HexKit

Contact

Website: http://www.settworks.com/pages/tools/hexkit

Forum: http://www.settworks.com/forum/hexkit

Email: tanuki@gmail.com

Package Contents

Behaviors

Scripts in this folder are `MonoBehavior` components. They can be attached to scene objects and prefabs, and their effects can be seen in the editor.

- `HexLocation` lets you manually arrange objects on a hexagonal grid.
- `HexFacing` controls the rotation of an object by 1/12th-circle increments.

Logic

These classes **do not** inherit `MonoBehavior`, and cannot be attached directly to game objects! Instead, they are used within your other `MonoBehavior` scripts to handle logic related to hexagon grids.

- `HexCoord` represents a regular hexagon, and its location on a gridded plane.
- `HexKit` provides functions that operate across multiple cells on the `HexCoord` plane.
- `HexPathNode` and `HexRayHit` are used as argument and return types within `HexKit`.

To use these classes, your script file must contain the following line. Place it at the top of your script, outside any brackets:

```
using Settworks.Hexagons;
```

The sub-folder `Internal Classes\` contains only private classes. They are used within other `HexKit` scripts, but they cannot be used directly and you will never need to handle them.

Example

This folder contains a simple scene demonstrating several features of `HexKit`. Reading through its control script can help to get a feel for how `HexKit` works in practice.

HexLocation

Manages the relationship between an object's position in Unity space and its location on a hexagon grid.

Inspector Fields

When this behavior is attached to an object, the following parameters are available through the Inspector.

`Location`	*`HexCoord` grid cell containing this object. Updates as object is moved.*
`→ q, r`	*If edited directly, the object snaps to the center of its new cell.*
`Grid Scale`	*Ratio of Unity units to `HexCoord` units. Must be greater than zero.*
`Auto Snap`	*If enabled, the object always snaps to the center of its grid cell.*

Class Members

These fields, properties and methods are available from scripts.

Fields

`HexCoord` `location`	*as inspector Location, above.*
`float` `gridScale`	*as inspector Grid Scale, above.*
`bool` `autoSnap`	*as inspector Auto Snap, above.*

Setting these fields will automatically update the object's status the same as when edited through the Inspector, but not until the next `Update()`. If you want the object's state to change immediately, use the setter methods below.

Methods

`void` `SetPosition(`**`Vector3`** `position)`
Sets `transform.localPosition`, updates `location`, and moves to the center of `location` if `autoSnap` is enabled.

`void` `SetLocation(`**`HexCoord`** `location)`
Moves this object to the center of the supplied `location`.

`void` `SetAutoSnap(`**`bool`** `isEnabled)`
Enable or disable `autoSnap`. When enabled, snaps immediately.

`void` `SetGridScale(`**`float`** `scale)`
Sets `gridScale`. If `autoSnap` is enabled, moves the object to stay in the same `location`; otherwise updates `location` to use the new `gridScale`.

`void` `Snap()`
Snap to the center of `location`, regardless of `autoSnap` status.

HexFacing

Represents the six cardinal and six diagonal directions of a hexagon grid.

Inspector Fields

`Facing` — *This value is constrained from 0 to 5.5, in steps of 0.5. Whole numbers correspond to straight lines on the hexagon grid. Half steps correspond to diagonals.*

`Rotation Mode` — *A selection box with five options:*

- `Passive` — *Object rotation updates `Facing`. Changes to `Facing` rotate the object.*
- `Snap` — *Same as `Passive`, but any rotation snaps to the closest `Facing` direction.*
- `Enforce` — *The object cannot be rotated directly, only by setting `Facing`.*
- `Disable` — *The object's rotation is locked at zero, regardless of `Facing`.*
- `Independent` — *Object rotation has no effect on `Facing`, and vice versa.*

Only the Z axis is affected by `HexFacing`. X and Y rotations are unaffected.

Class Members

Scripts have access to the following.

