

Hexagonal Grids

from [Red Blob Games](#)

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

2022, Feb 2023, Oct 2024, Mar 2025

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. To get an offline copy of this page, use your browser's File → Save As (preserves interactivity) or File → Print (loses interactivity).

● [Geometry](#)

● [Coordinate systems](#)

● [Conversions](#)

● [Neighbors](#)

● [Distances](#)

● [Line drawing](#)

● [Range](#)

● [Rotation](#)

● [Reflection](#)

● [Rings and Spirals](#)

● [Field of view](#)

● [Hex to pixel](#)

● [Pixel to hex](#)

● [Rounding](#)

● [Map storage](#)

● [Wraparound maps](#)

● [Pathfinding](#)

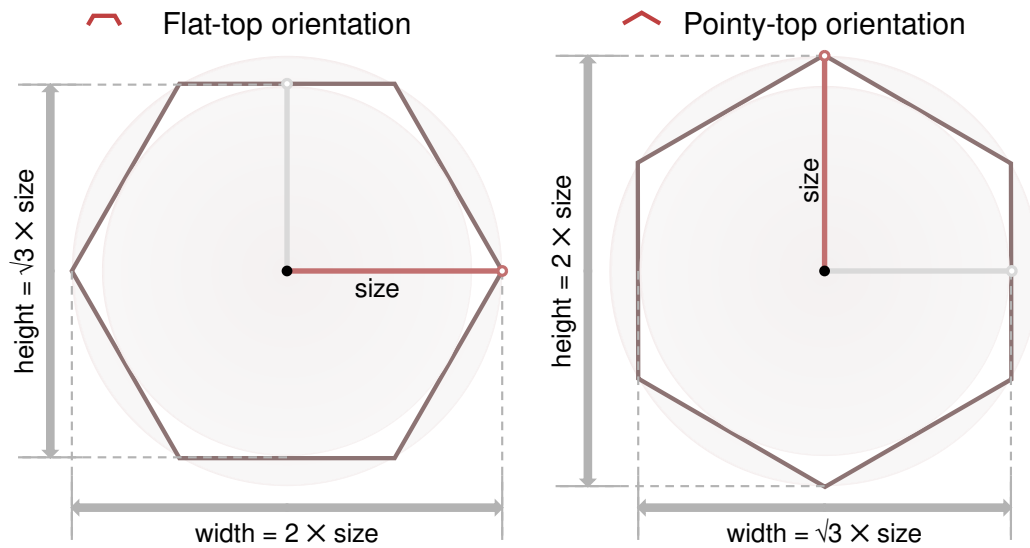
● [More reading](#)

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 any 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.

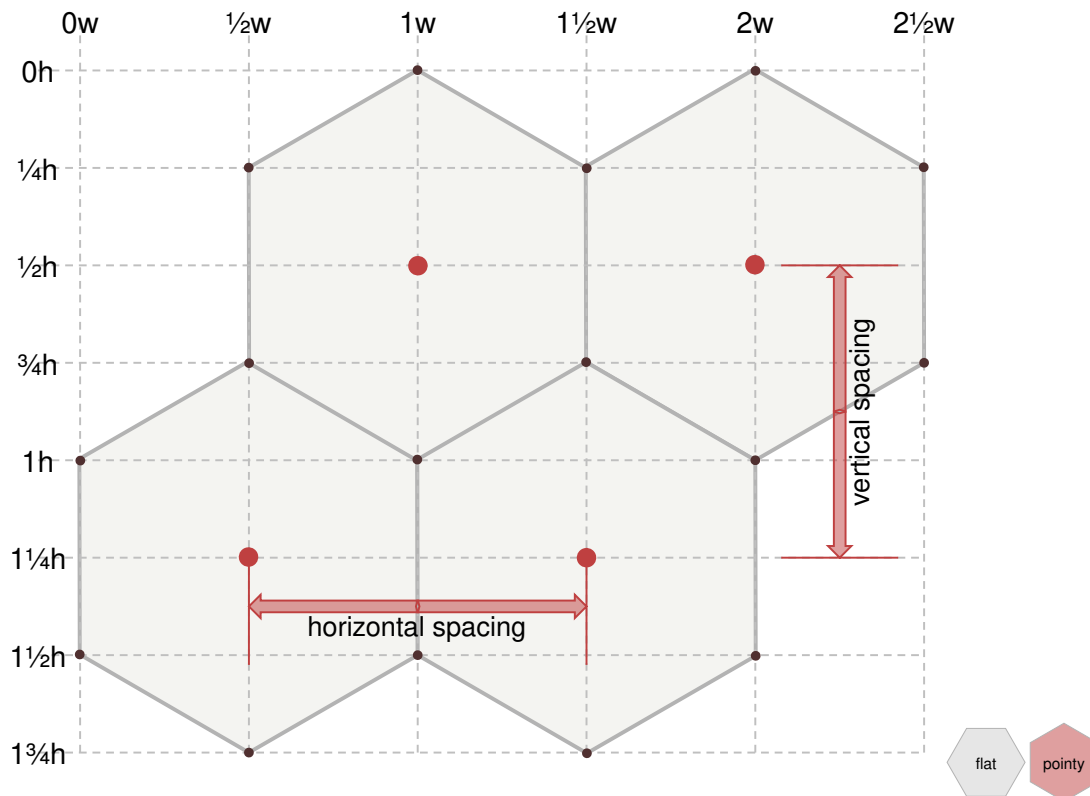


The size of a regular hexagon can be described by either the inner circle[?], touching the edges, or the outer circle[?], touching the corners. On this page, I call the outer radius "**size**". The width[?] and height[?] are defined in terms of the diameters of the two circles.

Spacing

#

Next we want to put several hexagons together. The spacing will depend on both the outer circle's radius (`size`) and the inner circle's radius (`inradius`).



In the **flat top** orientation, the horizontal distance between adjacent hexagons centers is **horiz** = $\frac{3}{4} * \text{width} = \frac{3}{2} * \text{size}$. The vertical distance is **vert** = height = $\sqrt{3} * \text{size} = 2 * \text{inradius}$.

In the **pointy top** orientation, the horizontal distance between adjacent hexagon centers is **horiz** = width == $\sqrt{3} * \text{size} == 2 * \text{inradius}$. The vertical distance is **vert** = $\frac{3}{4} * \text{height} == \frac{3}{2} * \text{size}$.

Some games use pixel art for hexagons that does not match an exactly regular polygon, and these formulas will have to be adjusted slightly. We'll do that in the [hex to pixel](#) and [pixel to hex](#) functions.

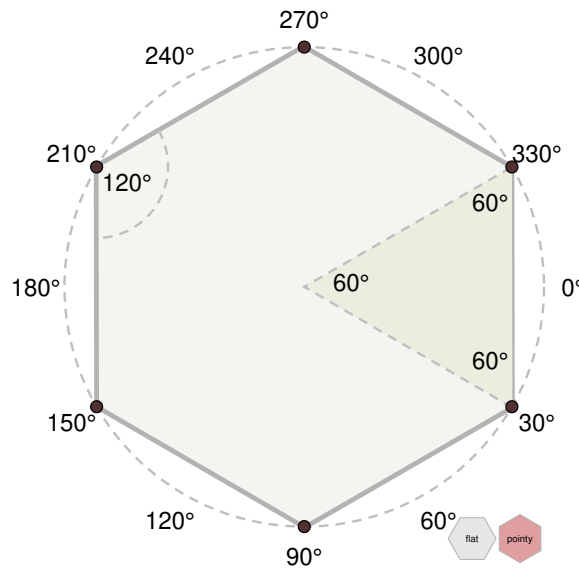
Angles

#

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)`.

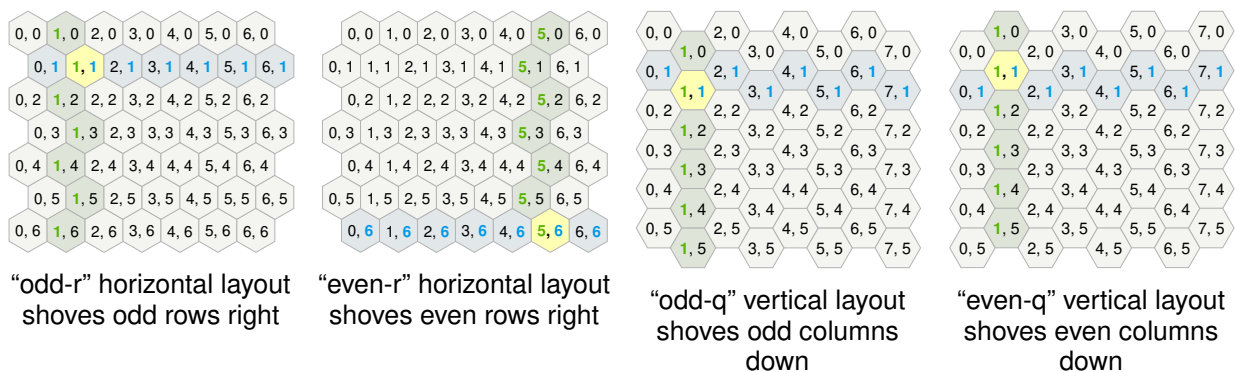
The difference between the two orientations is a rotation, and that causes the angles to change: **flat top** angles are $0^\circ, 60^\circ, 120^\circ, 180^\circ, 240^\circ, 300^\circ$ and **pointy top** angles are $30^\circ, 90^\circ, 150^\circ, 210^\circ, 270^\circ, 330^\circ$. Note that the diagrams 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).

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.



Cube coordinates

#

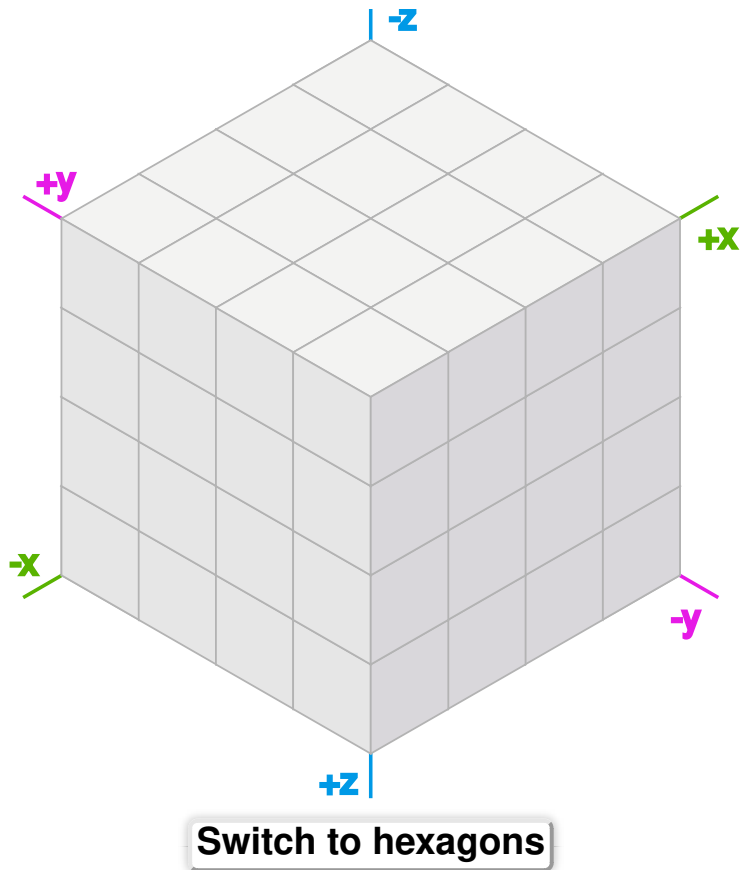
Another way to look at hexagonal grids is to see that there are *three* primary axes, unlike the *two* we have for square grids. There's an elegant symmetry with these.

