Hex Grid Geometry for Game Developers (1.3)

Herman Tulleken

June 7, 2018

Contents

Introduction

This document is an introduction to the math for working on a hex grid, and should be useful to you if you want to implement geometric algorithms on such a grid.

I assume you know how vectors work, and how we use them in rectangular systems to do geometry.

This article is divided into three main sections. The first deals with vectors on a hex grid, and basic operations defined on them. It uses concepts that should already be familiar to you from "square" math.

The second part deals with shapes; their equations, and properties of special shapes. Like the the previous section, the ideas should be familiar, although the formulas are different.

The third deals with algorithms and techniques based grid point division. This tool is not commonly used for square grids, but the ideas follow from integer division for 1D math.

Credit and Further Reading

The hex-rounding function is from Amit Patel's page on hexagons: Hexagonal Grids. This page has many interactive examples, and also deals with some other hex coordinate systems. It is also from Patel's Implementation of Hex Grids that I first got the idea to use floats instead of only integers for coordinates, which drastically simplifies a lot of the math, and makes the system a lot closer to "ordinary" rectangular coordinate math. Patel wrote quite a bit on various implementation issues, and his work is worth a read:

- Hex grids: simplifying the variants
- Hex grids: procedurally generating code
- Hex grids: choosing data structures and generating algorithms
- Hex grids: code generator
- Hex grids: more on coordinate names
- Hex grids: finishing up the code generator project

Changes v.1.3

Fixed a typo[1] on page 18: "where p, q and r are integers" now reads "where d, q and r are integers".

[1] Thanks to Daniel Toffetti who alerted me to this.

Changes v.1.2

This document was first published in June 2015. Version 1.2 is only a minor revision: it fixes a few typos[2], and the layout is slightly different. Because I lost the original source, I had to recreate it, and it is possible a few errors crept in in the process. If you spot any, please let me know[3].

[2] Some of which were pointed out by "user69513" on Math Stackexchange.

[3] herman@gamelogic.co.za

Hex Vectors

Coordinate System

By choosing a simple coordinate system, we can get many of the conveniences of square grids to apply to hex grids, so that we can deduce geometric information by performing calculations on coordinates. For a rectangular coordinate system:

- we put down two identical real lines on the plane at a 90 degrees angle from each other,
- we call the one the x-axis and the other y-axis,
- and for any point, we can draw parallel lines towards the axes,
- and jot down the two numbers where these parallel lines cross the axes,
- and call them the x and y coordinates of the point.

There is a bit of mathematical machinery before we get to a vector (which is really an offset from one point to another), but since we won't distinguish between points and vectors, I will skip over this.

Once we have vectors, we can define a few operations that allow us to do geometric computations using their coordinates. These include vector addition, multiplication with a real number, multiplication with matrices, and a handful of other multiplication-like operators (such as the dot product).

To get to vectors for a hex system, we follow the same procedure as above, with a minor difference in the first step:

- we put down two identical real lines on the plane at a 60 degree angle from each other.

This difference is minor, but it gives this system the following property: integer coordinates fall on the faces of a hex grid, and we call these points **grid points**.

As with the square system, we can define some operations, and for the most part things work exactly the same:

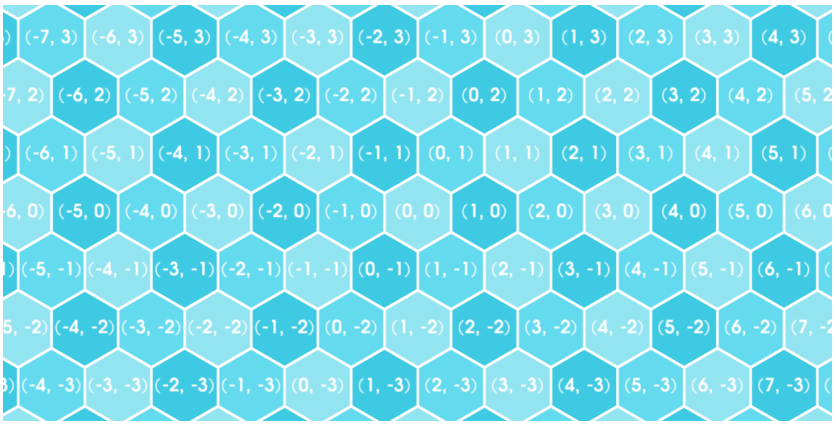

Figure 1: A hex Grid

- We can add and subtract vectors to denote displacements in this hex world.
- We can multiply vectors with scalars to denote scaling.
- We can write equations for lines, circles, and other shapes.
- We can use matrices for transformations such as scaling, reflection, rotation, shearing.

But there some differences:

- Distances are given by a different formula.
- Rotation and reflection matrices have different formulas.
- Angles between vectors are calculated using different formulas.
- We have slightly different identities that relate coordinates with magnitude and rotation with respect to the x-axis.
- We have a different definition of the dot product.

We will from time to time find it useful to use an auxiliary coordinate, which can be calculated from the other two:

$$z = -x - y. \tag{1}$$

This is mostly to make formulas more concise and symmetrical. This also makes it easier to see certain patterns when comparing formulas with the square system. The three lines $x = 0$, $y = 0$, and $z = 0$ are the **major axes**; the three lines $x = y$, $y = z$ and $z = x$ are the **minor axes**.

We sometimes use special notation for the unit vectors:

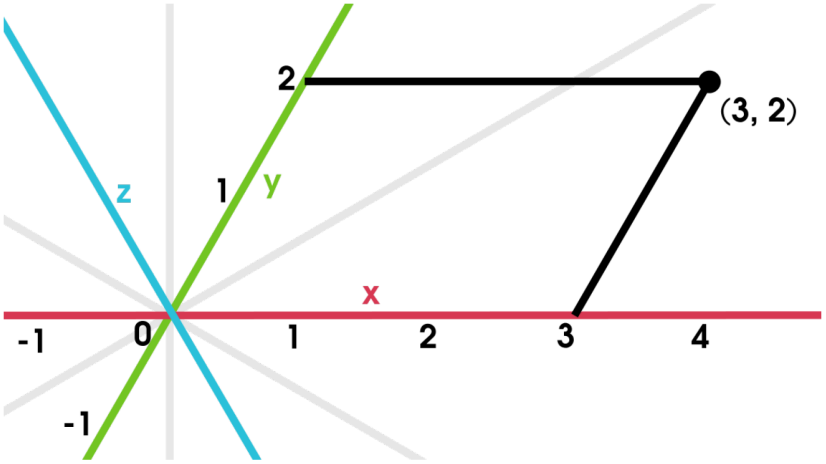

Figure 2: Hex coordinate system. The major axes are shown in red (x), green (y) and blue (z). The minor axes are shown in grey. If we draw lines parallel to the x- and y-axes, they intersect these axes at the points marked 3 and 2, so the point has coordinates $(3, 2)$.

$$\mathbf{e}_x = (1, 0) \tag{2}$$
$$\mathbf{e}_y = (0, 1) \tag{3}$$
$$\mathbf{e}_z = (-1, -1). \tag{4}$$

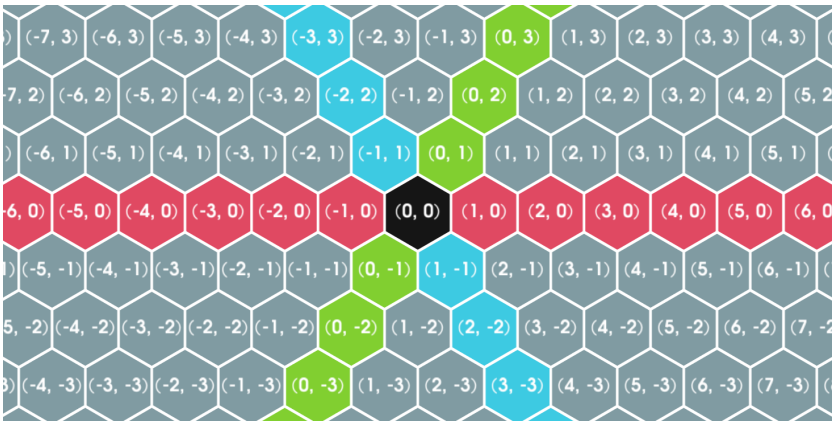

Figure 3: The major axes shown on a hex grid.

Grid points and hex rounding. We now introduce a special operation, which we call "hex rounding"; it gives for any point, the closest grid point. This operation divides the plane into hexagons–except for at the edges of these hexagons, the operation gives us the coordinate at the center of the hexagon the point "belongs" to.

If $\mathbf{v}$ is a vector, then the closest grid point is denoted by $\langle \mathbf{v} \rangle$. It is this operator that makes hex-coordinates useful; without it, there is little benefit of using hex-coordinates over rectangular coordinates.