Enumeration

```
RotationMode    { Passive, Snap, Enforce, Disable, Independent }
```

Fields & Properties

```
float facing                as inspector Facing, above.
RotationMode rotationMode   as inspector Rotation Mode, above.
int halfSextant             Read-only; equals (facing * 2).
```

If `facing` is truncated to an integer, it matches the format of `HexCoord` neighbor and corner indexing. The `halfSextant` property matches `HexCoord` half-sextant indexing. As with HexLocation, automatic changes to object state don't happen until `Update()`; use methods for immediate changes.

Methods

```
void SetAngle(float angle)
```
Sets local Z rotation, then applies the current `rotationMode` policy.

```
void SetFacing(float facing)
```
Sets `facing`, then applies the current `rotationMode` policy.

```
void SetHalfSextant(int halfSextant)
```
Equivalent to `SetFacing(halfSextant * 0.5f)`

```
void setRotationMode(RotationMode mode)
```
Sets `rotationMode`, then applies the new policy to local Z rotation, then to facing.

HexCoord

Represents the location and geometry of one cell on a gridded plane of regular hexagons with sides 1 unit long.

Concepts

This is a brief overview of concepts essential to fully utilizing `HexCoord`. For a more thorough primer, visit http://www.redblobgames.com/grids/hexagons/.

QR Coordinates

`HexCoord` indexes cells in a hexagon grid using two non-perpendicular axes. The `q` axis maps directly to Unity's `x` axis, but the positive `r` axis is rotated only 60 degrees from `q` instead of the 90 degrees of the Cartesian `y` axis. This system was chosen because it allows vector operations: adding and subtracting works the same with `HexCoord` as with `Vector2`.

Cubic Coordinates

The QR system is a simplification of a three-axis system, based on the observation that a hexagon is the maximal cross-section of a cube. When viewed from an angle equidistant from all three axes, the standard XYZ coordinate system can be used to index a hexagonal plane. Multiple coordinate sets project to the same location on the plane, but unique indices can be achieved with the constraint that `{x + y + z = 0}`. Furthermore, with that constraint in place, knowing position on any two axes allows you to derive the third. This is the basis for the QR system: two projected cubic axes, with the third axis inferred.

Polar Coordinates

On a hexagon grid, a circle approximates to a hexagon. This allows convenient use of polar coordinates. Using real angles isn't optimal with integer grid indices, so HexCoord uses a ring position index instead. (Conversion from real angles is available.) At each radius, cells are numbered counterclockwise starting from 0 (on the positive `q` axis). The number of hexagons in a given ring is that ring's radius times 6.

Offset Coordinates

Another way of indexing a hexagon grid, and probably the most common, is to simulate the perpendicular Cartesian axes by “staggering” indices. Horizontal motion follows a straight line of grid cells on the `x` axis, but vertical motion zig-zags to follow the `y` axis. This can simplify working in rectangular spaces (*e.g.* the screen), and many references and tutorials on hexagon geometry use this system, so `HexCoord` supports conversion to and from it.

Neighbors, Corners, Sextants

Each grid cell has six neighbors, and six corners. These are indexed in counterclockwise order. The cell to the right (`q + 1`) is neighbor `0`, and the upper corner of their shared side is corner `0`. Each neighbor and corner index can also represent the central angle of a sextant, which is a 60 degree arc (one sixth of a circle) radiating from the center of the hexagon. Neighbor sextants are bounded by the two corners on that side of the hexagon, and corner sextants are bounded by the midpoints of the two sides adjacent to that corner.

Class Members

All public members of `HexCoord` are described below. These conventions apply throughout:

- A cell's position in Unity space is its geometric center.
- Angles are in radians. Angles taken as method arguments are normalized internally, and this also applies to rotational indices (e.g. neighbor, corner, polar).
- An angle or rotational index returned by a method is **not** guaranteed to be normalized.

Constructor

`HexCoord``(`**`int`** `q, `**`int`** `r)`

Initializes a new instance of the `HexCoord` struct with the specified `q` and `r` values.

Fields & Properties

`int` `q` — *Position on the `q` axis. Equivalent to position on the cubic `x` axis.*

`int` `r` — *Position on the `r` axis. Equivalent to position on the cubic/offset `y` axis.*

`int` `Z` — *Position on the cubic `z` axis, `-q-r`.*

`int` `O` — *Position on the offset `x` axis, `r/2 + q`.*

Static Property

`HexCoord` `origin`

The cell at grid position `(0,0)`.