Let's take a cube grid and **slice** out a diagonal plane at

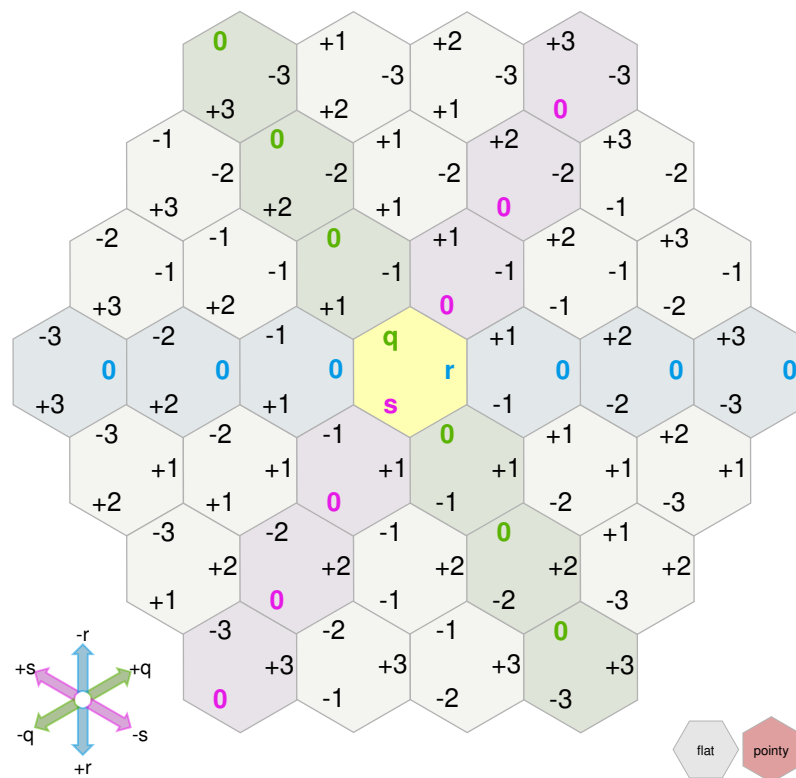
$x + y + z = 0$. This is a *weird* idea but it helps us with hex grid algorithms:

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.



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 to calculate neighbors.

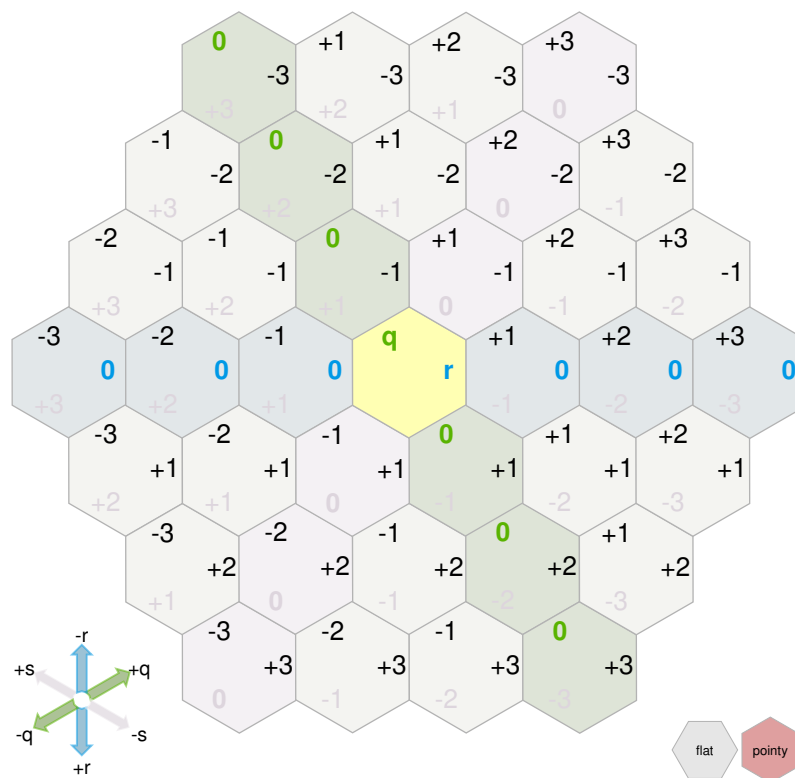
The cube coordinates are a reasonable choice for a hex grid coordinate system. The constraint is that $q + r + s = 0$ so the algorithms must preserve that. The constraint also ensures that there's a canonical coordinate for each hex.

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.

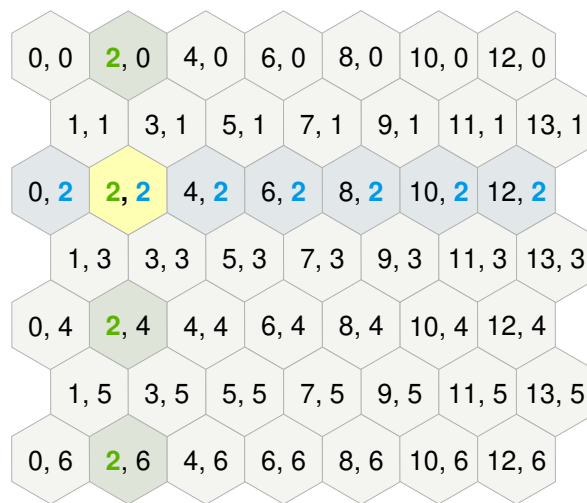


The axial/cube system allows us to add, subtract, multiply, and divide with hex coordinates. The offset coordinate systems do not allow this, and that's part of what makes algorithms simpler with axial/cube coordinates.

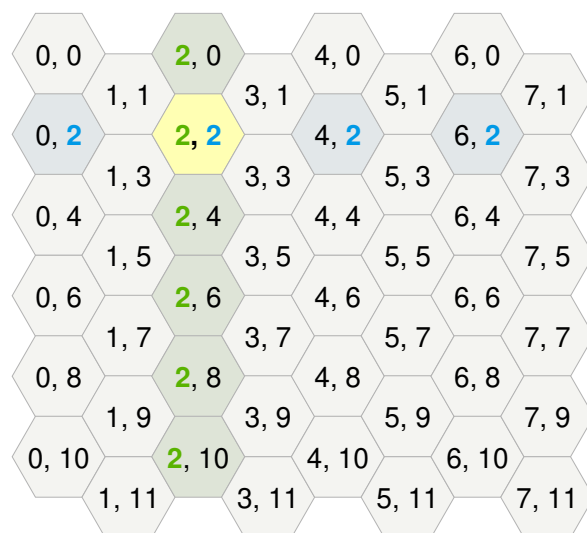
Doubled coordinates

#

Consider doubled instead of offset coordinates. 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 $(col + row) \bmod 2 == 0$. 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](#)^[4], rot.js calls it [double width](#)^[5], and [this paper](#)^[6] 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](#).

In addition to the [flat spiral coordinate systems](#) shown later on this page, there are nested/recursive spiral systems. See [this question](#)^[7] on stackoverflow, or this [Spiral Architecture for Machine Vision](#)^[8] (1996), or this [diagram about "generalized balanced ternary" coordinates](#)^[9], or this [An isomorphism between the p-adic integers and a ring associated with a tiling of N-space by permutohedra](#)^[10] (1994) ([DOI](#)^[11]), or this [discussion on reddit](#)^[12]. There's a [Clojure library](#)^[13] implementing Generalized Balanced Ternary. Also see the Gosper Curve, [here](#)^[14] and [here](#)^[15].

Recommendations

#

What do I recommend?

	Offset	Doubled	Axial	Cube
Pointy rotation	evenr, oddr	doublewidth		
Flat rotation	evenq, oddq	doubleheight	axial	cube
Other rotations	no		yes	

	Offset	Doubled	Axial	Cube
Vector operations (add, subtract, scale)	no	yes	yes	yes
Array storage	rectangular	no [*]	rhombus [*]	no [*]
Hash storage	any shape		any shape	
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**. Note that Axial (q, r) and Cube (q, r, s) are essentially the same system. Store coordinates as Axial, and calculate s in algorithms that need it.

Coordinate conversion

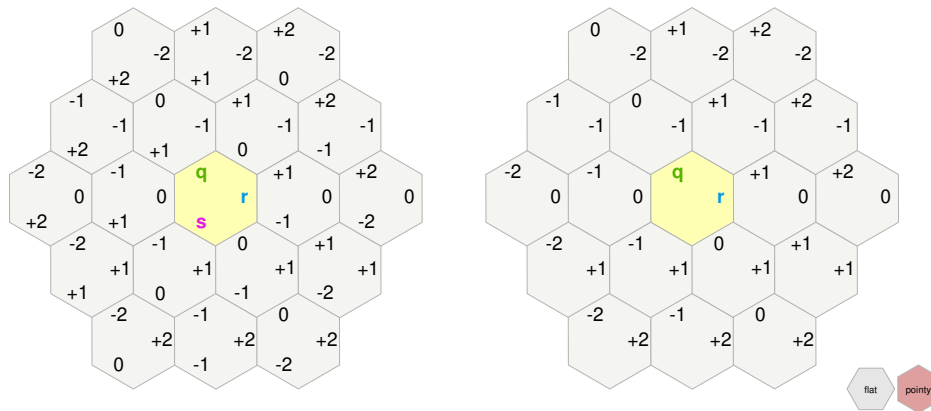
#

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 system. In the Cube system, we store the third coordinate, s . In the Axial system, we 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 only in the algorithms that need it.

Offset coordinates

#

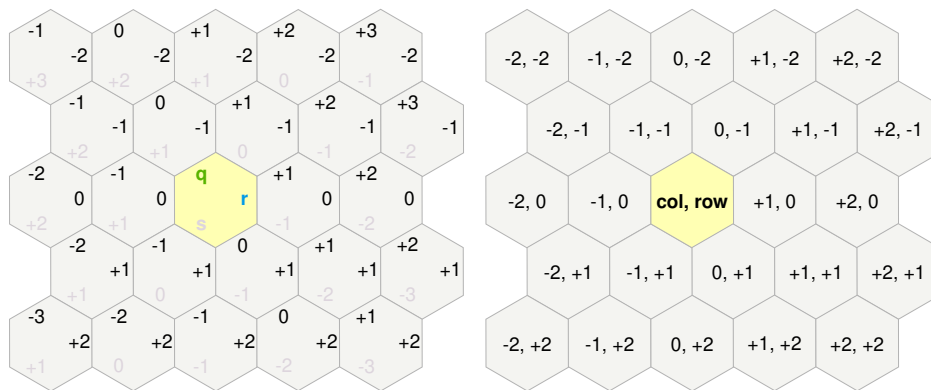
Determine which type of offset system you use; ***-r** are pointy top; ***-q** are flat top. The conversion is different for each.

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

function oddr_to_axial(hex):
    var parity = hex.row&1
```

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

- **odd-r** shoves odd rows by $+\frac{1}{2}$ column
- **even-r** shoves even rows by $+\frac{1}{2}$ column
- **odd-q** shoves odd columns by $+\frac{1}{2}$ row
- **even-q** shoves even columns by $+\frac{1}{2}$ row



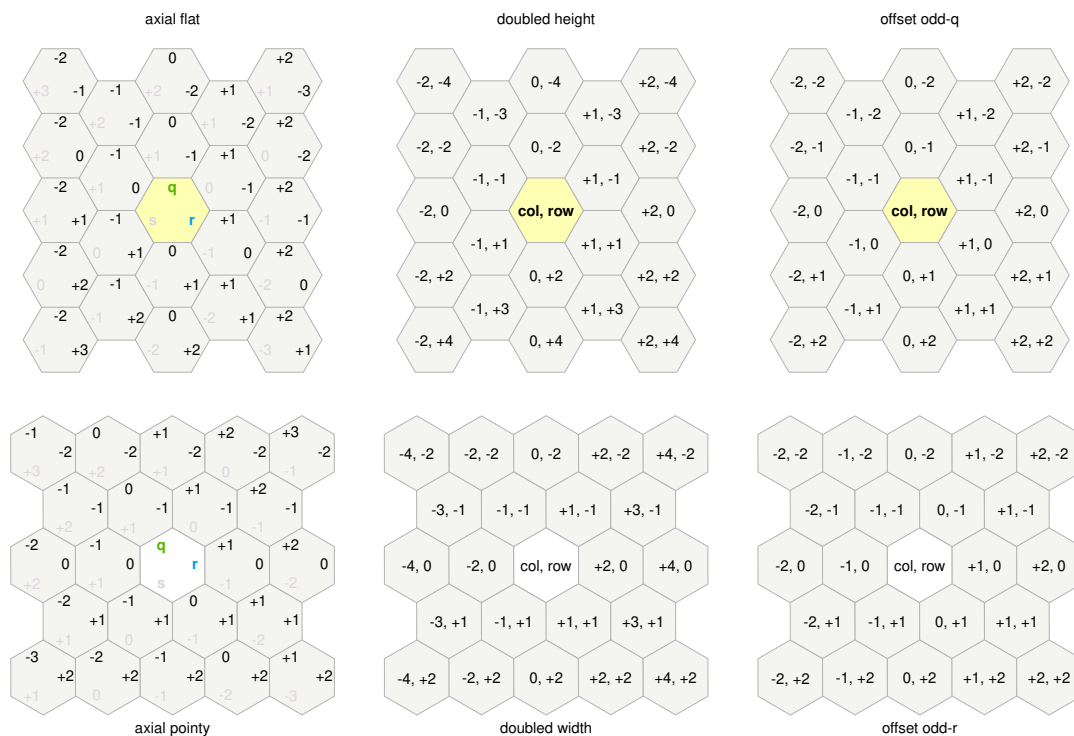
Convert to/from ● **axial** or ○ **cube**.

Implementation note: I use $a \& 1$ ([bitwise and](#)^[16]) instead of $a \% 2$ ([modulo](#)^[17]) 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

#

Compared to offset coordinates, Double height coordinates double the [row](#). Double width coordinates double the [col](#).