Let $\langle a \rangle$ be the integer closest to a, and let $d_x = |x - \langle x \rangle|$, $d_y = |y - \langle y \rangle|$, and $d_z = |z - \langle z \rangle|$. Then we can calculate $\langle \mathbf{v} \rangle$ as

$$\langle \mathbf{v} \rangle = \begin{cases} (-\langle y \rangle - \langle z \rangle, \langle y \rangle) & \text{if } d_x = \max\{d_x, d_y, d_z\} \\ (\langle x \rangle, -\langle x \rangle - \langle z \rangle) & \text{if } d_y = \max\{d_x, d_y, d_z\} \\ (\langle x \rangle, \langle y \rangle) & \text{if } d_z = \max\{d_x, d_y, d_z\} \end{cases}. \tag{5}$$

Grid points have a few special properties:

- The sum (and difference) between two grid vectors is another grid vector.
- Scaling a grid-vector by an integer is always a grid vector.
- Any grid point can be written as a linear combination of $(1, 0)$ and $(0, 1)$, with integer coefficients.
- Rotating a grid vector by 60 degrees (and any integer multiple thereof) gives another grid vector.
- Reflecting a grid vector about a line which contains two neighboring grid points gives another grid vector.

Vector addition and scalar multiplication. For hex grids, vector addition and scalar multiplication is the same as for square systems. As an example of these vectors in action, let's find the coordinates of the center of an edge, and of a vertex.

The midpoint of an edge between two hexagons is given by $(\mathbf{u} + \mathbf{v})/2$, where $\mathbf{u}$ and $\mathbf{v}$ are the coordinates of the faces of the hexagons. These points are always in the form $(m/2, n/2)$ where m and n are integers.

The vertex between three neighboring hexagons is given by $(\mathbf{u} + \mathbf{v} + \mathbf{w})/3$. These are always in the form $(m/3, n/3)$ where m and n are integers.

Norms

Norms are functions that assign real numbers to vectors that coincide with our notion of length. They are useful not only for determining distances between points, but also to write equations for common shapes.

The Euclidean length is the norm we know from square geometry, and for hex coordinates is given by:

$$|\mathbf{v}| = \sqrt{\frac{x^2 + y^2 + z^2}{2}}. \tag{6}$$

As with square systems, we can find the distance between two points by taking the magnitude of their difference:

$$d(\mathbf{u}, \mathbf{v}) = |\mathbf{v} - \mathbf{u}|. \tag{7}$$

In calculations on a hex grid, the **hex norm** is occasionally useful[4]:

$$|\mathbf{v}|_h = \frac{|x| + |y| + |z|}{2} = \max\{|x|, |y|, |z|\}. \tag{8}$$

[4] It is a good exercise to prove that the two quantities in the equation are in fact equal.

We can define a related distance metric:

$$d_h(\mathbf{u}, \mathbf{v}) = |\mathbf{v} - \mathbf{u}|_h. \tag{9}$$

For grid points, this gives the minimum number of hexes we have to enter moving from one point to the next. It is analogous to two different norms in the square world: the taxicab norm and the chess norm. The taxicab norm (the sum of the absolute values of the coordinates) gives the minimum number of squares we have to enter to move from one square cell to another if we are only allowed orthogonal moves. The chess distance (the maximum of te absolute values of the coordinates) is the minimum number of squares we have to enter to move from one cell to another if we are also allowed diagonal movements.

We can either view the hex distance as a taxicab norm, where the roads are at 60 degree angles instead of 90 and there are three directions instead of two; or as the chess norm where there are not any diagonals because of the structure of the hex grid.

The following are not norms; we will call them **pseudonorms**. The down-triangle pseudo-norm and up-triangle pseudo-norm are defined by

$$|\mathbf{v}|_\triangledown = \max\{x, y, z\} \tag{10}$$

$$|\mathbf{v}|_\triangle = \max\{-x, -y, -z\}. \tag{11}$$

These pseudo-norms are used to write compact equations for equilateral triangles on a hex grid. We will get back to this later, but to give you an idea, $|\mathbf{v}|_\triangledown \leq r$ is an equilateral triangle pointing down, with sides of length r.

The triangle psuedo-norms are related by the identity

$$|-\mathbf{v}|_\triangledown = |\mathbf{v}|_\triangle. \tag{12}$$

These can be combined to define the **star norm**:

$$|\mathbf{v}|_s = \min\{|\mathbf{v}|_\triangledown, |\mathbf{v}|_\triangle\}. \tag{13}$$

The star norm is used to write compact equations for regular hexagrams (or stars).

The triangle pseudonyms are also related to the hex norm:

$$|\mathbf{v}|_h = \max\left\{|\mathbf{v}|_\bigtriangledown, |\mathbf{v}|_\bigtriangleup\right\}. \tag{14}$$

Transformations

In our normal geometry on a square coordinate system, we can represent a large class of useful transformations (of points and sets of points) as multiplication with matrices. The same trick works in a hex coordinate system, although the matrices we use look slightly different.

Rotation. This is the general formula for the rotation matrix:

$$R(\theta) = \begin{pmatrix} \sin(\theta + \frac{2\pi}{3}) & \sin\theta \\ -\sin\theta & \sin(\theta + \frac{\pi}{3}) \end{pmatrix}. \tag{15}$$

To rotate a vector, we simply multiply it with the matrix above. And for multiples of 60 degrees, the rotation matrix only has integer components, and so rotating a grid point by multiples of 60 degrees will give another grid point. Notice the occurrences of the sine shifted by various multiples of 60 degrees; we will see this often. In square systems, we have occurrences of the sine shifted by 90 degrees (remember that cosine is just a convenient abbreviation for $\sin(x + \frac{\pi}{2})$.

Scaling. Uniform scaling works the same as in square systems

$$S(s) = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix}. \tag{16}$$

The following matrix "scales" the coordinates independently, and is really a shear:

$$S(s_1, s_2) = \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix}. \tag{17}$$

Combining this shearing with rotations, we can get more general shears. For this purpose, it is more useful to only scale one component. We can get all possible shears along one shearing-axis by using a matrix of the form:

$$R(-\theta)S(s, 1)R(\theta). \tag{18}$$

We can get all two-axes shears by combining two of these:

$$R(-\theta_1)S(s_1, 1)R(\theta_1)R(-\theta_2)S(s_2, 1)R(\theta_2). \tag{19}$$

Reflection about x-axis. Reflection about the x-axis can be done using the following transformation matrix.

$$R_x = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}. \tag{20}$$

We can get reflections about other lines through the origin using rotations

$$R(-\theta)R_x R(\theta). \tag{21}$$

Trigonometry

When using vectors for doing geometry, it is not very common to use trigonometry, but it is useful to know how hex coordinates are related to lengths and angles of the vectors. This is, for example, helpful in finding equations for certain shapes. Here are a few basic identities

$$x = \frac{2r}{\sqrt{3}} \sin\left(\theta + \frac{2\pi}{3}\right) \tag{22a}$$

$$y = \frac{2r}{\sqrt{3}} \sin\theta \tag{22b}$$

$$z = \frac{2r}{\sqrt{3}} \sin\left(\theta - \frac{2\pi}{3}\right) \tag{22c}$$

And

$$x^2 + y^2 + z^2 = 2r^2 \tag{23}$$

$$\cot\theta = \frac{x - z}{\sqrt{3}y} \tag{24}$$

$$\sin\left(\theta + \frac{2\pi}{3}\right) + \sin\theta + \sin\left(\theta - \frac{2\pi}{3}\right) = 0 \tag{25}$$

$$\sin^2\left(\theta + \frac{2\pi}{3}\right) + \sin^2\theta + \sin^2\left(\theta - \frac{2\pi}{3}\right) = \frac{3}{2} \tag{26}$$

Products

The reason why we don't use trigonometry bso often is that we can use special products (the dot and perp dot product) to by-step trigonometric calculations.

For a hex system, we define the dot product as:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z. \tag{27}$$

The cosine of the angle between two vectors is given by:

$$\cos\alpha = \frac{\mathbf{u} \cdot \mathbf{v}}{\sqrt{3}\,|\mathbf{u}|\,|\mathbf{v}|}. \tag{28}$$

We can also define a cross-like product. Although it has the same definition as the perp dot product, it really involves a 60 degree rotation on the vector rather than a 90 degree one, so it would be a bit of a misnomer.

$$\mathbf{u} \times \mathbf{v} = u_x v_y - u_y v_x. \tag{29}$$

If it bothers you that z-coordinates are not involved in this definition, you would be glad to know that the following holds:

$$\begin{aligned} \mathbf{u} \times \mathbf{v} &= u_x v_y - u_y v_x && \text{(30a)} \\ &= u_y v_z - u_z v_y && \text{(30b)} \\ &= u_z v_x - u_x v_z && \text{(30c)} \\ & && \text{(30d)} \end{aligned}$$

We can get the (signed) angle from $\mathbf{u}$ to $\mathbf{v}$ using this:

$$\sin \alpha = \frac{2\mathbf{u} \times \mathbf{v}}{\sqrt{3}\,|\mathbf{u}|\,|\mathbf{v}|}. \tag{31}$$

Shapes

We will be mostly interested in grid shapes, shapes that contain only grid points. We will also talk about grid lines, grid triangles, grid circles, and so on to mean in each case these shapes only contain grid points. Non-grid shapes are still useful in interim calculations, and we will use the terms real shapes, real lines, and so on (real as in real number, not real as in not fake)

Lines

Axis-aligned grid lines are described by one of these equations:

$$\begin{aligned} x &= k && \text{(32a)} \\ y &= k && \text{(32b)} \\ z &= k, && \text{(32c)} \end{aligned}$$

with x, y and k integers.

