Grid Framework

1.7.2

Generated by Doxygen 1.8.7

Tue Jan 27 2015 13:09:34

Contents

Chapter 1

Index

Welcome to the user manual and scripting reference for Grid Framework. This document consists of two parts, the manual and the API reference. Read the manual to get a good understanding of what Grid Framework is and how it works, and read the reference for details about the API. Aside from that you can also jump straight into the included examples (in your project view under *Grid Framework/Examples*) and see Grid Framework right in action.

An online version of this documentation can be found under `this URL`. Feel free to bookmark the link and and throw away the local documentation files. They can be found in the *WebPlayerTemplates* folder of your project. If you delete the local documentation the help menu will automatically forward you to the online URL as well.

Note

> Some images have been scaled down automatically in order to maintain the flow of text. In that case you have to right-click them and choose to open in a new tab or window to see the image in its full resolution. The maximum size of images scales with font size, so increasing font size will enlarge the images as well until they reach their native size.

Table of Contents

The user manual consist of the following pages:

- Page Overview gives you a rundown of all included files.
- Page Understanding the concept of grids explains the general idea behind grids in Grid Framework.
- Page Abstract Grids contains information on the bases class all grids inherit from.
- Page Rectangular Grids discusses rectangular grids.
- Page Hexagonal grids discusses hexagonal grids.
- Page Polar grids discusses polar grids.
- Page Rendering and drawing a Grid explains to display grids in the editor and in the game and the associated performance costs.
- Page Getting Started explains how to find the features you are looking for in the interface.
- Page Events features an introductory example for built-in events in Grid Framework.
- Page Extending Grid Framework demonstrates two ways of extending the capabilities of Grid Framework to your liking without changing the source code.
- Page Debugging a grid explains hot to debug and analyze your grids.
- Page Legacy Support contains information on features that were either dropped or changed and how to adapt your code.

- Page Playmaker support discusses the built-in Playmaker actions and how to write your own actions. (Playmaker is a separate addon for Unity and not affiliated with Grid Framework)
- Page Changelog contains the changelog, it's the same file you can also find under Grid Framework/↩ Changelog.

You can find the API scripting reference under *Classes* either in the sidebar or in the top bar. Form there you have various sorting options.

Note

Due to limitations of Doxygen the API can only be documented with C# syntax. Despite this you can still write your code in UnityScript, there are no C# exclusive features. Most of the examples are written in both languages as well, except for the ones that make use of special language features not present in UnityScript.

Chapter 2

Overview

Preface

Thank you for choosing Grid Framework for Unity3D. My goal was to create an easy to use setup where the user only needs to say what kind of grid he or she wants and let the computer do all the math. After all, isn't that why computers where invented in the first place?

This package provides you with a new Grid component which can be added to any GameObject in the scene, lots of functions to use with the grid and an editor panel for aligning and scaling objects in your scene. Every grid is three-dimensional and infinite in size. You can draw your grids in the editor using gizmos, render them at runtime and even use Grid Framework along with `Vectrosity` if you have a license for it.

By buying this package you support further development and you are entitled to all future upgrades for free. My long-term goal is to create a complete grid framework with rectangular grids, hex grids, triangular grids, polar grids and even pathfinding... we'll see. In the meantime please enjoy the current system and thank you for your support.

HiPhish

Setup

Nothing special is needed if you get Grid Framework form the Unity Asset Store. If you accidentally moved some files and are getting compilation errors you can restore things back to normal by placing the files in the same locations as described below. You can also delete the global *Grid Framework* folder or any of its contents without breaking anything if you don't need it.

All features of Grid Framework, including the documentation, can be accessed through the menu bar, see the Getting Started page for more information, there is no reason to attach scripts manually.

What is included

The package contains four global folders, one called *Grid Framework*, the other *Editor*, the third one *Plugins* and the fourth one *WebPlayerTemplates*.

The *Grid Framework* folder contains the documentation, the changelog and two folders, *Examples* and *Debug*.

The *Examples* folder contains complete showcase scenes which demonstrate how to implement real gameplay functionality. I recommend new users to take a look at some of the scripts to see Grid Framework right in action. All scripts are written both in C# and JavaScript (if possible) and are extensively commented. The Vectrosity examples need a copy of Vectrosity installed to work. `Vectrosity` is a separate vector line drawing extension for Unity written by Eric5h5 and not related to Grid Framework. Please see the rendering section for more information.

The *Debug* folder contains a prefab and a script, both used for debugging the grid. Use the prefab to examine the grid both in the editor and during runtime.

The *Plugins* folder contains the folder *Grid Framework* containing the scripts *GFRectGrid.cs*, *GFHexGrid.cs* and

GFPolarGrid.cs, and the folders *Abstract* (containing *GFGrid.cs* and *GFLayeredGrid.cs*), *Extension Methods* (containing *GFTransformExtensions.cs* and *GFVectorThreeExtensions.cs*), *Vectors* (containing *GFBoolVector3.cd* and *GFColorVector3.cs*), *Grid Rendering* (containing *GFGridRenderCamera.cs*, which needs to be attached to a camera if you want to render your grids at runtime and *GFGridRenderManager.cs*) and a folder named *Legacy Support* containing zipped scripts for deprecated methods, included for optional backwards compatibility.

Out of these only *GFRectGridcs*, *GFHexGrid.cs* and *GFPolarGrid.cs* are relevant to end-users, *GFGrid.cs* and *G↩ FLayerdGrid.cs* are abstract classes and cannot be added to Objects (more on that later) and the rest just contain supplementary classes and functions or handy extension methods I used.

The *Editor* folder contains the folder *Grid Framework* with *GFGridAlignPanel.cs*, the alignment and scaling panel, *GFMenuItems.cs* for putting entries in Unity's menus and the folder *Inspectors*. That folder contains the abstract base class *GFGridEditor* and its derived classes, *GFRectGridEditor*, *GFHexGridEditor* and *GFPolarGridEditor*, for the actual inspector panels. None of these scripts are of interest to end-users, they just embed Grid Framework into the Unity Editor's interface.

Chapter 3

Understanding the concept of grids

Anatomy of a grid

Each grid is defined by a handful of basic properties, some of them common to all, some of them specific to a certain type. The common ones are the position and rotation, which are directly taken from the `Transform` of the `GameObject` they are attached to. The origin of each grid lies where its position is. When we look at a grid we see certain patterns: edges, vertices where edges cross, faces, which are areas enclosed by edges, and boxes, which are spaces enclosed by faces. Aside from edges (which are just pairs of vertices), all of these can get coordinates assigned. We can then either determine on which vertex, face or box of the grid we are or where a certain vertex, face or box lies in world space. The sections for each of the grids contain more detailed information.

The GFGridPlane

The 3D nature of these grids allow us to move in any direction, but sometimes you want or have to restrict things to 2D. In Unity planes are defined by their normal vector, but in grids we can make use of the grids' restrictive nature, so we only have to specify if it's an XY-, XZ- or YZ-plane. The scripting reference section contains details about the proper syntax.

Chapter 4

Abstract Grids

There are two abstract base classes, GFGrid and its child GFLayeredGrid, from which other grids inherit. These classes exist to provide an abstract template and they cannot be instantiated. It is very similar to Unity's abstract `Collider` class and its subclasses `BoxCollider`, `SphereCollider`, `MeshCollider` and so on. The child classes have all members of the parent class, but the exact implementation of methods can vary.

The GFGrid class

All grids inherit from the GFGrid class. GFGrid is an abstract class, which means it cannot do anything on its own and it cannot be attached to any object. Rather it serves as a template for what its children, the specific grids, have to be able to do, but not how they do it. You say that GFRectGrid, GFHexGrid and GFPolarGrid "implement" GFGrid.

Of course you can still reference any grid by its respective type like this:

```
var myGrid: GFRectGrid // UnityScript
GFRectGrid myGrid // C#
```

The downside is that if you change your opinion and want another kind of grid you need to change your source code as well. Or maybe you don't know what type of grid you are dealing with. Writing `var myGrid:  GFGrid` lets you use any sort of grid you desire and always picks the right implementation of the function. So the following code will always return the appropriate vector for any grid type:

```
var myGrid: GFGrid;
var myVec: Vector3 = myGrid.NearestVertexW(transform.position);
```

Please see the scripting reference sections for detailed information on the API.

The GFLayeredGrid class

Layered grids are all grids that are two-dimensional and stacked on top of each other, like layers. Currently this category contains hexagonal grids and polar grids. This class encapsulates their two common members: the *depth*, i.e. how far apart these layers are, and the *plane* (XY, XZ or YZ) the grid is aligned to. Like all grids layered grids are children of the aforementioned GFGrid.

Chapter 5

Rectangular Grids

The GFRectGridClass

This is your standard rectangular grid. It is defined by `spacing`, a Vector3 that describes how wide the faces of the grid are, and the shearing of the grid's axes. All three coordinates, X, Y and Z, can be adjusted individually.

The central vertex has the grid coordinates (0, 0, 0) and each coordinate goes in the direction of the grid's corresponding axis. Practically speaking, the vertex (1, 2, 3) would be one unit to the right, two units above and three units in front of the central vertex (all values in local space and multiples of `spacing`). Thus vertex coordinates are the same as the default grid coordinates.

The coordinates for faces and boxes follow this pattern as well and they are given by their central point. This means the coordinates of the result from `NearestBoxG` will always contain half a unit:

```
var box = myGrid.NearestBoxG (transform.position) // something like (1.5, 2.5, -0.5)
```

The same applies to faces, however the coordinate that lies on the plane (e.g. Z for XY grids) will be a whole number:

```
var tile = myGrid.NearestFaceG (transform.position, GFGridPlane.XY) // something like (1.5, 2.5, -1)
```

Note

This behaviour is different from previous versions. There the extra half unit was subtracted from the result, giving faces and boxes whole numbers. The problem with this is that it made conversion between grid- and world coordinates complicated and it was just a general mess. If you want the old behaviour back read the Legacy Support section.

Shearing

Shearing, or shear mapping, linearly displaces each point of the grid in a fixed direction proportional to its distance from a plane parallel to that direction. In three-dimensional grids we have three axes, each of which can be sheared along the other two axes, resulting in a total of six individual shearing factors. Shearing stacks on top of rotation and is stored in the custom `Vector6` class.

Let's look at an example: If we set the *XY*-shearing to *2* each point will be moved up the *Y*-axis. By how much it will be moved depends on the shearing factor and the *X*-coordinate of the point: *(x,y)* will be mapped to *(x, 2x + y)*. The further away a point is from the origin, the more it will be displaced.

The nomenclature is as follows: the first letter tells us which axis is displaced, the second letter tells us in what direction it is displaced. *XY* means the *X*-axis is sheared by that factor into the positive direction of the *Y*-axis; if the factor is negative, then it is sheared into the negative direction. No shearing is indicated by setting the factor to *0*.

The most likely use of shearing is for "isometric" 2D graphics where rotation cannot be used. Using shearing allows us to keep the grid perpendicular to the camera while still making it appear distorted. As an example, consider the popular 2:1 dimetric look on an *XY*-grid: set the spacing to *(2, 1, 1)* and the shearing to *(-1/2, 0, 2, 0, 0, 0)*. For a 3:2 mapping the spacing would be *(3, 2, 1)* and the shearing would be *(-2/3, 3/2, 0, 0, 0, 0)*.

Shearing always maintains the volume of grid boxes, but the surface area of grid faces is only maintained if the plane they lie in is not skewed. This means if you shear your grid's *X*- and *Y*-axes around each other, but not the *Z*-axis, then the surface area of *XY*-faces will remain the same.

Chapter 6

Hexagonal grids

The GFHexGrid class

Hex grids resemble honeycomb patterns, they are composed of regular hexagons arranged in such a way that each hex shares an edge with another hex. The distance between two neighbouring hexes is always the same, unlike rectangular grids where two rectangles can be diagonal to each other.

The size of a hex is determined by the grid's `radius`, the distance between the centre of a hex and one of its vertices. A honeycomb pattern is two-dimensional and the grid's `gridPlane` determines whether it's an XY-, XZ- or YZ grid. These honeycomb patterns are then stacked on top each other, the distance between two of them being defined by the grid's `depth`, to form a 3D grid.

The angle between two vertices is always 60° or 2/3. The distance between opposite vertices is twice the radius. The distance between adjacent hexes is the same as the distance between adjacent edges, `sqrt(3)*radius`.

Different kinds of hex grids

There are several ways to define a hex grid. For one, you can have hexes with pointy sides (default) or with flat sides. As mentioned above, you have three different grid planes. For the sake of simplicity I'll be using the terms for XY-grids from now on. If you have another grid plane replace X, Y and Z with the respective coordinate, meaning for a top-down XZ grid X would refer to the local "X"- axis, "Y" to the local Z-axis and "Z" to the local Y-axis.

There are multiple ways of drawing a hex grid as well: currently the supported patterns are rectangle, rhombus and herringbone, all in up-and downwards variants plus a compact one for the rectangle.

Coordinate systems in hex grids

In rectangular grids we place the grid along the vertices, but for hex grids it's more intuitive to use the faces instead with the central face of the central hex as the origin.

Currently there are four coordinate systems available (plus world coordinates of course): cubic, rhombic, upwards herringbone and downwards herringbone. I will explain the various coordinate systems and their respective advantages and disadvantages in detail below.

Converting between coordinate systems

At any time you can convert between any of the coordinate systems (including world). The method to call follows the following convention: `OriginalToTarget(point)` where *Original* and *Target* are either `World`, `Cubic`, `Rhombic`, `HerringU` or `HerringD` and *point* is either a Vector3 (world, rhombic or herringbone) or a Vector4 (cubic). There is also the coordinate system `Grid`, it is equivalent to `HerringU`.

Cubic coordinates

In old games, like Qbert, the game world would appear to be in 3D with cubes, but it was actually in 2D and the cubes were actually hexagons; you could use a three dimensional coordinate system in the game and then embed it into the two dimensional plane. The cubic coordinate system follows the same idea.

There are three axes, called *X*, *Y* and *Z*. At any given point their sum is always 0. This is because in a Qbert game every time we move in one of the three directions we also move away from another. Or in other words, the axes span a plane going diagonally through space, resulting in the condition.

On top of that there is a fourth *W* coordinate. The first three coordinates tell us the position on the plane and the fourth coordinate tells us the distance from the central plane. It is given as a multiple of `depth`.

This coordinate system is very nice to use, because its three-dimension nature fits well with the six-sided hexagons, just as the two-dimensional cartesian coordinate systems fits four-sided squares. Many algorithms are very easy to write this way, and that is the the reason why it is used internally in Grid Framework and all other coordinate expressions are conversions. The downside is that it can be rather awkward to understand if you are just looking for a simple coordinate system.

Rhombic coordinates

Rhombic coordinates are a simplification of cubic coordinates. Instead of three axes we use only two, like in a cartesian coordinates system, but the angle between axes is 60° instead of 90°. If the hexes have pointy sides moving up moves you north and moving right moves you north-east. If the hexes have flat sides moving up moves you north-east and moving right moves you east.

The third coordinate, *Z*, is again the distance from the central plane as a multiple of `depth`. This coordinate system is fairly intuitive and a good choice for most tasks.

Herringbone coordinates

The herringbone pattern ows its name to the fact that it resembles the skeleton of a fish. One axis (Y for pointy sides and X for flat sides) is the regular cartesian axis, while the other axis zig-zags up and down (pointy sides) or left and right (flat sides).

Whether the zig-zagging axis is moving up (right for flat sides) or down (left for flat sides) depends on whether the left column (lower row for flat sides) is even or odd. In upwards coordinates the odd columns are shifted upwards (left) relative to the even ones, while in the downwards coordinate system they are shifted downwards (right).

While this coordinate system is *very* intuitive it raises an important issue: where exactly on the grid a pair of coordinates points to depends on whether it is even or odd. Let's say we have our player at (0,0) in a pointy side grid and want to move north-east. Then we add (1,0) to player's position. However, if we want to move north-east again now we have to add (1,1) to the new position, because we are now in an odd column and the X-axis goes downwards. Adding (0,1) would move south-east.

If you want to use herringbone coordinates you will always have to make a distinction between even and odd cases. That's why I wouldn't recommend using this coordinate system for game mechanics such as movement. On the other hand, its rectangular structure makes it a good choice for building level maps.

Nearest vertices, faces and boxes

The methods for finding the nearest *something* adhere to the following convention: *NearestSomethingX (point)* where *point* is a point in world space, *something* is either `Vertex`, `Face` or `Box` and *X* is either `W` (world), `C` (cubic), `R` (rhombic), `HU` (upwards herringbone), `HD` (downwards herringbone) or `G` (grid, equivalent to `HU`). In rectangular grids we need to specify a *plane* for faces, here the plane of the grid is used. You can omit that parameter and even if you specify one it will be ignored.

The plane of hex grids

As mentioned in the sections *The GFGridPlane*, *The GFLayeredGrid class* and *Different kinds of hex grids*, hex grids can be oriented to different planes. When you are making a top-down game your will most likely be working on an XZ-grid and your object will move in X- and Y directions. Consequently your entire game logic will be about the X- and Z coordinate of their position.

Therefore it makes sense that grid coordinates follow this pattern as well. The X-, Y- and Z components of rhombic and herringbone coordinates will be mapped to their respective counterparts, i.e. in an XZ-grid the X coordinate stays X, the Y coordinates becomes Z and the Z coordinate becomes Y. Cubic (and later barycentric) coordinates are not affected by this, they already have more dimensions than the surrounding space, so it makes no sense to transform the embedding mapping.

Aligning and scaling to hex grids

In the case of rectangular grids objects get aligned either on or between faces depending on whether their scale is an even or odd multiple of the spacing. In hex grids this distinction makes no sense, it's like trying to fit a square peg into a round hole, so instead the object's pivot point will be aligned with the center of the nearest hex. I also looked at other hex-based games and how they handled it and they used the same pivot point approach.

Usually and object's pivot point is its centre, you can either change it in your 3D modeling application or make the object a child of an empty object and use that parent as the pivot point.

Chapter 7

Polar grids

The GFPolarGrid class

In this section I will refer to the grids a *polar* for the sake of simplicity, even though they are technically *cylindrical*. Any cylindrical coordinate system can be regarded as polar simply by omitting the third coordinate.

The defining characteristics of polar grids are `radius`, `sectors` and `depth`. There are two coordinate systems, a standard cylindrical one and a grid coordinate system similar to what the other grids use.

The `radius` is the radius of the innermost circle around the origin and the radii of all the other circles are multiples of it. The next value, `sectors`, tells us into how many sectors we divide the circles. It's an integer value of at least one (the circle itself is one huge sector). We derive the angle between sectors by dividing 2 (or 360°) by the amount of sectors. We cannot set the angle directly, because the sum of all angles has to form a full circle, thus only certain values are possible. Finally, `depth` gives us how far apart two grid planes are. If you want to use pure polar coordinates you can ignore this value, but it is relevant if you want to use cylindrical coordinates (i.e. 3D).

Coordinates are processes as Vector3 values. From now on I will only be explaining coordinates in a XY-grid, for other grids the roles of the X-, Y- and Z-component are different to reflect the way cartesian coordinates would be handled. Please refer to the following table to see how to interpret the values.

	XY-grid	**XZ-grid**	**YZ-grid**
X-component	radius	height	radius
Y-component	polar angle	polar angle	height
Z-component	height	radius	polar angle

What are polar and cylindrical coordinates?