Methods

`Vector2` `Position()`

The position of this cell in Unity space.

`int` `AxialLength()`

This cell's maximum absolute position on any cubic axis. Equal to the number of grid steps from `origin`, and to the radius of the polar ring containing this cell.

`int` `AxialSkew()`

This cell's minimum absolute position on any cubic axis. Equivalent to the number of grid steps from `origin` which deviate from the most-traveled axis.

`float` `PolarAngle()`

The angle from `origin` to this cell. Returns `0` at `origin`.

`int` `PolarIndex()`

The polar index of this cell.

`int` `NeighborSextant()`

The neighbor-indexed sextant from `origin` containing this cell. Returns `0` at `origin`.

int CornerSextant()
The corner-indexed sextant from `origin` containing this cell. Returns `0` at `origin`.

int HalfSextant()
Ignoring fractions, `HalfSextant/2` is equivalent to `CornerSextant` and `(1+HalfSextant)/2` is equivalent to `NeighborSextant`.

HexCoord Neighbor(**int** index)
This cell's immediate neighbor at `index`.

IEnumerable<HexCoord> Neighbors(**int** first = 0)
This cell's six neighbors in counterclockwise order, starting with `first`.

HexCoord PolarNeighbor(**bool** CCW = false)
This cell's immediate neighbor on the ring at its distance from `origin`. Selects the counterclockwise neighbor if `CCW` is `true`, clockwise by default.

Vector2 Corner(**int** index)
The position of the corner of this cell at `index`, in Unity space.

IEnumerable<Vector2> Corners(**int** first = 0)
The positions of this cell's six corners in Unity space, in counterclockwise order, starting with `first`.

float CornerPolarAngle(**int** index)
The angle from `origin` to the corner of this cell at `index`.

int PolarBoundingCornerIndex(**bool** CCW = false)
Returns the index for one of two corners which define an arc from `origin` that contains this entire cell. Selects the counterclockwise corner if `CCW` is `true`, clockwise by default. Returns `0` at `origin`.

float PolarBoundingAngle(**bool** CCW = false)
As `PolarBoundingCornerIndex()` but returns the corner's angle from `origin`.

Vector2 PolarBoundingCorner(**bool** CCW = false)
As `PolarBoundingCornerIndex()` but returns the corner's position in Unity space.

HexCoord SextantRotation(**int** sextants)
Rotates this cell counterclockwise around `origin`, in sextant increments.

HexCoord Mirror(**int** axis = 1)
Mirrors this cell across a line defined by two opposite corners of `origin`. `Axis` is either of those corners.

HexCoord Scale(**int** factor)
Multiplies this cell's `q` and `r` by `factor`.

HexCoord Scale(**float** factor)
Multiplies this cell's `q` and `r` by `factor`, truncating results to integers.

Vector2 ScaleToVector(**float** factor)
Multiplies this cell's `q` and `r` by `factor`. The resulting `Vector2` is in QR space!

bool IsOnCartesianLine(**Vector2** a, **Vector2** b)
Returns true if this cell touches the infinite line passing through points `a` and `b`.

bool IsOnCartesianLineSegment(**Vector2** a, **Vector2** b)
Returns true if this cell touches the line segment between points `a` and `b`.

bool IsWithinRectangle(**HexCoord** cornerA, **HexCoord** cornerB)
Returns true if this cell lies within (or is on the border of) the rectangle defined by the positions of `cornerA` and `cornerB` in Unity space.

Operators

+ $q'=q_1+q_2$, $r'=r_1+r_2$
- $q'=q_1-q_2$, $r'=r_1-r_2$
== $q_1==q_2$ && $r_1==r_2$
!= $q_1!=q_2$ || $r_1!=r_2$

Cast

explicit (**Vector2**)**HexCoord**
Note: the resulting `Vector2` is in QR space!

Override Methods

string ToString()
"(q,r)"

bool Equals(**object** o)
Returns `true` if `o` is a `HexCoord` with `q` and `r` equal to this `HexCoord`.

int GetHashCode()
Calls `GetHashCode()` on a `uint64` constructed from `q` as high bits and `r` as low bits.