We can also write equations in parametric form:

$$\begin{aligned} \mathbf{v} &= \mathbf{p} + n\mathbf{e}_x && \text{(33a)} \\ \mathbf{v} &= \mathbf{p} + n\mathbf{e}_y && \text{(33b)} \\ \mathbf{v} &= \mathbf{p} + n\mathbf{e}_z. && \text{(33c)} \end{aligned}$$

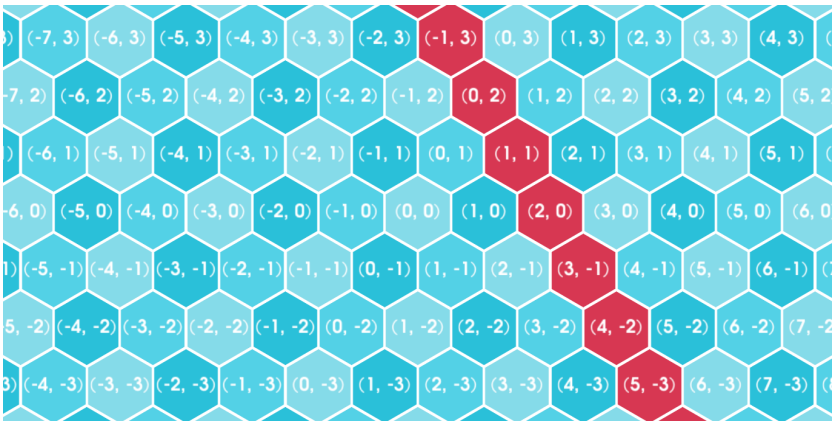

Figure 4: The line $z = -2$.

where $\mathbf{p}$ is any point in the line and n goes through all integers.

It is trivial to determine whether these lines are parallel (they must have the same equation form), and if not, where they intersect. For example, the lines $y = 3$ and $x = 5$ intersect in the point $(5, 3)$. The lines $y = 3$ and $z = -x - y = -5$ intersect in the point $(2, 3)$.

What about "lines" that look the one in Figure 5?

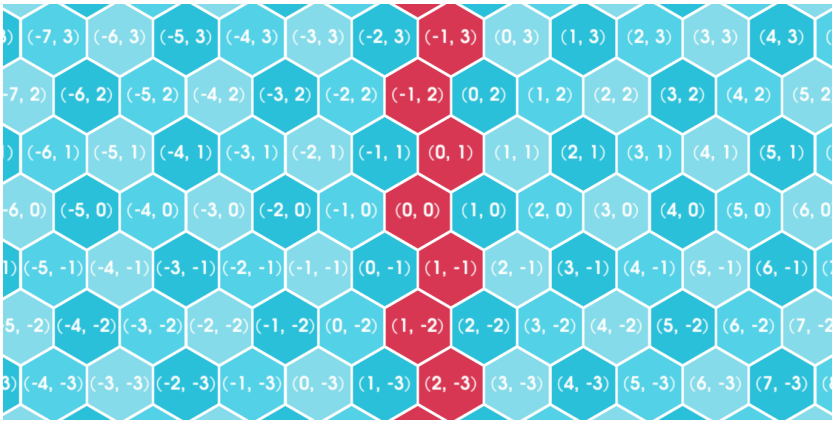

Figure 5: An impure line.

For these types of lines, it is necessary to recognize it as a grid approximation of a real line. Now there are many lines that have the same approximation, and we can choose any one to work with. The above grid line can be described by this equation:

$$v = \langle (1/4, 0) + n(-1/2, 1) \rangle \,. \tag{34}$$

With these type of equations we usually try to avoid choices that give points on the edges of hexagons, so that the behavior of our rounding function on edges does not affect the line. If, in the

example above, we chose $\mathbf{v} = \langle n(-1/2, 1)\rangle$, we could get either of these lines, depending on how the rounding function works.

More generally, then, we have this general description for grid lines:

$$\mathbf{v} = \langle \mathbf{p} + nt\mathbf{q}\rangle \tag{35}$$

where $\mathbf{p}$ and $\mathbf{q}$ are not necessarily grid points, and n goes through all integers, and t is a number with which we control the density of lines. If it is too big, we get holes in the lines and we call them sparse. If it is too small, we get repeated points, and we call the lines dense. There is a unique value for t that ensures that the grid line has no holes and no repeated points, and we call such lines pure.

Unlike real lines, non-parallel grid lines do not always intersect in a single point, and parallel lines that don't coincide can intersect.

Finding all the points of intersection can be somewhat tricky. For non-parallel lines one point of intersection can be found easily: the grid point closest to the intersection of the associated real lines, and we know that other intersection points must lie close to this.

- Pure lines that are not parallel always intersect in at least one point. All points of intersection are distinct.
- Sparse lines may not always intersect.
- Dense lines that are not parallel always intersect in at least one point. Points of intersection may not be distinct.
- Two lines are parallel if their $\mathbf{q}$-values are the same.
- Two parallel grid lines are the same if $\langle \mathbf{p}\rangle = \langle \mathbf{p}'\rangle$ and $\langle \mathbf{p} + t\mathbf{q}\rangle = \langle \mathbf{p}' + t\mathbf{q}\rangle$.

Halfplanes

Halfplanes are shapes with all points to the side of the borderline and including the borderline. They are described with inequalities with this general form (for real half-planes):

$$(\mathbf{v} - \mathbf{p}) \times \mathbf{q} \leq 0, \tag{36}$$

where $\mathbf{p}$ is a point on the borderline of the half-plane, and $\mathbf{q}$ is in the direction of the borderline (so that the defined halfplane lies on the right of $\mathbf{q}$).

For real halfplanes, the edges are the line $(\mathbf{v} - \mathbf{p}) \times \mathbf{q} = 0$. For grid lines, it is often more useful to exclude the border, and use

$$(\mathbf{v} - \mathbf{p}) \times \mathbf{q} < 0, \tag{37}$$

instead so that potential rounding errors are avoided.

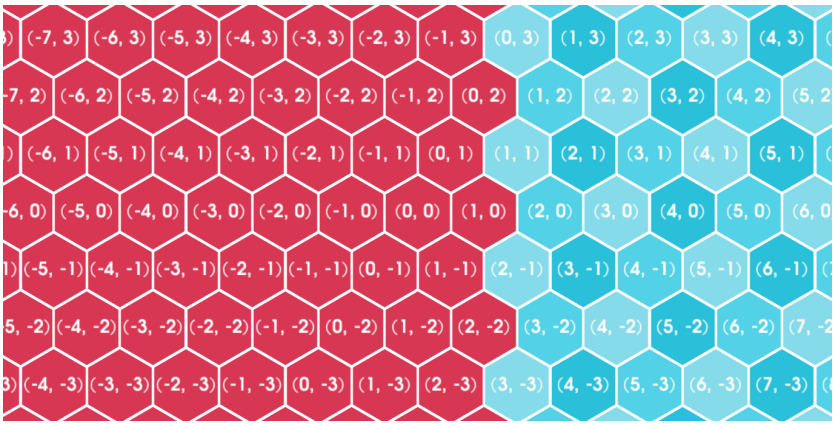

Figure 6: The grid halfplane $2x + y - 3 < 0$.

Polygons

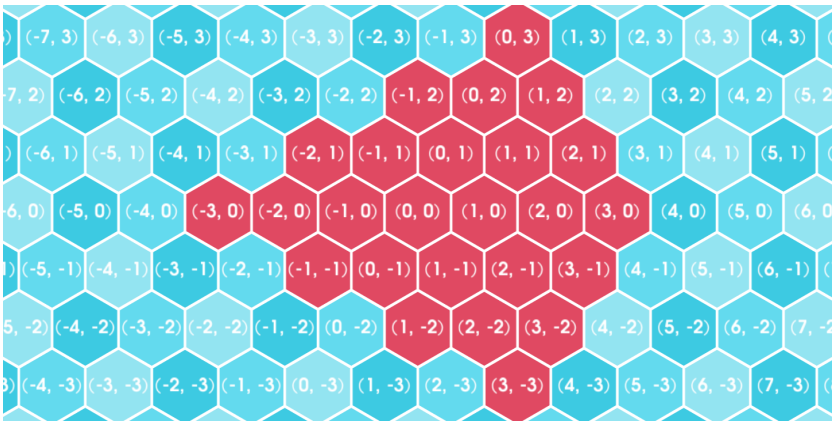

Figure 7: The polygon $P((-3,0),(3,-3),(3,0),(0,3);\mathbf{v}) \leq 0$.

A convex polygon T with n vertices $\mathbf{t}_i$ (in anti-clockwise order) is given by the intersection of halfplanes $(\mathbf{v} - \mathbf{t}_i)(\mathbf{t}_{i+1} - \mathbf{t}_i) \leq 0$.