Polar grids are based on polar coordinates. In the usual cartesian coordinate system we identify a given point through its distance from the origin by looking at its distance from the coordinate axes (or if you prefer linear algebra, as the linear combination of the coordinate system's basis vectors).

In a `polar coordinate` system we identify a point by its absolute distance (radius or radial coordinate) from the origin (pole) and its angle (angular coordinate, polar angle or azimuth) around a given axis (polar axis).

`Cylindrical coordinates` are polar coordinates with a third axis added for the distance (height or altitude) from the reference plane (the plane containing the origin). We call this third axis the cylindrical or longitudinal axis.

Degree VS radians

There are two common ways to measure angles: as degrees, ranging from 0° to 360° and as `radians` ranging from 0 to 2. Since radians is the preferred method in mathematics and other branches of science Grid Framework is using radians internally for all angles. Some values can be read or written in degrees as well, but the result is always internally stored as radians. Keep in mind that you can convert between degree and radians using Unity's own `` `Mathf` `` class.

Float VS angle

No matter whether you are using radians of degree, an angle is always saved as a float type value, either on its own or as part of a vector. Since float numbers can be negative or exceed the range of radians (2) and degree (360°) there need to be some restrictions in place.

If the number is larger than the range it gets wrapped around, meaning 750° becomes 30° (since 750 = 2 $*$ 360 + 30). If the number is negative it gets subtracted from a full circle (after being wrapped around). In the case of -750° we would get 330° (since 330 = 360 - 30 and 750 = 2 $*$ 360 + 30).

These restrictions are always applied internally when you are using a polar grid's methods and accessors, so you won't have to worry about anything. Just be aware of what's happening in case your circles start going the other way around or something like that.

The polar coordinate system

This is a standard cylindrical coordinate system and internally it is the default coordinate system. In a standard XY-grid the first coordinate is the *radius*, the second coordinate is the *polar angle* and the third coordinate is the *height*, all distances measured in world length. This coordinate system is not affected by `radius`, `sectors` or `depth`, it's great if you want to have full control over your points. For general information about polar and cylindrical grids please refer to a mathematical textbook (or Wikipedia ^^).

The grid coordinate system

This one is similar to the above, except the values now are directly influenced by `radius`, `sectors` and `depth`. The *radius* gives us the distance from the origin as a multiple of the grid's `radius` and the *height* is a multiple of the grid's `depth`. The curious value is the *angle*, because rather than being the angle in radians or degrees it tells us how many sectors we are away from the polar axis. The maximum number is the amount of segments.

This coordinate system is great if you want to think in terms of "grid points", for example in a board game. If you want to move the player one unit towards the outside and two units counter-clockwise you could write

```
var myPlayerPosition: Vector3 = myGrid.WorldToGrid(player.transform.position);
myPlayerPosition += Vector3(1, 2, 0);
player.transform.position = myGrid.GridToWorld(myPlayerPosition);
```

Note how you don't need to know anything about the grid and its values, you just concentrate on the movement itself.

Displaying the grid

Polar grids contain circles, but the computer cannot draw circles, it can only draw straight lines. Since a polar grid where only straight lines connect the sectors would look bad we have a `smoothness` value. What smoothing does is it adds extra "sectors" between the sectors to break the lines int shorter segments, creating a rounder look. Keep in mind that this does affect performance, so you have to find a good balance between look and performance. Usually a single digit number is already good enough and the more sectors you have, the less smoothing you need.

When using the `size` of a grid the values represent how large the grid is, how far the angle goes and how many layers to display. The first and third value cannot get lower than 0 and the second values is bounded between 0 and 2 (360°). If you decide to use a custom range the angle of the from-value can be larger than the to-value without problems. The angle values loop around as described in *Float VS angle*.

Aligning and scaling

When aligning a transform only its Z-scale is taken in account, just as with rectangular grids, but the X- and Y value is ignored, because it makes no sense to force something straight into a circle. Instead they will snap both on and in between circle segments and radial lines, depending on which is closer. The same rule applies to scaling as well.

Chapter 8

Rendering and drawing a Grid

Render VS Draw

Grid Framework lets you both draw and render the grid. While they share the same points and give similar results they work in different ways: drawing uses gizmos, rendering is done directly at a low level. Drawings won't be visible in the game during runtime but they are visible in the editor, while the rendered grid is only visible in game view at runtime. Both serve the same purpose and together they complete each other. Use drawing when you want you to see the grid and use rendering when you want the player to see the grid.

Setup

Make sure your grid has the `renderGrid` flag set to *true* and set the line's width. Attach the `GFGridRender↩ Camera` component to all your cameras which are supposed to render the grid. Usually this would be your default camera. In the inspector you can make the camera render the grid even if it's not the current main camera; this could be useful for example if you want the grid to appear on a mini-map.

Absolute and relative size

By default the size (and `renderFrom`/`renderTo`) is interpreted as absolute lengths in world units. This means a grid with an X-size of 5 will always be five wold units wide to both sides, regardless of its other values like spacing or radius. Setting the `useRelativeSize` flag will interpret the `size` (and `renderFrom`/`renderTo`) as relative lengths measured in grid space.

In other words, a rectangular grid with an X-`spacing` of 1.5 and an X-`size` of 3 will have an absolute width of 6 (3 x 2) world units and a relative width of 9 (3 x 1.5 x 2) world units.

Rendering Range

By default the part of the grid that gets rendered or drawn is defined by its `size` setting. At the centre lies the origin of the grid and from there on it spreads by the specified size in each direction. This means if for example your grid has a `size` of (2, 3,1) it will spread two world units both left and right from the origin. The rendering respects rotation and position of the game object, but not scale. Scale is analogous to spacing.

If you want, you can set your own rendering range. In the inspector set the flag for `useCustomRendering↩ Range`, which will let you specify your own range. The only limitation here is that each component of `renderFrom` has to be smaller or equal to its corresponding component in `renderTo`. You can also set these values in code, please refer to the scripting manual. Despite its name the rendering range applies to both rendering and drawing.

Shared Settings

The rendering shares some options with the drawing. They have the same colours and you can toggle the axes individually. The members `hideGrid` and `hideOnPlay` are only for drawing, you can toggle rendering through the `renderGrid` flag. If you set the the `useSeparateRenderColor` to *true* can can also have separate colours for rendering and drawing.

Rendering with Vectrosity

`Vectrosity` is a separate vector line drawing solution developed by Eric5h5 and not related to Grid Framework. If you own a license you can easily extract points from a grid in a format fit for use with Vectrosity. The included examples show how to combine both packages. For more information on how to use them together please refer to Grid Framework's scripting reference and Vectrosity's own documentation. Just like with rendering you can either use the grid's `size` (default) or a custom range.

Rendering Performance

Rendering, as well as drawing, is a two-step process: first we need to calculate the two end points for each line and then we need to actually render all those lines. Since version 1.2.4 Grid Framework will cache the calculated points, meaning as long as you don't change your grid the points won't be re-calculated again. This means we don't need to waste resources calculating the same values over and over again. However, the second step cannot be cached, we need to pass every single point to Unity's GL class every frame.

For rectangular grids every line stretches from one end of the grid to the other, so it doesn't matter how long the lines are, only how dense they are.

Hex grids on the other hand consist of many line segments, basically every hex adds three lines to the count (lines shared between hexes are drawn only once), outer hexes add even more. This makes a hex grid more expensive than a rectangular grid of the same size.

The performance cost of polar grids depends largely one the amount of `sectors` and the `smoothness`, the more you have, the more expensive it gets. Keep in mind that the more sectors your grid has, the less smoothing you need to achieve a round look.

To improve performance you could adjust the rendering range of your grid dynamically during gameplay to only render the area the player will be able to see. Of course frequently changing the range forces a recalculation of the points and defeats the purpose of caching. Still, the gain in performance can be worth it, just make sure to adjust the range only at certain thresholds. Also keep in mind that if something is set not to render it won't just be invisible, it will not be rendered at all and prevent the loops from running. Turning off an axis or having a flat grid can make a noticeable difference (a 100 x 100 x 0 grid will perform much better than a 100 x 100 x 1 grid).

The *seemingly endless grid* example shows how you can create the illusion of a huge grid without actually having to render the whole thing. We only render the part that will be visible plus some extra buffer. Only when the camera has been moved ten world units from the last fixed position we readjust the rendering range, thus forcing a recalculation of the draw points. This is a compromise between performance and flexibility, we can still display a large grid without the huge overhead of actually having a large grid.

Chapter 9

Getting Started

You can access this documentation from within Unity by going to Help → Grid Framework Documentation

Setting up a new grid

Choose a GameObject in your scene that will carry the grid. Go to Component → Grid Framework and choose the grid component you want to add. Alternatively you can create a grid from scratch by going to GameObject → Create Grid. Once you have your grid in place you can then position it by moving and rotating the GameObject, you can change the settings and you can reference it through scripting.

Creating grids programatically

All grids inherit from the *GFGrid* class, which in turn inherits from Unity's own *MonoBehaviour* class (the class of all scripts). As such grids can be created programatically the same way as any other component in Unity.

The usual approach is to use the `AddComponent` method of the *GameObject* class to add a new grid to the scene

```
GameObject go;                                   // object to carry the grid
GFRectGrid grid = go.AddComponent<GFRectGrid>(); // reference to the new grid
```

Here we use the rectangular grid as an example, but any grid works the same way. Also keep in mind that the abstract grid classes cannot be instantiated, they only serve as abstraction.

Referencing a grid

You can reference a grid the same way you would reference any other component:

```
var myGrid: GFGrid = myGameObject.GetComponent.<GFGrid>();// UnityScript
GFGrid myGrid = myGameObject.GetComponent<GFGrid>();      // C#
```

This syntax is the `generic` version of `GetComponent` which returns a value of type GFGrid instead of Component. See the Unity Script reference for more information. You can access any grid variable or method directly like this:

```
myVector3 = myGrid.WorldToGrid(transform.position);
```

If you need a special variable that is exclusive to a certain grid type (like `spacing` for rectangular grids) you will need to use that specific type rather than the more general `GFGrid`:

```
var myGrid: GFRectGrid = myGameObject.GetComponent.<GFRectGrid>();
```

Drawing the grid

A grid can be drawn using gizmos to give you a visual representation of what is otherwise infinitely large. Note that gizmos draw the grid for you, but they don't render the grid in a finished and published game during runtime. If you wish the grid to be visible during runtime you need to render it.

In the editor you can set flags to hide the grid altogether, hide it only in play mode, draw a sphere at the origin and set colours. If you cannot see the grid even though `hideOnPlay` is disabled make sure that "Gizmos" in the upper right corner of the game view window is enabled.

The Grid Align Panel

You can find this window under Window → Grid Align Panel. To use it, simply drag & drop an object with a grid from the hierarchy into the Grid field. The buttons are pretty self- explanatory. You can also set which layers will be affected, this is especially useful if you want to manipulate large groups of objects at the same time without affecting the rest of the scene. If Ignore Root Objects is set Align Scene and Scale Scene will ignore all objects that have no parent and at least one child. If Include Children is not set then child objects will be ignored. Auto snapping makes all objects moved in editor mode snap automatically to the grid. They will only snap if they have been selected, so you could turn this option on and click on all the objects you want to align quickly and then turn the option off again without affecting the other objects in the scene. You can also set individual axes to not be affected by the aligning functions.

For polar grids there are a few specific options which will only be visible when you drop a polar grid into the grid field. These options include auto- rotating aroudn the grid's origin, similar to auto snapping, and buttons to rotate and align at the same time.

Chapter 10

Events

What is an event?

Delegates and events are a native C# feature that lets you execute code when certain things happen. The *sender* sends out an event and the *listeners* who have subscribed to that event receive a notification and call an appropriate function. For example, if the player receives an invulnerability power-up it could send an event to all enemies, making them run away from the player. The important part is that the sender has no idea who is listening, it's the listeners who have to subscribe.

GridChangedEvent

In Grid Framework there is an event sent every time one of the grid's properties is changed. You could use this in order to know when you have to perform recalculations and when it is save to store values. I'm not going to explain the details of events here, that would exceed the scope of the user manual, but I will give a simple example code. If you need more information I recommend the official `MSDN Event Tutorials`. Since this is a .NET feature it will only work in C#, not UnityScript.

Note

> Since there is no proper way in Unity to be notified when an object's Transform (position, rotation or scale) has been changed the event will *not* be fired when the grid's Transform is being altered.

```
using UnityEngine;
using System.Collections;

public class EventTest : MonoBehaviour {
    public GFGrid grid; // the grid we use for reference

    // subscribe to the event when the script becomes active
    void OnEnable() {
        grid.GridChangedEvent += new GFGrid.
      GridChangedDelegate (SomeMethod);
    }

    // unsubscribe to the event when the script becomes inactive
    void OnDisable() {
        grid.GridChangedEvent -= new GFGrid.
      GridChangedDelegate (SomeMethod);
    }

    // the method to call; it is important that the return type and parameters match the delegate
    void SomeMethod( GFGrid changedGrid ) {
        Debug.Log( "Changes have been made to the grid with name " + changedGrid.name);
    }
}
```

The first thing we need to do is subscribe to the event; a good place for this would be when the script becomes active. In order to subscribe you call the object's event and then add via += the method that will be called as a delegate. A delegate is similar to a function pointer in C or C++, it holds a reference to a function. It is important that the return type and amount and type of parameters matches the one of the delegate, but the actual contents of the method are entirely up to you. We can also unsubscribe to an event, which is done using −= when we want to stop listening.

Chapter 11

Extending Grid Framework

Grid Framework, much like Unity 3D itself, is written as a multi-purpose framework. You can write your own code and reference Grid Framework's API just as you would reference Unity's API, but sometimes you might find yourself wanting not just to use grid Framework, but to expand on it. Maybe there is some method or variable that would fit greatly in your project but that's too specific for me to include by default.

This is where this section of the manual comes into play. I strongly advise against changing the source code, you will either have to re-apply your changes after each update or refrain from updating and thus miss bug fixes and other improvements. Instead, you should use inheritance or extension methods. While these techniques can be accomplished using UnityScript it is not what the language has been created for. I will use C# in this article and I recommend you to do so as well, since C# was designed with these features in mind.

Inheritance

Inheritance is an integral part of object oriented programing, so I won't go into detail, instead I will provide an example. Let's say you you have an array of special points and you want to store them directly inside the (rectangular) grid, as well as a method to output those points. In that case you would inherit from the rectangular grid class and just add the extra code:

```
using UnityEngine;
using System.Collections;

public class ExtendedGrid : GFRectGrid {
    public Vector3[] specialPoints;

    public void PrintPoints () {
        if (specialPoints == null)
            return;
        foreach (Vector3 point in specialPoints)
            Debug.Log (point);
    }
}
```

Obviously this is not a very useful extension, but you get the idea. Now that we have a custom class we can also create a custom inspector for it. If you don't the default inspector provided by Unity will be displayed, so this step is not mandatory. To create a custom inspector inherit from the base class's inspector and add your extra code to it:

```
using UnityEngine;
using UnityEditor;
using System; // needed to the Array class
using System.Collections;

[CustomEditor (typeof(ExtendedGrid))]
public class ExtendedGridEditor : GFRectGridEditor {
    private int listSize;

    public override void OnInspectorGUI () {
        StandardFields ();
        ListField ((ExtendedGrid)grid);

        if (GUI.changed)
```

```
            EditorUtility.SetDirty (target);
    }

    private void ListField (ExtendedGrid xGrid) {
        if (xGrid.specialPoints == null)
            xGrid.specialPoints = new Vector3[0];

        listSize = Mathf.Max (EditorGUILayout.IntField ("Special points amount", listSize), 0);
        if (listSize != xGrid.specialPoints.Length)
            Array.Resize<Vector3> (ref xGrid.specialPoints, listSize);

        for (int i = 0; i < listSize; i++) {
            xGrid.specialPoints [i] = EditorGUILayout.Vector3Field ("Special point " + (i+1), xGrid.
      specialPoints [i]);
        }
    }
}
```

This inspector lets you set and remove entries to the array in the editor. I will not cover Unity's editor GUI scripting as that topic is outside the scope of this article, but I will explain this example.

Since the extended grid inherits from rectangular grids our new inspector will inherit from the rectangular grid's inspector as well. The private variable keeps track of how long we want the array of points to be. Then we override the `OnInspectorGUI` method to display all our fields. The method `StandardFields` simply displays all the fields the base inspector also has, this is handy if you want to use my inspector layout and then just add your own elements to it. If you want more control you should look into the actual code of the inspectors, every group of GUI elements is encapsulated into its own method, so you can mix them as you please. After the standard fields we place our new extra code, again encapsulated into its own method to keep things clean. Finally we finish it up as usual with `EditorUtility.SetDirty`.

The `ListField` method takes a parameter of type ExtendedGrid. Each grid inspector holds a reference to its grid object via the `grid` variable, but its type is the abstract `GFGrid`, so we need to typecast it to its more specific runtime type if we want to access the array. The rest of the code is just regular inspector GUI scripting.

Extension Methods

Extension methods are a way to add methods to existing classes without having or needing access to the implementation of the class. Once a method has been added this way it will be treated the same way as other methods and it will even show up in autocomplete if your editor supports it. I made use of extension methods in the "lights out" example, you can see a practical application there, here I will go into the theory.

Start by creating a new C# script, the name doesn't matter, so use something that makes sense you, I will use "GridPrintExtension". Delete the default contents of the file created by Unity and add the following code to it:

```
using UnityEngine;
using System.Collections;

public static class GridPrintExtension {

}
```

That class is where our extension method will be written. Let's write it...

```
using UnityEngine;
using System.Collections;

public static class GridPrintExtension {
    public static void PrintGrid (this GFGrid grid) {
        Debug.Log (grid.size.x);
    }
}
```

Let's go over the code; `public` should be self-explanatory, the method has to be public so the grid class can see it and it also needs to be static because the containing class is static as well. Next up are the return type, the name of the method and the parameter list. The extension method does not belong to any instance, but the extended class has to treat is as if it did belong to an instance of it. That is what the first parameter is for, it always starts with `this`, then the the name of the class we extend and finally an identifier which can be any name.

That's essentially it, we can now use this method in our scripting. However, let's take it a step further, what if we want one common method name, but different implementations in every specific grid? In the lights out example each grid type applies different rules as to what an adjacent flied is. We will use a simple if-else construct:

```
using UnityEngine;
using System.Collections;

public static class GridPrintExtension {
    public static void PrintGrid (this GFGrid grid) {
        Type t = grid.GetType();
        if (t == typeof(GFRectGrid)) {
            PrintRect ((GFRectGrid) grid);
        } else if (t == typeof(GFHexGrid)) {
            PrintHex ((GFHexGrid) grid);
        } else if (t == typeof(GFPolarGrid)) {
            PrintPolar ((GFPolarGrid) grid);
        } else {
            Debug.Log (grid.size.x);
        }
    }

    private static void PrintRect (GFRectGrid grid) {
        Debug.Log (grid.spacing.x);
    }
    private static void PrintHex (GFHexGrid grid) {
        Debug.Log (grid.radius);
    }
    private static void PrintPolar (GFPolarGrid grid) {
        Debug.Log (grid.sectors);
    }
}
```

The default case is not needed, it's just for demonstration. As you can see we print the contents of a member variable that is specific to a certain grid type without needing to know what type of grid will call the extension method. The specific methods are set to private to make sure they can't be called outside of the extension method, but that's just a matter of personal preference and choice.