```
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)
```

Convert to/from ☒ **axial** or ☐ **cube**.

To convert from doubled to offset, chain the calls, e.g. `ax-`

`ial_to_oddq(doublewidth_to_axial(hex))`, or inline the

call to write a direct conversion function. I include the direct conversion

functions in the sample code in the [implementation guide](#).

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)
]

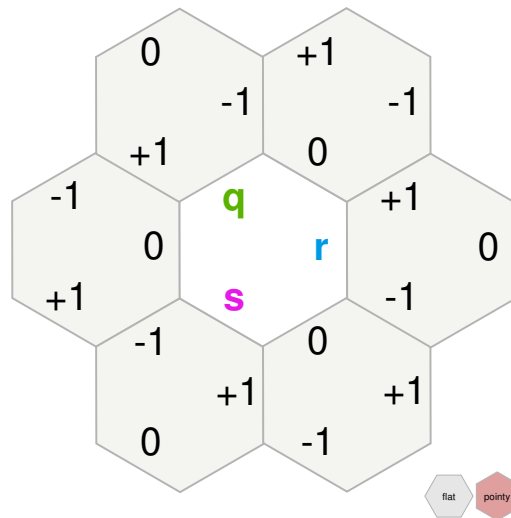
function cube_direction(direction):
    return cube_direction_vectors[direction]

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` func-

tion does. Axial and Doubled coordinates also support this, but the Offset coordinates do not.



Axial coordinates

#

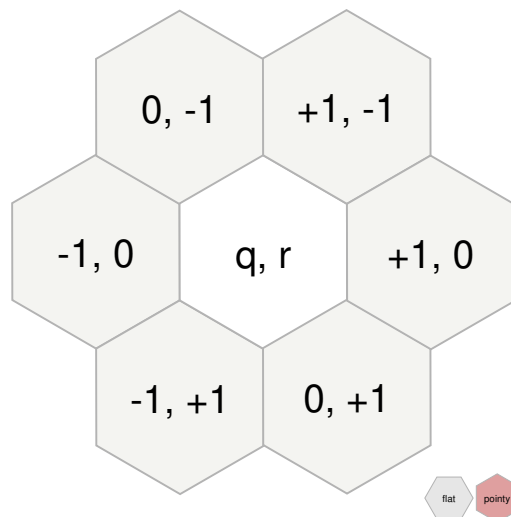
Since axial is the same as cube except not storing the third coordinate, the code is the same as the previous section except we won't write out the third coordinate:

```
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.*

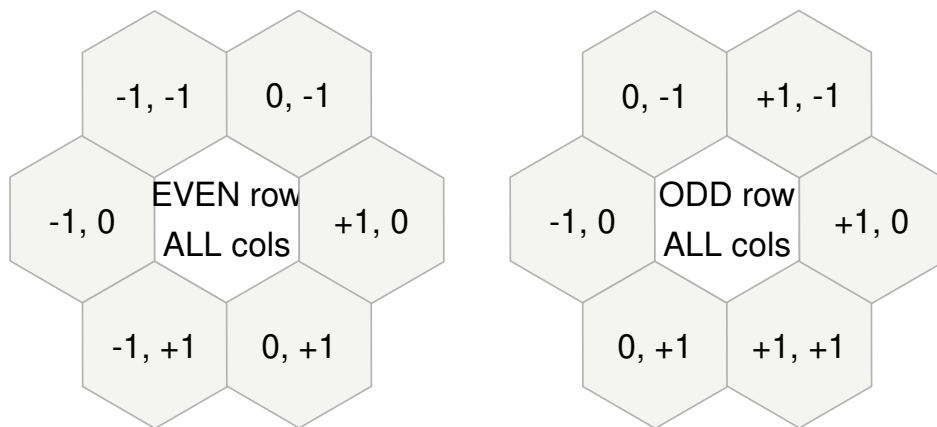
Since the movement vector is different for odd and even columns/rows, we will need two separate lists of neighbors. **Pick a grid type** to see the corresponding code.

```
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 -
```

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

#

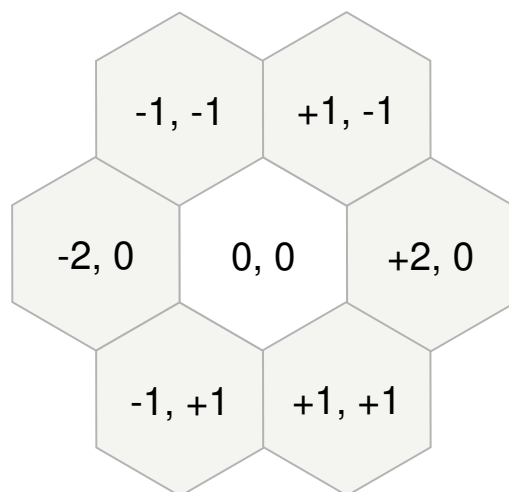
Unlike offset coordinates, the neighbors for doubled coordinates do *not* depend on which column/row we're on. They're the same everywhere, like axial/cube coordinates. Also unlike offset coordinates, we can safely subtract and add doubled coordinates, which makes them easier to work with than offset 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,
    Cube(-2, +1, +1), Cube(-1, +2, -1), Cube(+1,
```

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

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



#

#

```
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))
// or: (abs(a.q - b.q) + abs(a.r - b.r) + abs(a.s - b.s))
```

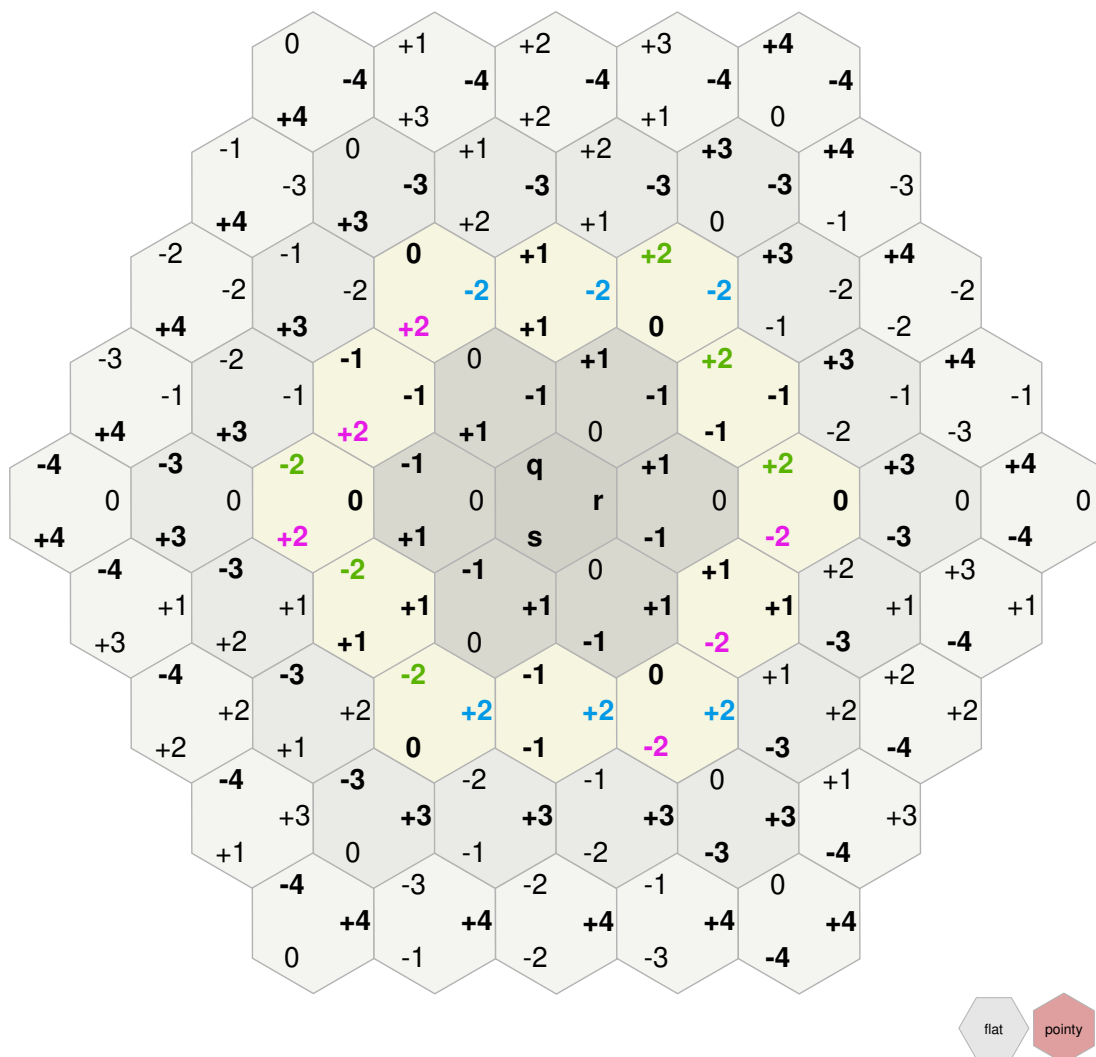
23.09.25, 13:09

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 $\text{abs}(\text{vec.q} - \text{vec.r})$, $\text{abs}(\text{vec.r} - \text{vec.s})$, $\text{abs}(\text{vec.s} - \text{vec.q})$ to figure out which of the 6 “wedges” a hex is in; see [diagrams here](#).



Xiangguo Li's 2013 paper [*Storage and addressing scheme for practical*](#)

[*hexagonal image processing*](#).^[18] ([DOI](#)^[19]) gives a formula for Euclidean distance, which can be adapted to axial coordinates:

$$\text{sqrt}(dq^2 + dr^2 + dq*dr).$$

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”](#)^[20] formula 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^[21] 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^[22]:

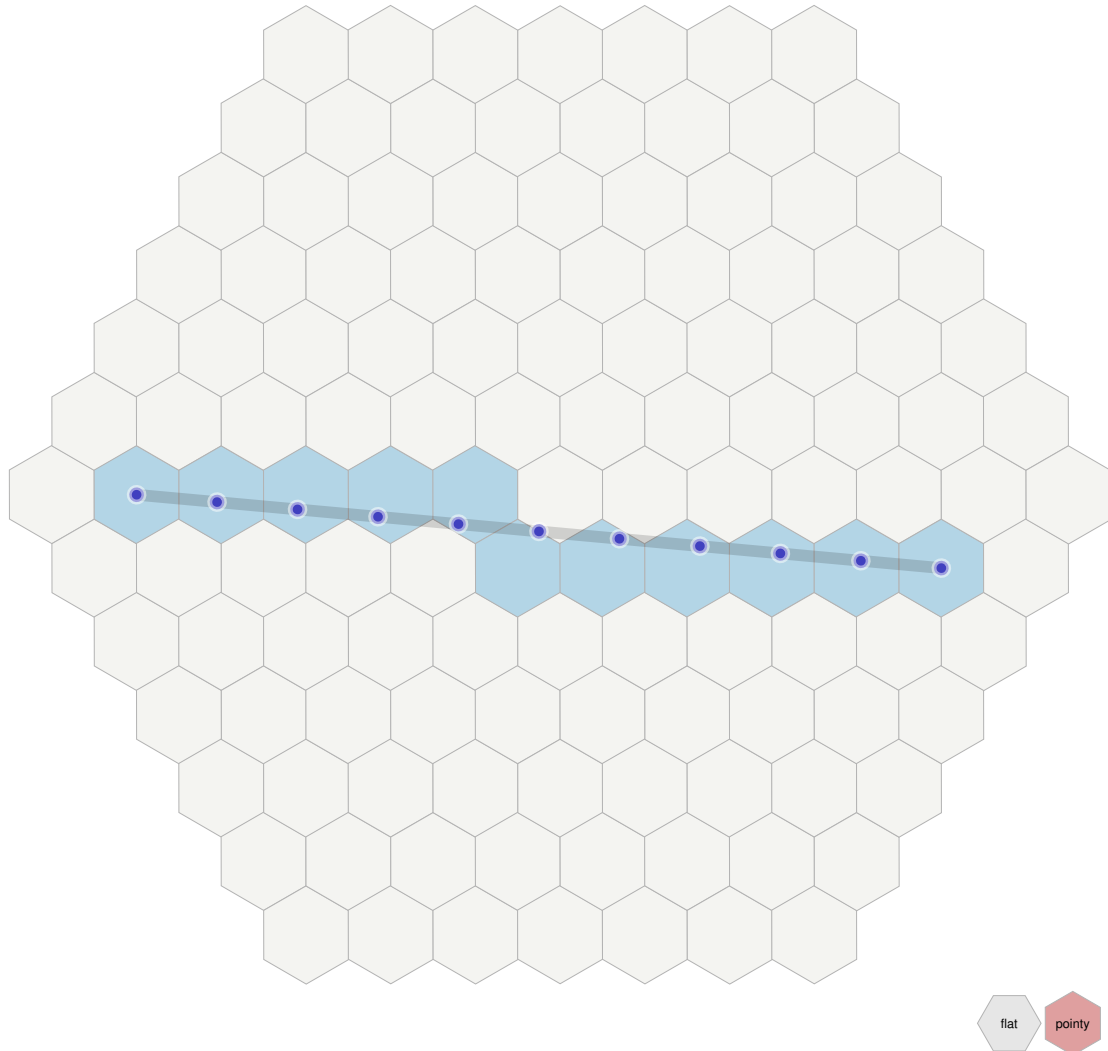
```
function doublewidth_distance(a, b) :
    var dcol = abs(a.col - b.col)
    var drow = abs(a.row - b.row)
    return drow + max(0, (dcol-drow)/2)
```