We define the polygon function

$$P(T;\mathbf{v}) = P(\mathbf{t}_0, \cdots, \mathbf{t}_{n-1};\mathbf{v}) = \max_{i=0..n-1} \left\{ (\mathbf{v} - \mathbf{t}_i) \times (\mathbf{t}_{i+1} - \mathbf{t}_i) \right\}, \quad (38)$$

where $\mathbf{t}_0 = \mathbf{t}_n$. A polygon is then defined by

$$P(T;\mathbf{v})0. \quad (39)$$

Arbitrary polygons can be represented as the union of convex polygons. The union of polygons T_i is given by

$$\min_i \left\{ P(T_i;\mathbf{v}) \right\} \leq 0. \quad (40)$$

The area of any parallelogram is given by the cross product of the

vectors from one vertex to the two adjacent ones:

$$A = \frac{\sqrt{3}}{2} |\mathbf{u} \times \mathbf{v}| . \tag{41}$$

The area of a triangle between vectors $\mathbf{u}$ and $\mathbf{v}$ is given by

$$A = \frac{\sqrt{3}}{4} |\mathbf{u} \times \mathbf{v}| . \tag{42}$$

The area of a polygon with vertices (in clockwise order) $\mathbf{t}_0, \cdots, \mathbf{t}_{n-1}$ is given by

$$A = \frac{\sqrt{3}}{4} \left| \sum_{i=1}^{n-2} (\mathbf{t}_i - \mathbf{t}_0) \times (\mathbf{t}_{i+1} - \mathbf{t}_0) \right| . \tag{43}$$

The formula holds even if the polygon is not convex.

Circles

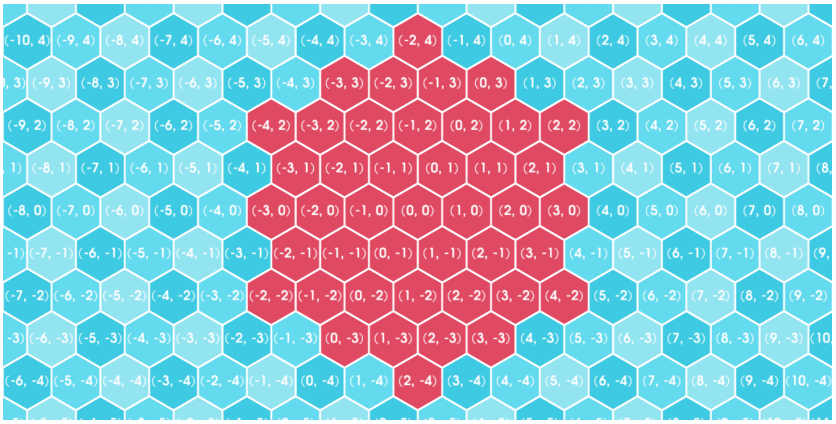

Figure 8: The grid circle $|\mathbf{v} - \mathbf{p}| < \frac{9}{4}$.

The equation for a circle of radius r at a point $\mathbf{p}$ is given by:

$$|\mathbf{v} - \mathbf{p}| \leq \sqrt{2}r. \tag{44}$$

The parametrical form is given by

$$\mathbf{v} = \frac{\sqrt{2}r}{3} \left(\sin\left(t + \frac{2\pi}{3}\right), \sin t \right) + \mathbf{p}. \tag{45}$$

Axis-aligned Triangles

The triangle psuedo-norms we defined earlier can be used to write equations for axis-aligned triangles. For a down-triangle centered at $\mathbf{p}$ with side length $3r$, we have

$$|\mathbf{v} - \mathbf{p}|_\triangledown \leq r \tag{46a}$$

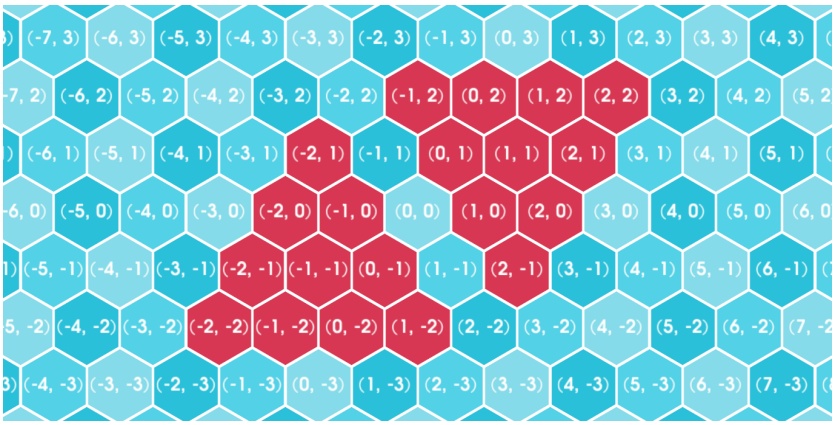

Figure 9: The triangles $|\mathbf{v} - (1,1)|_{\triangledown} \leq 1$ and $|\mathbf{v} + (1,1)|_{Delta} \leq 1$.

and for an up-triangle, we have

$$|\mathbf{v} - \mathbf{p}|_{\triangle} \leq r \tag{46b}$$

Up triangles are bounded by the lines $x = p_x + r$, $y = p_y + r$, and $z = p_z + r$, and down triangles by the lines $x = p_x - r$, $y = p_y - r$, and $z = p_z - r$. These triangles are always equilateral, and the length of their sides is given by $3r$. By taking the equations of the lines of the triangle sides two at a time, we can easily calculate the three vertices of the triangle. For up triangles:

$$\mathbf{t}_1 = (p_x + r, p_y + r) \tag{47a}$$
$$\mathbf{t}_2 = (p_x + r, p_y - 2r) \tag{47b}$$
$$\mathbf{t}_3 = (p_x - 2r, p_y + r) \tag{47c}$$

and for down triangles:

$$\mathbf{t}_1 = (p_x - r, p_y - r) \tag{48a}$$
$$\mathbf{t}_2 = (p_x - r, p_y + 2r) \tag{48b}$$
$$\mathbf{t}_3 = (p_x + 2r, p_y - r). \tag{48c}$$

The center $\mathbf{p}$ can be found as the average of the three vertices:

$$\mathbf{p} = \frac{\mathbf{t}_1 + \mathbf{t}_2 + \mathbf{t}_3}{3}. \tag{49}$$

The radius is a third of any of the three sides:

$$r = \frac{\mathbf{t}_2 - \mathbf{t}_1}{3} = \frac{\mathbf{t}_3 - \mathbf{t}_2}{3} = \frac{\mathbf{t}_1 - \mathbf{t}_3}{3}. \tag{50}$$

Axis-aligned Quadrangles, Pentagons, Hexagons and Hexagrams

In hex space, convex polygons can have up to six sides.

All quadrangles in hex-space are either parallelograms or isosceles trapezoids. Pentagons always have two pairs of sides parallel. Hexagons have three pairs of sides parallel.

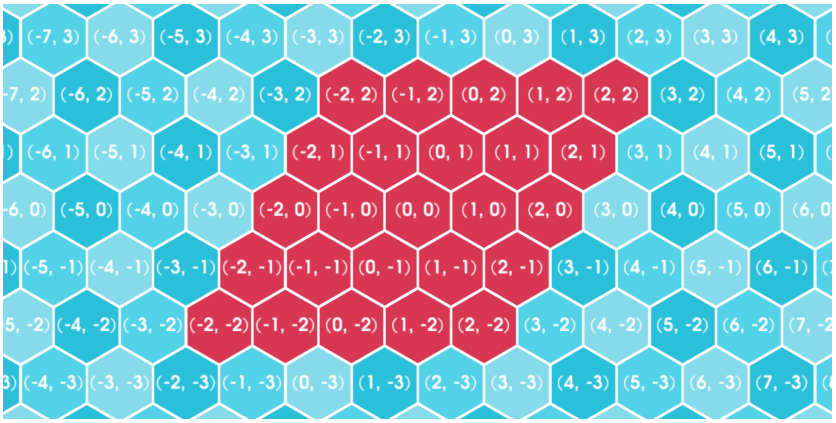

Figure 10: Parallelogram

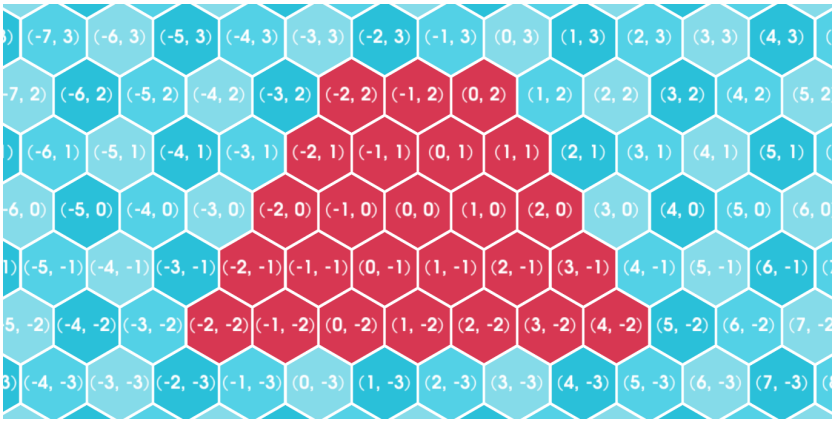

Figure 11: Trapezoid

Axes-aligned rhombi have one of the following equations:

$$\max\left\{|x - p_x|, |y - p_y|\right\} \leq r \tag{51a}$$
$$\max\left\{|x - p_x|, |z - p_z|\right\} \leq r \tag{51b}$$
$$\max\left\{|y - p_y|, |z - p_z|\right\} \leq r. \tag{51c}$$