Chapter 12

Debugging a grid

The Debug Sphere

Under Grid Framework/Debug you will find a prefab called Debug Sphere. This prefab instantiates a small sphere with the script GridDebugger attached. Just drop a grid onto the sphere, choose the function you want to debug, choose a plane if necessary and move the sphere around. You will see gizmo drawings and get values printed to the console for the function you have chosen.

Debug Scripts

UnderComponent $\rightarrow$ Grid Framework $\rightarrow$ Debug you'll find the script use for the sphere prefab as well as a script for debugging various conversions in polar grids.

Chapter 13

Legacy Support

Grid Framework is constantly being worked on and most changes happen under the hood, so you don't notice them. However, there are times when I find myself regretting a choice or decision and where changing or removing the offending part is the only reasonable thing to do in the long run. This part of the documentation is here to list what changes were made and how to adapt your code. Note that this does not include things that were clearly broken and had to be fixed, so if there is a change not listed here it was a bug, not a feature.

NearestVertex/Face/Box

In previous versions the results of `NearestFaceG` and `NearestBoxG` of rectangular grids, as well as `NearestVertexW` of hexagonal grids, had returned shifted values to give them whole numbers as coordinates. This was meant to make things simpler, but in reality it was just more confusing and made no real sense. In version 1.3.2 this was changed to normal, as one would expect. You can either change your code and subtract the extra shift, or you unzip the file GFNearestLegacy.cs.zip found under under Plugins/Grid Framework/Legacy Support and append the letter L (for *legacy*) to the methods in *your* code (e.g. turn `NearestFaceG` into `NearestFaceGL`).

Vertex matrix

Various methods for building and reading an array of Vector3 for storing vertices have been dropped in version 1.3.2. The methods were bloated and too specific, in most cases it was better to just write a custom solution for storing coordinates. They have been moved into extension methods, so if you want them back you must unzip the file GFVertexMatrixLegacy.cs.zip, which can be found under Plugins/Grid Framework/Legacy Support.

FindNearestSomething and GetSomethingCoordinates

The methods `FindNearestVertex/Face/Box` and `FindNearestVertex/Face/Box` have been renamed to `NearestVertex/Face/BoxW` and `NearestVertex/Face/BoxG` respectively in version 1.2.5. This was done in preparation for polar grids, which have two coordinate systems rather than one (hex grids at that point only had odd herringbone coordinates). The old methods are still present in name, but they will throw deprecation warnings when used. Run a simple search&replace to fix it.

Chapter 14

Playmaker support

Grid Framework supports actions for `Playmaker`, the visual scripting tool for Unity 3D. Playmaker is a separate add on and not affiliated with Grid Framework, so you will need a separate license for it.

Grid Framework comes with a set of actions covering almost its entire API; you can set and get the grid's attributes and call the grid's method's.

Activating Grid Framework actions

In order to prevent compilation errors for users without Playmaker the scripts have been prevented from compiling using the preprocessor. To make the scripts compile you will have to toggle them on.

In the menu bar under *Component* -> *Grid Framework* -> *Toggle Playmaker actions* you can turn the scripts on and off. If successful, a message will be printed to the console informing you whether support was enabled or disabled. If something goes wrong an error will be printed instead.

This mechanism edits the source code of the scripts directly to uncomment or comment a specific line of code, so it is strongly advised that you do not move the scripts and don't change the preprocessor directives, or else this menu item might not work anymore.

Using Grid Framework actions

You use Grid Framework actions basically the same way you use any other Playmaker actions. The most important thing to note is that each Grid Framework action requires a grid; usually it will try to default to the owner of the FSM, but you can set it to something else if you wish to, simply by drag&dropping the grid onto the field. A grid is strictly required and the actions will not work without one.

Actions fall into two basic categories: methods and getters/setters. Methods have the same name as the grid method they invoke and getters are prefixed with *Get* while setters are prefixed with *Set*. Each action has the expected tooltips, but for complete information on the underlying calls you will have to refer to the API documentation. Overloaded methods from the API are all bundled in one action that lets you specify default values.

The amount of actions is very large, larger than any of the categories Playmaker ships by default, so it is recommended that you use the search field of Playmakers's action browser. Find the method or getter/setter you wish to invoke in the API documentation and then search for that name and it should show up.

Writing your own Playmaker actions

If you are familiar with Playmaker you can write your own actions. For the sake of consistency you will want to use the same inheritance pattern as was used for the official Grid Framework actions. While it is not strictly required that you follow the same pattern to produce working actions it will allow your actions to inherit any future improvements from the base classes. Any new Playmaker action always inherits from Playmaker's `FsmStateAction` class, from there on the inheritance tree is as follows:

```
FsmStateAction // required base class by Playmaker
|- FsmGFStateAction<T> : FsmStateAction where T : GFGrid
   |- FsmGFStateActionGetSet<T> : FsmGFStateAction<T> where T : GFGrid
   |  |- FsmGFStateActionGetSetGrid : FsmGFStateActionGetSet<GFGrid>
   |  |- FsmGFStateActionGetSetRect : FsmGFStateActionGetSet<GFRectGrid>
   |  |- FsmGFStateActionGetSetLayered<T> : FsmGFStateActionMethod<T> where T :
      GFLayeredGrid
   |     |- FsmGFStateActionGetSetHex : FsmGFStateActionGetSetLayered <GFHexGrid>
   |     |- FsmGFStateActionGetSetPolar : FsmGFStateActionGetSetLayered <GFPolarGrid>
   |
   |- FsmGFStateActionMethod<T> : FsmGFStateAction<T> where T : GFGrid
      |- FsmGFStateActionMethodGrid : FsmGFStateActionMethod<GFGrid>
      |- FsmGFStateActionMethodRect : FsmGFStateActionMethod<GFRectGrid>
      |- FsmGFStateActionMethodLayered<T> : FsmGFStateActionMethod<T> where T :
      GFLayeredGrid
         |- FsmGFStateActionMethodHex : FsmGFStateActionMethodLayered<GFHexGrid>
         |- FsmGFStateActionMethodPolar : FsmGFStateActionMethodLayered<GFPolarGrid>
```

The class `FsmGFStateAction` is the base class for all Grid Framework actions, it contains all the common basic member variables and methods. From there on the rest of the inheritance tree is only used to keep things organised.

Most important are the `grid` and `gridGameObject` member variables. The latter holds the reference to the GameObject that holds the grid and and usually defaults to the owner of the FSM. The former is used to cache the reference to the grid and its type is the type of the generic `T` parameter.

Finally, the `abstract void DoAction()` method provides a place where to implement your actual action. Since it is abstract it does not have any implementation, you instead override it in you own sub-class and it is then called when the action fires.

Missing actions

Due to limitations in Playmaker's API some setters, getters and method calls couldn't yet be implemented as actions. These are any actions that would take in or return enums (like angle modes or grid planes) or Vector4 variables. Variables of type `GFBoolVector3` and `GFColorVector3` are only supported in getters and setters by using three separate variables. While this shouldn't make any difference in most cases it is worth pointing out.

Of course, once Playmaker becomes able to support these features the actions will be added to the package.

Chapter 15

Changelog

Version 1.7.x

Version 1.7.2

- *Fixed:* Null exception on polar grids when getting Vectrosity points if the grid is not being rendered.

Version 1.7.1

- *Fixed:* The grid align panel now correctly respect or ignores rotation when auto-snapping.

Version 1.7.0

This release features a number of new coordinate systems and corresponding rendering shapes.

- *New:* Downwards herringbone coordinate system for hex grids
- *New:* Downwards rectangle rendering shape to accompany the new coordinate system.
- *New:* Downwards rhombic coordinate system.
- *New:* Downwards rhombic rendering shape to accompany the new coordinate system.
- *New:* Up- and downwards herringbone rendering shape.
- *Fixed:* The grid align panel now correctly respect or ignores rotation when aligning.

Version 1.6.x

Version 1.6.0

- *New:* Hex-grids can now render in a rhombic shape.

Version 1.5.x

Version 1.5.3

- Compatibility with Unity 5.

Version 1.5.2

- *Fixed:* Changing the `depth` of polar grids affected the cylindric lines wrongly.

Version 1.5.1

- *Fixed:* Compilation errors when toggling on Playmaker actions.

Version 1.5.0

Introducing shearing for rectangular grids.

- *New:* Rectangular grids can now store a `shearing` field to distort them.
- *New:* Custom `Vector6` class for storing the shearing.
- *API change:* The odd herringbone coordinate system has been renamed to upwards herringbone. The corresponding methods use the `HerringU` pre- or suffix instead of `HerringOdd`; the old methods still work but are marked as depracated.
- *API change:* The enumeration `GFAngleMode` has been renamed `AngleMode` and moved into the `GridFramework` namespace.
- *API change:* The enumeration `GridPlane` has been moved into the `GridFramework` namespace. It is no longer part of the `GFGrid` class.
- *API change:* The class `GFColorVector3` has been renamed `ColorVector3` and moved into the `GridFramework.Vectors` namespace.
- *API change:* The class `GFBoolVector3` has been renamed `BoolVector3` and moved into the `GridFramework.Vectors` namespace.
- *Enhanced:* Vectrosity methods without parameters can now pick betweem size and custom range automatically.
- *Fixed:* Vectrosity methods were broken in previous version.
- Updated the documentation.

Version 1.4.x

Version 1.4.2

This release is a major overhaul of the rendering and drawing routines and fixes some issues with coordinate conversion.

- *Fixed:* Wrong rotation when using a rotated grid and an origin offset.
- *Fixed:* Wrong result when convertig coordinates in a hex grid rotated along the X- or Y axis.
- *Fixed:* Setting the `relativeSize` flag for polar grids now interprets the range properly in grid coordinates.
- *Fixed:* Wrong accessibility for `NearestVertexHO` and `NearestBoxHO` for hex grids.
- *New:* Polar grids can now render continuously at any range instead of discretely at smoothness steps.

Version 1.4.1

- *Fixed:* compilation error in one of the Playmaker actions (setter and getter for depth of layered grids).

Version 1.4.0

- Introducing Playmaker support: Almost the entire Grid Framework API can no be used as Playmaker actions (some parts of the API are ouside the capabilies of Playmaker for now)
- Updated the documentation to include a chapter about Playmaker and how to write your own Grid Framework actions.
- *Fixed:* the origin offset resetting every time after exiting play mode.

Version 1.3.x

Version 1.3.8

- *Fixed:* wrong calculation result in `CubicToWorld` and all related methods in hex grids.

Version 1.3.7

- *Fixed:* compilation error, sometimes the program might refuse to compile if a script used one of the functions NearestVertexW, NearestBoxW or NearestBoxW.
- Auto-complete support: Grid Framework's API documentation will now show up in MonoDevelop's auto-complete feature. There is no need to jump between editor and documentation anymore, it's all integrated.

Version 1.3.6

- Changing the origin offset of a grid now takes effect instantly.

Version 1.3.5

- Added a new event for when the grid changes in such a way that if would need to be redrawn.
- Some of the exmples were broken when Unity updated to version 4.3, now they should be working again.
- Overhauled the *undo* system for the grid align panel to remove the now obsolete Unity undo methods.

Version 1.3.4

- Added the ability to add a position offset to the grids. This moves the origin of a grid by the offset relative to the object's Transform position. In the API this is represented by the `originOffset` member of the `GFGrid` class.
- Added a chapter about extending Grid Framework without changing the source code to the manual. Everything described there are just standard .NET features, the chapter is intended for people who were not aware of the potential or unfamiliar with it.

Version 1.3.3

- Values of GFColorVector3 and GFBoolVector3 were not persistent in version 1.3.2, fixed this now.
- Examples *Movement with Walls* and *Sliding Puzzle* were broken after version 1.3.2, fixed them now.
- The documentation can now be read online as well. Just delete the offline documentation from *WebPlayer↩ Templates* and the help menu will notice that it's missing and open the web URL.

Version 1.3.2

- Hex Grids: new coordinate systems, see the manual page about Hexagonal grids for more information.
- New HTML documentation generated with Doxygen replaces the old one.
- Fixed a bug in `Angle2Rotation` when the grid's rotation was not a multiple of 90°.
- *New example:* generate a terrain mesh similar to old games like SimCity from a plain text file and have it align to a grid.
- *New example:* a rotary phone dial that rotates depending on which number was clicked and reports that number back. A great template for disc-shaped GUIs.

Some existing methods have changed in this release, please consult the Legacy Support page of the user manual.

- Rect Grids: changed the way `NearestBoxG` works, now there is no offset anymore, it returns the actual grid coordinates of the box. Just add `0.5 * Vector.one` to the result in your old methods.
- Rect Grids: changed the way `NearestFaceG` works, just like above. Add `0.5 * Vector3.one - 0.5 * i` to the result in your old methods (where `i` is the index of the plane you used).
- Hex grids: Just like above, nearest vertices of hex grids return their true coordinates for whatever coordinate system you choose.

I am sorry for these changes so late , but I realize this differentiation made things more complicated in the end than they should have been. It's better to have one unified coordinate system instead. Read the Legacy Support to learn how to get the old behaviour back.

Version 1.3.1

- Fixed an edge case for `AlignVector3` in rectangular grids.
- in the runtime snapping example you can now click-drag on the grids directly and see `AlignVector3` in action (turn on gizmos in game view to see).
- added the *PointDebug* script to the above example for that purpose.
- Changed the way movement is done in the grid-based movement example, now the sphere will always take the straight path.

Version 1.3.0

Introducing polar grids to Grid Framework: comes with all the usual methods and two coordinate systems.

- Added `up`, `right` and `forward` members to rectangular grids.
- Added `sides`, `width` and `height` members to hex grids.
- Added the enum `GFAngleMode {radians, degree}` to specify an angle type; currently only used in methods of polar grids.
- Added the enum `HexDirection` for cardinal directions (north, north-east, east, ...) in hex grids.
- Added the `GetDirection` method to hex grids to convert a cardinal direction to a world space direction vector.
- Added a lot of minor conversion methods for rotation, angles, sectors and so on in hex grids
- Hex grids and polar grids now both inherit from `GFLayeredGrid`, which in return inherits from `GFGrid`.
- The Lights Off example now features a polar grid as well.
- Procedural mesh generation for grid faces in the Lights Off example.
- Mouse handling in runtime snapping example changed because it was confusing a lot of users who just copy-pasted the code.

Version 1.2.x

Version 1.2.5

This release serves as a preparation for Version 1.3.0, which will add polar grids

- the methods 'NearestVertex/Face/BoxW' and 'NearestVertex/Face/BoxG' replace 'FindNearestVertex/Face/↩ Box' and 'FindNearestVertex/Face/Box' respectively. This is just a change in name, as the old nomenclature was confusing and makes no sense for grids with multiple coordinate systems, but the syntax stays the same. The old methods will throw compiler warnings but will still work fine. You can run a Search&Replace through your scripts to get rid of them.
- The 'GFBoolVector3' class can now be instantiated via 'GFBoolVector3.True' and 'GFBoolVector3.False' to create an all-_true_ or all-_false_ vector.
- Similarly you can use `GFColorVector3.RGB`, `GFColorVector3.CMY` and `GFColorVector3.↩ BGW` for half-transparent standard colour vectors
- Various code cleanup.

Version 1.2.4

- Performance improvement by caching draw points. As long as the grid hasn't been changed the method CalculateDrawPoints will reuse the existing points instead of calculating them again.
- Added explanation about rendering performance to the user manual. It explains what exactly happens, what lowers performance and what techniques can improve performance.
- *New exmple:* A seemingly endless grid scrolls forever. This is achieved by adjusting the rendering range dynamically and we add a little buffer to make use of the new caching feature.

Version 1.2.3

- Added the ability to use a separate set of colours for rendering and drawing.
- Added the ability to have the size of drawings/renderings be relative to the spacing of the grid instead of absolute in world coordinates.
- Some examples were broken after the last update after adding accessors to the code, fixed now.

Version 1.2.2

- Fixed a typo that could prevent a finished project from building correctly.
- *New example:* a sliding block puzzle working entirely without physics.
- Removed the variables `minimumSpacing` and `minimumRadius` from `GFRectGrid` and `GFHexGrid`. Instead they both use accessors that limit spacing and radius to 0.1.
- The members `size`, `renderTo` and `renderFrom` are now using accessors as well, this prevents setting them to nonsensical values.
- Removed the redundant *Use Custom Rendering Range* flag in the inspector (doesn't change anything in the API though, it's just cosmetic)
- The foldout state for *Draw & Render Settings* in the inspector should stick now (individual for both grid types).
- Several minor tweaks under the hood.

Version 1.2.1

- Updated the Lights Off example to use hex grids.

Version 1.2.0

Introducing hexagonal grids: use hexagons instead of rectangles for your grids. Comes with all the methods you've come to know from rectangular grids and uses a herringbone pattern as the coordinate system.

- The movement example scripts now take a 'GFGrid' instead of a 'GFRectGrid', allowing the user to use both rectangular and hexagonal grids without changing the code.

Version 1.1.x

Version 1.1.10

- *New method:* 'ScaleVector3' lets you scale a `Vector3` to the grid instead of a `Transform`.

Version 1.1.9

- *New method:* `AlignVector3` lets you align a single point represented as a `Vector3` instead of a `Transform` to a grid.
- Added the ability to lock a certain axes when calling `AlignTransform` and `AlignVector3`.
- Added a new constructor to both `GFBoolVector3` and `GFColorVector3` that lets you pass one parameter that gets applied to all components.
- You can now lock axes in the Grid Align Panel as well.
- Aligning objects via the Grid Align Panel which already are in place won't do anything, meaning they won't create redundant Undo entries anymore.
- Fixed an issue in `GetVectrosityPointsSeperate`.
- Renamed the classes `BoolVector3` and `ColorVector3` to `GFBoolVector3` and `GFColor↩ Vector3` to avoid name collision.
- The member `size` has always been a member of `GFGrid`, not `GFRectGrid`, I fixed that mistake in the documentation.
- Minor code cleanup and removing redundant code.

Version 1.1.8

- Previously if you unloaded a level with a grid that was rendered the game could have crashed. Fixed that issue.

Version 1.1.7

- Fixed a typo that prevented adding the `GFGridRenderCamera` component from the menu bar.
- *New example:* design your levels in a plain text file and use Grid Framework and a text parser to build them during runtime. No need to change scenes when switching levels, faster than placing blocks by hand and great for user-made mods.
- *New example:* a continuation of the grid-based movement example where you can place obstacles on the grid and the sphere will avoid them. Works without using any physics or colliders.

Version 1.1.6

Important: The classes `Grid` and `RectGrid` have been renamed to `GFGrid` and `GFRectGrid`. This was done to prevent name collision with classes users might already be using or classes from other extensions. I apologize for the inconvenience.

- Minor code cleanup and performance increase in `GFRectGrid`.

Version 1.1.5

- Custom rendering range affects now drawing as well.

Version 1.1.4

- Fixed an issue where lines with width would be rendered on top of objects even though they should be underneath.

Version 1.1.3

- Support for custom range for rendering instead of the grid's `size`.
- From now on all files should install in the right place on their own, no more moving scripts manually.

Version 1.1.2

- Integration into the menu bar.
- Vectrosity support.
- Documentation split into a separate user manual and a scripting reference.