```
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 end-points.
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, i/N)))
    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](#)^[23] 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](#)^[24].
- There are many variants of line drawing. Sometimes you'll want ["super cover"](#)^[25]. 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](#)^[26]), describes a variant of Bresenham's line drawing algorithm for hexagonal grids. The same author has another paper, *The Generation of Circular Arcs on Hexagonal Grids* ([DOI](#)^[27]).

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  $-N \leq q \leq +N$ :
    for each  $-N \leq r \leq +N$ :
        for each  $-N \leq s \leq +N$ :
            if  $q + r + s == 0$ :
                results.append(cube_add(center, (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  $-N \leq q \leq +N$ :
    for each  $\max(-N, -q-N) \leq r \leq \min(+N, -q+N)$ :
        var  $s = -q-r$ 
        results.append(cube_add(center, Cube(q, r, s)))

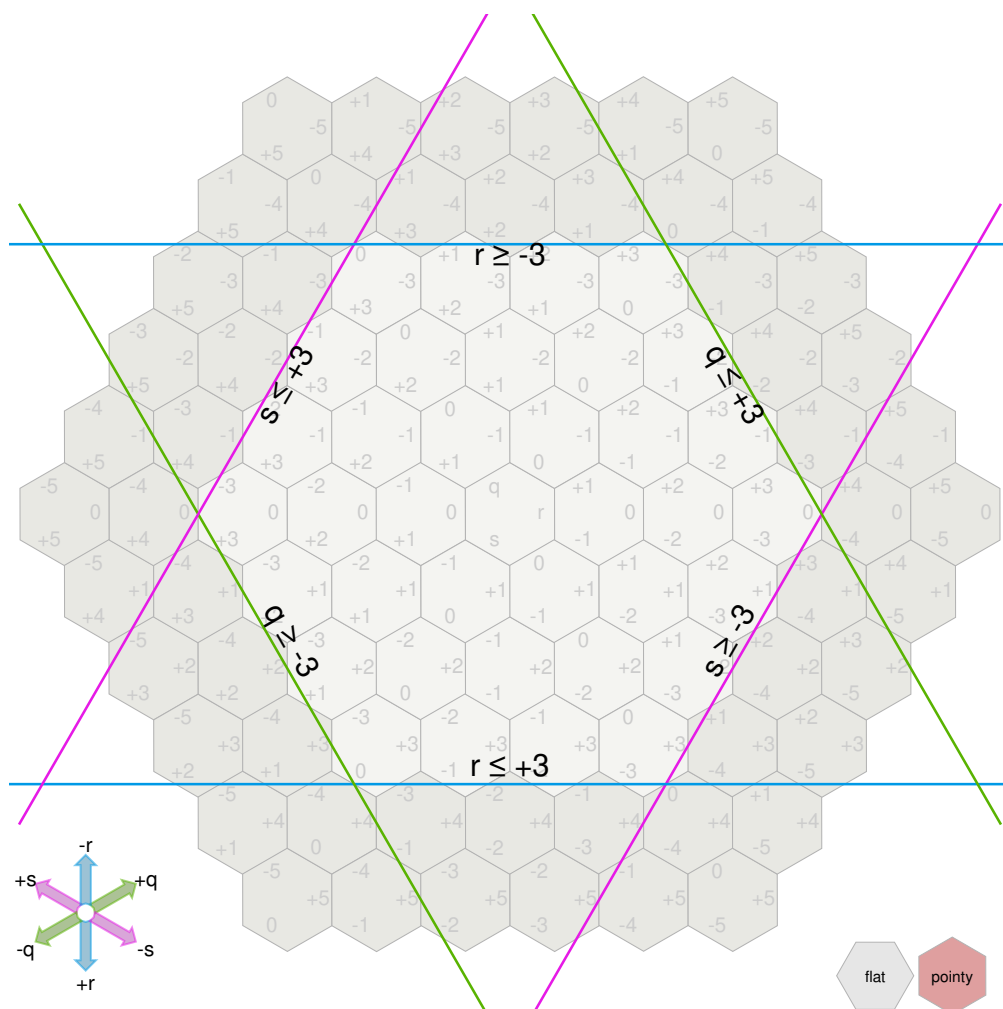
```

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](#)^[28]). We pick all the hexes that satisfy all six inequalities. This loop also works nicely with axial coordinates:

```

var results = []
for each  $-N \leq q \leq +N$ :
    for each  $\max(-N, -q-N) \leq r \leq \min(+N, -q+N)$ :
        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.

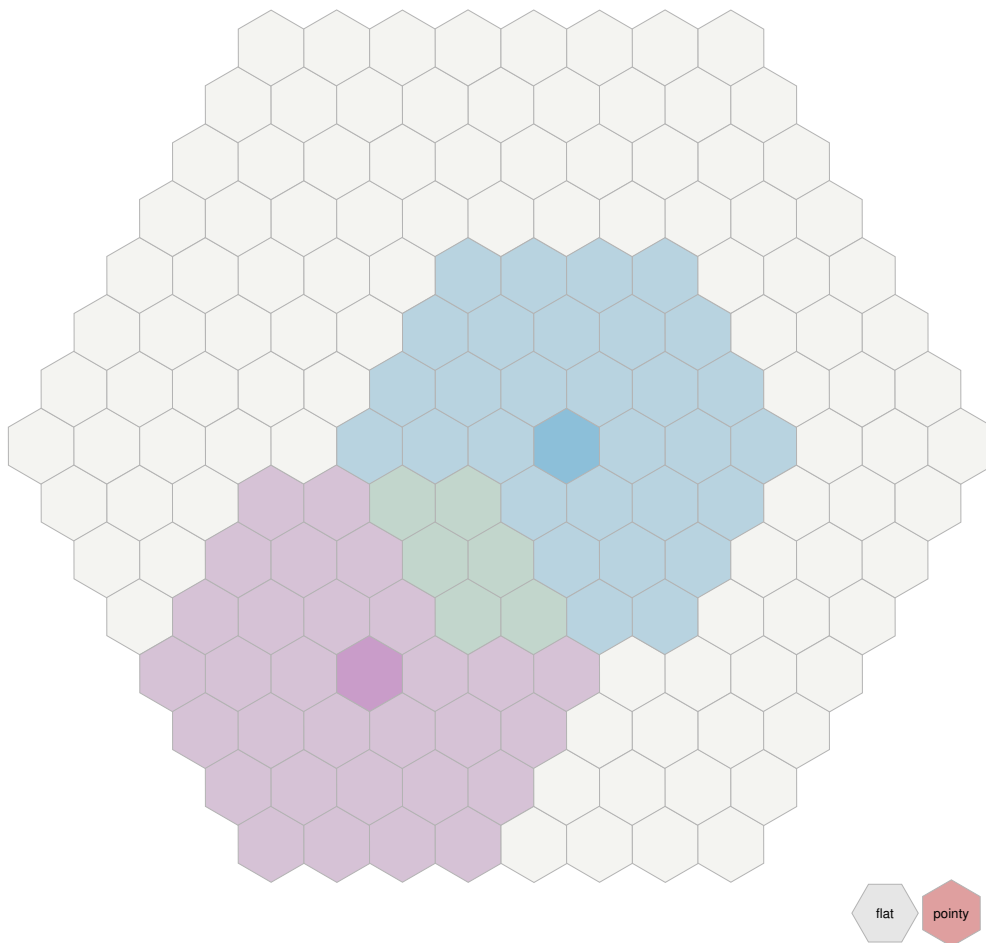
You can either think of this problem algebraically or geometrically. Algebraically, each hexagonally-shaped region is expressed as inequality constraints of the form $-N \leq dq \leq +N$, and we're going to solve for the intersection of those constraints. Geometrically, each region is a cube in 3D space, and we're going to intersect two cubes in 3D space to form a cuboid^[29] 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 $-N \leq dq \leq +N$ into a more general form, $q_{\min} \leq q \leq q_{\max}$, and set $q_{\min} = \text{center.q} - N$ 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  $q_{\min} \leq q \leq q_{\max}$ :
    for each  $\max(r_{\min}, -q-s_{\max}) \leq r \leq \min(r_{\max}, -q$ 
         $\text{results.append}(\text{Hex}(q, r))$ 
```

The intersection of two ranges $a \leq q \leq b$ and $c \leq q \leq d$ is $\max(a, c) \leq q \leq \min(b, d)$. 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^[30] (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 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]:
```

```

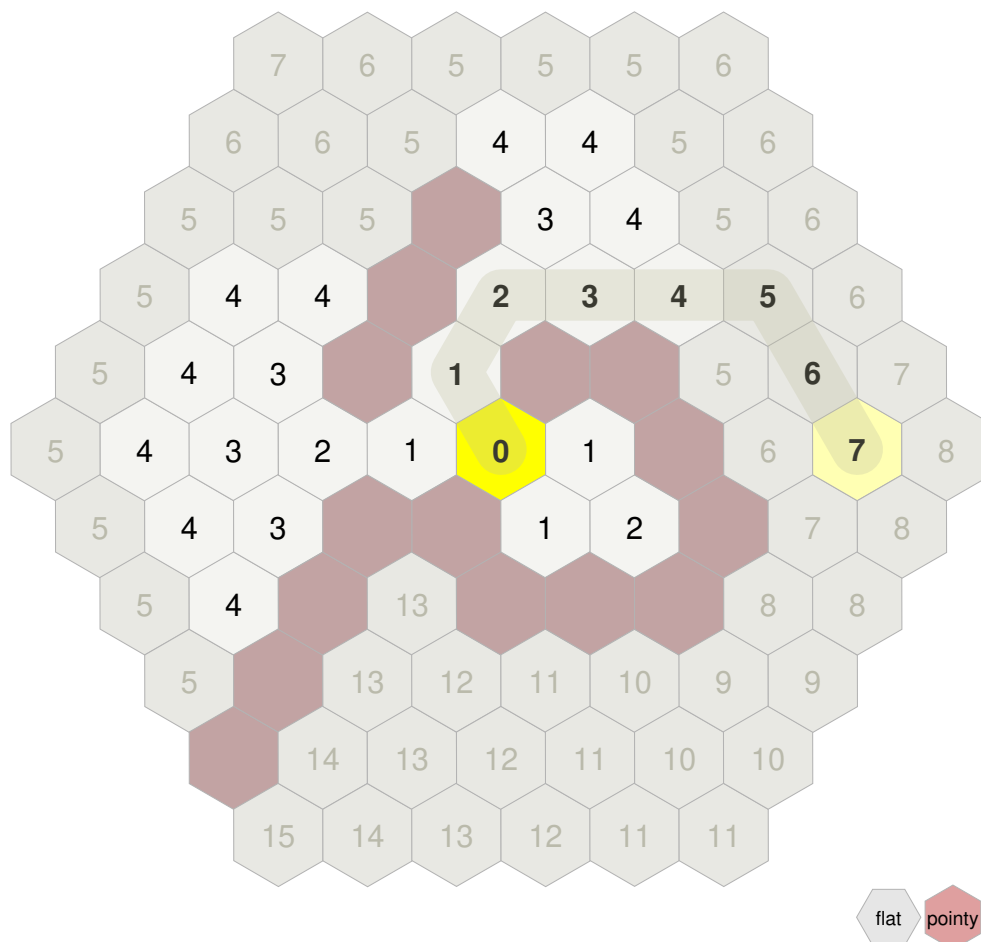
for each 0 ≤ dir < 6:
    var neighbor = hex_neighbor(hex,
    if neighbor not in visited and n
        add neighbor to visited
    fringes[k].append(neighbor)

