**Isometric Cube Coordinates**   
**by** [**Charles Fu**](mailto:ccwf@plato.klab.caltech.edu)   
4 Jun 1994

Courtesy of [Amit Patel](http://www-cs-students.stanford.edu/%7Eamitp)

Taken from a posting to rec.games.programmer

In article <2sinkh$kj@controversy.math.lsa.umich.edu>,

Zachary S. Delproposto wrote:

>Joseph Hlebasko (hlebasko@lilly.com) wrote:

>: |2 | is x above or below the line? You can use Y=MX+B and search

>: this way (brute force)

>: |\_\_\_\_ or if your hexes are small enough you can pre-calculate

>: the coords and store in

>: a table and do a table search.

>This is a very good description, and is quite similar to what I did in

>a program I wrote. However, the triangle part was easy -- I used

>the region functions in the Windows API (now under DOS, it's a bit

>more difficult!).

Am I the only one who has found all the solutions presented so far less than elegant? Maybe it's just my mathematical background. Anyways, I presented my solution to this problem in this group, oh, almost a decade ago now. Since I didn't save the article :-), I put pencil to paper and rederived the results. (Actually, this is slightly cleaner than my original solution: I've improved a little over the years. :-) Basically, for hex grids I advocated using a symmetric system of three axes each 120 degrees apart (obviously, one is redundant). For example, have x increasing to the right, y increasing to the upper left, and z increasing to the lower left. Then, your hexes will have coordinates like this:

|  |
| --- |
| \_\_\_\_  /  \_\_\_\_/ 0 1-1\_\_\_\_  //  -1 1 0 \_\_\_\_// 1 0-1\_\_\_  / /  \_\_\_\_/ 0 0 0\_\_\_\_/ 2 -1 -1  //  -1 0 1 \_\_\_\_// 1-1 0\_\_  / /  \_\_\_\_/ 0-1 1\_\_\_\_/  //  \_\_\_\_// |

The formula for which hex a point belongs to is then

|  |
| --- |
| \_\_\_\_ \_\_\_\_ \_  rint { rint(r) - [ 0.5 - max |rint(r )-r | ] sum rint(r ) }  i i i i i |

where r is your ungridded position, rint is your rounding function of choice, and the top bar indicates a vector. The zig-zag is not directly relevant and is explained at the end of the article.

Of course, this is just a pretty formula and not an efficient implementation. An efficient implementation would go something like this:

/\* doubles x, y, and z hold the initial raw ungridded position \*/

double rx, ry, rz;

int ix, iy, iz, s;

ix = rx = rint(x);

iy = ry = rint(y);

iz = rz = rint(z);

s = ix + iy + iz;

if(s) {

double abs\_dx = fabs(rx-x),

abs\_dy = fabs(ry-y), abs\_dz = fabs(rz-z);

if(abs\_dx >= abs\_dy && abs\_dx >= abs\_dz) ix -= s; /\* abs\_dx is max. \*/

else if(abs\_dy >= abs\_dx && abs\_dy >= abs\_dz)

iy -= s; /\* abs\_dy is max. \*.

else iz-=s;

}

/\* ix, iy, iz now hold the coordinates of the hex. \*/

Obviously, further machine implementation optimizations can be made depending upon the speed with which doubles can be cast to ints, whether or not rint and fabs are inlined and implemented in hardware, whether or not your machine uses a IEEE floating point representation, pipelining issues, and so forth. The if-else if-else can also be improved.

Note that the algorithm is symmetric in x,y,z since the axes are symmetric. Also, x+y+z is always zero. These two facts make algorithms more intuitive and bugs much easier to spot. Similarity in instructions may also improve chances for compiler or hand optimization. If desired, z can be computed whenever needed to reduce the memory usage by a few measly bytes.

Of course, if you don't use my coordinate system, you will need to convert in and out of it. This should be pretty simple. For example, if you use the popular system with x increasing by one every hex to the right and y increasing by one every hex up with y going 0.5,1.5,2.5,... on the odd columns, then your x is the same as my x and your y is just offset from mine by x/2 (and my z=-x-y as pointed out above).

**\*\*mathematical details below--not for the faint-hearted\*\***

How was the above formula derived? The insight is that a uniform hex grid is the isometric projection of an infinite grid of unit cubes whose centers satisfy the equation x+y+z=0. Thus, hairy problems with hexes in 2-D become nicer problems with cubes in 3-D in very much the same way that using homogenous coordinates linearizes projective transformations.

(An isometric projection is an orthographic, i.e., non-perspective, projection onto the x+y+z=0 plane. It is one of the standard views used by draftsmen: the one with x, y, and z 120 degrees from each other, just like my axes.)

In this particular case, the problem of determining which hex contains a given point becomes the trivial problem of which cube contains a point. The rest of the code just transforms from the x+y+z=0 plane to the cube grid and vice versa. That's it.

With the cube grid system, problems like counting the number of hexes between points also becomes trivial. The system also has interesting bizarre properties such as lines of constant x zigzagging to follow hex sides as shown in the diagram above. If you want a Euclidean metric, just stick with the rotated coordinate system and avoid the fancy projection except when discrete hexes are needed.

Final note: my advocacy of this approach got no support when I first posted it years ago. Perhaps it will stir more interest now. There does seem to be a gradual trend toward mathematical literacy amongst game designers and CG programmers. Perhaps there's hope for me yet. :-)

-ccwf

... Anatomy is very poor... See how that muscle connects?... And

that perspective, \_yeesh\_!... Do you know what a \_vanishing point\_

is? And as for \_faces\_...

-Scott McCloud, \_Understanding Comics\_

[Related Posts Plugin for WordPress, Blogger...](http://www.linkwithin.com/)