Version 1.1.1

- Line width for rendering now possible.

Version 1.1.0

- Introducing grid rendering.
- New inspector panel for `RectGrid`.

Version 1.0.x

Version 1.0.1

- Fixed rotation for cube shaped debug gizmos.

Version 1.0.0

- Initial release.

Chapter 16

Namespace Index

16.1 Packages

Here are the packages with brief descriptions (if available):

GridFramework . . . . ??
GridFramework.Vectors . . . . ??

Chapter 17

Hierarchical Index

17.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

GridFramework.Vectors.BoolVector3 . . . . **??**
GridFramework.Vectors.ColorVector3 . . . . **??**
GFGrid . . . . **??**
- GFLayeredGrid . . . . **??**
  - GFHexGrid . . . . **??**
  - GFPolarGrid . . . . **??**
- GFRectGrid . . . . **??**

GridFramework.Vectors.Vector6 . . . . **??**

Chapter 18

Class Index

18.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

GridFramework.Vectors.BoolVector3
A class that holds three booleans as X-, Y- and Z-value. . . . . **??**
GridFramework.Vectors.ColorVector3
A class that holds three colours as X-, Y- and Z-value. . . . . **??**
GFGrid
Abstract base class for all Grid Framework grids. . . . . **??**
GFHexGrid
A grid consting of flat hexagonal grids stacked on top of each other . . . . **??**
GFLayeredGrid
The parent class for all layered grids. . . . . **??**
GFPolarGrid
A polar grid based on cylindrical coordinates. . . . . **??**
GFRectGrid
A standard three-dimensional rectangular grid. . . . . **??**
GridFramework.Vectors.Vector6
Class representing six adjacent float values for the shearing of a rectangular grid. . . . . . . **??**

Chapter 19

Namespace Documentation

19.1 Package GridFramework

Namespaces

- package Vectors

Enumerations

- enum AngleMode

  Radians or degrees.
- enum GridPlane

  Enum for one of the three grid planes.

19.1.1 Enumeration Type Documentation

19.1.1.1 enum GridFramework.AngleMode

This is a simple enum for specifying whether an angle is given in radians for degrees. This enum is so far only used in methods of GFPolarGrid, but I decided to make it global in case other grids in the future will use it was well.

19.1.1.2 enum GridFramework.GridPlane

This enum encapsulates the three grid planes: XY, XZ and YZ. You can also get the integer of enum items, where the integer corresponds to the missing axis (X = 0, Y = 1, Z = 2): `// UnityScript: var myPlane: GridPlane = GridPlane.XZ; var planeIndex: int = (int)myPlane; // sets the variable to 1`

`// C# GridPlane myPlane = GridPlane.XZ; int planeIndex = (int)myPlane;`

19.2 Package GridFramework.Vectors

Classes

- class BoolVector3

  A class that holds three booleans as X-, Y- and Z-value.
- class ColorVector3

A class that holds three colours as X-, Y- and Z-value.

- class Vector6

  Class representing six adjacent float values for the shearing of a rectangular grid.

Enumerations

- enum Axis2 {
  Axis2.xy, Axis2.xz, Axis2.yx, Axis2.yz,
  Axis2.zx, Axis2.zy }

  Enumeration of the possible axis combinations.
- enum Axis { Axis.x, Axis.y, Axis.z }

  Enumeration of the three possible axes.

19.2.1 Enumeration Type Documentation

19.2.1.1 enum **GridFramework.Vectors.Axis**

Enumerator

x	integer value 0
y	integer value 1
z	integer value 2

19.2.1.2 enum **GridFramework.Vectors.Axis2**

The enumeraion's corresponding integer numers start at 0 and increment by one.

Enumerator

xy	integer value 0
xz	integer value 1
yx	integer value 2
yz	integer value 3
zx	integer value 4
zy	integer value 5

Chapter 20

Class Documentation

20.1 GridFramework.Vectors.BoolVector3 Class Reference

A class that holds three booleans as X-, Y- and Z-value.

Public Member Functions

- BoolVector3 (bool x, bool y, bool z)

  Creates a new bool vector with given X, Y and Z components.
- BoolVector3 ()

  Creates an all-`false` BoolVector3.
- BoolVector3 (bool condition)

  Creates a new BoolVector3 set to a condition.

Properties

- bool x `[get, set]`

  X component of the bool vector.
- bool y `[get, set]`

  Y component of the bool vector.
- bool z `[get, set]`

  Z component of the bool vector.
- bool this[int index] `[get, set]`

  Access the X, Y or Z components using [0], [1], [2] respectively.
- static BoolVector3 False `[get]`

  Creates a new all-`false` BoolVector3.
- static BoolVector3 True `[get]`

  Creates a new all-`true` BoolVector3.

20.1.1 Detailed Description

This class groups three booleans together, similar to how Vector3 groups three float numbers together. Just like Vector3 you can read and assign values using x, y, or an indexer.

20.1.2 Constructor & Destructor Documentation

20.1.2.1 GridFramework.Vectors.BoolVector3.BoolVector3 (bool *x*, bool *y*, bool *z*)

Parameters

x	X value.
y	Y value.
z	Z value.

20.1.2.2 GridFramework.Vectors.BoolVector3.BoolVector3 ( )

Creates an all-`false` BoolVector3.

20.1.2.3 GridFramework.Vectors.BoolVector3.BoolVector3 (bool *condition*)

Parameters

condition	The value to be used for all components.

reates a new BoolVector3 set to `condition`.

20.1.3 Property Documentation

20.1.3.1 BoolVector3 GridFramework.Vectors.BoolVector3.False [static], [get]

This is the same as calling `BoolVector3(false)`.

20.1.3.2 bool GridFramework.Vectors.BoolVector3.this[int index] [get], [set]

Parameters

index	The index.

Access the x, y, z components using [0], [1], [2] respectively. Example: `BoolVector3 b = new Bool←`
`Vector3(); b[1] = true; // the same as b.y = true`

20.1.3.3 BoolVector3 GridFramework.Vectors.BoolVector3.True [static], [get]

This is the same as calling `BoolVector3(true)`.

20.1.3.4 bool GridFramework.Vectors.BoolVector3.x [get], [set]

20.1.3.5 bool GridFramework.Vectors.BoolVector3.y [get], [set]

20.1.3.6 bool GridFramework.Vectors.BoolVector3.z [get], [set]

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/Vectors/GFBoolVector3.cs

20.2 GridFramework.Vectors.ColorVector3 Class Reference

A class that holds three colours as X-, Y- and Z-value.

Public Member Functions

- ColorVector3 (Color x, Color y, Color z)

  Creates a new colour vector with given X, Y and Z components.
- ColorVector3 (Color color)

  Creates a one-colour ColorVector3.

Properties

- Color x `[get, set]`

  X component of the colour vector.
- Color y `[get, set]`

  Y component of the colour vector.
- Color z `[get, set]`

  Z component of the colour vector.
- Color this[int index] `[get, set]`

  Access the X, Y or Z components using [0], [1], [2] respectively.
- static ColorVector3 RGB `[get]`

  Shorthand writing for `ColorVector3()`
- static ColorVector3 CMY `[get]`

  Shorthand writing for `ColorVector3(Color(0,1,1,0.5), Color(1,0,1,0.5), Color(1,1,0,0.↩ 5))`
- static ColorVector3 BGW `[get]`

  Shorthand writing for `ColorVector3(Color(0,0,0,0.5), Color(0.5,0.5,0.5,0.5), Color(1,1,1,0.↩ 5))`

20.2.1 Detailed Description

This class groups three colours together, similar to how Vector3 groups three float numbers together. Just like Vector3 you can read and assign values using x, y, or an indexer.

20.2.2 Constructor & Destructor Documentation

20.2.2.1 GridFramework.Vectors.ColorVector3.ColorVector3 (Color *x*, Color *y*, Color *z*)

Parameters

x	X-value of the new vector.
y	Y-value of the new vector.
z	Z-value of the new vector.

summary>Creates a standard RGB ColorVector3.

Creates a new standard RGB ColorVector3 where all three colours have their alpha set to 0.5.

20.2.2.2 GridFramework.Vectors.ColorVector3.ColorVector3 (Color *color*)

Parameters

color	The colur for all ccomponents.

Creates a new ColorVector3 where all components are set to the same colour.

20.2.3 Property Documentation

20.2.3.1 ColorVector3 GridFramework.Vectors.ColorVector3.BGW [static], [get]

20.2.3.2 ColorVector3 GridFramework.Vectors.ColorVector3.CMY [static], [get]

20.2.3.3 ColorVector3 GridFramework.Vectors.ColorVector3.RGB [static], [get]

20.2.3.4 Color GridFramework.Vectors.ColorVector3.this[int index] [get], [set]

Parameters

index	The index.

Access the x, y, z components using [0], [1], [2] respectively. Example: `ColorVector3 c = new Color↩ Vector3(); c[1] = true; // the same as c.y = true`

20.2.3.5 Color GridFramework.Vectors.ColorVector3.x [get], [set]

20.2.3.6 Color GridFramework.Vectors.ColorVector3.y [get], [set]

20.2.3.7 Color GridFramework.Vectors.ColorVector3.z [get], [set]

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/Vectors/GFColorVector3.cs

20.3 GFGrid Class Reference

Abstract base class for all Grid Framework grids.

Inheritance diagram for GFGrid:

Public Member Functions

- delegate void GridChangedDelegate (GFGrid grid)

  A delegate for handling events when the grid has been changed in such a way that it requires a redraw
- abstract Vector3 WorldToGrid (Vector3 worldPoint)

  Converts world coordinates to grid coordinates.
- abstract Vector3 GridToWorld (Vector3 gridPoint)

  Converts grid coordinates to world coordinates.
- abstract Vector3 NearestVertexW (Vector3 worldPoint, bool doDebug)

  Returns the world position of the nearest vertex.
- Vector3 NearestVertexW (Vector3 worldPoint)
- abstract Vector3 NearestFaceW (Vector3 worldPoint, GridPlane plane, bool doDebug)

  Returns the world position of the nearest face.
- Vector3 NearestFaceW (Vector3 worldPoint, GridPlane plane)
- abstract Vector3 NearestBoxW (Vector3 worldPoint, bool doDebug)

  Returns the world position of the nearest box.
- Vector3 NearestBoxW (Vector3 worldPoint)
- abstract Vector3 NearestVertexG (Vector3 worldPoint)

Returns the grid position of the nearest vertex.

- abstract Vector3 NearestFaceG (Vector3 worldPoint, GridPlane plane)

  Returns the grid position of the nearest Face.
- abstract Vector3 NearestBoxG (Vector3 worldPoint)

  Returns the grid position of the nearest box.
- abstract Vector3 AlignVector3 (Vector3 pos, Vector3 scale, BoolVector3 ignoreAxis)

  Fits a position vector into the grid.
- Vector3 AlignVector3 (Vector3 pos)
- Vector3 AlignVector3 (Vector3 pos, BoolVector3 lockAxis)
- Vector3 AlignVector3 (Vector3 pos, Vector3 scale)
- void AlignTransform (Transform theTransform, bool rotate, BoolVector3 ignoreAxis)

  Fits a Transform inside the grid (without scaling it).
- void AlignTransform (Transform theTransform)
- void AlignTransform (Transform theTransform, BoolVector3 lockAxis)
- void AlignTransform (Transform theTransform, bool rotate)
- abstract Vector3 ScaleVector3 (Vector3 scl, BoolVector3 ignoreAxis)

  Scales a size vector to fit inside a grid.
- Vector3 ScaleVector3 (Vector3 scl)
- void ScaleTransform (Transform theTransform, BoolVector3 ignoreAxis)

  Scales a Transform to fit the grid (without moving it).
- void ScaleTransform (Transform theTransform)
- void RenderGrid (Vector3 from, Vector3 to, ColorVector3 colors, int width=0, Camera cam=null, Transform camTransform=null)

  Renders the grid at runtime
- virtual void RenderGrid (int width=0, Camera cam=null, Transform camTransform=null)
- void RenderGrid (Vector3 from, Vector3 to, int width=0, Camera cam=null, Transform camTransform=null)
- void DrawGrid (Vector3 from, Vector3 to)

  Draws the grid using gizmos.
- void DrawGrid ()
- Vector3[] GetVectrosityPoints (Vector3 from, Vector3 to)

  Returns an array of Vector3 points ready for use with Vectrosity.
- Vector3[] GetVectrosityPoints ()
- Vector3[][] GetVectrosityPointsSeparate (Vector3 from, Vector3 to)

  Returns an array of arrays of Vector3 points ready for use with Vectrosity.
- Vector3[][] GetVectrosityPointsSeparate ()

Public Attributes

- ColorVector3 axisColors = new ColorVector3()

  Colours of the axes when drawing and rendering.
- bool useSeparateRenderColor = false

  Whether to use the same colours for rendering as for drawing.
- ColorVector3 renderAxisColors = new ColorVector3(Color.gray)

  Separate colours of the axes when rendering.
- bool hideGrid = false

  Whether to hide the grid completely.
- bool hideOnPlay = false

  Whether to hide the grid in play mode.
- BoolVector3 hideAxis = new BoolVector3()

  Whether to hide just individual axes.
- bool drawOrigin = false

Whether to draw a little sphere at the origin of the grid.

- bool renderGrid = true

  Whether to render the grid at runtime.

- Material renderMaterial = null

  The material for rendering, if none is given it uses a default material.

Properties

- bool relativeSize `[get, set]`

  Whether the drawing/rendering will scale with spacing.

- virtual Vector3 size `[get, set]`

  The size of the visual representation of the grid.

- virtual Vector3 renderFrom `[get, set]`

  Custom lower limit for drawing and rendering.

- virtual Vector3 renderTo `[get, set]`

  Custom upper limit for drawing and rendering.

- Vector3 originOffset `[get, set]`

  Offset to add to the origin

- bool useCustomRenderRange `[get, set]`

  Use your own values for the range of the rendering.

- int renderLineWidth `[get, set]`

  The width of the lines used when rendering the grid.

Events

- GridChangedDelegate GridChangedEvent

  An even that gets fired

20.3.1 Detailed Description

This is the standard class all grids are based on. Aside from providing a common set of variables and a template for what methods to use, this class has no practical meaning for end users. Use this as reference for what can be done without having to specify which type of grid you are using. For anything more specific you have to look at the child classes.

20.3.2 Member Function Documentation

20.3.2.1 void GFGrid.AlignTransform (Transform *theTransform,* bool *rotate,* BoolVector3 *ignoreAxis*)

Parameters

theTransform	The Transform to align.
rotate	Whether to rotate to the grid.
ignoreAxis	Which axes should be ignored.

Fits an object inside the grid by using the object's Transform. Setting `doRotate` makes the object take on the grid's rotation. The parameter `lockAxis` makes the function not touch the corresponding coordinate.

The resulting position depends on *AlignVector3* , so please look up how that method works.

20.3.2.2 void GFGrid.AlignTransform (Transform *theTransform*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It aligns and rotates the Transform while respecting all axes, it is equal to `AlignTransform(theTransform, true, new BoolVector3(false));`

20.3.2.3 void GFGrid.AlignTransform (Transform *theTransform*, BoolVector3 *lockAxis*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It aligns and rotates the Transform but leaves the axes to the user, it is equal to `AlignTransform(theTransform, true, lockAxis);`

20.3.2.4 void GFGrid.AlignTransform (Transform *theTransform*, bool *rotate*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It aligns and respects all axes to the user, but leaves the decision of rotation to the user, it is equal to `AlignTransform(theTransform, rotate, new BoolVector3(false));`

20.3.2.5 abstract Vector3 GFGrid.AlignVector3 (Vector3 *pos*, Vector3 *scale*, BoolVector3 *ignoreAxis*) `[pure virtual]`

Returns

Aligned position vector.

Parameters

pos	The position to align.
scale	A simulated scale to decide how exactly to fit the poistion into the grid.
ignoreAxis	Which axes should be ignored.

Fits a position inside the grid by using the object's transform. The exact position depends on whether the components of *scale* are even or odd and the exact implementation can be found in the subclasses. The parameter *ignoreAxis* makes the function not touch the corresponding coordinate.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.6 Vector3 GFGrid.AlignVector3 (Vector3 *pos*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It aligns the position while respecting all axes and uses a default size of 1 x 1 x 1, it is equal to `AlignVector3(pos, Vector3.one, new BoolVector3(false));`

20.3.2.7 Vector3 GFGrid.AlignVector3 (Vector3 *pos*, BoolVector3 *lockAxis*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It aligns the position and uses a default size of 1 x 1 x 1 while leaving the axes to the user, it is equal to `AlignVector3(pos, Vector3.one, lockAxis);`

20.3.2.8 Vector3 GFGrid.AlignVector3 (Vector3 *pos*, Vector3 *scale*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It aligns the position and respects the axes while using a default size of 1 x 1 x 1, it is equal to `AlignVector3(pos, scale, new BoolVector3(false));`

20.3.2.9 void GFGrid.DrawGrid (Vector3 *from*, Vector3 *to*)

Parameters

from	Lower limit of the drawing.
to	Upper limit s drawing.

This method draws the grid in the editor using gizmos. There is usually no reason to call this method manually, you should instead set the drawing flags of the grid itself. However, if you must, call this method from inside `OnDrawGizmos`.

20.3.2.10 void GFGrid.DrawGrid ( )

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Uses the size as limits, it is equivalent to `DrawGrid(-size, size);`

20.3.2.11 Vector3 [ ] GFGrid.GetVectrosityPoints (Vector3 *from*, Vector3 *to*)

Returns

Array of points in local space.

Parameters

from	Lower limit for the points.
to	Upper limit for the points.

Returns an array of Vector3 containing the points for a discrete vector line in Vectrosity. One entry is the starting point, the next entry is the end point, the next entry is the starting point of the next line and so on.

20.3.2.12 Vector3 [ ] GFGrid.GetVectrosityPoints ( )

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Uses the grid's `size` or `renderFrom` and `renderTo` as limits, equal to `useCustomRenderRange ? GetVectrosityPoints(renderFrom, renderTo) : GetVectrosityPoints(-size, size);`

20.3.2.13 Vector3 [ ][ ] GFGrid.GetVectrosityPointsSeparate (Vector3 *from*, Vector3 *to*)

Returns

Jagged array of three arrays, each containing the points of a single axis.

Parameters

from	Lower limit for the points.
to	Upper limit for the points.

This method is very similar to `GetVectrosityPoints`, except that the points are in separate arrays for each axis. This is useful if you want to treat the lines of each axis differently, like having different colours.

20.3.2.14 Vector3 [ ][ ] GFGrid.GetVectrosityPointsSeparate ( )

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Uses the grid's `size` or `renderFrom` and `renderTo` as limits, equal to `useCustomRenderRange ? GetVectrosityPointsSeparate(renderFrom, renderTo) : GetVectrosityPointsSeparate(-size, size);`

20.3.2.15 delegate void GFGrid.GridChangedDelegate (GFGrid *grid*)

Parameters

grid	The grid that calls the delegate

This is the delegate type for methods to be called when changes to the grid occur. It is best used together with the GridChangedEvent event.

20.3.2.16 abstract Vector3 GFGrid.GridToWorld (Vector3 *gridPoint*) `[pure virtual]`

Returns

World coordinates of the grid point.

Parameters

gridPoint	Point in grid space.