```

```

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.

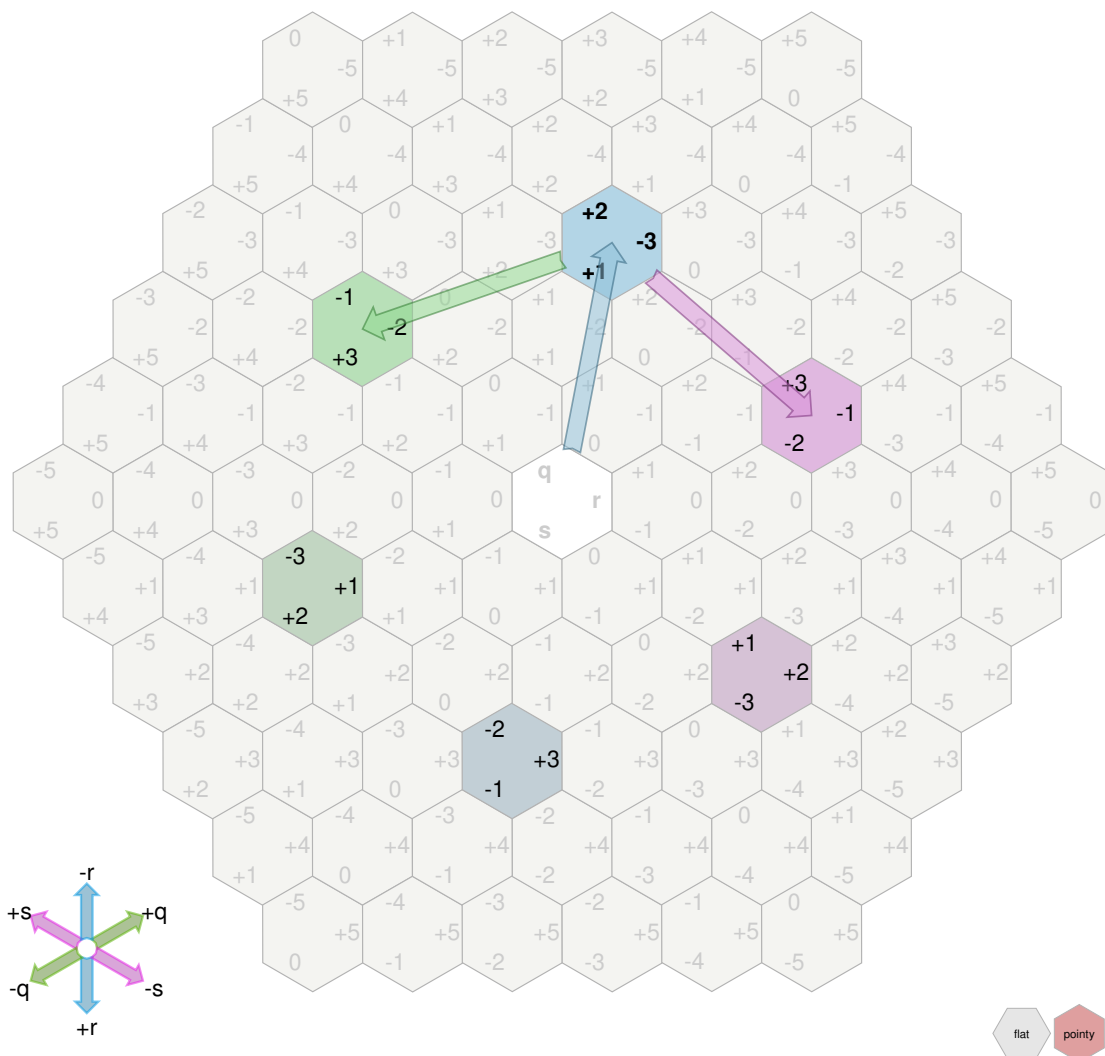
A rotation 60° right (clockwise ↻) shoves each coordinate one slot to

the left \leftarrow :

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

A rotation 60° left (counter-clockwise \curvearrowright) shoves each coordinate one slot to the right \rightarrow :

	[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.

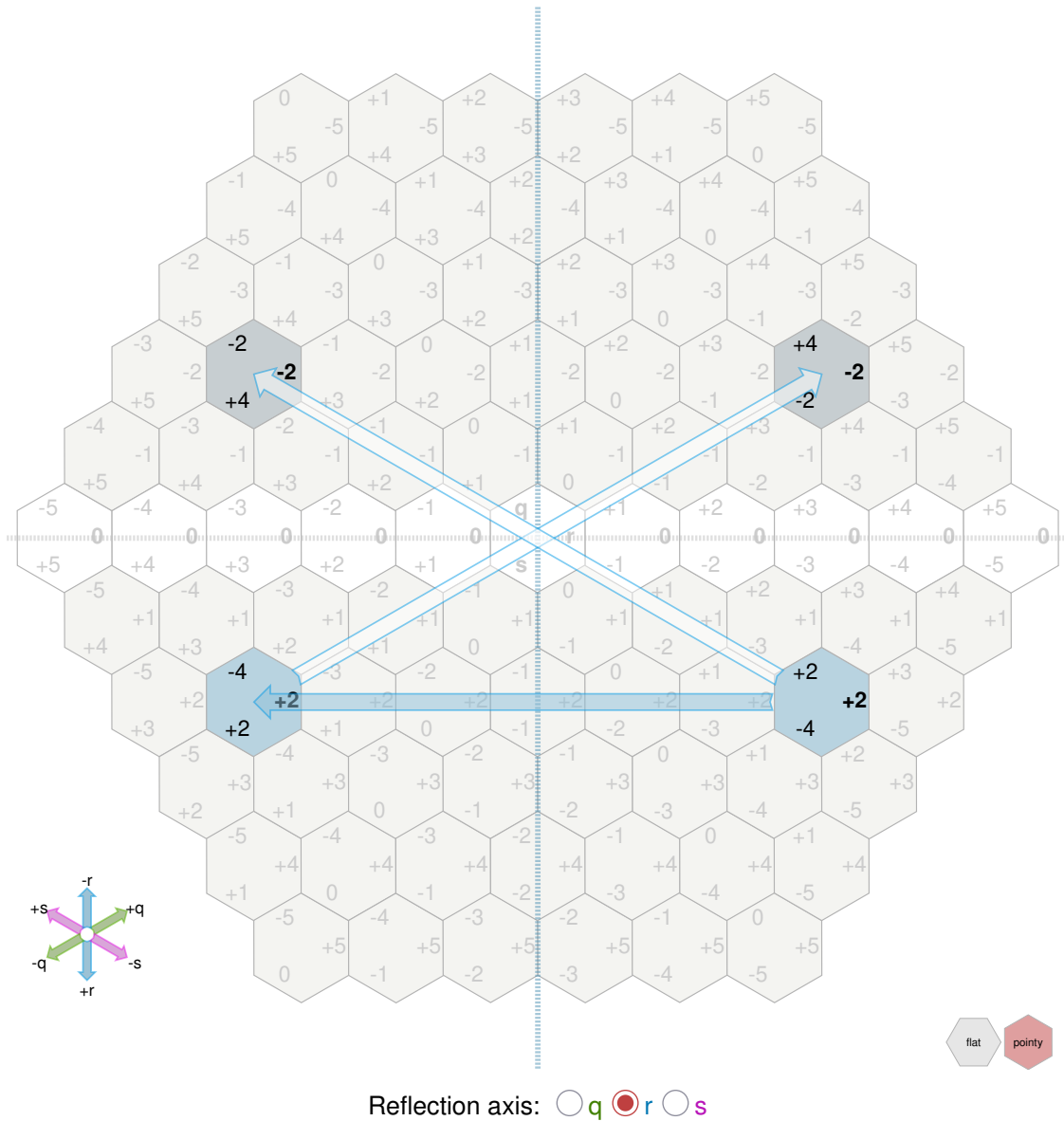
It's several conversion steps but each step is short. You can shortcut some of these steps by defining rotation directly on axial coordinates, but hex vectors don't work for offset coordinates and I don't know a shortcut for offset coordinates. Also see [this stackexchange discussion](#)^[31] for other 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 re-

flecting over. The axis we're reflecting over stays the same.



```
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

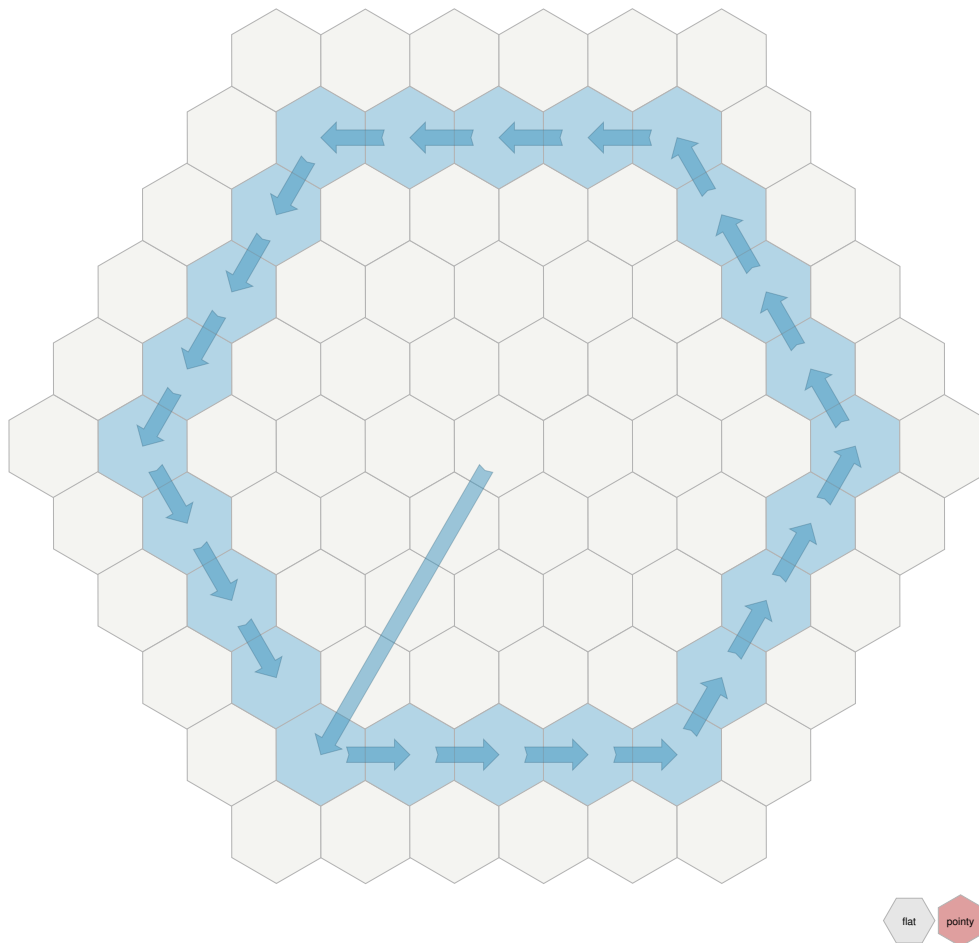
_

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, l

function cube_ring(center, radius):
    var results = []
    # this code doesn't work for radius == 0; can
    var hex = cube_add(center,
                        cube_scale(cube_direction
    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 a circumference of `6 * radius` steps it ends up back where it started.