Regular hexagons are given by the equation

$$|\mathbf{v} - \mathbf{p}|_h \leq r. \tag{52}$$

Regular hexagrams (star shapes) are given by the following equation:

$$|\mathbf{v} - \mathbf{p}|_s \leq r. \tag{53}$$

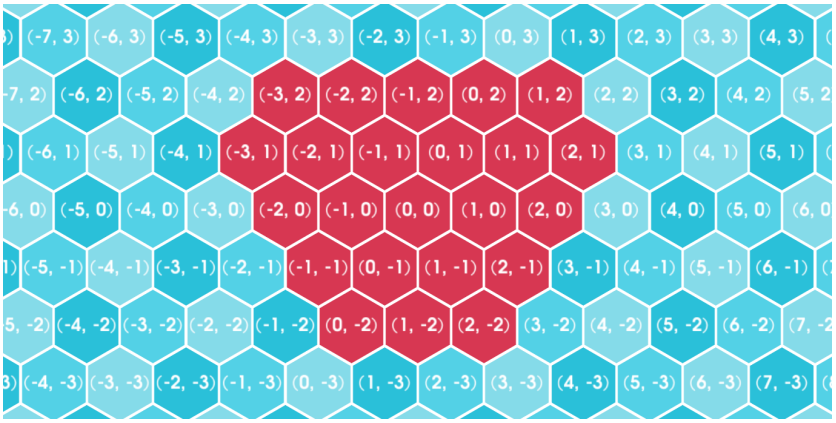

Figure 12: Hexagon

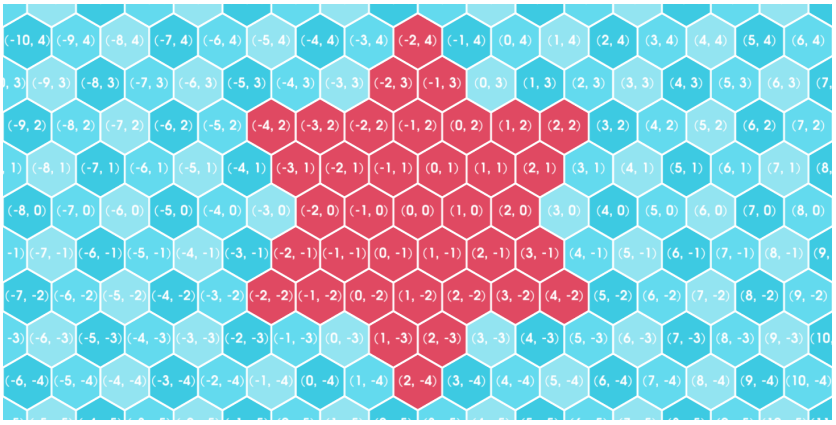

Figure 13: Regular Hexagram

Grid Point Division

Definitions

This section extends the ideas of division in 1D to 2D

For integers, recall the division algorithm: each integer n can be written in the form $n = qd + r$, where d, q and r are integers, and $0 \leq r < d$. It will be useful for us to extend this to real numbers. The above statement hold if we let n, d and r be any real numbers.

For real numbers, we define

$$v \operatorname{div} d = \left\lfloor \frac{v}{d} \right\rfloor \tag{54a}$$

$$v \bmod d = v - (v \operatorname{div} d)d. \tag{54b}$$

We then have, for the division algorithm

$$n = qd + r \tag{55}$$

q and r given by

$$q = n \operatorname{div} d \tag{56a}$$

$$r = n \bmod d. \tag{56b}$$

For real numbers, let's look at what these operations really mean, so that the 2D version is more intuitive. Let us partition the real line into half-open segments of length d, with each segment given by $P_q = (qd, (q+1)d)$ with q some integer.

Now for any real number n, if $n \in P_q$, then q is given by $n \operatorname{div} d$. So this operation tells us in which partition the number lies. The number $r = n \bmod d$ tells us how "far" into the partition the number lies. When we view the partitions as being the same, then r is a useful identifier that we can use to determine if two numbers in different partitions are the same.

For example, let $d = 2\pi$, and let's view the real number as angles. Then the partitions are really the same (as each partitions has the same angles as any other partition). In this case, $r = n \bmod 2\pi$ gives us the canonical angle, and we can use it to determine whether two angles are really the same. For example, $-\pi \bmod 2\pi = \pi$, and $3\pi \bmod 2\pi = \pi$, so we know that $-\pi$ and 3π are really the same angle, and moreover, that they are both the same as the angle π.

When it comes to 2D, we want to partition the plane into parallelograms, and we want the mod and div operators to tell us, for any point in the plane, which partition that point is in, and its position within that partition. In the following sections we will see how this is useful for a wide variety of algorithms.

Let $D = \begin{pmatrix} \mathbf{g} \\ \mathbf{h} \end{pmatrix}$ be a 2×2 matrix, with $\mathbf{g}$ and $\mathbf{h}$ two sides of a parallelogram. Then we make the following definitions:

$$\mathbf{v} \operatorname{div} D = \left\lfloor \mathbf{v} D^{-1} \right\rfloor \tag{57a}$$

$$\mathbf{v} \bmod D = \mathbf{v} - (\mathbf{v} \operatorname{div} D) D. \tag{57b}$$

where the floor of a matrix can be obtained by flooring all the components of the matrix. We can write every vector $\mathbf{v}$ in the form

$$\mathbf{v} = m\mathbf{g} + n\mathbf{h} + \mathbf{r} = \mathbf{q}D + r, \tag{58}$$

where $\mathbf{q} = (m, n)$ is a grid point, and $\mathbf{r}$ is a vector contained in the parallelogram bounded by $\mathbf{g}$ and $\mathbf{h}$, and given by

$$\mathbf{q} = \mathbf{v} \operatorname{div} D \tag{59a}$$

and

$$\mathbf{r} = \mathbf{v} \bmod D. \tag{59b}$$

As in the 1D case, the div operator tells us in which partition a point lies, and the mod operator tells us where in that parallelogram that points lies. Let us formalise this a bit. Let $P_{0,0}$ be the paralellogram bounded by vectors $\mathbf{g}$ and $\mathbf{h}$. Then if the plane is partitioned into paralellograms $P_{i,j} = \{(x, y) \mid (x, y) - i\mathbf{g} - j\mathbf{h} \in P_{0,0}\}$, then the div operator tells us in which parallelogram any points lies, and the mod operator tells us where inside the parallelogram the point lies.

The following identities are useful for computing these operations:

$$\mathbf{v} \operatorname{div} D = (\mathbf{v} \times \mathbf{h}, \mathbf{g} \times \mathbf{v}) \operatorname{div} (\mathbf{g} \times \mathbf{h}) \tag{60a}$$

$$\mathbf{v} \bmod D = (\mathbf{v} \times \mathbf{h}, \mathbf{g} \times \mathbf{v}) \bmod (\mathbf{g} \times \mathbf{h}) \tag{60b}$$

Uniform Spatial Partitioning

We saw how we can easily partition the plane into parallelograms. But what about other shapes, such as in Figure]reffig:HexagonPartitions?

Suppose P is a parallelogram with matrix D, and R is any set of points, and suppose that the function $f_D : R \to P$ defined by $f_D(r) = r \bmod D$ is a bijection.

Then we can partition space now into regions with the same shape as R.

For a point $\mathbf{v}$, we calculate which point in R has the same remainder when divided by D as $\mathbf{v}$. This is the location within the region of the point, and is given by $f_D^{-1}(\mathbf{v} \bmod D)$. We subtract this from $\mathbf{v}$;

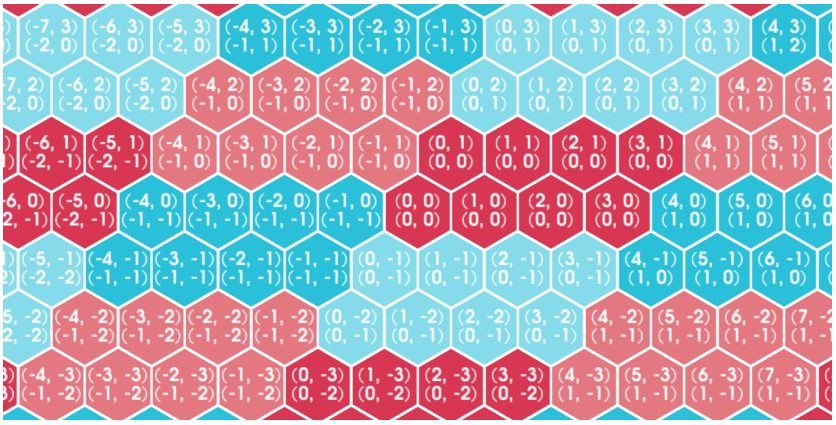

Figure 14: Partition with $\mathbf{g} = (4, -1)$ and $\mathbf{h} = (0, 2)$. In each cell, the top coordinate is the coordinate of the point $\mathbf{v}$, and the bottom coordinate is the partition index $\mathbf{v}$ div D.