Takes in a point in grid space and converts it to world space. Some grids have several coordinate system, so look into the specific class for conversion methods from other coordinate systems.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.17 abstract Vector3 GFGrid.NearestBoxG (Vector3 *worldPoint*) `[pure virtual]`

Returns

Grid position of the nearest box.

Parameters

worldPoint	Point in world space.

Similar to NearestVertexG, it returns the grid coordinates of a box in the grid. Since the box is enclosed by several vertices, the returned value is the point in between all of the vertices.

This is just an abstract template for the method, look into the specific class for exact implementation.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.18 abstract Vector3 GFGrid.NearestBoxW (Vector3 *worldPoint,* bool *doDebug*) `[pure virtual]`

Returns

World position of the nearest box.

Parameters

worldPoint	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Similar to NearestVertexW, it returns the world coordinates of a box in the grid. Since the box is enclosed by several vertices, the returned value is the point in between all of the vertices. If *doDebug* is set a small gizmo box will drawn there.

This is just an abstract template for the method, look into the specific class for exact implementation.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.19 Vector3 GFGrid.NearestBoxW (Vector3 *worldPoint*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

20.3.2.20 abstract Vector3 GFGrid.NearestFaceG (Vector3 *worldPoint,* GridPlane *plane*) `[pure virtual]`

Returns

Grid position of the nearest face.

Parameters

worldPoint	Point in world space.
plane	Plane on which the face lies.

Similar to NearestVertexG, it returns the grid coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices. You also need to specify on which plane the face lies (optional for hex- and polar grids).

This is just an abstract template for the method, look into the specific class for exact implementation.

Implemented in GFRectGrid, and GFLayeredGrid.

20.3.2.21 abstract Vector3 GFGrid.NearestFaceW (Vector3 *worldPoint,* GridPlane *plane,* bool *doDebug*) `[pure virtual]`

Returns

World position of the nearest face.

Parameters

worldPoint	Point in world space.
plane	Plane on which the face lies.
doDebug	If set to `true` draw a sphere at the destination.

Similar to NearestVertexW, it returns the world coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices. You also need to specify on which plane the face lies (optional for hex- and polar grids). If *doDebug* is set a small gizmo face will drawn there.

This is just an abstract template for the method, look into the specific class for exact implementation.

Implemented in GFRectGrid, and GFLayeredGrid.

20.3.2.22 Vector3 GFGrid.NearestFaceW (Vector3 *worldPoint,* GridPlane *plane*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

20.3.2.23 abstract Vector3 GFGrid.NearestVertexG (Vector3 *worldPoint*) `[pure virtual]`

Returns

Grid position of the nearest vertex.

Parameters

worldPoint	Point in world space.

Returns the position of the nerest vertex in grid coordinates from a given point in world space.

This is just an abstract template for the method, look into the specific class for exact implementation.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.24 abstract Vector3 GFGrid.NearestVertexW (Vector3 *worldPoint,* bool *doDebug*) [pure virtual]

Returns

World position of the nearest vertex.

Parameters

worldPoint	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Returns the world position of the nearest vertex from a given point in world space. If *doDebug* is set a small gizmo sphere will be drawn at that position. This is just an abstract template for the method, look into the specific class for exact implementation.

This is just an abstract template for the method, look into the specific class for exact implementation.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.25 Vector3 GFGrid.NearestVertexW (Vector3 *worldPoint*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

20.3.2.26 void GFGrid.RenderGrid (Vector3 *from,* Vector3 *to,* ColorVector3 *colors,* int *width* = `0`, Camera *cam* = `null`, Transform *camTransform* = `null`)

Parameters

from	Lower limit
to	Upper limit
colors	Colors for rendering
width	Width of the line
cam	Camera for rendering
camTransform	Transform of the camera

Renders the grid with lower and upper limit, a given line width and individual colours for the three axes. If the lines have line width 1 they will be exactly one pixel wide, and if they have a larger with they will be rendered as billboards (always facing the camera). If there is no camera and camera Transform passed this won't be possible and the lines will default back to one pixel width.

It is not necessary to call this method manually, rather you should just set the `renderGrid` flag to `true` and let Grid Framework take care of it. However, if you want total control use this method, usually from within an `OnPostRender` method.

20.3.2.27 virtual void GFGrid.RenderGrid (int *width* = `0`, Camera *cam* = `null`, Transform *camTransform* = `null`) [virtual]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Renders the grid using `size` for lower and upper limits, equal to `RenderGrid(-size, size, useSeparateRenderColor ? renderAxisColors : axisColors, width, cam, camTransform);`

Reimplemented in GFPolarGrid.

20.3.2.28 void GFGrid.RenderGrid (Vector3 *from*, Vector3 *to*, int *width* = `0`, Camera *cam* = `null`, Transform *camTransform* = `null`)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Renders the grid using `axisColors` (or `renderAxisColors` if `useSeparate↵ RenderColor` is `true`) as colours, equal to `RenderGrid(from, to, useSeparateRenderColor ? renderAxisColors : axisColors, width, cam, camTransform);`

20.3.2.29 void GFGrid.ScaleTransform (Transform *theTransform*, BoolVector3 *ignoreAxis*)

Parameters

theTransform	The Transform to scale.
ignoreAxis	The axes to ignore.

Scales a Transform to fit inside a grid. The parameter *ignoreAxis* makes the function not touch the corresponding coordinate.

The resulting position depends on ScaleVector3, so please look up how that method works.

20.3.2.30 void GFGrid.ScaleTransform (Transform *theTransform*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It scales the Transform while respecting all axes, it is equal to `ScaleTransform(the↵ Transform, new BoolVector3(false));`

20.3.2.31 abstract Vector3 GFGrid.ScaleVector3 (Vector3 *scl*, BoolVector3 *ignoreAxis*) `[pure virtual]`

Returns

The re-scaled vector.

Parameters

scl	The vector to scale.
ignoreAxis	The axes to ignore.

This method takes in a vector representing a size and fits it inside the grid. The *ignoreAxis* parameter lets you ignore individual axes.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.2.32 Vector3 GFGrid.ScaleVector3 (Vector3 *scl*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. It scales the size while respecting all axes, it is equal to `ScaleVector3(scl, new BoolVector3(false));`

20.3.2.33 abstract Vector3 GFGrid.WorldToGrid (Vector3 *worldPoint*) `[pure virtual]`

Returns

Grid coordinates of the world point.

Parameters

worldPoint	Point in world space.

Takes in a point in world space and converts it to grid space. Some grids have several coordinate system, so look into the specific class for conversion methods to other coordinate systems.

Implemented in GFHexGrid, GFPolarGrid, and GFRectGrid.

20.3.3 Member Data Documentation

20.3.3.1 ColorVector3 GFGrid.axisColors = new ColorVector3()

The colours are stored as three separte entries, corresponding to the three separate axes. They will be used for both drawing an rendering, unless useSeparateRenderColor is set to `true`.

20.3.3.2 bool GFGrid.drawOrigin = false

If set to `true` a small gizmo sphere will be drawn at the origin of the grid. This is not a rendering, so it wil not appear in the game, it is intended to make selecting the grid in the editor easier.

20.3.3.3 BoolVector3 GFGrid.hideAxis = new BoolVector3()

This hides the individual axes rather than the whole grid.

20.3.3.4 bool GFGrid.hideGrid = false

If set to `true` the grid will be neither drawn nor rendered at all, it then takes precedence over all the other flags.

20.3.3.5 bool GFGrid.hideOnPlay = false

This is similar to hideGrid, but only active while in play mode.

20.3.3.6 ColorVector3 GFGrid.renderAxisColors = new ColorVector3(Color.gray)

By default the colours of axisColors are used for rendering, however if you set useSeparateRenderColor to `true` these colours will be used instead. Otherwise this does nothing.

20.3.3.7 bool GFGrid.renderGrid = true

The grid will only be rendered if this flag is set to `true`, otherwise you won't be able to see the grid in the game.

20.3.3.8 Material GFGrid.renderMaterial = null

You can use you own material if you want control over the shader used, otherwise this default material will be used:

```
new Material("Shader \"Lines/Colored Blended" {" + "SubShader { Pass { " +
" Blend SrcAlpha OneMinusSrcAlpha " + " ZWrite Off Cull Off Fog { Mode Off
} " + " BindChannels {" + " Bind "vertex", vertex Bind "color", color }" +
"} } }" )
```

20.3.3.9 bool GFGrid.useSeparateRenderColor = false

If you set this flag to `true` the rendering will use the colours of renderAxisColors, otherwise it will default to axis←
Colors. This is useful if you want to have different colours for rendering and drawing. For example, you could have a clearly visible grid in the editor to work with and a barely visible grid in the game while playing.

20.3.4 Property Documentation

20.3.4.1 Vector3 GFGrid.originOffset `[get], [set]`

By default the origin of grids is at the world position of their gameObject (the position of the Transform), this offset allows you to move the grid's pivot point by adding a value to it. Keep in mind how this will affect the various grid coordinate systems, they are still relative to the grid's origin, not the Transform.

In other words, if a point at grid position (1, 2, 0) is at world position (4, 5, 0) and you add an offset of (1, 1, 0), then point's grid position will still be (1, 2, 0), but its world position will be (5, 6, 0). Here is an example:

```
GFGrid myGrid;
Vector3 gPos = new Vector3 (1, 2, 3);
Vector3 wPos = myGrid.GridToWorld (gPos);
Debug.Log (wPos); // prints (4, 5, 0)

myGrid.pivotOffset = new Vector3 (1, 1, 0);
wPos = myGrid.GridToWorld (gPos);
Debug.Log (wPos); // prints (5, 6, 0)
```

20.3.4.2 bool GFGrid.relativeSize `[get], [set]`

`true` if the grid's size is in grid coordinates; `false` if it's in world coordinates.

Set this to `true` if you want the drawing to have relative size, i.e. to scale with the spacing/radius or whatever the specific grid uses. Otherwise set it to `false`.

See also: size, renderFrom, renderTo

20.3.4.3 virtual Vector3 GFGrid.renderFrom `[get], [set]`

Custom lower limit for drawing and rendering.

When using a custom rendering range this is the lower left backward limit of the rendering and drawing.

See also: relativeSize, ,

20.3.4.4 int GFGrid.renderLineWidth `[get], [set]`

The width of the render line.

The width of the rendered lines, if it is set to 1 all lines will be one pixel wide, otherwise they will have the specified width in world units.

20.3.4.5 virtual Vector3 GFGrid.renderTo `[get], [set]`

Custom upper limit for drawing and rendering.

When using a custom rendering range this is the upper right forward limit of the rendering and drawing.

See also: relativeSize, ,

20.3.4.6 virtual Vector3 GFGrid.size `[get], [set]`

The size of the grid's visual representation.

Defines the size of the drawing and rendering of the grid. Keep in mind that the grid is infinitely large, the drawing is just a visual representation, stretching on all three directions from the origin. The size is either absolute or relative to the grid's other parameters, depending on the value of relativeSize.

If you set useCustomRenderRange to `true` that range will override this member. The size is either absolute or relative to the grid's other parameters, depending on the value of relativeSize.

See also: relativeSize, useCustomRenderRange

20.3.4.7 bool GFGrid.useCustomRenderRange `[get], [set]`

`true` if using a custom range for rendering and drawing; otherwise, `false`.

If this flag is set to `true` the grid rendering and drawing will use the values of renderFrom and renderTo as limits. Otherwise it will use the size instead.

20.3.5 Event Documentation

20.3.5.1 GridChangedDelegate GFGrid.GridChangedEvent

This is the event that gets fired when one of the grid's properties is changed. If the Transform (position or rotation) is changed this event will only be fired if there is a camera trying to render the grid or some other method tries to draw the gird (like drawing in the editor or calling GetVectrosityPoints). You can learn more about events ont he Events page of the user manual.

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/Abstracts/GFGrid.cs

20.4 GFHexGrid Class Reference

A grid consting of flat hexagonal grids stacked on top of each other

Inheritance diagram for GFHexGrid:

Public Types

- enum HexOrientation { HexOrientation.PointySides, HexOrientation.FlatSides }

  orientation of hexes, pointy or flat sides.

- enum HexTopOrientation { HexTopOrientation.FlatTops, HexTopOrientation.PointyTops }

  orientation of hexes, flat or pointy tops.

- enum HexGridShape {
  HexGridShape.Rectangle, HexGridShape.CompactRectangle, HexGridShape.RectangleDown, HexGrid↩
  Shape.Rhombus,
  HexGridShape.RhombusDown, HexGridShape.HerringboneUp, HexGridShape.HerringboneDown }

- enum HexDirection {
  HexDirection.N, HexDirection.NE, HexDirection.E, HexDirection.SE,
  HexDirection.S, HexDirection.SW, HexDirection.W, HexDirection.NW }

  Cardinal direction of a vertex.

Public Member Functions

- override Vector3 WorldToGrid (Vector3 worldPoint)

  Converts world coordinates to grid coordinates.
- override Vector3 GridToWorld (Vector3 gridPoint)

  Converts grid coordinates to world coordinates
- Vector3 WorldToHerringU (Vector3 world)

  Returns the upwards herringbone coordinates of a point in world space.
- Vector3 WorldToHerringD (Vector3 world)

  Returns the downwards herringbone coordinates of a point in world space.
- Vector3 WorldToRhombic (Vector3 world)

  Returns the rhombic coordinates of a point in world space.
- Vector3 WorldToRhombicD (Vector3 world)

  Returns the downwards rhombic coordinates of a point in world space.
- Vector4 WorldToCubic (Vector3 world)

  Returns the cubic coordinates of a point in world space.
- Vector3 HerringUToWorld (Vector3 herring)

  Returns the world coordinates of a point in upwards herringbone coordinates.
- Vector3 HerringUToHerringD (Vector3 herring)

  Returns the downwards herringbone coordinates of a point in upwards- coordinates.
- Vector3 HerringUToRhombic (Vector3 herring)

  Returns the rhombic coordinates of a point in upwards herringbone coordinates.
- Vector3 HerringUToRhombicD (Vector3 herring)

  Returns the downwards rhombic coordinates of a point in upwards herringbone coordinates.
- Vector4 HerringUToCubic (Vector3 herring)

  Returns the cubic coordinates of a point in upwards herringbone coordinates.
- Vector3 HerringDToWorld (Vector3 herring)

  Returns the world coordinates of a point in downwards herringbone coordinates.
- Vector3 HerringDToHerringU (Vector3 herring)

  Returns the upwards herringbone coordinates of a point in downwards- coordinates.
- Vector3 HerringDToRhombic (Vector3 herring)

  Returns the rhombic coordinates of a point in downwards herringbone coordinates.
- Vector3 HerringDToRhombicD (Vector3 herring)

  Returns the downwards rhombic coordinates of a point in downwards herringbone coordinates.
- Vector4 HerringDToCubic (Vector3 herring)

  Returns the cubic coordinates of a point in downwards herringbone coordinates.
- Vector3 RhombicToWorld (Vector3 rhombic)

  Returns the world coordinates of a point in rhombic coordinates.
- Vector3 RhombicToHerringU (Vector3 rhombic)

  Returns the upwards herringbone coordinates of a point in rhombic coordinates.
- Vector3 RhombicToRhombicD (Vector3 rhombic)

  Returns the downwards rhombic coordinates of a point in rhombic coordinates.
- Vector3 RhombicToHerringD (Vector3 rhombic)

  Returns the downwards herringbone coordinates of a point in rhombic coordinates.
- Vector4 RhombicToCubic (Vector3 rhombic)

  Returns the cubic coordinates of a point in rhombic coordinates.
- Vector3 RhombicDToWorld (Vector3 rhombic)

  Returns the world coordinates of a point in downwards rhombic coordinates.
- Vector3 RhombicDToHerringU (Vector3 rhombic)

  Returns the upwards herringbone coordinates of a point in downwards rhombic coordinates.
- Vector3 RhombicDToHerringD (Vector3 rhombic)

Returns the downwards herringbone coordinates of a point in downwards rhombic coordinates.

- Vector3 RhombicDToRhombic (Vector3 rhombic)

  Returns the downwards herringbone coordinates of a point in downwards rhombic coordinates.
- Vector4 RhombicDToCubic (Vector3 rhombic)

  Returns the cubic coordinates of a point in downwards rhombic coordinates.
- Vector3 CubicToWorld (Vector4 cubic)

  Returns the world coordinates of a point in cubic coordinates.
- Vector3 CubicToHerringU (Vector4 cubic)

  Returns the upwards herring coordinates of a point in cubic coordinates.
- Vector3 CubicToRhombic (Vector4 cubic)

  Returns the rhombic coordinates of a point in cubic coordinates.
- Vector3 CubicToRhombicD (Vector4 cubic)

  Returns the downwards rhombic coordinates of a point in cubic coordinates.
- override Vector3 NearestVertexW (Vector3 world, bool doDebug=false)

  Returns the world coordinates of the nearest vertex.
- override Vector3 NearestFaceW (Vector3 world, bool doDebug)

  Returns the world coordinates of the nearest face.
- override Vector3 NearestBoxW (Vector3 fromPoint, bool doDebug)

  Returns the world coordinates of the nearest box.
- override Vector3 NearestVertexG (Vector3 world)

  Returns the grid position of the nearest vertex.
- override Vector3 NearestFaceG (Vector3 world)

  Returns the grid position of the nearest face.
- override Vector3 NearestBoxG (Vector3 world)

  Returns the grid position of the nearest box.
- Vector3 NearestVertexHO (Vector3 world)

  Returns the upwards herring position of the nearest vertex.
- Vector3 NearestFaceHO (Vector3 world)

  Returns the upwards herring position of the nearest face.
- Vector3 NearestBoxHO (Vector3 world)

  Returns the grid position of the nearest box.
- Vector3 NearestVertexR (Vector3 world)

  Returns the rhombic position of the nearest vertex.
- Vector3 NearestFaceR (Vector3 world)

  Returns the rhombic position of the nearest face.
- Vector3 NearestBoxR (Vector3 world)

  Returns the rhombic position of the nearest box.
- Vector4 NearestVertexC (Vector3 world)

  Returns the cubic position of the nearest vertex.
- Vector4 NearestFaceC (Vector3 world)

  Returns the cubic position of the nearest face.
- Vector4 NearestBoxC (Vector3 world)

  Returns the cubic position of the nearest box.
- override Vector3 AlignVector3 (Vector3 pos, Vector3 scale, BoolVector3 lockAxis)

  Fits a position vector into the grid.
- override Vector3 ScaleVector3 (Vector3 scl, BoolVector3 lockAxis)

  Scales a size vector to fit inside a grid.

Properties

- float radius `[get, set]`

  Distance from the centre of a hex to a vertex.
- HexOrientation hexSideMode `[get, set]`

  Pointy sides or flat sides.
- HexTopOrientation hexTopMode `[get, set]`

  Flat tops or pointy tops.
- HexGridShape gridStyle `[get, set]`

  The shape of the overall grid, affects only drawing and rendering, not the calculations.
- float side `[get]`

  1.5 times the radius.
- float height `[get]`

  Full width of the hex.
- float width `[get]`

  Distance between vertices on opposite sides.

Additional Inherited Members

20.4.1 Detailed Description

A regular hexagonal grid that forms a honeycomb pattern. It is characterized by the `radius` (distance from the centre of a hexagon to one of its vertices) and the `depth` (distance between two honeycomb layers). Hex grids use a herringbone pattern for their coordinate system, please refer to the user manual for information about how that coordinate system works.

