage for axial coordinates. Mersereau's paper *The processing of hexagonally sampled two-dimensional signals* (DOI[67]) describes signal processing on axial coordinates.
- There's a paper that calls cube coordinates **R3 coordinates*: Her, I., *Geometric transformations on the hexagonal grid*, IEEE Transactions on Image Processing (1995) (DOI[68]) It covers coordinates, correspondence to cube coordinates, rounding, reflections, scaling, shearing, and rotation. A paper from the same author (DOI[69]) covers distances.
- The Reddit discussion[70] and Hacker News discussion[71] and MetaFilter discussion[72] have more comments and links.

The code that powers this page is partially procedurally generated! The core algorithms are in lib.js, generated from my guide to implementation. There are a few more algorithms in hex-algorithms.js. The interactive diagrams are in diagrams.js and index.js, using Vue.js to inject into the templates in index.bxml (xhtml I feed into a preprocessor).

There are more things I want to do for this guide. I'm keeping a list on Notion[73]. Do you have suggestions for things to change or add? Comment below.

Email me redblobgames@gmail.com, or tweet @redblobgames, or comment:

Endnotes

[1]: http://www-cs-students.stanford.edu/~amitp/gameprog.html#hex
[2]: http://www-cs-students.stanford.edu/~amitp/Articles/Hexagon2.html
[3]: http://www-cs-students.stanford.edu/~amitp/Articles/HexLOS.html
[4]: http://www-cs-students.stanford.edu/~amitp/game-programming/grids/
[5]: https://en.wikipedia.org/wiki/Hexagon#Parameters
[6]: http://web.archive.org/web/20090205120106/http://sc.tri-bit.com/Hex_Grids
[7]: http://ondras.github.io/rot.js/manual/#hex/indexing
[8]: https://www.researchgate.net/publication/235779843_Storage_and_addressing_scheme_for_practical_hexagonal_image_processing?_sg=fIKEA6rk1KmOpC4LBjQJN_-NBuiR1KJtJt-XeYRXnd0z_MNUrB2gjb2FKV3iBoKg988P2xHCpQ
[9]: https://gamedev.stackexchange.com/questions/71785/converting-between-spiral-honeycomb-mosaic-and-axial-hex-coordinates
[10]: https://tex.stackexchange.com/questions/275490/is-there-an-easy-way-to-number-a-hexagonal-spiral
[11]: https://opus.lib.uts.edu.au/bitstream/2100/280/11/02Whole.pdf
[12]: http://www.pyxisinnovation.com/pyxwiki/index.php?title=Generalized_Balanced_Ternary
[13]: https://www.sciencedirect.com/science/article/pii/0166218X9200186P
[14]: http://old.reddit.com/r/gamedev/comments/19wmvn/a_data_structure_for_a_game_board_with_hexagonal/c8s9qbe
[15]: https://en.wikipedia.org/wiki/Bitwise_operation#AND
[16]: https://en.wikipedia.org/wiki/Modulo_operation

[17]: https://www.researchgate.net/publication/235779843_Storage_and_addressing_scheme_for_practical_hexagonal_image_processing?_sg=flKEA6rk1KmOpC4LBjQJN_-NBuiR1KJtJt-XeYRXnd0z_MNUrB2gjb2FKV3iBoKg988P2xHCpQ
[18]: https://doi.org/10.1117/1.JEI.22.1.010502
[19]: http://3dmdesign.com/development/hexmap-coordinates-the-easy-way
[20]: http://ondras.github.io/rot.js/manual/#hex/indexing
[21]: http://ondras.github.io/rot.js/manual/#hex/indexing
[22]: https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)
[23]: http://zvold.blogspot.com/2010/02/line-of-sight-on-hexagonal-grid.html
[24]: http://stackoverflow.com/questions/3233522/elegant-clean-special-case-straight-line-grid-traversal-algorithm
[25]: https://doi.org/10.1111/1467-8659.1210027
[26]: http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/
[27]: https://en.wikipedia.org/wiki/Cuboid
[28]: http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/
[29]: http://gamedev.stackexchange.com/questions/15237/how-do-i-rotate-a-structure-of-hexagonal-tiles-on-a-hexagonal-grid/
[30]: https://en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF
[31]: http://www-cs-students.stanford.edu/~amitp/Articles/HexLOS.html
[32]: https://github.com/jbochi/duelo
[33]: https://s3.amazonaws.com/jbochi/layout.html
[34]: http://www.adammil.net/blog/v125_Roguelike_Vision_Algorithms.html
[35]: http://www.roguebasin.com/index.php?title=Pre-Computed_Visibility_Tries
[36]: http://www.roguebasin.com/index.php?title=Field_of_Vision
[37]: http://www.wolframalpha.com/input/?i=inv+%7B%7Bsqrt%283%29%2C+sqrt%283%29%2F2%7D%2C+%7B0%2C+3%2F2%7D%7D
[38]: http://www.wolframalpha.com/input/?i=inv+%7B%7B3%2F2%2C+0%7D%2C+%7Bsqrt%283%29%2F2%2C+sqrt%283%29%7D%7D
[39]: https://observablehq.com/@jrus/hexround