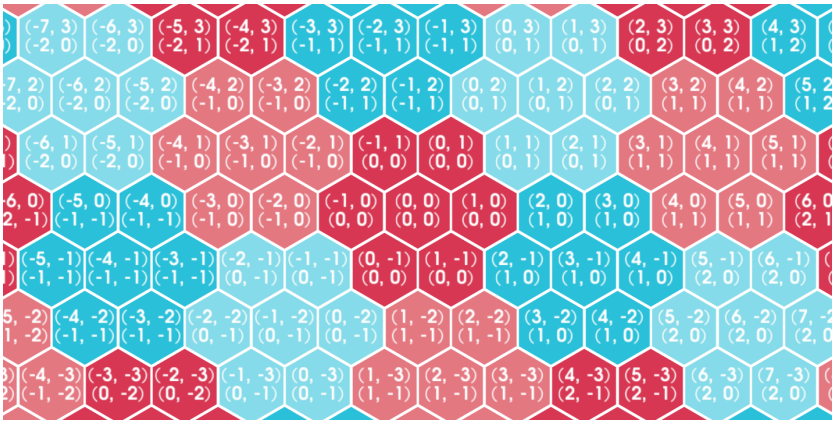

Figure 15: Partitioning a plane in hexagons.

this gives as the corner of the paralellogram. We can now divide this by D to find the partition index.

The partition index is given by

$$\mathbf{v} \operatorname{div} (R, D) = (\mathbf{v} - f_D^{-1}(\mathbf{v} \bmod D)) \operatorname{div} D \tag{61a}$$

and the location in the partition is

$$\mathbf{v} \bmod (R, D) = f_D^{-1}(\mathbf{v} \bmod D) \operatorname{div} D \tag{61b}$$

Wrapped Grids

Wrapped grids are often used in games. They have some properties that makes them attractive for game worlds:

- They are finite.
- They allow infinite movement along any direction.
- They have a different topology which makes different things possible. For example, they allow the player to catch an enemy from behind or the front.

These combine to allow us to simulate the feeling of being on a sphere, as is used in Civilization type games (even though it is really a topological torus). In square grid worlds, we easily implement wrapping using modular arithmetic on coordinates. And in hex worlds, we can use the same approach if our grid is in the shape of a parallelogram. But hex grids allows for another type of wrapping if our grid is in the shape of a hexagon.[5]

[5] There is a nice interactive example here: https://www.redblobgames.com/grids/hexagons/#wraparound.

While parallelogram wrapping resembles the surface on a torus, hexagon wrapping resembles the surface of a twisted torus. Figure 14 shows the same grid as Figure 15.

This wrapping has the strange effect that if you continue in a straight line, you will visit each cell before returning to the original cell. You can always hit an enemy by shooting in any (axis-aligned) direction!

Without the concept of point division we introduced earlier, this type of wrapping is difficult to implement: if you go out of the one end, it is not clear how to calculate the coordinate of where you will come in.

But with point division it is really trivial.

What we usually want is to calculate the following: where we are when we start at point $\mathbf{v}$ when travelling along $\mathbf{u}$. In a non-wrap grid, our location would be given by $\mathbf{u} + \mathbf{v}$. In a wrapped grid, the calculation is the same, except that we apply a wrapping function W to the result.

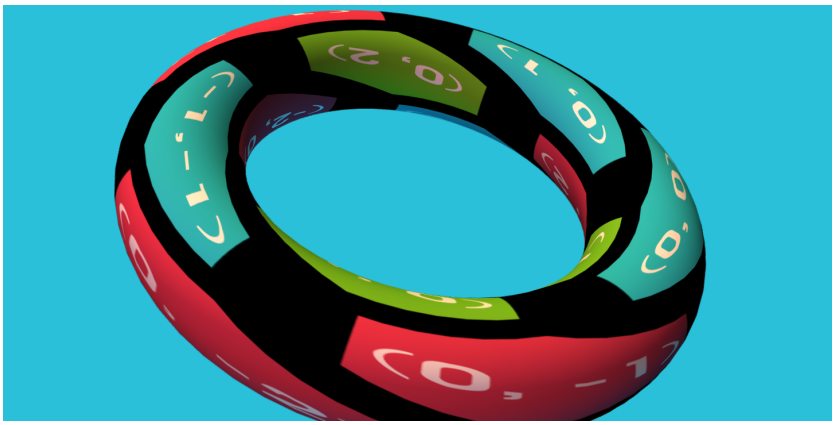

Figure 16: Wrapped Hex Grid on Torus. Notice the line formed by the grid points $(0,-2)$, $(0,-1)$, $(0,0)$, $(0,1)$, $(0,2)$, and $(-2,0)$.

The wrapping function is given by

$$W(\mathbf{v}) = \mathbf{v} \bmod (R, D) \tag{62}$$

where R is the points of our grid (the points for which $W(\mathbf{v}) = \mathbf{v}$), and D is the matrix of parallelogram that describes the repitition of our wrapping. There are two basic wrappings for hexagons that corresponds to whether the topological torus is twisted left or right. When R is a hexagon with sides of length n, two possible values for each type D are given by

$$D_L = \begin{pmatrix} 2n-1 & 1-n \\ n-1 & n \end{pmatrix} \tag{63a}$$

and

$$D_R = \begin{pmatrix} 2n-1 & -n \\ n & n-1 \end{pmatrix}. \tag{63b}$$

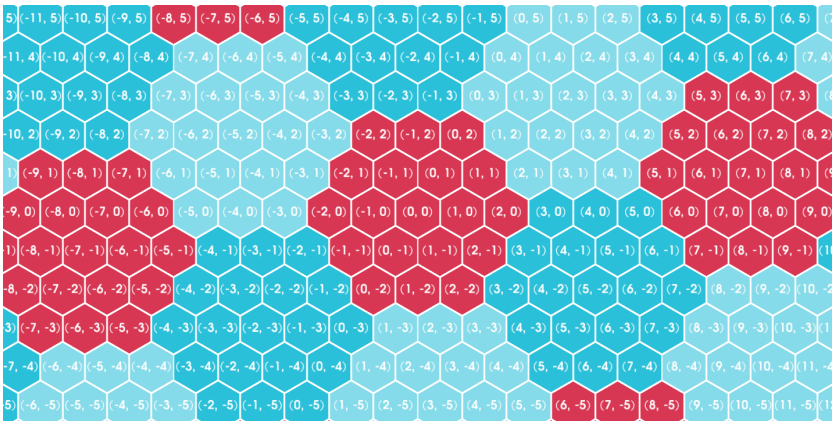

Figure 17: Hexagon wrapped with $D_L(3)$. Notice the line formed by the grid points $(0,-2)$, $(0,-1)$, $(0,0)$,$(0,1)$, $(0,2)$, and $(-2,0)$.

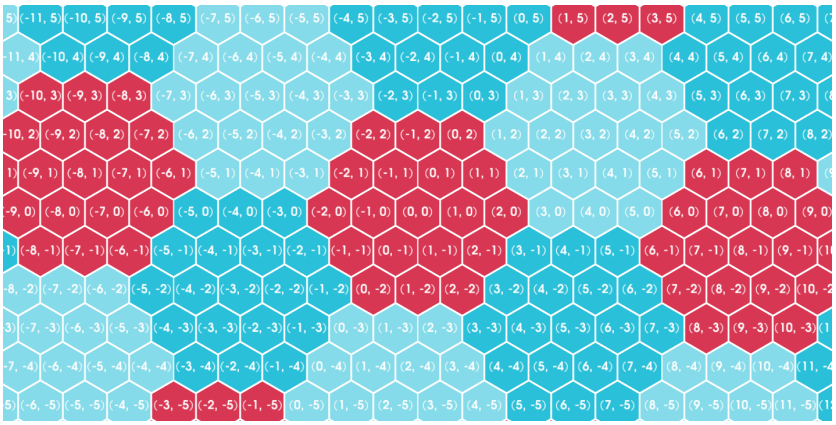

Figure 18: Hexagon wrapped with $D_R(3)$.

Recursive Spatial Partitioning

Squares have the nice property that we can dissect them and get squares again. Or put differently, we can stack $n \times n$ squares and get a square. We make use of this property in many algorithms and data-structures: quad trees, sampling, noise generation. Moreover, using modular and integer division, we can work on multi-res grids with ease (for example, if we impose a grid of half the resolution on an existing grid, we can get low res coordinates by integer-diving high-res by two). These manipulations are so trivial we hardly notice that we are doing something special.

When we try to apply those ideas to hex grids, it seems they break down. No number of hexes can be stacked to get a pure hexagon shape, or put differently, we cannot dissect a hexagon into smaller hexagons.

But hexagons can be stacked to form approximate hexagons, as shown in the figures below. And these can be stacked together again, and again. So some form of recursive partitioning is possible. The trick is now to get it into a form that makes it possible for us to write useful algorithms.

We already saw that we can partition space into hexagons. By applying this recursively, we can get partitions of the shapes in the images shown.

Let

$$P_0(\mathbf{v}) = \mathbf{v}. \tag{64}$$

Then the k-level partition is given by:

$$P_k(\mathbf{v}) = P_{k-1}(\mathbf{v}) \operatorname{div} (R, D) \tag{65}$$

where R is the 7-point hexagon, and $D = D_L(1)$.

While we used the ideas here to find somewhat regular subdivisions, they still work when our stacks are not hexagon(ish). In fact, they work for any region R if translated copies of R tile the plane and each copy has 6 neighboring copies. And because we don't ever use any property of hex grids (we only calculate directly on the coordinates), we can use the exact same scheme on square grids. For example, we can use the 5-cell cross as a primitive element, that gives us recursive space divisions that are not squares. And we can use the same scheme of noise generation on these cross shapes, and get results with far less artifacts (admittedly, at a higher computational cost).