20.4.2 Member Enumeration Documentation

20.4.2.1 enum GFHexGrid.HexDirection

The cardinal position of a vertex relative to the centre of a given hex. Note that using N and S for pointy sides, as well as E and W for flat sides does not make sense, but it is still possible.

Enumerator

N North.

NE South.

E East.

SE South-East.

S South.

SW South-West.

W West.

NW North-West.

20.4.2.2 enum GFHexGrid.HexGridShape

summary>Shape of the drawing and rendering.

Different shapes of hexagonal grids: `Rectangle` looks like a rectangle with every odd-numbered column offset, `CompactRectangle` is similar with the odd-numbered colums one hex shorter.

Enumerator

Rectangle Rectangular upwards herringbone pattern.

CompactRectangle Rectangular upwards herringbone pattern with the top of every odd column clipped.

RectangleDown Rectangular downwards herringbone pattern.

Rhombus Rhobus-like upwards pattern.

RhombusDown Rhobus-like downwards pattern.

HerringboneUp Upwards herringbone pattern.

HerringboneDown Downwards herringbone pattern.

20.4.2.3 enum GFHexGrid.HexOrientation

There are two ways a hexagon can be rotated: `PointySides` has flat tops, parallel to the grid's X-axis, and `FlatSides` has pointy tops, parallel to the grid's Y-axis.

Enumerator

PointySides Pointy east and west, flat north and south, equal to `HexTopOrientation.FlatTops`.

FlatSides Flat east and west, pointy north and south, equal to `HexTopOrientation.PointyTops`.

20.4.2.4 enum GFHexGrid.HexTopOrientation

There are two ways a hexagon can be rotated: `FlatTops` has flat tops, parallel to the grid's X-axis, and `Pointy↩ Tops` has pointy tops, parallel to the grid's Y-axis. This enum is best used to complement `HexOrientation`.

Enumerator

FlatTops Flat north and south, pointy east and west, equal to `HexOrientation.PointySides`.

PointyTops Pointy north and south, flat east and west, equal to `HexOrientation.FlatSides`.

20.4.3 Member Function Documentation

20.4.3.1 override Vector3 GFHexGrid.AlignVector3 (Vector3 *pos*, Vector3 *scale*, BoolVector3 *lockAxis*) `[virtual]`

Parameters

pos	The position to align.
scale	A simulated scale to decide how exactly to fit the position into the grid.
lockAxis	Which axes should be ignored.

Returns

The vector3.

Aligns a poistion vector to the grid by positioning it on the centre of the nearest face. Please refer to the user manual for more information. The parameter lockAxis makes the function not touch the corresponding coordinate.

Implements GFGrid.

20.4.3.2 Vector3 GFHexGrid.CubicToHerringU (Vector4 *cubic*)

Parameters

cubic	Point in cubic coordinates.

Returns

Point in upwards herring coordinates.

Takes a point in cubic coordinates and returns its upwards herring position.

20.4.3.3 Vector3 GFHexGrid.CubicToRhombic (Vector4 *cubic*)

Parameters

cubic	Point in cubic coordinates.

Returns

Point in rhombic coordinates.

Takes a point in cubic coordinates and returns its rhombic position.

20.4.3.4 Vector3 GFHexGrid.CubicToRhombicD (Vector4 *cubic*)

Parameters

cubic	Point in cubic coordinates.

Returns

Point in downwards rhombic coordinates.

Takes a point in cubic coordinates and returns its rhombic position.

20.4.3.5 Vector3 GFHexGrid.CubicToWorld (Vector4 *cubic*)

Parameters

cubic	Point in cubic coordinates.

Returns

Point in world coordinates.

Takes a point in cubic coordinates and returns its world position.

20.4.3.6 override Vector3 GFHexGrid.GridToWorld (Vector3 *gridPoint*) `[virtual]`

Parameters

gridPoint	Point in grid space (upwards herringbone coordinate system).

Returns

World coordinates of the grid point.

This is the same as calling `HerringUToWorld`, because upwards herringbone is the default grid coordinate system.

Implements GFGrid.

20.4.3.7 Vector4 GFHexGrid.HerringDToCubic (Vector3 *herring*)

Parameters

herring	Point in downwards herringbone coordinates.

Returns

Point in cubic coordinates.

Takes a point in downwards herringbone coordinates and returns its cubic position.

20.4.3.8 Vector3 GFHexGrid.HerringDToHerringU (Vector3 *herring*)

Parameters

herring	Point in downwards herringbone coordinates.

Returns

Point in upwards herringbone coordinates.

Takes a point in downwards herringbone coordinates and returns its upwards herringbone position.

20.4.3.9 Vector3 GFHexGrid.HerringDToRhombic (Vector3 *herring*)

Parameters

herring	Point in downwards herringbone coordinates.

Returns

Point in rhombic coordinates.

Takes a point in downwards herringbone coordinates and returns its rhombic position.

20.4.3.10 Vector3 GFHexGrid.HerringDToRhombicD (Vector3 *herring*)

Parameters

herring	Point in downwards herringbone coordinates.

Returns

Point in downwards rhombic coordinates.

Takes a point in downwards herringbone coordinates and returns its downwards rhombic position.

20.4.3.11 Vector3 GFHexGrid.HerringDToWorld (Vector3 *herring*)

Parameters

herring	Point in downwards herringbone coordinates.

Returns

Point in world coordinates.

Takes a point in downwards herringbone coordinates and returns its world position.

20.4.3.12 Vector4 GFHexGrid.HerringUToCubic (Vector3 *herring*)

Parameters

herring	Point in upwards herringbone coordinates.

Returns

Point in cubic coordinates.

Takes a point in upwards herringbone coordinates and returns its cubic position.

20.4.3.13 Vector3 GFHexGrid.HerringUToHerringD (Vector3 *herring*)

Parameters

herring	Point in upwards herringbone coordinates.

Returns

Point in downwards herringbone coordinates.

Takes a point in upwards herringbone coordinates and returns its downwards herringbone position.

20.4.3.14 Vector3 GFHexGrid.HerringUToRhombic (Vector3 *herring*)

Parameters

herring	Point in upwards herringbone coordinates.

Returns

Point in rhombic coordinates.

Takes a point in upwards herringbone coordinates and returns its rhombic position.

20.4.3.15 Vector3 GFHexGrid.HerringUToRhombicD (Vector3 *herring*)

Parameters

herring	Point in upwards herringbone coordinates.

Returns

Point in downwards rhombic coordinates.

Takes a point in upwards herringbone coordinates and returns its downwards rhombic position.

20.4.3.16 Vector3 GFHexGrid.HerringUToWorld (Vector3 *herring*)

Parameters

herring	Point in upwards herringbone coordinates.

Returns

Point in world coordinates.

Takes a point in upwards herringbone coordinates and returns its world position.

20.4.3.17 Vector4 GFHexGrid.NearestBoxC (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Cubic position of the nearest box.

Returns the cubic position of the nearest box from a given point in upwards herring coordinates.

20.4.3.18 override Vector3 GFHexGrid.NearestBoxG (Vector3 *world*) `[virtual]`

Parameters

world	Point in world space.

Returns

Grid position of the nearest box.

This is just a shortcut for `NearestBoxHO`.

Implements GFGrid.

20.4.3.19 Vector3 GFHexGrid.NearestBoxHO (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Grid position of the nearest box.

Returns the world position of the nearest box from a given point in upwards herring coordinates.

20.4.3.20 Vector3 GFHexGrid.NearestBoxR (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Rhombic position of the nearest box.

Returns the rhombic position of the nearest box from a given point in upwards herring coordinates.

20.4.3.21 override Vector3 GFHexGrid.NearestBoxW (Vector3 *fromPoint,* bool *doDebug*) [virtual]

Parameters

fromPoint	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Returns

World position of the nearest box.

Returns the world position of the nearest box from a given point in world space. Since the box is enclosed by several vertices, the returned value is the point in between all of the vertices. If `doDebug` is set a gizmo sphere will be drawn at that position.

Implements GFGrid.

20.4.3.22 Vector4 GFHexGrid.NearestFaceC (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Cubic position of the nearest face.

This method takes in a point in world space and returns the cubic coordinates of the nearest face.

20.4.3.23 override Vector3 GFHexGrid.NearestFaceG (Vector3 *world*) [virtual]

Parameters

world	Point in world space.

Returns

Grid position of the nearest face.

This is just a shortcut for `NearestFaceHO`.

Implements GFLayeredGrid.

20.4.3.24 Vector3 GFHexGrid.NearestFaceHO (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Grid position of the nearest face.

This method takes in a point in world space and returns the upwards herring coordinates of the nearest face.

20.4.3.25 Vector3 GFHexGrid.NearestFaceR (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Rhombic position of the nearest face.

This method takes in a point in world space and returns the rhombic coordinates of the nearest face.

20.4.3.26 override Vector3 GFHexGrid.NearestFaceW (Vector3 *world,* bool *doDebug*) `[virtual]`

Parameters

world	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Returns

World position of the nearest vertex.

Returns the world position of the nearest vertex from a given point in world space. If `doDebug` is set a small gizmo sphere will be drawn at that position.

Implements GFLayeredGrid.

20.4.3.27 Vector4 GFHexGrid.NearestVertexC (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Cubic position of the nearest vertex.

This method takes in a point in world space and returns the cubic coordinates of the nearest vertex.

20.4.3.28 override Vector3 GFHexGrid.NearestVertexG (Vector3 *world*) `[virtual]`

Parameters

world	Point in world space.

Returns

Grid position of the nearest vertex.

This is just a shortcut for `NearestVertexHO`.

Implements GFGrid.

20.4.3.29 Vector3 GFHexGrid.NearestVertexHO (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Grid position of the nearest vertex.

This method takes in a point in world space and returns the upwards herring coordinates of the nearest vertex.

20.4.3.30 Vector3 GFHexGrid.NearestVertexR (Vector3 *world*)

Parameters

world	Point in world space.

Returns

Rhombic position of the nearest vertex.

This method takes in a point in world space and returns the rhombic coordinates of the nearest vertex.

20.4.3.31 override Vector3 GFHexGrid.NearestVertexW (Vector3 *world*, bool *doDebug* = `false`) `[virtual]`

Parameters

world	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Returns

World position of the nearest vertex.

Returns the world position of the nearest vertex from a given point in world space. If `doDebug` is set a small gizmo sphere will be drawn at that position.

Implements GFGrid.

20.4.3.32 Vector4 GFHexGrid.RhombicDToCubic (Vector3 *rhombic*)

Parameters

rhombic	Point in downwards rhombic coordinates.

Returns

Point in cubic coordinates.

Takes a point in downwards rhombic coordinates and returns its cubic position.

20.4.3.33 Vector3 GFHexGrid.RhombicDToHerringD (Vector3 *rhombic*)

Parameters

rhombic	Point in downwards rhombic coordinates.

Returns

Point in downwards herring coordinates.

Takes a point in downwards rhombic coordinates and returns its downwards herring position.

20.4.3.34 Vector3 GFHexGrid.RhombicDToHerringU (Vector3 *rhombic*)

Parameters

rhombic	Point in downwards rhombic coordinates.

Returns

Point in upwards herring coordinates.

Takes a point in downwards rhombic coordinates and returns its upwards herring position.

20.4.3.35 Vector3 GFHexGrid.RhombicDToRhombic (Vector3 *rhombic*)

Parameters

rhombic	Point in downwards rhombic coordinates.

Returns

Point in downwards herring coordinates.

Takes a point in downwards rhombic coordinates and returns its downwards herring position.

20.4.3.36 Vector3 GFHexGrid.RhombicDToWorld (Vector3 *rhombic*)

Parameters

rhombic	Point in downwards rhombic coordinates.

Returns

Point in world coordinates.

Takes a point in downwards rhombic coordinates and returns its world position.

20.4.3.37 Vector4 GFHexGrid.RhombicToCubic (Vector3 *rhombic*)

Parameters

rhombic	Point in rhombic coordinates.

Returns

Point in cubic coordinates.

Takes a point in rhombic coordinates and returns its cubic position.

20.4.3.38 Vector3 GFHexGrid.RhombicToHerringD (Vector3 *rhombic*)

Parameters

rhombic	Point in rhombic coordinates.

Returns

Point in downwards herring coordinates.

Takes a point in rhombic coordinates and returns its downwards herring position.

20.4.3.39 Vector3 GFHexGrid.RhombicToHerringU (Vector3 *rhombic*)

Parameters

rhombic	Point in rhombic coordinates.

Returns

Point in upwards herring coordinates.

Takes a point in rhombic coordinates and returns its upwards herring position.

20.4.3.40 Vector3 GFHexGrid.RhombicToRhombicD (Vector3 *rhombic*)

Parameters

rhombic	Point in rhombic coordinates.

Returns

Point in downwards rhombic coordinates.

Takes a point in rhombic coordinates and returns its downwards rhombic position.

20.4.3.41 Vector3 GFHexGrid.RhombicToWorld (Vector3 *rhombic*)

Parameters

rhombic	Point in rhombic coordinates.

Returns

Point in world coordinates.

Takes a point in rhombic coordinates and returns its world position.

20.4.3.42 override Vector3 GFHexGrid.ScaleVector3 (Vector3 *scl,* BoolVector3 *lockAxis*) `[virtual]`

Returns

The re-scaled vector.

Parameters

scl	The vector to scale.
lockAxis	The axes to ignore.

This method takes in a vector representing a size and scales it to the nearest multiple of the grid's radius and depth. The `lockAxis` parameter lets you ignore individual axes.

Implements GFGrid.

20.4.3.43 Vector4 GFHexGrid.WorldToCubic (Vector3 *world*)

Parameters

world	Point in world coordinates.

Returns

Point in cubic coordinates.

This method takes a point in world space and returns the corresponding rhombic coordinates. The cubic coordinate system uses four axes; X, Y and Z are used to fix the point on the layer while W is which layer of the grid the point is on, relative to the grid's central layer. The central hex has coordinates (0, 0, 0, 0) and the sum of the first three coordinates is always 0.

20.4.3.44 override Vector3 GFHexGrid.WorldToGrid (Vector3 *worldPoint*) `[virtual]`

Parameters

worldPoint	Point in world space.

Returns

Grid coordinates of the world point (upwards herringbone coordinate system).

This is the same as calling `WorldToHerringU`, because upwards herringbone is the default grid coordinate system.

Implements GFGrid.

20.4.3.45 Vector3 GFHexGrid.WorldToHerringD (Vector3 *world*)

Parameters

world	Point in world coordinates.

Returns

Point in downwards herringbone coordinates.

This method takes a point in world space and returns the corresponding downwards herringbone coordinates. Every odd numbered column is offset downwards, giving this coordinate system the herringbone pattern. This means that the Y coordinate directly depends on the X coordinate. The Z coordinaate is simply which layer of the grid is on, relative to the grid's central layer.

20.4.3.46 Vector3 GFHexGrid.WorldToHerringU (Vector3 *world*)

Parameters

world	Point in world coordinates.

Returns

Point in upwards herringbone coordinates.

This method takes a point in world space and returns the corresponding upwards herringbone coordinates. Every odd numbered column is offset upwards, giving this coordinate system the herringbone pattern. This means that the Y coordinate directly depends on the X coordinate. The Z coordinaate is simply which layer of the grid is on, relative to the grid's central layer.

20.4.3.47 Vector3 GFHexGrid.WorldToRhombic (Vector3 *world*)

Parameters

world	Point in world coordinates.

Returns

Point in rhombic coordinates.

This method takes a point in world space and returns the corresponding rhombic coordinates. The rhombic coordinate system uses three axes; the X-axis rotated 30° counter-clockwise, the regular Y-axis, and the Z coordinate is which layer of the grid the point is on, relative to the grid's central layer.

20.4.3.48 Vector3 GFHexGrid.WorldToRhombicD (Vector3 *world*)

Parameters

world	Point in world coordinates.

Returns

Point in downwards rhombic coordinates.

This method takes a point in world space and returns the corresponding downwards rhombic coordinates. The downwards rhombic coordinate system uses three axes; the X-axis rotated 300° counter-clockwise, the regular Y-axis, and the Z coordinate is which layer of the grid the point is on, relative to the grid's central layer.

20.4.4 Property Documentation

20.4.4.1 HexGridShape GFHexGrid.gridStyle `[get], [set]`

The shape when drawing or rendering the grid. This only affects the grid's appearance, but not how it works.

20.4.4.2 float GFHexGrid.height `[get]`

This is the full vertical height of a hex. For pointy side hexes this is the distance from one edge to its opposite (`sqrt(3) * radius`) and for flat side hexes it is the distance between two opposite vertixes (`2 * radius`).

20.4.4.3 HexOrientation GFHexGrid.hexSideMode `[get], [set]`

Whether the grid has pointy sides or flat sides. This affects both the drawing and the calculations.

20.4.4.4 HexTopOrientation GFHexGrid.hexTopMode `[get], [set]`

Whether the grid has flat tops or pointy tops. This is directly connected to `hexSideMode`, in fact this is just an accessor that gets and sets the appropriate value for it.

20.4.4.5 float GFHexGrid.radius `[get], [set]`

This refers to the distance between the centre of a hexagon and one of its vertices. Since the hexagon is regular all vertices have the same distance from the centre. In other words, imagine a circumscribed circle around the hexagon, its radius is the radius of the hexagon. The value may not be less than 0.1 (please contact me if you really need lower values).

20.4.4.6 float GFHexGrid.side `[get]`

Shorthand writing for `1.5f * radius` (read-only).

20.4.4.7 float GFHexGrid.width `[get]`

This is the full horizontal width of a hex. For pointy side hexes this is the distance from one vertex to its opposite (`2 * radius`) and for flat side hexes it is the distance between to opposite edges (`sqrt(3) * radius`).

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/GFHexGrid.cs

20.5 GFLayeredGrid Class Reference

The parent class for all layered grids.

Inheritance diagram for GFLayeredGrid:

Public Member Functions

- override Vector3 NearestFaceW (Vector3 worldPoint, GridPlane plane, bool doDebug)

    Returns the world position of the nearest face.
- override Vector3 NearestFaceG (Vector3 worldPoint, GridPlane plane)

    Returns the grid position of the nearest Face.

Properties

- float depth `[get, set]`

    How far apart layers of the grid are.
- GridPlane gridPlane `[get, set]`

    What plane the layers are on.

Additional Inherited Members

20.5.1 Detailed Description

This class serves as a parent for all grids composed out of two-dimensional grids stacked on top of each other (currently only hex- and polar grids). These grids have a plane (orientation) and a "depth" (how densely stacked they are). Other than keeping common values and internal methods in one place, this class has not much practical use. I recommend yu ignore it, it is documented just for the sake of completion.

20.5.2 Member Function Documentation

20.5.2.1 override Vector3 GFLayeredGrid.NearestFaceG (Vector3 *worldPoint*, GridPlane *plane*) `[virtual]`

Returns

Grid position of the nearest face.

Parameters

worldPoint	Point in world space.
plane	Plane on which the face lies.

Similar to NearestVertexG, it returns the grid coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices. You also need to specify on which plane the face lies (optional for hex- and polar grids).

This is just an abstract template for the method, look into the specific class for exact implementation.

Implements GFGrid.