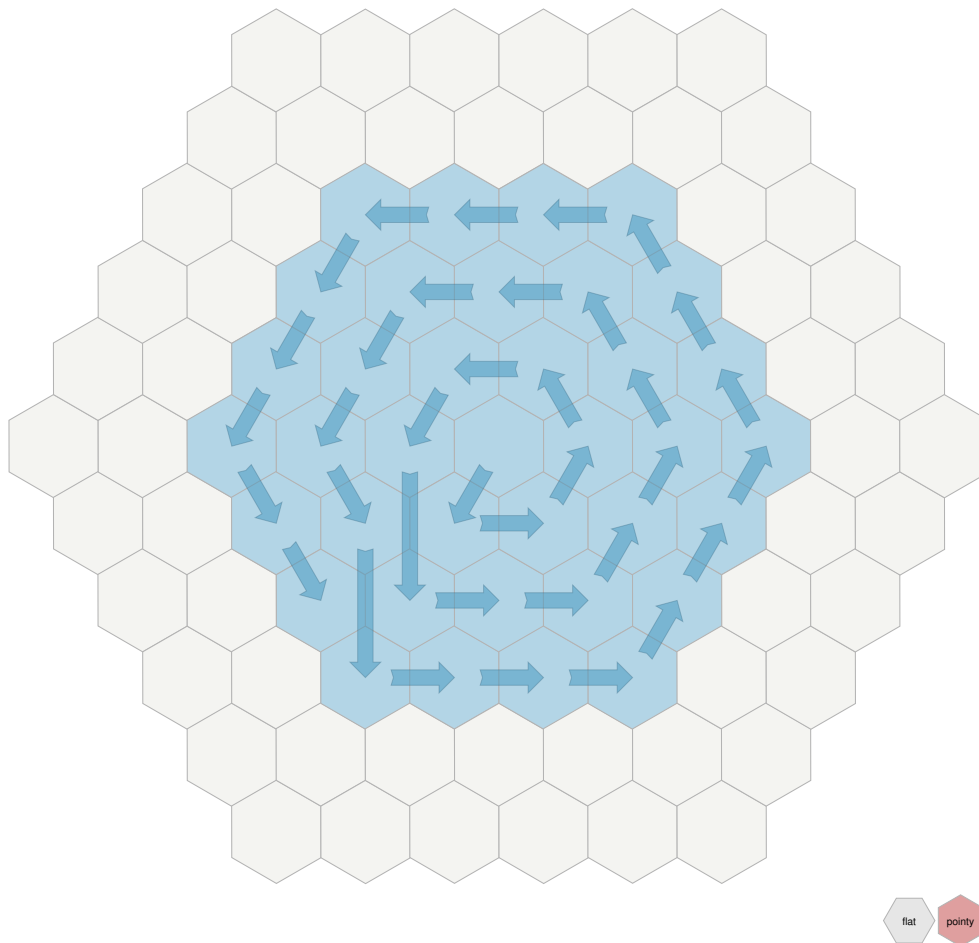
The scale, add, and neighbor operations also work on axial and doubled coordinates, so the same algorithm can be used. For offset coordinates, convert to one of the other formats, generate the ring, and convert back.

Spiral rings

#

Traversing the N 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  $1 \leq k \leq \text{radius}$ :
        results = list_append(results, cube_ring
    return results
```



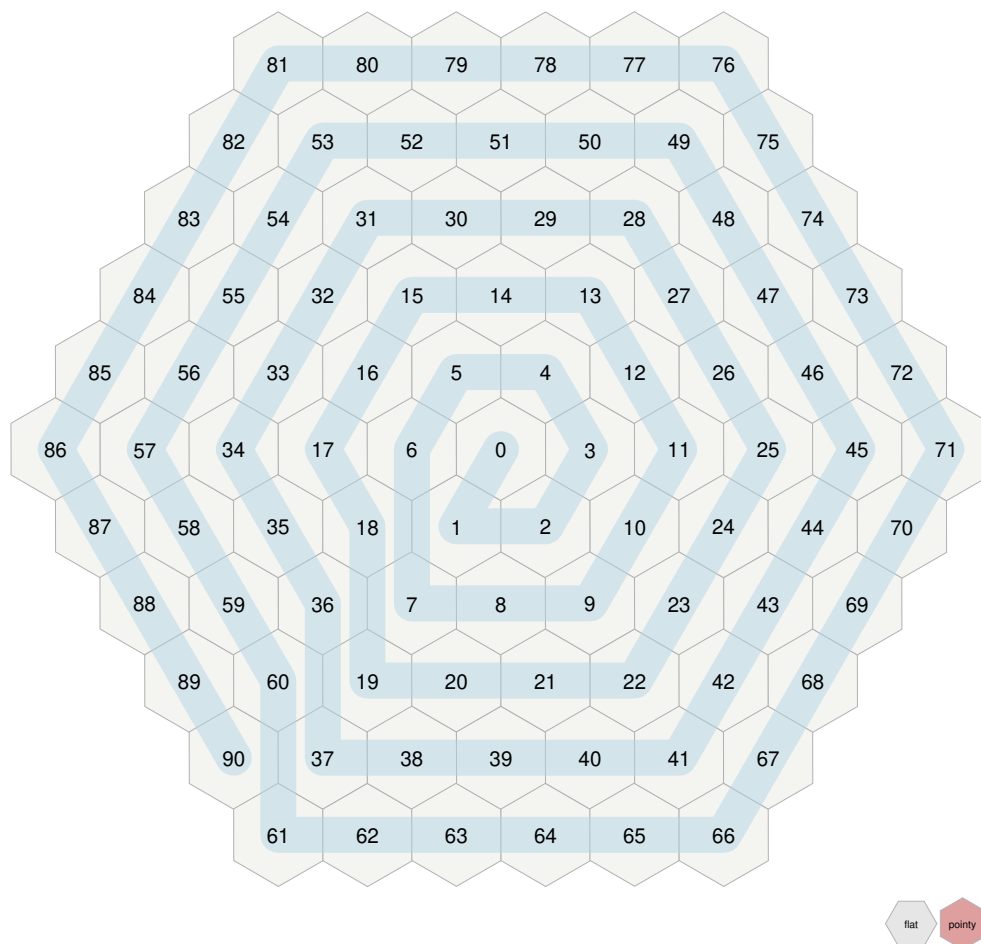
Spirals also give us a way to *count* how many hexagon tiles are in the larger hexagon. The center is 1 hex. The circumference of the k -th ring is $6 * k$ hexes. The sum for N rings is $1 + 6 * \text{sum}(1 \text{ to } N)$. Using [this formula](#)^[32], that simplifies to $1 + 3 * N * (N+1)$. For rings, there will be 61 hexes.

Visiting the hexes this way can also be used to calculate [movement range](#).

Spiral coordinates

#

Following the spiral path above lets us build one of many *spiral coordinate systems*:



The main ingredients we need are the radius[?] (`spiralindex_to_radius`), start of each ring[?] (`spiralindex_start_of_ring`), start of each segment[?], and position within the segments[?].

```
function spiralindex_start_of_ring(radius):
    return 1 + 3 * radius * (radius - 1)

function spiralindex_to_radius(index):
    // solve for 'radius' in equation: index = 1
    return floor((sqrt(12 * index - 3) + 3) / 6)

function spiral_to_cube(index):
    center = Hex(0, 0, 0)
    radius = spiralindex_to_radius(index)
    ringstart = spiralindex_start_of_ring(radius)
    return cube_ring(center, radius)[index - ringstart]
```

```

function cube_to_spiral(hex):
    center = Hex(0, 0, 0)
    radius = cube_distance(hex, center)
    ring_hexes = cube_ring(center, radius)
    for each 0 ≤ i < ring_hexes.length:
        if hex == ring_hexes[i]:
            return i + spiralindex_start_of_ring

```

To convert from a cube/axial coordinate to a spiral index is trickier. In the example `cube_to_spiral` function I linearly search the hexes in a ring, but this can be optimized.

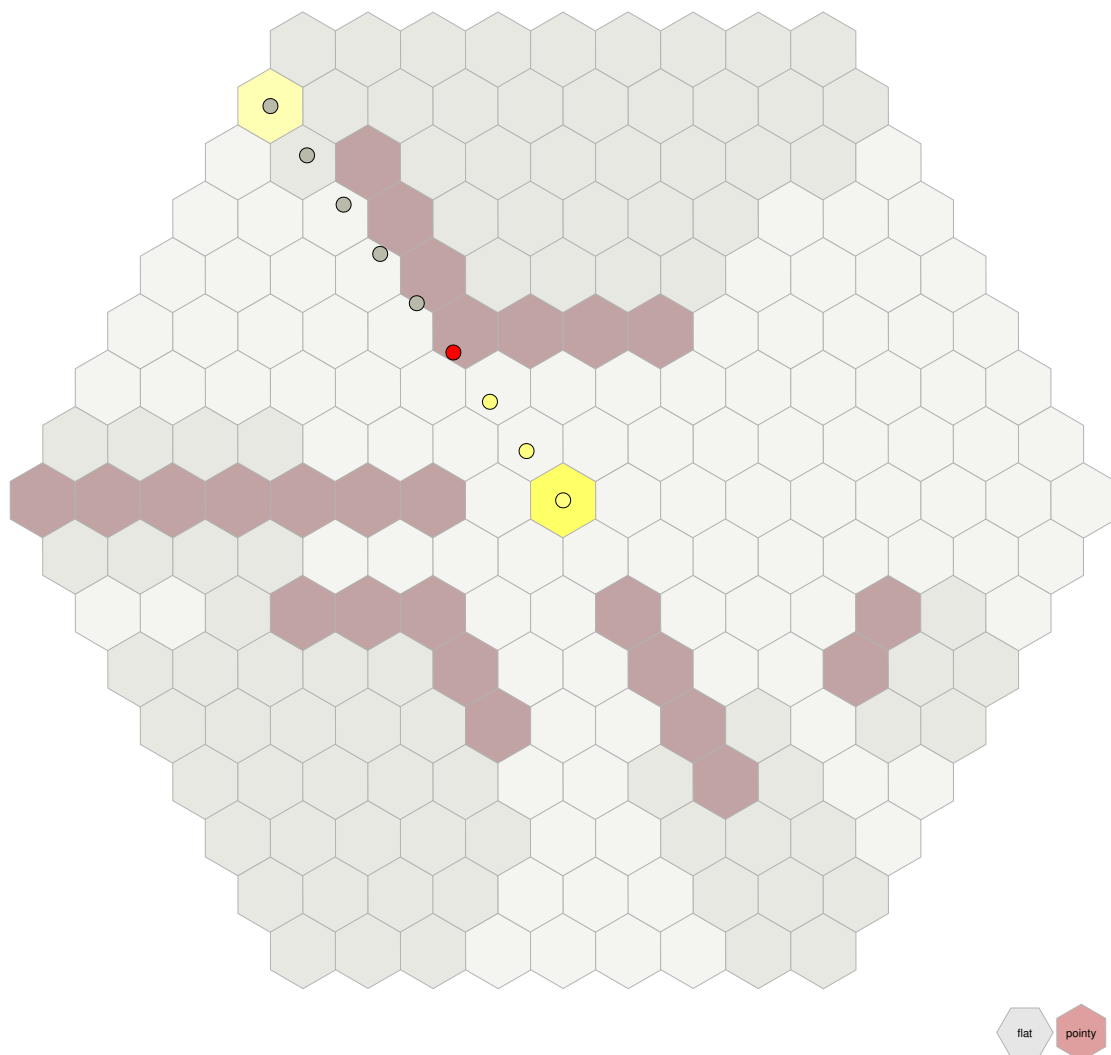
I haven't yet used any of the spiral coordinate systems in a real project. There are many possible variants, including [outside-in](#)^[33], 1-based, making sure numbers are adjacent, alternating clockwise/counterclockwise, and probably more. See [ljedrz/hex-spiral](#)^[34] and [lucidBrot/hex-gridspiral](#)^[35] for some implementations.

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](#)^[36] describes a “start at center and move outwards” algorithm to calculate field of view. 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 algo-

rithms for square grids (see [Roguelike Vision Algorithms](#)^[37] and [Pre-Computed Visibility Tries](#)^[38] and [Field of Vision](#)^[39]); some of them might be adapted for hex grids.

Hex to pixel

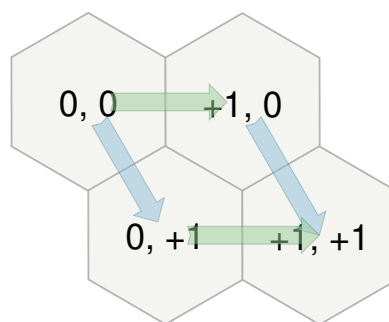
#

For hex to pixel, it's useful to review the [size and spacing diagram](#) at the top of the page where we defined the `horiz` and `vert` spacing between adjacent hexagons.

Axial coordinates

#

For axial coordinates, the way to think about hex to pixel conversion is to look at the *basis vectors*. The arrow $(0,0) \rightarrow (1,0)$ is the **q** basis vector ($x=\sqrt{3}$, $y=0$) and $(0,0) \rightarrow (0,1)$ 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) :
    // hex to cartesian
    var x = (sqrt(3) * hex.q + sqrt(3)/2 * hex
    var y = (                                3./2 * hex
    // scale cartesian coordinates
```

```

x = x * size
y = y * size
return Point(x, y)

```

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

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

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) :
    // hex to cartesian
    var x = sqrt(3) * (hex.col + 0.5 * (hex.row & 1))
    var y = 3./2 * hex.row
    // scale cartesian coordinates
    x = x * size
    y = y * size
    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):
    // hex to cartesian
    var x = sqrt(3)/2 * hex.col
    var y =      3./2 * hex.row
    // scale cartesian coordinates
    x = x * size
    y = y * size
    return Point(x, y)

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

Mod: non-zero origin

#

Some projects have grids that are not centered at 0,0. We can adapt any of the hex-to-pixel functions above by *chaining* one additional operation to the end:

```
function *_to_pixel(hex):
    // hex to cartesian
    ...
    // scale cartesian coordinates
    ...
    // translate cartesian coordinates
    x = x + origin.x
    y = y + origin.y
    return Point(x, y)
```

Later, when converting pixel to hex, we'll undo this by subtracting the origin at the beginning.