Static Methods

int Distance(**HexCoord** a, **HexCoord** b)
The number of grid steps from `a` to `b`, equivalent to `(a - b).AxialLength()`.

HexCoord NeighborVector(**int** index)
As `origin.Neighbor(index)`.

IEnumerable<HexCoord> NeighborVectors(**int** first = 0)
As `origin.Neighbors(first)`.

int AngleToNeighborIndex(**float** angle)
Index of the neighbor of `origin` entered by a ray at `angle` from its position.

float NeighborIndexToAngle(**int** index)
Angle of a ray from `origin` to its immediate neighbor at `index`.

Vector2 CornerVector(**int** index)
As `origin.Corner(first)`.

IEnumerable<Vector2> CornerVectors(**int** first = 0)
As `origin.Corners(first)`.

int AngleToCornerIndex(**float** angle)
Index of the corner of `origin` nearest to a ray at `angle` from its position.

float CornerIndexToAngle(**int** index)
Angle of a ray from `origin` to its corner at `index`.

int NormalizeRotationIndex(**int** index, **int** cycle = 6)
Normalizes `index` to lie between `0` and `cycle - 1`.

bool IsSameRotationIndex(**int** a, **int** b, **int** cycle = 6)
Returns true if `a` and `b` are equal after normalizing to `cycle`.

HexCoord AtPosition(**Vector2** position)
The cell containing a Unity-space `position`.

HexCoord AtPolar(**int** radius, **int** index)
The cell at polar `index` on the ring at `radius` from `origin`.

int FindPolarIndex(**int** radius, **float** angle)
The polar index of the cell at `radius` centered nearest to `angle`.

HexCoord AtOffset(**int** x, **int** y)
The cell at offset coordinates `(x,y)`.

HexCoord FromQRVector(**Vector2** QRvector)
Finds the cell containing a non-integer location in QR space.

Vector2 VectorXYtoQR(**Vector2** XYvector)
Transforms a `Vector2` from XY (Unity) space to QR space.

Vector2 VectorQRtoXY(**Vector2** QRvector)
Transforms a `Vector2` from QR space to XY (Unity) space.

HexCoord[] CartesianRectangleBounds(**Vector2** cornerA, **Vector2** cornerB)
Returns an array of two HexCoords defining the minimum rectangle in QR space containing every cell intersecting the Unity-space rectangle defined by cornerA and cornerB.

HexKit

Static class providing a library of hexagon grid operations and related tools.

Concepts

HexKit makes use of enumerable methods, function passing, and `out` arguments. These C# features are briefly explained here.

Enumerable Methods

Not to be confused with the enumeration (`enum`) type, an enumerable method is one whose return type is `IEnumerable` or the generic `IEnumerable<T>`. (`HexKit` uses only the latter.) These interfaces reside in the `System.Collections` and `System.Collections.Generic` namespaces respectively, and `System.Linq` gives them many useful extension methods. There are several ways to use this data type, but the simplest is the `foreach` statement, an iterator which steps through a data set operating on each element with a block of code. This is convenient in its own right; but when used with `IEnumerable`, the “data set” is actually a function generating data on demand.

Here's a practical example. `HexKit` can generate a “supercover” line: given starting and ending coordinates, it finds all the grid cells passed through by a real line between the center points of the two ends. This can be used for line of sight tests by passing the method as a `foreach` data source. In the code block, check whether the current cell obstructs the line. If not, let it continue; but if so, line of sight is broken. You can break the loop right there and the rest of the line is never generated, saving cycles.

Passing Functions

In C#, functions are just another data type. They can be stored in variables, and passed as arguments to other functions. Many `HexKit` methods use functions as arguments, to avoid placing any constraints on how you store and organize your data. You can supply these arguments in two ways: pass in a named function defined elsewhere (*e.g.* a method in your class), or define it in-line with a lambda expression. The only restriction is that the function you supply must have the right argument and return types.

`out` Arguments

Put simply, an `out` argument represents an additional return value. Instead of passing data in, this argument takes a variable into which data will be put. `HexKit` uses this for its pathfinding methods, which return `true` or `false` depending on whether a path was found; the path itself, if one exists, is passed through an `out` argument.