20.5.2.2 override Vector3 GFLayeredGrid.NearestFaceW (Vector3 *worldPoint*, GridPlane *plane*, bool *doDebug*) `[virtual]`

Returns

World position of the nearest face.

Parameters

worldPoint	Point in world space.
plane	Plane on which the face lies.
doDebug	If set to `true` draw a sphere at the destination.

Similar to NearestVertexW, it returns the world coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices. You also need to specify on which plane the face lies (optional for hex- and polar grids). If *doDebug* is set a small gizmo face will drawn there.

This is just an abstract template for the method, look into the specific class for exact implementation.

Implements GFGrid.

20.5.3 Property Documentation

20.5.3.1 float GFLayeredGrid.depth `[get], [set]`

Depth of grid layers.

Layered grids are made of an infinite number of two-dimensional grids stacked on top of each other. This determines how far apart those layers are. The value cannot be lower than 0.1 in order to prevent contradictory values.

20.5.3.2 GridPlane GFLayeredGrid.gridPlane `[get], [set]`

The plane on which the grid is aligned.

Layered grids are made of an infinite number of two-dimensional grids stacked on top of each other. This determines the orientation of these layers, i. e. if they are XY-, XZ- or YZ-layers.

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/Abstracts/GFLayeredGrid.cs

20.6 GFPolarGrid Class Reference

A polar grid based on cylindrical coordinates.

Inheritance diagram for GFPolarGrid:

Public Member Functions

- override Vector3 WorldToGrid (Vector3 worldPoint)

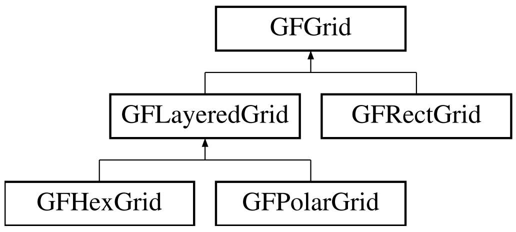
  Converts from world to grid coordinates.
- override Vector3 GridToWorld (Vector3 gridPoint)

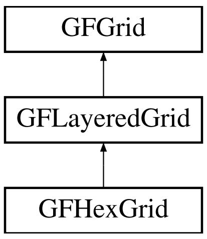
  Converts from grid to world coordinates.
- Vector3 WorldToPolar (Vector3 worldPoint)

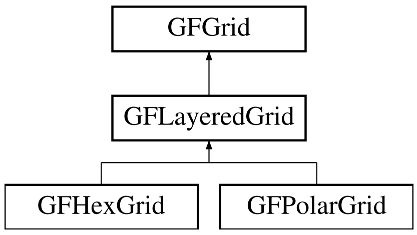
  Converts from world to polar coordinates.
- Vector3 PolarToWorld (Vector3 polarPoint)

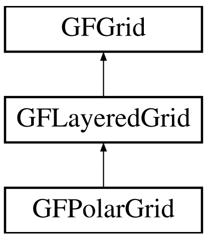
  Converts from polar to world coordinates.
- Vector3 GridToPolar (Vector3 gridPoint)

  Converts a point from grid to polar space.
- Vector3 PolarToGrid (Vector3 polarPoint)


GFGrid
GFLayeredGrid
GFRectGrid
GFHexGrid
GFPolarGrid


GFGrid
GFLayeredGrid
GFHexGrid


GFGrid
GFLayeredGrid
GFHexGrid
GFPolarGrid


GFGrid
GFLayeredGrid
GFPolarGrid

Converts a point from polar to grid space.

- float Angle2Sector (float angle, AngleMode mode=AngleMode.radians)

  Converts an angle (radians or degree) to the corresponding sector coordinate.
- float Sector2Angle (float sector, AngleMode mode=AngleMode.radians)

  Converts a sector to the corresponding angle coordinate (radians or degree).
- Quaternion Angle2Rotation (float angle, AngleMode mode=AngleMode.radians)

  Converts an angle around the origin to a rotation.
- Quaternion Sector2Rotation (float sector)

  Converts a sector around the origin to a rotation.
- Quaternion World2Rotation (Vector3 worldPoint)

  Converts a world position to a rotation around the origin.
- float World2Angle (Vector3 worldPoint, AngleMode mode=AngleMode.radians)

  Converts a world position to an angle around the origin.
- float World2Sector (Vector3 worldPoint)

  Converts a world position to the sector of the grid it is in.
- float World2Radius (Vector3 worldPoint)

  Converts a world position to the radius from the origin.
- override Vector3 NearestVertexW (Vector3 worldPoint, bool doDebug)

  Returns the world position of the nearest vertex.
- override Vector3 NearestFaceW (Vector3 world, bool doDebug)

  Returns the world position of the nearest face.
- override Vector3 NearestBoxW (Vector3 worldPoint, bool doDebug)

  Returns the world position of the nearest box.
- override Vector3 NearestVertexG (Vector3 worldPoint)

  Returns the grid position of the nearest vertex.
- override Vector3 NearestFaceG (Vector3 world)

  Returns the grid position of the nearest Face.
- override Vector3 NearestBoxG (Vector3 worldPoint)

  Returns the grid position of the nearest box.
- Vector3 NearestVertexP (Vector3 worldPoint)

  Returns the grid position of the nearest vertex.
- Vector3 NearestFaceP (Vector3 worldPoint)

  Returns the polar position of the nearest Face.
- Vector3 NearestBoxP (Vector3 worldPoint)

  Returns the polar position of the nearest box.
- void AlignRotateTransform (Transform transform, BoolVector3 lockAxis)

  Aligns and rotates a Transform.
- void AlignRotateTransform (Transform theTransform)
- override Vector3 AlignVector3 (Vector3 pos, Vector3 scale, BoolVector3 ignoreAxis)

  Fits a position vector into the grid.
- override Vector3 ScaleVector3 (Vector3 scl, BoolVector3 ignoreAxis)

  Scales a size vector to fit inside a grid.
- override void RenderGrid (int width=0, Camera cam=null, Transform camTransform=null)

Properties

- override Vector3 size `[get, set]`

  Overrides the size of GFGrid to make sure the angular coordinate is clamped appropriately between 0 and 2 (360°) or 0 and sectors.

- override Vector3 renderFrom `[get, set]`

  Overrides the property of GFGrid to make sure that the radial value is positive and the angular value is wrapped around properly.

- override Vector3 renderTo `[get, set]`

  Overrides the property of GFGrid to make sure that the angular value is wrapped around properly.

- float radius `[get, set]`

  The radius of the inner-most circle of the grid.

- int sectors `[get, set]`

  The amount of sectors per circle.

- int smoothness `[get, set]`

  Divides the sectors to create a smoother look.

- float angle `[get]`

  The angle of a sector in radians.

- float angleDeg `[get]`

  The angle of a sector in degrees.

Additional Inherited Members

20.6.1 Detailed Description

A grid based on cylindrical coordinates. The caracterizing values are radius, sectors and depth. The angle values are derived from sectors and we use radians internally. The coordinate systems used are either a grid-based coordinate system based on the defining values or a regular cylindrical coordinate system. If you want polar coordinates just ignore the height component of the cylindrical coordinates.

It is important to note that the components of polar and grid coordinates represent the radius, radian angle and height. Which component represents what depends on the gridPlane. The user manual has a handy table for that purpose.

The members size, renderFrom and renderTo are inherited from GFGrid, but slightly different. The first component of all three cannot be lower than 0 and in the case of renderFrom it cannot be larger than the first component of `renderTo`, and vice-versa. The second component is an angle in radians and wraps around as described in the user manual, no other restrictions. The third component is the height, it's bounded from below by 0.01 and in the case of renderFrom it cannot be larger than renderTo and vice-versa.

20.6.2 Member Function Documentation

20.6.2.1 void GFPolarGrid.AlignRotateTransform (Transform *transform,* BoolVector3 *lockAxis*)

Parameters

transform	The Transform to align.
lockAxis	Axis to ignore.

Aligns a Transform and the rotates it depending on its position inside the grid. This method cobines two steps in one call for convenience.

20.6.2.2 void GFPolarGrid.AlignRotateTransform (Transform *theTransform*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Use this overload when you want to appy the alignment on all axis, then you don't have to specity them.

20.6.2.3 override Vector3 GFPolarGrid.AlignVector3 (Vector3 *pos*, Vector3 *scale*, BoolVector3 *ignoreAxis*) `[virtual]`

Parameters

pos	The position to align.
scale	A simulated scale to decide how exactly to fit the poistion into the grid.
ignoreAxis	Which axes should be ignored.

Returns

Aligned position vector.

Fits a position inside the grid by using the object's transform. Currently the object will snap either on edges or between, depending on which is closer, ignoring the *scale* passed, but I might add an optionfor this in the future.

Implements GFGrid.

20.6.2.4 Quaternion GFPolarGrid.Angle2Rotation (float *angle*, AngleMode *mode* = `AngleMode.radians`)

Returns

Rotation quaterion which rotates arround the origin by *angle* .

Parameters

angle	Angle in either radians or degrees.
mode	The mode of the angle, defaults to radians.

This method returns a quaternion which represents a rotation within the grid. The result is a combination of the grid's own rotation and the rotation from the angle. Since we use an angle, this method is more suitable for polar coordinates than grid coordinates. See Sector2Rotation for a similar method that uses sectors.

20.6.2.5 float GFPolarGrid.Angle2Sector (float *angle*, AngleMode *mode* = `AngleMode.radians`)

Returns

Sector value of the angle.

Parameters

angle	Angle in either radians or degress.
mode	The mode of the angle, defaults to radians.

This method takes in an angle and returns in which sector the angle lies. If the angle exceeds 2 or 360° it wraps around, nagetive angles are automatically subtracted from 2 or 360°.

Let's take a grid with six sectors for example, then one sector has an agle of 360° / 6 = 60°, so a 135° angle corresponds to a sector value of 130° / 60° = 2.25.

20.6.2.6 Vector3 GFPolarGrid.GridToPolar (Vector3 *gridPoint*)

Returns

Point in polar space.

Parameters

gridPoint	Point in grid space.

Converts a point from grid to polar space. The main difference is that grid coordinates are dependent on the grid's parameters, while polar coordinates are not.

20.6.2.7 override Vector3 GFPolarGrid.GridToWorld (Vector3 *gridPoint*) [virtual]

Returns

World coordinates of the grid point.

Parameters

gridPoint	Point in grid space.

Converts a point from grid space to world space.

Implements GFGrid.

20.6.2.8 override Vector3 GFPolarGrid.NearestBoxG (Vector3 *worldPoint*) [virtual]

Returns

Grid position of the nearest box.

Parameters

worldPoint	Point in world space.

Similar to NearestVertexG, it returns the grid coordinates of a box in the grid. Since the box is enclosed by several vertices, the returned value is the point in between all of the vertices.

Implements GFGrid.

20.6.2.9 Vector3 GFPolarGrid.NearestBoxP (Vector3 *worldPoint*)

Returns

Polar position of the nearest box.

Parameters

worldPoint	Point in world space.

Similar to NearestVertexP, it returns the polar coordinates of a box in the grid. Since the box is enclosed by several vertices, the returned value is the point in between all of the vertices.

20.6.2.10 override Vector3 GFPolarGrid.NearestBoxW (Vector3 *worldPoint*, bool *doDebug*) [virtual]

Returns

World position of the nearest box.

Parameters

worldPoint	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Similar to NearestVertexW, it returns the world coordinates of a box in the grid. Since the box is enclosed by several vertices, the returned value is the point in between all of the vertices. If *doDebug* is set a small gizmo box will drawn there.

Implements GFGrid.

20.6.2.11 override Vector3 GFPolarGrid.NearestFaceG (Vector3 *world*) [virtual]

Returns

Grid position of the nearest face.

Parameters

world	Point in world space.

Similar to NearestVertexG, it returns the grid coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices.

Implements GFLayeredGrid.

20.6.2.12 Vector3 GFPolarGrid.NearestFaceP (Vector3 *worldPoint*)

Returns

Polar position of the nearest face.

Parameters

worldPoint	Point in world space.

Similar to NearestVertexP, it returns the polar coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices.

20.6.2.13 override Vector3 GFPolarGrid.NearestFaceW (Vector3 *world,* bool *doDebug*) [virtual]

Returns

World position of the nearest face.

Parameters

world	Point in world space.
doDebug	If set to `true` draw a sphere at the destination.

Similar to NearestVertexW, it returns the world coordinates of a face on the grid. Since the face is enclosed by several vertices, the returned value is the point in between all of the vertices. If *doDebug* is set a small gizmo face will drawn there.

Implements GFLayeredGrid.

20.6.2.14 override Vector3 GFPolarGrid.NearestVertexG (Vector3 *worldPoint*) [virtual]

Returns

Grid position of the nearest vertex.

Parameters

worldPoint	Point in world space.

Returns the position of the nerest vertex in grid coordinates from a given point in world space.

Implements GFGrid.

20.6.2.15 Vector3 GFPolarGrid.NearestVertexP (Vector3 *worldPoint*)

Returns

Polar position of the nearest vertex.

Parameters

worldPoint	Point in world space.

Returns the position of the nerest vertex in polar coordinates from a given point in world space.

20.6.2.16 override Vector3 GFPolarGrid.NearestVertexW (Vector3 *worldPoint,* bool *doDebug*) [virtual]

Returns

World position of the nearest vertex.

Parameters

worldPoint	Point in world space.
doDebug	If set to `true` do debug.

Returns the world position of the nearest vertex from a given point in world space. If *doDebug* is set a small gizmo sphere will be drawn at that position.

Implements GFGrid.

20.6.2.17 Vector3 GFPolarGrid.PolarToGrid (Vector3 *polarPoint*)

Returns

Point in grid space.

Parameters

polarPoint	Point in polar space.

Converts a point from polar to grid space. The main difference is that grid coordinates are dependent on the grid's parameters, while polar coordinates are not.

20.6.2.18 Vector3 GFPolarGrid.PolarToWorld (Vector3 *polarPoint*)

Returns

Point in world space.

Parameters

polarPoint	Point in polar space.

Converts a point from polar space to world space.

20.6.2.19 override void GFPolarGrid.RenderGrid (int *width* = `0`, Camera *cam* = `null`, Transform *camTransform* = `null`) `[virtual]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Renders the grid using `size` for lower and upper limits, equal to `RenderGrid(-size, size, useSeparateRenderColor ? renderAxisColors : axisColors, width, cam, camTransform);`

Reimplemented from GFGrid.

20.6.2.20 override Vector3 GFPolarGrid.ScaleVector3 (Vector3 *scl*, BoolVector3 *ignoreAxis*) `[virtual]`

Parameters

scl	The vector to scale.
ignoreAxis	The axes to ignore.

Returns

The re-scaled vector.

Scales a given scale vector to the nearest multiple of the grid's radius and depth, but does not change its position. The parameter *ignoreAxis* makes the function not touch the corresponding coordinate.

Implements GFGrid.

20.6.2.21 float GFPolarGrid.Sector2Angle (float *sector*, AngleMode *mode* = `AngleMode.radians`)

Returns

Angle value of the sector.

Parameters

sector	Sector number.
mode	The mode of the angle, defaults to radians.

This method takes in a sector coordinate and returns the corresponding angle around the origin. If the sector exceeds the amount of sectors of the grid it wraps around, nagetive sctors are automatically subtracted from the maximum.

Let's take a grid with six sectors for example, then one sector has an agle of $360° / 6 = 60°$, so a 2.25 sector corresponds to an angle of $2.25 * 60° = 135°$.

20.6.2.22 Quaternion GFPolarGrid.Sector2Rotation (float *sector*)

Returns

Rotation quaterion which rotates arround the origin.

Parameters

sector	Sector coordinate inside the grid.

This is basically the same as Angle2Rotation, excpet with sectors, which makes this method more suitable for grid coordinates than polar coordinates.

20.6.2.23 float GFPolarGrid.World2Angle (Vector3 *worldPoint,* AngleMode *mode* = `AngleMode.radians`)

Returns

Angle between the point and the grid's "right" axis.

Parameters

worldPoint	Point in world space.
mode	The mode of the angle, defaults to radians.

This method returns which angle around the grid a given point in world space has.

20.6.2.24 float GFPolarGrid.World2Radius (Vector3 *worldPoint*)

Returns

Radius of the point from the grid.

Parameters

worldPoint	Point in world space.

This method returns the distance of a world point from the grid's radial axis. This is not the same as the point's distance from the grid's origin, because it doesn't take "height" into account. Thus it is always less or equal than the distance from the origin.

20.6.2.25 Quaternion GFPolarGrid.World2Rotation (Vector3 *worldPoint*)

Returns

The rotation.

Parameters

worldPoint	Point in world space.

This method compares the point's position in world space to the grid and then returns which rotation an object should have if it was at that position and rotated around the grid.

20.6.2.26 float GFPolarGrid.World2Sector (Vector3 *worldPoint*)

Returns

Sector the point is in.

Parameters

worldPoint	Point in world space.

This method returns which which sector a given point in world space is in.

20.6.2.27 override Vector3 GFPolarGrid.WorldToGrid (Vector3 *worldPoint*) [virtual]

Returns

Grid coordinates of the world point.

Parameters

worldPoint	Point in world space.

Converts a point from world space to grid space. The first coordinate represents the distance from the radial axis as multiples of radius, the second one the sector and the thrid one the distance from the main plane as multiples of depth. This order applies to XY-grids only, for the other two orientations please consult the manual.

Implements GFGrid.

20.6.2.28 Vector3 GFPolarGrid.WorldToPolar (Vector3 *worldPoint*)

Returns

Point in polar space.

Parameters

worldPoint	Point in world space.

Converts a point from world space to polar space. The first coordinate represents the distance from the radial axis, the second one the angle in radians and the thrid one the distance from the main plane. This order applies to XY-grids only, for the other two orientations please consult the manual.

20.6.3 Property Documentation

20.6.3.1 float GFPolarGrid.angle [get]

This is a read-only value derived from `sectors`. It gives you the angle within a sector in radians and it's a shorthand writing for `(2.0f * Mathf.PI) / sectors`

20.6.3.2 float GFPolarGrid.angleDeg [get]

The same as `angle` except in degrees, it's a shorthand writing for `360.0f / sectors`

20.6.3.3 float GFPolarGrid.radius [get], [set]

The radius.

The radius of the innermost circle and how far apart the other circles are. The value cannot go below 0.01.

20.6.3.4 override Vector3 GFPolarGrid.renderFrom [get], [set]

Custom lower limit for drawing and rendering.

relativeSize useCustomRenderRange renderTo Aside from the additional constraint for the radial and angular value the same rules apply as for the base class, meeaning the vector cannot be greater than renderTo.

20.6.3.5 override Vector3 GFPolarGrid.renderTo `[get], [set]`

Custom upper limit for drawing and rendering.

relativeSize useCustomRenderRange renderFrom Aside from the additional constraint for the angular value the same rules apply as for the base class, meeaning the vector cannot be lower than renderFrom.

20.6.3.6 int GFPolarGrid.sectors `[get], [set]`

Ampunt of sectors.

The amount of sectors the circles are divided into. The minimum values is 1, which means one full circle.

20.6.3.7 override Vector3 GFPolarGrid.size `[get], [set]`

The size of the grid's visual representation.

relativeSize useCustomRenderRange Aside from the additional constraint for the angular value the same rules apply as for the base class, meaning no component can be less than 0.

20.6.3.8 int GFPolarGrid.smoothness `[get], [set]`

Smoothness of the grid segments.