Mod: pixel sizes

#

Some projects need hexagons to fit a specific size. We can use the formulas from the [Geometry](#) section of this page to change the scaling for any of the hex-to-pixel functions to *separately* multiply by x and y scales.

Desired size width= × height= in pixels

```
function *_to_pixel(hex) :
    // hex to cartesian
    ...
    // scale cartesian coordinates
    x = x * (17 / sqrt(3))
    y = y * (24 / 2.)
```

☐ flat top or ☒ pointy top

Later, [when converting pixel to hex](#), we'll undo this by dividing by the x and y scales.

To further simplify, we can *inline* the scaling into matrix multiply to cancel out the `sqrt(3)`. For example, here's the Axial code before inlining:

```
function pointy_hex_to_pixel(hex) :
    // hex to cartesian
    var x = (sqrt(3) * hex.q + sqrt(3)/2 * hex.r)
    var y = (3./2 * hex.r)
    // scale cartesian coordinates
    x = x * (17 / sqrt(3))
    y = y * (24 / 2.)
    return Point(x, y)
```

Here's the inlined version with the `sqrt(3)` canceled out:

```
function pointy_hex_to_pixel(hex) :
    var x = 17 * (hex.q + 1./2 * hex.r)
    var y = 24 * (3./4 * hex.r)
    return Point(x, y)
```

Similarly, inlining the scaling into the offset or doubled conversion will simplify the code.

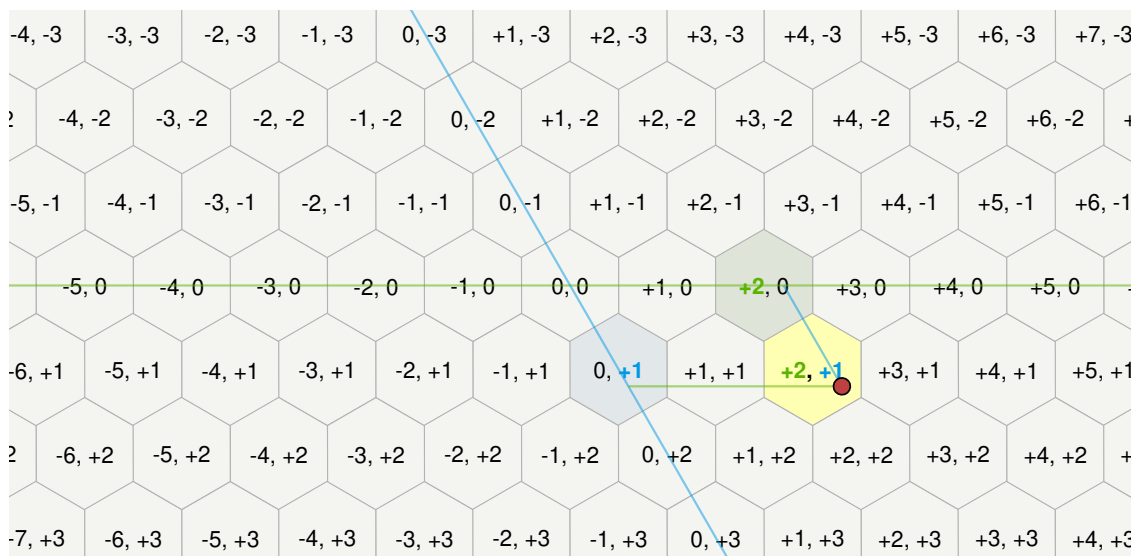
Pixel to Hex

#

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/cube coordinates. For offset and doubled coordinates, I first convert pixel to axial/cube, and then axial/cube to offset/doubled, but there are more direct algorithms also.

Axial coordinates

#



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:

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

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

Matrix for: ○ flat top or ● pointy top

To invert the hex-to-pixel process into a pixel-to-hex process we first invert the scaling, then [invert the pointy-top hex-to-pixel matrix](#)^[40] into a pixel-to-hex matrix:

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

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) :
    // invert the scaling
    var x = point.x / size
    var y = point.y / size
    // cartesian to hex
    var q = (sqrt(3)/3 * x - 1./3 * y)
    var r = (
        2./3 * y)
    return axial_round(Hex(q, r))
```

Code for: ○ flat top or ● pointy top

This algorithm reuses the `round()` function needed in line drawing. I'm making a [list of other algorithms](#) that don't use the `round()` function.

Offset coordinates

#

If you use offset coordinates, you can convert to pixel to axial, then axial to offset. There are also more direct algorithms that I need to study; browse [this list](#). In one project I used [a pixel map](#), a precomputed array of pixels that stores the hex coordinate at each.

Doubled coordinates

#

The [hex-to-pixel](#) algorithm for doubled coordinates is a straight multi-

ply, so the pixel-to-hex will divide by those same amounts. After that, we need to round to the nearest hex. I haven't worked that out yet for doubled coordinates.

Mod: non-zero origin

#

For hex-to-pixel, we implemented [a non-zero origin](#) by inserting an addition at the *end* of the chain. To invert this operation, we need to insert a subtraction at the *beginning* of the chain:

```
function *_to_hex(point) :
    // invert the addition
    x = x - origin.x
    y = y - origin.y
    // invert the scaling
    ...
    // cartesian to hex
    ...
```

This pattern is not limited to hexagons. Any chain of operations $p \rightarrow A \rightarrow B \rightarrow C \rightarrow q$ can be inverted by inverting each individual operation and performing them in reverse order: $q \rightarrow C^{-1} \rightarrow B^{-1} \rightarrow A^{-1} \rightarrow p$.

Mod: pixel sizes

#

For hex-to-pixel, we implemented [non-uniform scaling](#) by changing the multiplying by `size` to multiplying by a different scale for x and y. To invert this operation, we need to divide by those scaling values:

Desired size width= × height= in pixels