Class Members

HexKit contains only static methods. They fall into several categories, explained below.

Mesh Generation

These methods create new meshes every time they are called. If multiple game objects using the meshes generated here will only ever differ in scale and/or position, consider sharing the same mesh between them. Note: the radius of a regular hexagon is the distance from its center to any corner, which is also the length of each side.

```
Mesh CreateMesh()
```
Creates a 4-triangle regular hexagon mesh of radius 1.

```
Mesh CreateOpenMesh(float inner = 0, float outer = 1)
```
Creates a 12-triangle regular hexagon "outline" mesh of radius `outer`, with a central opening of radius `inner`. Both arguments are constrained to be not less than 0, and `inner` is constrained to be not greater than `outer`.

Spatial Grouping

Methods in this category enumerate grid cells within geometric regions.

```
IEnumerable<HexCoord> WithinRange(HexCoord center, int radius,
                                  bool border = false)
```
Enumerates all cells within `radius` of `center`. If `border` is `true`, instead enumerates only those cells at the outer edge of this range.

```
IEnumerable<HexCoord> WithinRect(HexCoord cornerA, HexCoord cornerB,
                                 bool border = false)
```
Enumerates all cells for which `HexCoord.IsWithinRectangle(cornerA, cornerB)` would return true. If `border` is `true`, instead enumerates only those cells at the outer edge of this rectangle.

```
IEnumerable<HexCoord> Ring(HexCoord center, int radius)
```
Enumerates the same cells as `WithinRange(center, radius, true)`, in the order of their `HexCoord` polar indices. Unless this ordering is useful, prefer `WithinRange` for speed.

```
IEnumerable<HexCoord> Line(HexCoord from, HexCoord to,
                           bool supercover = false)
```
Enumerates cells along a line between `from` and `to`. The supplied endpoints are not returned. By default this is a Bresenham line, one cell across at all points. If `supercover` is `true`, every cell the real line crosses is returned.

Radiant Effects

Field of view is one common application of these methods, but with appropriate function parameters they are useful for any area-effect that is blocked or diminished by obstacles. **Warning:** so long as there are visible arcs, these methods don't stop! Limit their range by returning `null` for cells outside a reasonable area.

```
IEnumerable<HexCoord> Radiate(HexCoord origin,
                              Func<HexCoord, bool?> IsTransparent)
```

Enumerates cells which can be "seen" from `origin`. Each cell is checked against `IsTransparent`; this function should return `true` if a cell provides no obstruction, `false` for a cell which blocks those behind it but will itself be returned if not blocked, and `null` for a blocking cell which will never be returned.

```
IEnumerable<HexRayHit> Radiate(HexCoord origin, int intensity,
                               Func<HexRayHit, int?> Attenuation)
```

Enumerates cells from center where `intensity` is greater than `0`. Each cell is checked against `Attenuation`. With `Attenuation >= 0`, cells beyond this one receive that much less `intensity`. With `Attenuation < 0` or `Attenuation == null`, cells beyond this one are completely blocked. A cell is returned if it receives `intensity > 0`, except that `null` cells are never returned.

Pathfinding

HexKit performs three kinds of pathfinding: bounded spread (a.k.a. movement range), shortest path to nearest qualifying target, and shortest path to a single specific target. Each of these has two variants: one optimized for uniform grids where all movement has the same cost, and is either allowed or blocked; and one where movement costs can vary, including asymmetrical costs where A → B is not the same as B → A. The `HexPathNode` type is used to encapsulate path data.

Uniform vs. Non-Uniform

On a uniform grid, the cost to move between adjacent cells is always 1. Uniform pathfinding methods take a function argument `Predicate<HexCoord> IsObstacle`. This function should return `true` if that grid cell cannot be entered, otherwise `false`.

Non-uniform methods take `Func<HexPathNode, HexCoord, uint> MoveCost`. The first argument is the grid cell the path is moving out of. This is a `HexPathNode` to allow direction of motion and total distance of the path so far to be factored in if desired. The second argument is the cell being entered. The returned value is what it costs to enter the new cell from the old one – not the total path cost, only this single movement. Lower costs allow easier movement, to a minimum of 1. Returning `0` means that this movement is completely blocked.