Unity's GL class can only draw straight lines, so in order to get the sectors to look round this value breaks each sector into smaller sectors. The number of smoothness tells how many segments the circular line has been broken into. The amount of end points used is smoothness + 1, because we count both edges of the sector.

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/GFPolarGrid.cs

20.7 GFRectGrid Class Reference

A standard three-dimensional rectangular grid.

Inheritance diagram for GFRectGrid:

Public Member Functions

- override Vector3 WorldToGrid (Vector3 worldPoint)

  Converts world coordinates to grid coordinates.
- override Vector3 GridToWorld (Vector3 gridPoint)

  Converts grid coordinates to world coordinates.
- override Vector3 NearestVertexW (Vector3 worldPoint, bool doDebug)
- override Vector3 NearestFaceW (Vector3 worldPoint, GridPlane plane, bool doDebug)

  Returns the world position of the nearest face.
- override Vector3 NearestBoxW (Vector3 worldPoint, bool doDebug)

  Returns the world position of the nearest box.
- override Vector3 NearestVertexG (Vector3 worldPoint)

  Returns the grid position of the nearest vertex.
- override Vector3 NearestFaceG (Vector3 worldPoint, GridPlane plane)

  Returns the grid position of the nearest face.
- override Vector3 NearestBoxG (Vector3 worldPoint)

Returns the grid position of the nearest box.

- override Vector3 AlignVector3 (Vector3 pos, Vector3 scale, BoolVector3 ignoreAxis)

  Fits a position vector into the grid.
- override Vector3 ScaleVector3 (Vector3 scl, BoolVector3 ignoreAxis)

  Scales a size to fit inside the grid.

Properties

- Vector3 spacing `[get, set]`

  How large the grid boxes are.
- Vector6 shearing `[get, set]`

  How the axes are sheared.
- Vector3 right `[get]`

  Direction along the X-axis of the grid in world space.
- Vector3 up `[get]`

  Direction along the Y-axis of the grid in world space.
- Vector3 forward `[get]`

  Direction along the Z-axis of the grid in world space.

Additional Inherited Members

20.7.1 Detailed Description

Your standard rectangular grid, the characterising values is its spacing, which can be set for each axis individually.

20.7.2 Member Function Documentation

20.7.2.1 override Vector3 GFRectGrid.AlignVector3 (Vector3 *pos*, Vector3 *scale*, BoolVector3 *ignoreAxis*) `[virtual]`

Returns

Aligned position vector.

Parameters

pos	The position to align.
scale	A simulated scale to decide how exactly to fit the poistion into the grid.
ignoreAxis	Which axes should be ignored.

This method aligns a point to the grid. The *scale* parameter is needed to simulate the "size" of point, which influences the resulting position like the scale of a Transform would do above. By default it's set to one on all axes, placing the point at the centre of a box. If a component of `scale` is odd that component of the vector will be placed between edges, otherwise it will be placed on the nearest edge. The `lockAxis` parameter lets you ignore individual axes.

Implements GFGrid.

20.7.2.2 override Vector3 GFRectGrid.GridToWorld (Vector3 *gridPoint*) `[virtual]`

Returns

World coordinates of the Grid point.

Parameters

gridPoint	Point in grid space.

The opposite of WorldToGrid, this returns the world position of a point in the grid. The origin of the grid is the world position of its GameObject and its axes lie on the corresponding axes of the Transform. Rotation is taken into account for this operation.

Implements GFGrid.

20.7.2.3 override Vector3 GFRectGrid.NearestBoxG (Vector3 *worldPoint*) `[virtual]`

Returns

Grid position of the nearest box.

Parameters

worldPoint	Point in world space.

Similar to `NearestBoxW`, except you get grid coordinates instead of world coordinates. Since faces lie between vertices all three values will always have +0.5 compared to vertex coordinates. Example: `GFRectGrid my←Grid; Vector3 worldPoint; Vector3 box = myGrid.NearestBoxG(worldPoint); // something like (2.5, -1.5, 3.5)`

Implements GFGrid.

20.7.2.4 override Vector3 GFRectGrid.NearestBoxW (Vector3 *worldPoint*, bool *doDebug*) `[virtual]`

Returns

World position of the nearest box.

Parameters

worldPoint	Point in world space.
doDebug	Whether to draw a small debug sphere at the box.

Similar to NearestVertexW, it returns the world coordinates of a box in the grid. Since the box is enclosed by eight vertices, the returned value is the point in between all eight of them. If `doDebug` is set a small gizmo box will drawn inside the box.

Implements GFGrid.

20.7.2.5 override Vector3 GFRectGrid.NearestFaceG (Vector3 *worldPoint*, GridPlane *plane*) `[virtual]`

Returns

Grid position of the nearest face.

Parameters

worldPoint	Point in world space.
plane	Plane on which the face lies.

Similar to NearestFaceW, except you get grid coordinates instead of world coordinates. Since faces lie between vertices two values will always have +0.5 compared to vertex coordinates, while the values that lies on the plane will have a round number. Example: `GFRectGrid myGrid; Vector3 worldPoint; Vector3 face = myGrid.NearestFaceG(worldPoint, GFGrid.GridPlane.XY); // something like (2.5, -1.5, 3)`

Implements GFGrid.

20.7.2.6 override Vector3 GFRectGrid.NearestFaceW (Vector3 *worldPoint*, GridPlane *plane*, bool *doDebug*) `[virtual]`

Returns

World position of the nearest face.

Parameters

worldPoint	Point in world space.
plane	Plane on which the face lies.
doDebug	Whether to draw a small debug sphere at the vertex.

Similar to NearestVertexW, it returns the world coordinates of a face on the grid. Since the face is enclosed by four vertices, the returned value is the point in between all four of the vertices. You also need to specify on which plane the face lies. If `doDebug` is set a small gizmo face will drawn inside the face.

Implements GFGrid.

20.7.2.7 override Vector3 GFRectGrid.NearestVertexG (Vector3 *worldPoint*) `[virtual]`

Returns

Grid position of the nearest vertex.

Parameters

worldPoint	Point in world space.

Similar to `NearestVertexW`, except you get grid coordinates instead of world coordinates.

Implements GFGrid.

20.7.2.8 override Vector3 GFRectGrid.NearestVertexW (Vector3 *worldPoint*, bool *doDebug*) `[virtual]`

Returns the world position of the nearest vertex from a given point in world space. If `doDebug` is set a small gizmo sphere will be drawn at the vertex position.

Implements GFGrid.

20.7.2.9 override Vector3 GFRectGrid.ScaleVector3 (Vector3 *scl*, BoolVector3 *ignoreAxis*) `[virtual]`

Returns

The re-scaled vector.

Parameters

scl	The vector to scale.
ignoreAxis	The axes to ignore.

Scales a size to the nearest multiple of the grid's spacing. The parameter *ignoreAxis* makes the function not touch the corresponding coordinate.

Implements GFGrid.

20.7.2.10 override Vector3 GFRectGrid.WorldToGrid (Vector3 *worldPoint*) `[virtual]`

Returns

Grid coordinates of the world point.

Parameters

worldPoint	Point in world space.

Takes in a position in wold space and calculates where in the grid that position is. The origin of the grid is the world position of its GameObject and its axes lie on the corresponding axes of the Transform. Rotation is taken into account for this operation.

Implements GFGrid.

20.7.3 Property Documentation

20.7.3.1 Vector3 GFRectGrid.forward `[get]`

Unit vector in grid scale along the grid's Z-axis.

The Z-axis of the grid in world space. This is a shorthand writing for `spacing.z * transform.forward`. The value is read-only.

20.7.3.2 Vector3 GFRectGrid.right `[get]`

Unit vector in grid scale along the grid's X-axis.

The X-axis of the grid in world space. This is a shorthand writing for `spacing.x * transform.right`. The value is read-only.

20.7.3.3 Vector6 GFRectGrid.shearing `[get], [set]`

Shearing vector of the grid.

How much the individual axes of the grid are skewed towards each other. For instance, this means the if *XY* is set to *2*, then for each point with grid coordinates _(x, y)_ will be mapped to _(x, y + 2x)_, while the uninvolved *Z* coordinate remains the same. For more information refer to the manual.

20.7.3.4 Vector3 GFRectGrid.spacing `[get], [set]`

The spacing of the grid.

How far apart the lines of the grid are. You can set each axis separately, but none may be less than 0.1, in order to prevent values that don't make any sense.

20.7.3.5 Vector3 GFRectGrid.up `[get]`

Unit vector in grid scale along the grid's Y-axis.

The Y-axis of the grid in world space. This is a shorthand writing for `spacing.y * transform.up`. The value is read-only.

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/GFRectGrid.cs

20.8 GridFramework.Vectors.Vector6 Class Reference

Class representing six adjacent float values for the shearing of a rectangular grid.

Inherits IEquatable< Vector6 >.

Public Member Functions

- Vector6 (float xy, float xz, float yx, float yz, float zx, float zy)

  Creates a new vector with given values.
- Vector6 (float value)

  Creates a new vector with all values set to the same value.
- Vector6 (Vector6 original)

  Creates a new vector from an existing vector.
- Vector6 ()

  Creates a new vector with undefined, but allocated values.
- void Set (float xy, float xz, float yx, float yz, float zx, float zy)

  Set the values of the vector manually.
- void Set (float value, int index)

  Set a value of the vector at a certain index.
- void Set (float value, Axis2 component)

  Set a value of the vector at a certain component.
- void Set (Vector2 values, int index)

  Set two values of the vector at a certain index.
- void Set (Vector2 values, Axis axis)

  Set two values of the vector at a certain axis.
- void Set (Vector6 original)

  Copies the values of another vector.
- void Add (Vector6 summand)

  Adds a vector to this vector.
- void Subtract (Vector6 subtrahend)

  Subtracts a vector from this vector.
- void Negate ()

  Negates a vector.
- void Scale (Vector6 factor)

  Scales this vector component-wise with another vector.
- void Scale (float factor)

  Scales this vector component-wise with a scalar factor.
- void Max (Vector6 comparedTo)

  Component-wise maximum of this vector and another one.
- void Min (Vector6 comparedTo)

  Component-wise minimum of this vector and another one.
- void Lerp (Vector6 towards, float t)

  Linearly interpolates a vector with another one.

Static Public Member Functions

- static Vector6 Add (Vector6 summand1, Vector6 summand2)

  Adds two vectors and returns the result as a new vector.
- static Vector6 Subtract (Vector6 minuend, Vector6 subtrahend)

  Subtracts two vectors and returns the result as a new vector.
- static Vector6 Negate (Vector6 value)

  Negates a vector and returns the result as a new vector.
- static Vector6 Scale (Vector6 factor1, Vector6 factor2)

  Scales a vector component-wise with another vector and returns the result as a new vector.
- static Vector6 Scale (Vector6 vector, float factor)

  Scales a vector component-wise with a scalar factor and returns the result as a new vector.

- static Vector6 Max (Vector6 lhs, Vector6 rhs)

    Creates a new vector as the component-wise maximum of two vectors.
- static Vector6 Min (Vector6 lhs, Vector6 rhs)

    Creates a new vector as the component-wise minimum of two vectors.
- static Vector6 Lerp (Vector6 from, Vector6 to, float t)

    Linearly interpolates two vectors and returns the result as a new vector.
- static Vector6 operator+ (Vector6 lhs, Vector6 rhs)

    Creates a new vector as the sum of two vectors.
- static Vector6 operator- (Vector6 lhs, Vector6 rhs)

    Creates a new vector as the difference of two vectors.
- static Vector6 operator- (Vector6 value)

    Creates a new vector as the negationn of a vector.
- static Vector6 operator∗ (Vector6 vector, float scalar)

    Creates a new vector as the component-wise product of two vectors.
- static Vector6 operator∗ (float scalar, Vector6 vector)

    Creates a new vector as the component-wise product of a vector and a scalar.
- static Vector6 operator/ (Vector6 vector, float scalar)

    Creates a new vector as the component-wise quotient of two vectors.

Properties

- float xy `[get, set]`

    Shearing of the X axis in Y direction.
- float xz `[get, set]`

    Shearing of the X axis in Z direction.
- float yx `[get, set]`

    Shearing of the Y axis in X direction.
- float yz `[get, set]`

    Shearing of the Y axis in Z direction.
- float zx `[get, set]`

    Shearing of the Z axis in X direction.
- float zy `[get, set]`

    Shearing of the Z axis in Y direction.
- float this[int index] `[get, set]`

    Shearing value at a specific index.
- static Vector6 zero `[get]`

    Creates a new vector with all values set to zero.
- Vector2 x `[get, set]`

    Accessor to the vector's XY and XZ components.
- Vector2 y `[get, set]`

    Accessor to the vector's YX and YZ components.
- Vector2 z `[get, set]`

    Accessor to the vector's ZX and ZY components.

20.8.1 Detailed Description

This class is based on Unity's own *Vector3* struct and can be used in a similar way. It resides in Grid Framework's own namespace to prevent collision with similar types from other plugins or a future official Unity *Vector6* type.

The API is mostly the same as for *Vector3*, except in places where it wouldn't make sense. The individual components are named to reflect their shearing nature accorrding to the following pattern: "ab" where *a* is either *x*, *y* or *z* and *b* is another axis different from *a*. Two adjacent values belonging to the same axis can also be accessed as a *Vector2*, like for example *yx* and *yz*, but not *xz* and *yx*.

Note that *Vector6* is a class, not a struct, unlike Unity's Vector3. This was done for technical reasons due to Unity. It is generally not an issue, but you should avoid throwing out *Vector6* left and right like you can do with *Vector3*. Also, keep in mind that classes are always passed by reference, while structs are passed by value. If you need to assign new values use the `Set` methods instead of assigning a new instance to a variable. If you need arithmetic or comparison methods there are always two available: one that operates on an instance and mutates it rather than creating a new instance, and a static one that does return a new instance but does not mutate its arguments. Also note that the operators create new instances rather than mutating existing ones.

The individual components are indexed as follows: *xy*=0, *xz*=1, *yx*=2, *yz*=3, *zx*=4, *zy*=5.

20.8.2 Constructor & Destructor Documentation

20.8.2.1 GridFramework.Vectors.Vector6.Vector6 (float *xy*, float *xz*, float *yx*, float *yz*, float *zx*, float *zy*)

20.8.2.2 GridFramework.Vectors.Vector6.Vector6 (float *value*)

20.8.2.3 GridFramework.Vectors.Vector6.Vector6 (Vector6 *original*)

20.8.2.4 GridFramework.Vectors.Vector6.Vector6 ( )

20.8.3 Member Function Documentation

20.8.3.1 void GridFramework.Vectors.Vector6.Add (Vector6 *summand*)

20.8.3.2 static Vector6 GridFramework.Vectors.Vector6.Add (Vector6 *summand1*, Vector6 *summand2*) [static]

20.8.3.3 void GridFramework.Vectors.Vector6.Lerp (Vector6 *towards*, float *t*)

20.8.3.4 static Vector6 GridFramework.Vectors.Vector6.Lerp (Vector6 *from*, Vector6 *to*, float *t*) [static]

20.8.3.5 void GridFramework.Vectors.Vector6.Max (Vector6 *comparedTo*)

20.8.3.6 static Vector6 GridFramework.Vectors.Vector6.Max (Vector6 *lhs*, Vector6 *rhs*) [static]

20.8.3.7 void GridFramework.Vectors.Vector6.Min (Vector6 *comparedTo*)

20.8.3.8 static Vector6 GridFramework.Vectors.Vector6.Min (Vector6 *lhs*, Vector6 *rhs*) [static]

20.8.3.9 void GridFramework.Vectors.Vector6.Negate ( )

20.8.3.10 static Vector6 GridFramework.Vectors.Vector6.Negate (Vector6 *value*) [static]

20.8.3.11 static Vector6 GridFramework.Vectors.Vector6.operator* (Vector6 *vector*, float *scalar*) [static]

20.8.3.12 static Vector6 GridFramework.Vectors.Vector6.operator* (float *scalar*, Vector6 *vector*) [static]

20.8.3.13 static Vector6 GridFramework.Vectors.Vector6.operator+ (Vector6 *lhs*, Vector6 *rhs*) [static]

20.8.3.14 static Vector6 GridFramework.Vectors.Vector6.operator- (Vector6 *lhs,* Vector6 *rhs*) `[static]`

20.8.3.15 static Vector6 GridFramework.Vectors.Vector6.operator- (Vector6 *value*) `[static]`

20.8.3.16 static Vector6 GridFramework.Vectors.Vector6.operator/ (Vector6 *vector,* float *scalar*) `[static]`

20.8.3.17 void GridFramework.Vectors.Vector6.Scale (Vector6 *factor*)

20.8.3.18 void GridFramework.Vectors.Vector6.Scale (float *factor*)

20.8.3.19 static Vector6 GridFramework.Vectors.Vector6.Scale (Vector6 *factor1,* Vector6 *factor2*) `[static]`

20.8.3.20 static Vector6 GridFramework.Vectors.Vector6.Scale (Vector6 *vector,* float *factor*) `[static]`

20.8.3.21 void GridFramework.Vectors.Vector6.Set (float *xy,* float *xz,* float *yx,* float *yz,* float *zx,* float *zy*)

20.8.3.22 void GridFramework.Vectors.Vector6.Set (float *value,* int *index*)

20.8.3.23 void GridFramework.Vectors.Vector6.Set (float *value,* Axis2 *component*)

20.8.3.24 void GridFramework.Vectors.Vector6.Set (Vector2 *values,* int *index*)

20.8.3.25 void GridFramework.Vectors.Vector6.Set (Vector2 *values,* Axis *axis*)

20.8.3.26 void GridFramework.Vectors.Vector6.Set (Vector6 *original*)

20.8.3.27 void GridFramework.Vectors.Vector6.Subtract (Vector6 *subtrahend*)

20.8.3.28 static Vector6 GridFramework.Vectors.Vector6.Subtract (Vector6 *minuend,* Vector6 *subtrahend*) `[static]`

20.8.4 Property Documentation

20.8.4.1 float GridFramework.Vectors.Vector6.this[int index] `[get]`, `[set]`

20.8.4.2 Vector2 GridFramework.Vectors.Vector6.x `[get]`, `[set]`

20.8.4.3 float GridFramework.Vectors.Vector6.xy `[get]`, `[set]`

20.8.4.4 float GridFramework.Vectors.Vector6.xz `[get]`, `[set]`

20.8.4.5 Vector2 GridFramework.Vectors.Vector6.y `[get]`, `[set]`

20.8.4.6 float GridFramework.Vectors.Vector6.yx `[get]`, `[set]`

20.8.4.7 float GridFramework.Vectors.Vector6.yz `[get]`, `[set]`

20.8.4.8 Vector2 GridFramework.Vectors.Vector6.z `[get]`, `[set]`

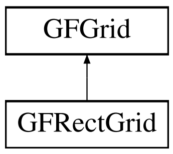
20.8.4.9 Vector6 GridFramework.Vectors.Vector6.zero `[static]`, `[get]`

20.8.4.10 float GridFramework.Vectors.Vector6.zx `[get]`, `[set]`

20.8.4.11 float GridFramework.Vectors.Vector6.zy `[get]`, `[set]`

The documentation for this class was generated from the following file:

- Assets/Plugins/Grid Framework/Vectors/Vector6.cs


GFGrid
GFRectGrid