```
function *_to_hex(hex) :
    // invert the scaling
    x = x / (17 / sqrt(3))
    y = y / (24 / 2.)
    // cartesian to hex
    ...
```

☐ flat top or ☒ pointy top

Similar to hex-to-pixel, inlining the scaling into the matrix multiply will simplify the code by canceling out the `sqrt(3)`. It's easier to reason about the code when the steps are separate, so I start out with separate code before combining the steps.

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](#)^[42] 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.

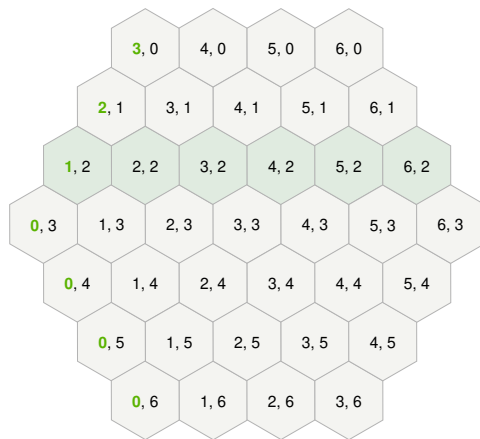
This algorithm is based on [Charles Fu's article from 1994](#)^[43]. His code contains the additional optimization that if $rx + ry + rz = 0$ there's no need to look at the error values and reset the largest component.

Patrick Surry has a [visualization showing why the rounding algorithm works](#)^[44]. Martin R. Han has a [different visualization showing why the rounding algorithm works](#)^[45].

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-top 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 $2*N+1 - \text{abs}(N-r)$.
 - **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-top 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

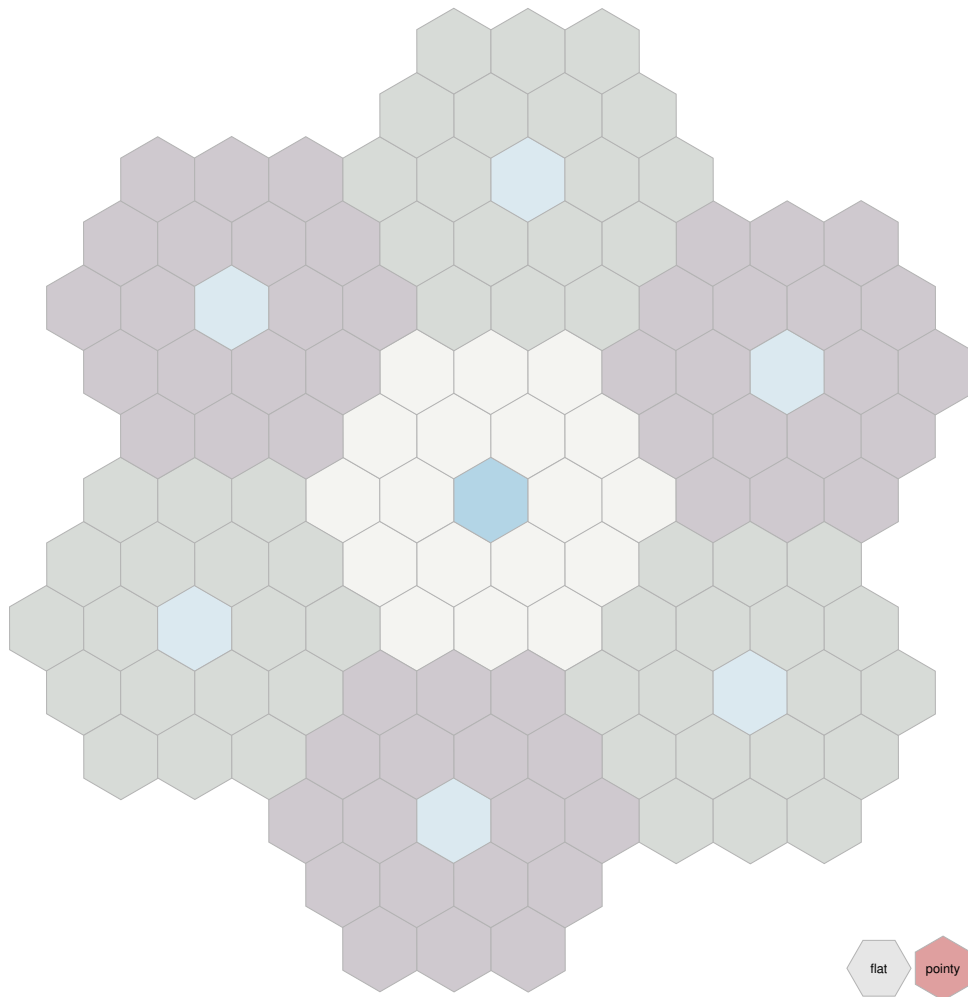
#

Sometimes you may 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](#)^[46] 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](#).

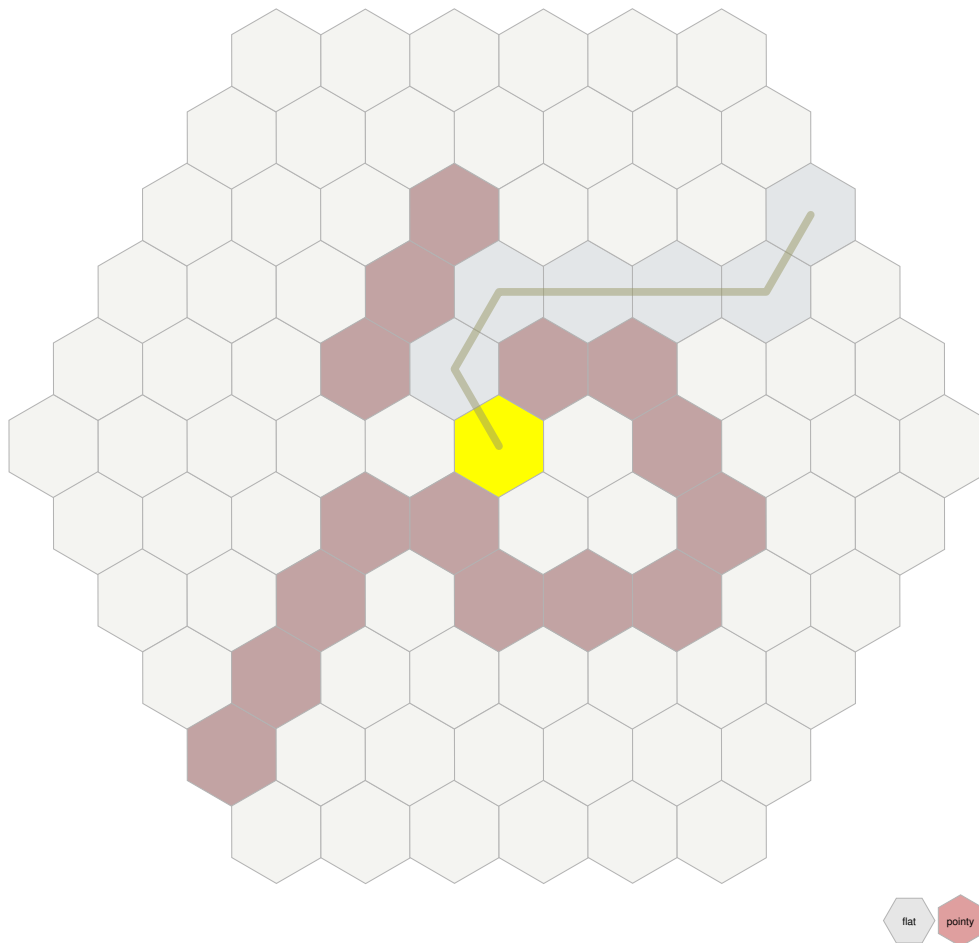


Related: Sander Evers has a [nice explanation of how to combine small hexagons into a grid of large hexagons](#)^[47] and also a [coordinate system to represent small hexagons within a larger one](#)^[48].

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](#)^[49] 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++, Python, C#, Haxe, Java, Javascript, Typescript, Lua, and Rust. I also link to existing libraries for C# (including Unity), Java, Objective C, Swift, Python, Ruby, and other languages.

- The best early guide I saw to the axial coordinate system was [Clark Verbrugge's guide](#)^[50], written in 1996.
- The first time I saw the cube coordinate system was from [Charles Fu's posting to rec.games.programmer](#)^[51] in 1994.
- [DevMag has a nice visual overview of hex math](#)^[52] including how to represent areas such as half-planes, triangles, and quadrangles. There's a [PDF article](#)^[53] that goes into more detail. **Highly recommended!** The [GameLogic Grids](#)^[54] 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](#)^[55].
- [Overview of hex coordinate types](#)^[56]: staggered (offset), interlaced, 3D (cube), and trapezoidal (axial).
- Hexnet explains how the [correspondence between hexagons and cubes](#)^[57] goes much deeper than what I described on this page, generalizing to higher dimensions.
- [The Rot.js library](#)^[58] has a list of hex coordinate systems: non-orthogonal (axial), odd shift (offset), double width (interlaced), cube.
- [Range for cube coordinates](#)^[59]: given a distance, which hexagons are that distance from the given one?
- [Distances on hex grids](#)^[60] using cube coordinates, and reasons to use cube coordinates instead of offset.
- [This guide](#)^[61] explains the basics of measuring and drawing

hexagons, using an offset grid.

- [Convert cube hex coordinates to pixel coordinates](#)^[62].
- [This thread](#)^[63] explains how to generate rings.
- Are there [pros and cons of “pointy top” and “flat top” hexagons](#)^[64]?
- [Line of sight in a hex grid](#)^[65] with offset coordinates, splitting hexes into triangles
- I printed out the PDF hex grids from [this page](#)^[66] while working out some of the algorithms.
- [Hexagonal Image Processing](#)^[67] ([DOI](#)^[68]) is an entire book that uses a hierarchical hexagonal coordinate system.
- 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](#)^[69]) It calls the axial system *oblique coordinates* and the offset systems *pseudohexagonal grids*.
- Snyder, Qi, Sander's paper *Coordinate system for hexagonal pixels* ([DOI](#)^[70]) describes gradients, diffusion, and map storage for axial coordinates. Mersereau's paper *The processing of hexagonally sampled two-dimensional signals* ([DOI](#)^[71]) describes signal processing on axial coordinates.
- There's a paper that calls cube coordinates **R3 coordinates*: Her, Innchyn, *Geometric Transformations on the Hexagonal Grid*, IEEE Transactions on Image Processing (1995) ([DOI](#)^[72]) It covers coordinates, correspondence to cube coordinates, rounding, reflections, scaling, shearing, and rotation. A paper from the same author ([DOI](#)^[73]) covers distances.
- The [Reddit discussion](#)^[74] and [Hacker News discussion](#)^[75] and [MetaFilter discussion](#)^[76] 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](#)^[77]. Do you have suggestions for things to change or add? Comment below.

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

Links

- [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]: https://web.archive.org/web/20090205120106/http://sc.tri-bit.com/Hex_Grids
- [5]: <https://ondras.github.io/rot.js/manual/#hex/indexing>
- [6]: https://www.researchgate.net/publication/235779843_Storage_and_addressing_scheme_for_practical_hexagonal_image_processing?_sg=flKEA6rk1KmOpC4LBjQJN_-NBuiR1KJtJt-XeYRXnd0z_MNUrB2gjb2FKV3iBoKg988P2xHCpQ
- [7]: <https://gamedev.stackexchange.com/questions/71785/convert-between-spiral-honeycomb-mosaic-and-axial-hex-coordinates>
- [8]: <https://opus.lib.uts.edu.au/bitstream/2100/280/11/02Whole.pdf>
- [9]: https://web.archive.org/web/20120303114550/http://www.pyxisinnovation.com/pyxwiki/index.php?title=Generalized_Balanced_Ternary
- [10]: <https://www.sciencedirect.com/science/article/pii/S0166218X9200186P>
- [11]: [https://doi.org/10.1016/0166-218X\(92\)00186-P](https://doi.org/10.1016/0166-218X(92)00186-P)
- [12]: https://old.reddit.com/r/gamedev/comments/19wmvn/a_data_structure_for_a_game_board_with_hexagonal/c8s9qbe/
- [13]: <https://github.com/SimonWailand/hexwrench>
- [14]: <https://patricksurry.github.io/posts/flowsnake/>
- [15]: <https://metacpan.org/pod/Math::PlanePath::FlowsnakeCentres>
- [16]: https://en.wikipedia.org/wiki/Bitwise_operation#AND
- [17]: https://en.wikipedia.org/wiki/Modulo_operation
- [18]: <https://scholar.google.com/scholar?q=Storage+and+addressing+scheme+for+practical+hexagonal+image+processing>
- [19]: <https://doi.org/10.1117/1.JEI.22.1.010502>
- [20]: <https://web.archive.org/web/20210302023226/http://3dmdesign.com/development/hexmap-coordinates-the-easy-way>

- [21]: <https://ondras.github.io/rot.js/manual/#hex/indexing>
- [22]: <https://ondras.github.io/rot.js/manual/#hex/indexing>
- [23]: [https://en.wikipedia.org/wiki/Digital_differential_analyzer_\(graphics_algorithm\)](https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm))
- [24]: <https://z vold.blogspot.com/2010/02/line-of-sight-on-hexagonal-grid.html>
- [25]: <https://stackoverflow.com/questions/3233522/elegant-clean-special-case-straight-line-grid-traversal-algorithm>
- [26]: <https://doi.org/10.1111/1467-8659.1210027>
- [27]: <https://doi.org/10.1111/1467-8659.1210021>
- [28]: <http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/>
- [29]: <https://en.wikipedia.org/wiki/Cuboid>
- [30]: <http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/>
- [31]: <https://gamedev.stackexchange.com/questions/15237/how-do-i-rotate-a-structure-of-hexagonal-tiles-on-a-hexagonal-grid/>
- [32]: https://en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF
- [33]: <https://x.com/BEBischof/status/969813312341917697>
- [34]: <https://github.com/ljedrz/hex-spiral/>
- [35]: <https://github.com/lucidBrot/hexgridspiral?tab=readme-ov-file#coordinate-systems>
- [36]: <http://www-cs-students.stanford.edu/~amitp/Articles/HexLOS.html>
- [37]: https://www.adammil.net/blog/v125_Roguelike_Vision_Algorithms.html
- [38]: https://www.roguebasin.com/index.php/Pre-Computed_Visibility_Tries
- [39]: https://www.roguebasin.com/index.php/Field_of_Vision
- [40]: <https://www.wolframalpha.com/input/?i=inv+%7B%7Bsqrt%283%29%2C+sqrt%283%29%2F2%7D%2C+%7B0%2C+3%2F2%7D%7D>
- [41]: <https://www.wolframalpha.com/input/?i=inv+%7B%7B3%2F2%2C+0%7D%2C+%7Bsqrt%283%29%2F2%2C+sqrt%283%29%7D%7D>
- [42]: <https://observablehq.com/@jrus/hexround>
- [43]: <http://www-cs-students.stanford.edu/~amitp/Articles/Hexagon2.html>
- [44]: <https://blocks.roadtolarissa.com/patricksurry/0603b407fa0a0071b59366219c67abca>
- [45]: <https://www.desmos.com/3d/86szmiocif>
- [46]: <https://gamedev.stackexchange.com/a/137603/2472>
- [47]: <https://observablehq.com/@sanderevers/hexagon-tiling-of-an-hexagonal-grid>
- [48]: <https://observablehq.com/@sanderevers/hexmod-representation>
- [49]: <https://www.redblobgames.com/pathfinding/a-star/introduction.html>
- [50]: <http://www-cs-students.stanford.edu/~amitp/Articles/HexLOS.html>

- [51]: <http://www-cs-students.stanford.edu/~amitp/Articles/Hexagon2.html>
- [52]: <http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/>
- [53]: <https://www.gamelogic.co.za/downloads/HexMath2.pdf>
- [54]: <https://gamelogic.co.za/grids/documentation-contents/quick-start-tutorial/gamelogics-hex-grids-for-unity-and-amit-patels-guide-for-hex-grids/>
- [55]: <https://playtechs.blogspot.com/2007/04/hex-grids.html>
- [56]: https://web.archive.org/web/20090205120106/http://sc.tri-bit.com/Hex_Grids
- [57]: <https://hexnet.org/content/permutohedron>
- [58]: <https://ondras.github.io/rot.js/manual/#hex/indexing>
- [59]: <https://stackoverflow.com/questions/2049196/generating-triangular-hexagonal-coordinates-xyz>
- [60]: <https://archive.ph/20141214082648/http://keekercdc.com/2011/03/hexagon-grids-coordinate-systems-and-distance-calculations/>
- [61]: <https://web.archive.org/web/20130608235236/https://www.br-gs.com/tutorial/hexagon-grid.html>
- [62]: <https://stackoverflow.com/questions/2459402/hexagonal-grid-coordinates-to-pixel-coordinates>
- [63]: <https://gamedev.stackexchange.com/questions/51264/get-ring-of-tiles-in-hexagon-grid>
- [64]: <https://gamedev.stackexchange.com/questions/49718/vertical-vs-horizontal-hex-grids-pros-and-cons>
- [65]: <https://web.archive.org/web/20121113035227/http://arges-systems.com/blog/2011/01/10/hex-grid-line-of-sight-revisited/>
- [66]: <https://incompetech.com/graphpaper/hexagonal/>
- [67]: <https://link.springer.com/book/10.1007/1-84628-203-9>
- [68]: <https://doi.org/10.1007/1-84628-203-9>
- [69]: <https://doi.org/10.1109/TC.1976.1674642>
- [70]: <https://doi.org/10.1117/12.348629>
- [71]: <https://doi.org/10.1109/PROC.1979.11356>
- [72]: <https://doi.org/10.1109/83.413166>
- [73]: <https://doi.org/10.1115/1.2919210>
- [74]: <https://old.reddit.com/r/gamedev/comments/1dz1tr/>
- [75]: <https://news.ycombinator.com/item?id=5809724>
- [76]: <https://www.metafilter.com/128649/Hexagonal-Grids>
- [77]: <https://redblobgames.notion.site/Hexagonal-Grids-7d2d4d624bc5483dafbe615d75ab3902>

Copyright © 2025 [Red Blob Games](#)

 [RSS Feed](#)

Created 11 Mar 2013 with [Haxe](#) and [D3.js](#) ; updated in 2018 to use [Vue.js](#) ; Last modified: 02 Aug 2025