Non-uniform methods also take `bool uniformOnEntry`, defaulting to `false`. Setting this to `true` means that your `MoveCost` function considers only the cell being entered, always returning the same value for that cell; this allows optimization of the path search.

Spread

The `Spread` methods perform movement cost analysis without searching for a target. They are bounded by a range, which is the maximum total cost of a path.

```
IEnumerable<HexPathNode> Spread(HexCoord origin, int range,
                                Predicate<HexCoord> IsObstacle)
```

Enumerates all grid paths from `origin`. Each cell reduces the remaining range of its path by 1, and a cell is not entered if the remaining range is zero.

```
IEnumerable<HexPathNode> Spread(HexCoord origin, int range,
                                Func<HexPathNode, HexCoord, uint> MoveCost,
                                bool uniformOnEntry = false,
                                bool permissive = false)
```

Enumerates all grid paths from `origin`. Each cell's `MoveCost` is deducted from the remaining range of its path. By default a cell not entered if its `MoveCost` is greater than the remaining range; if `permissive` is `true`, a cell is entered unless there is no remaining range.

Path to Nearest Target

These `Path` methods use the same core search logic as `Spread` (Dijkstra's algorithm), but are not bounded by a range and do not enumerate nodes during the search. Instead, the method terminates the first time `IsTarget` returns true for a cell, or when it runs out of legal nodes. **Warning:** if no target is found, these methods don't stop! Limit the search space by returning `IsObstacle==true` or `MoveCost==0` for cells outside a reasonable area.

```
bool Path(out HexPathNode[] path, HexCoord origin,
          Predicate<HexCoord> IsTarget, Predicate<HexCoord> IsObstacle)
```

Finds the shortest path from `origin` to any cell where `IsTarget` returns `true`, halting if no more cells are accessible. If a target is found, returns `true` and `path` contains the path to that target; if no target is found, returns `false` and `path` is `null`.

```
bool Path(out HexPathNode[] path, HexCoord origin,
          Predicate<HexCoord> IsTarget,
          Func<HexPathNode, HexCoord, uint> MoveCost,
          bool uniformOnEntry = false)
```

Path to Specific Target

With a single `target` known in advance, a much faster algorithm can be used (A*) and there is no danger of an infinite search. Otherwise, these behave the same as nearest-target.

```
bool Path(out HexPathNode[] path, HexCoord origin, HexCoord target,
          Predicate<HexCoord> IsObstacle)
```

Finds the shortest path from origin to target, halting if no more cells are accessible. If a target is found, returns `true` and `path` contains the path to that target. Otherwise, returns `false` and `path` is `null`.

```
bool Path(out HexPathNode[] path, HexCoord origin, HexCoord target,
          Func<HexPathNode, HexCoord, uint> MoveCost,
          bool uniformOnEntry = false)
```

HexPathNode

Decorates a `HexCoord` with information about that cell's place in a specific path.

Instances of this class cannot be created directly, but you'll need to handle them when using `HexKit` pathfinding features. `HexPathNode` represents one step in a particular path (a “node”) with three pieces of data: a `HexCoord` identifying the location of this node, the length (or cost) of the path up to and including this node, and the location of the previous step in the same path (the “parent node”).

Properties

`HexCoord` `Location`
: *The grid cell containing this path node.*

`int` `PathCost`
: *Total cost of the path up to and including this node.*

`int` `FromDirection`
: *`HexCoord` neighbor index to parent node's cell.*

`HexCoord` `Ancestor`
: *Parent node's cell, same as `this.Location.Neighbor(FromDirection)`.*

HexRayHit

Represents the collision of an intensity-weighted ray with a `HexCoord`.

Like `HexPathNode`, this class cannot be directly instantiated; but it must be handled when using the “intensity” variant of `Radiate()`.

Properties

`HexCoord` `Location`
: *The grid cell where this collision occurs.*

`int` `Intensity`
: *Intensity of the ray at the point of collision.*

`float` `Angle`
: *Angle from the collision point back to the ray's origin.*