[40]: https://bl.ocks.org/patricksurry/0603b407fa0a0071b59366219c67abca
[41]: http://gamedev.stackexchange.com/a/137603/2472
[42]: https://observablehq.com/@sanderevers/hexagon-tiling-of-an-hexagonal-grid
[43]: https://observablehq.com/@sanderevers/hexmod-representation
[44]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[45]: http://www-cs-students.stanford.edu/~amitp/Articles/HexLOS.html
[46]: http://www-cs-students.stanford.edu/~amitp/Articles/Hexagon2.html
[47]: http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/
[48]: http://www.gamelogic.co.za/downloads/HexMath2.pdf
[49]: http://gamelogic.co.za/grids/documentation-contents/quick-start-tutorial/gamelogics-hex-grids-for-unity-and-amit-patels-guide-for-hex-grids/
[50]: http://playtechs.blogspot.com/2007/04/hex-grids.html
[51]: http://web.archive.org/web/20090205120106/http://sc.tri-bit.com/Hex_Grids
[52]: http://ondras.github.io/rot.js/manual/#hex/indexing
[53]: http://stackoverflow.com/questions/2049196/generating-triangular-hexagonal-coordinates-xyz
[54]: http://keekerdc.com/2011/03/hexagon-grids-coordinate-systems-and-distance-calculations/
[55]: http://stackoverflow.com/questions/2459402/hexagonal-grid-coordinates-to-pixel-coordinates
[56]: http://gamedev.stackexchange.com/questions/51264/get-ring-of-tiles-in-hexagon-grid
[57]: http://www.battleanalysis.com/battlefield.html
[58]: http://gamedev.stackexchange.com/questions/49718/vertical-vs-horizontal-hex-grids-pros-and-cons
[59]: http://arges-systems.com/blog/2011/01/10/hex-grid-line-of-sight-revisited/
[60]: http://hexnet.org/content/permutohedron
[61]: http://incompetech.com/graphpaper/hexagonal/
[62]: https://www.springer.com/us/book/9781852339142
[63]: https://doi.org/10.1007/1-84628-203-9

[64]: http://hexgridutilities.codeplex.com/documentation
[65]: https://doi.org/10.1109/TC.1976.1674642
[66]: https://doi.org/10.1117/12.348629
[67]: https://doi.org/10.1109/PROC.1979.11356
[68]: https://doi.org/10.1109/83.413166
[69]: https://doi.org/10.1115/1.2919210
[70]: http://old.reddit.com/r/gamedev/comments/1dz1tr/
[71]: https://news.ycombinator.com/item?id=5809724
[72]: http://www.metafilter.com/128649/Hexagonal-Grids
[73]: https://www.notion.so/redblobgames/f8bc2f44fba94607afa9c06711d23245?v=0766432cb1534ce582ce35b33cbbef7e&p=7d2d4d624bc5483dafbe615d75ab3902

RSS Feed

Created 11 Mar 2013 with Haxe and D3.js ; updated in 2018 to use Vue.js ;
Last modified: 19 Apr 2022