And remember now that we can also represent other grids as hex or square grids, and the same ideas can be applied directly to those grids. So we have found an algorithmic tool that can be applied to any grid that is represented as either a square or hex grid. It is a bit tricky to find the coloring to use, but it can be made easier by offsetting the parallelogram slightly so that it does not fall on any edges or vertices. For example, if we want to recursively subdivide a tri-grid into hexes, we have to use a 9-coloring.

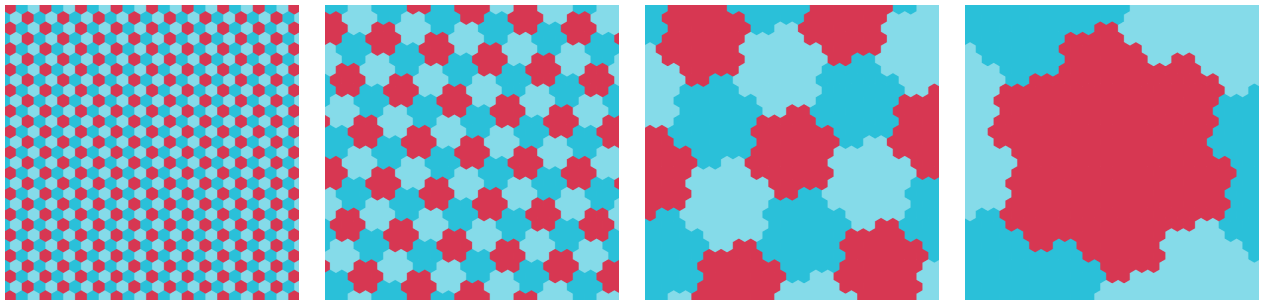

Figure 19: Stack hexagons together.

(Some extra tools are necessary for dealing with tilings which involve rotations or reflections, such as if we wanted to recursively subdivide a grid into triangle or rhomb shapes. I have not investigated these, but it seems that not much work should be required to handle these as well.)

The Gosper Curve

Suppose we have a grid partitioned at multiple resolutions, and we want to iterate through all points such that:

1. each point is one unit vector away from the previous point,
2. we don't leave a partition at any resolution before we visited all points in the partition.

For hex grids, one type of curve that has this property is called the

Gosper curve. (Strictly speaking, the Gosper curve is a fractal, and we are really interested in discrete approximations of it. We will call these approximations Gosper curves too.)

These curves are useful for a variety of applications; essentially, they give us a way to fake 2D algorithms using 1D. For example, the following images were generated using 1D spatially correlated noise.

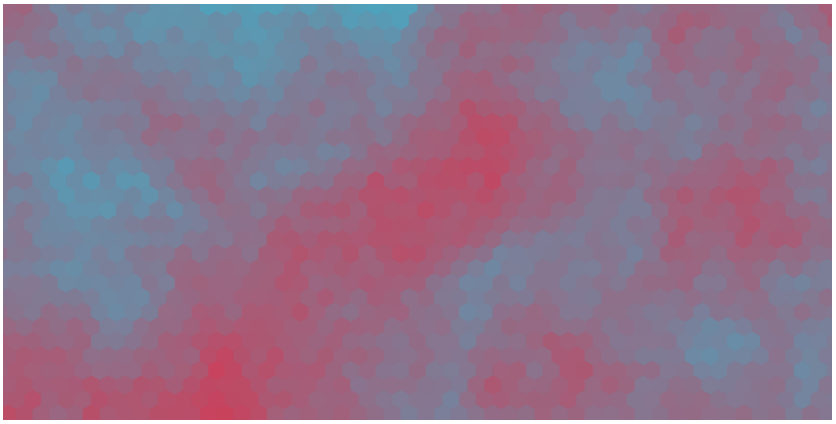

Figure 20: Noise.

These types of curves are neatly described by L-systems, and is easy to implement. If you don't know what an L-system is, here is a brief introduction:

1. An L-system is a set of rules that tells you how to replace symbols in a string with new strings.
2. The system has an axiom–a string of symbols as the starting string.

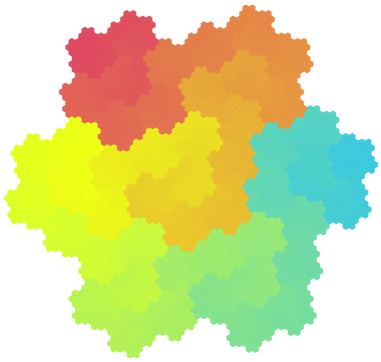

Figure 21: A Gosper traversal of a hex grid.

You can now take the axiom, and rewrite it using the rules, replacing each symbol with the appropriate string. You can do this repeatedly, giving you a new string each time.

These strings can then be interpreted as a sequence of commands to drive some other system to generate an object. To generate a Gosper curve, we will interpret the string as turtle graphics commands; commands that tell a "turtle" with a pen how to move with simple commands such as "go forward", "turn 60 degrees left" and so on. (This terminology comes from LOGO, an educational programming language famous for its use of turtle graphics.)

Here is the L-system for the Gosper curve, shown below: Axiom: X Rewrite Rules:

$$X \to X + YF + +YF - FX - -FXFX - YF +$$
$$Y \to -FX + YFYF + +YF + FX - -FX - Y.$$

Here, we interpret F to mean move one unit forward, $-$ to mean turn 60 degrees left, and $+$ to mean turn 60 degrees right. After k iterations of rewriting, we get a curve that spans y^k hexes.

Unfortunately, this does not describe cell-to-cell movements that we really want. However, each segment falls in a unique hexagon, in particular, their midpoints fall into the hexagons, and we can therefore easily get the movement that we want by calculating the midpoints of the segments and rounding those values to grid points.

The resulting sequence of points describes a path in a hex grid that performs a depth first traversal.

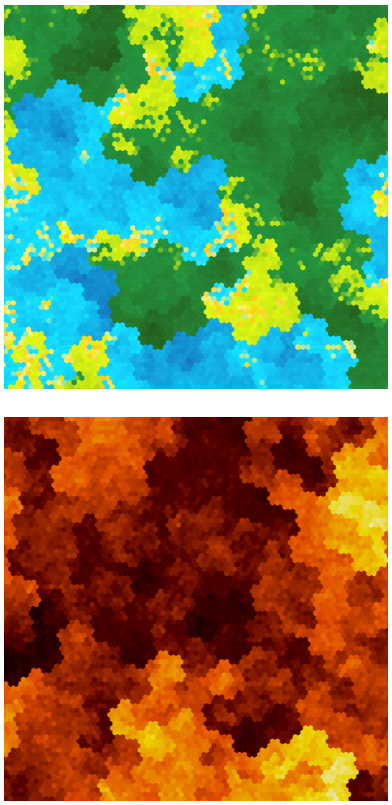

Figure 22: Procedural generation using a Gosper curve.

Grid Colorings

For the remainder of this section, we will only be concerned with grid points, so we restrict $\mathbf{v}$ and D to have only integer components, which means $\mathbf{r}$ will only have integer components too.

In many cases, we will only need $\mathbf{r}$ as a label; we don't actually care about the values of $\mathbf{r}$. For these cases, we make the following definition.

Let $c(\mathbf{v})$ be a function that assigns for each grid point within the parallelogram of D a unique integer between 0 and $N-1$, where N is the number of grid points in the parallelogram, given by

$$N = |\mathbf{g} \times \mathbf{h}|. \tag{66}$$

We extend this function to arbitrary grid points:

$$c(\mathbf{v}) = c(\mathbf{v} \bmod D) \tag{67}$$

We call this function a **coloring** of the grid by D. A coloring c by D has the property that for any two integers m and n, with $\mathbf{q} = (m, n)$,

$$c(\mathbf{v} + m\mathbf{g} + n\mathbf{h}) = c(\mathbf{v} + \mathbf{q}D) = c(\mathbf{v}). \tag{68}$$

It is a usual visual aid to color grid points by different colors depending on the value of $c(\mathbf{v})$, below we show a few such colorings.

I don't know of an algorithm to implement colorings for arbitrary $\mathbf{g}$ and $\mathbf{h}$. However, there is a relatively straightforward implementation if g is restricted to have $y = 0$. We can always choose suitable $\mathbf{g}$ and $\mathbf{h}$ with $g_y = 0$ for any coloring.

When $g_y = 0$, our coloring parallelogram always have n rows of equal length m. We can now use simple modular arithmetic and

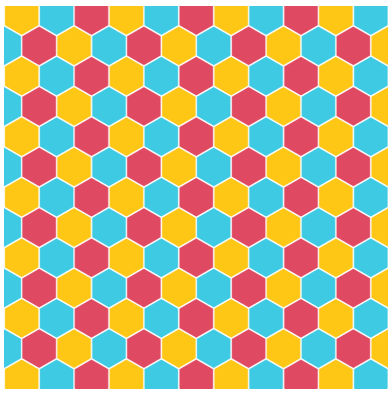

Figure 23: Regular 3-coloring, with $D = \begin{pmatrix} 0 & 3 \\ 1 & 1 \end{pmatrix}$.

offsets to calculate the colors as follows:

$$y_c = y \bmod n \tag{69a}$$
$$x_c = x \bmod m \tag{69b}$$

We combine these to get the final index:

$$c = y_c m + x_c. \tag{70}$$

In this scheme, the numbers in each row is typically not in order. However, the order is not important, since we use these color indices mostly for labeling. The fact that one point maps to 0 and another to 1 is not important. What is important is that they are different.

Three colorings are of particular importance; they are useful in a variety of ways.

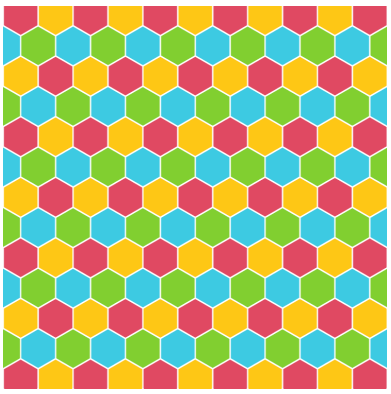

Figure 24: Regular 4-coloring, with $D = \begin{pmatrix} 0 & 2 \\ 2 & 2 \end{pmatrix}$.

Regular 3-coloring. This coloring is the only way to color a hex grid with the smallest number of colors so that no neighbors have the same color. This coloring is used for hexagonal chess, because it gives diagonally opposite cells the same colors, and therefor is a visual aid to movement of bishops, queens and kings. This coloring is used for representing triangular grids using hex grids (as explained in the next section).

Regular 4-coloring. In this coloring, opposite neighbors have the same color. We can use this coloring is we want to use a hex grid to represent faces and edges or faces and vertices simultaneously. The former makes this coloring useful in maze generation algorithms (such as Prim's algorithm). This coloring is used for representing rhombille grids using hex grids.

Regular 7-coloring. This coloring gives unique colors to each hex in partitioning into hexes. This coloring is used for representing floret pentagonal grids using hex grids.

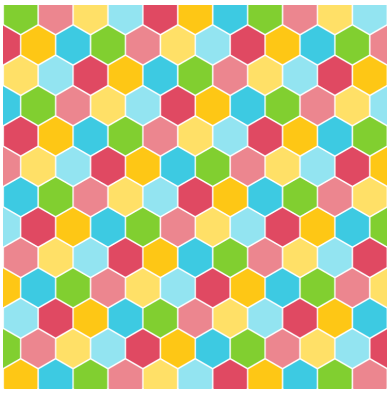

Figure 25: Regular 7-coloring, with $D = \begin{pmatrix} 0 & 7 \\ 4 & 1 \end{pmatrix}$.

Triangular grids

Triangular grids, despite their regular nature, are hard to work with algorithmically.

The usual coordination hides many of the symmetries, and make algorithms ugly and error prone, similar to the way in which different choices of coordinates for hex grids make algorithms ugly and error prone.

But there is a neat trick we can use to work with these grids, and get all the nice geometric advantages of hex grids.

We simply treat points on the hex grid as points on a tri grid grid. Figure 26 should explain what I mean:

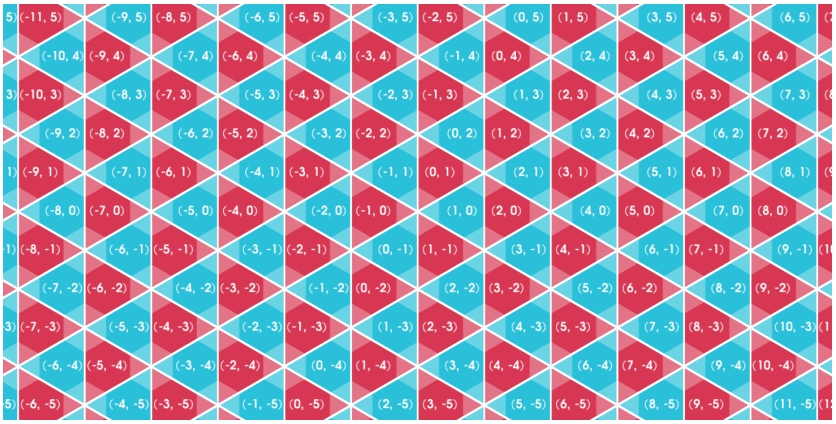

Figure 26: I hex grid superimposed oin a tri grid can be used to give a tri grid coordinates.

As you can see, not all hex points correspond to tri points. And at first glance, it looks like the coordinates do not capture all the information–when is a triangle up or down? How do we know a hex is not part of the system? But this information is actually nicely recovered using the 3-coloring you can see in the image.

Using this scheme, we can do all the geometry using the simple tools of hex grids.

In triangular grids, the grid lines we think of as important are parallel to the *minor axes*. They are either parallel, or intersect in *two* grid points. The grid point closest to the intersection of the associated real lines give you one of these points. The other can be found by checking all this point's neighbors.

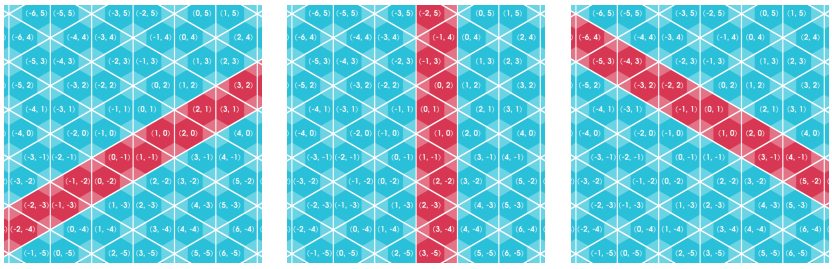

Figure 27: Lines corresponding to the equations $\mathbf{v} = (\frac{1}{2}, \frac{1}{2}) + nt(-1, 2)$, $\mathbf{v} = (1, \frac{1}{2}) + nt(1, 1)$, and $\mathbf{v} = (\frac{1}{2}, \frac{1}{2}) + nt(-2, 1)$.

We can use the same trick for other grids as well: for example, using appropriate colorings we can also represent rhombille grids and floret pentagon grids.

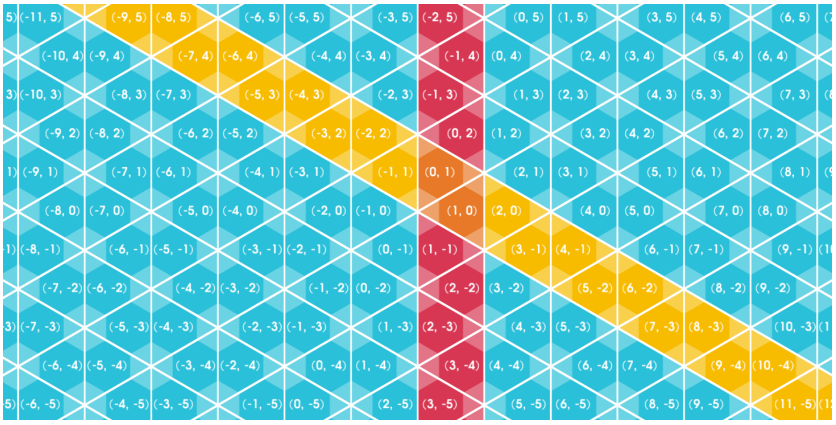

Figure 28: The intersection of two lines.

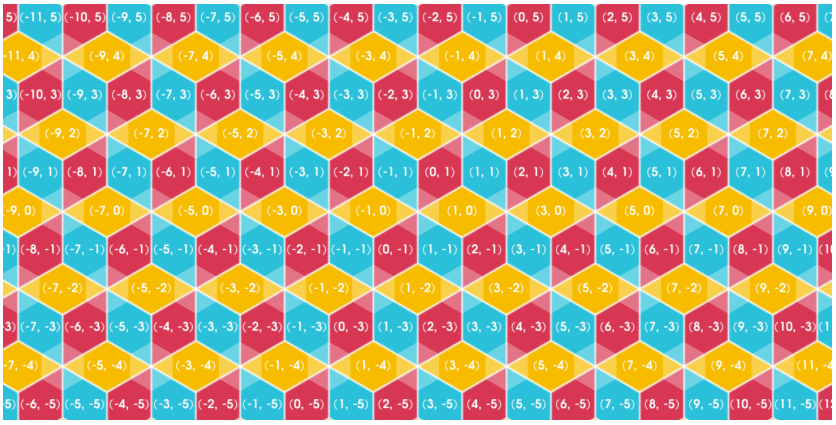

Figure 29: Rhombille Grid

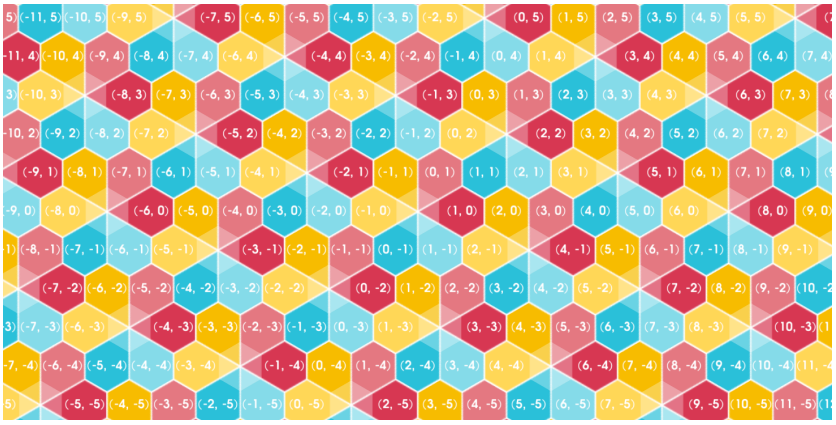

Figure 30: Floret Pentagon Grid