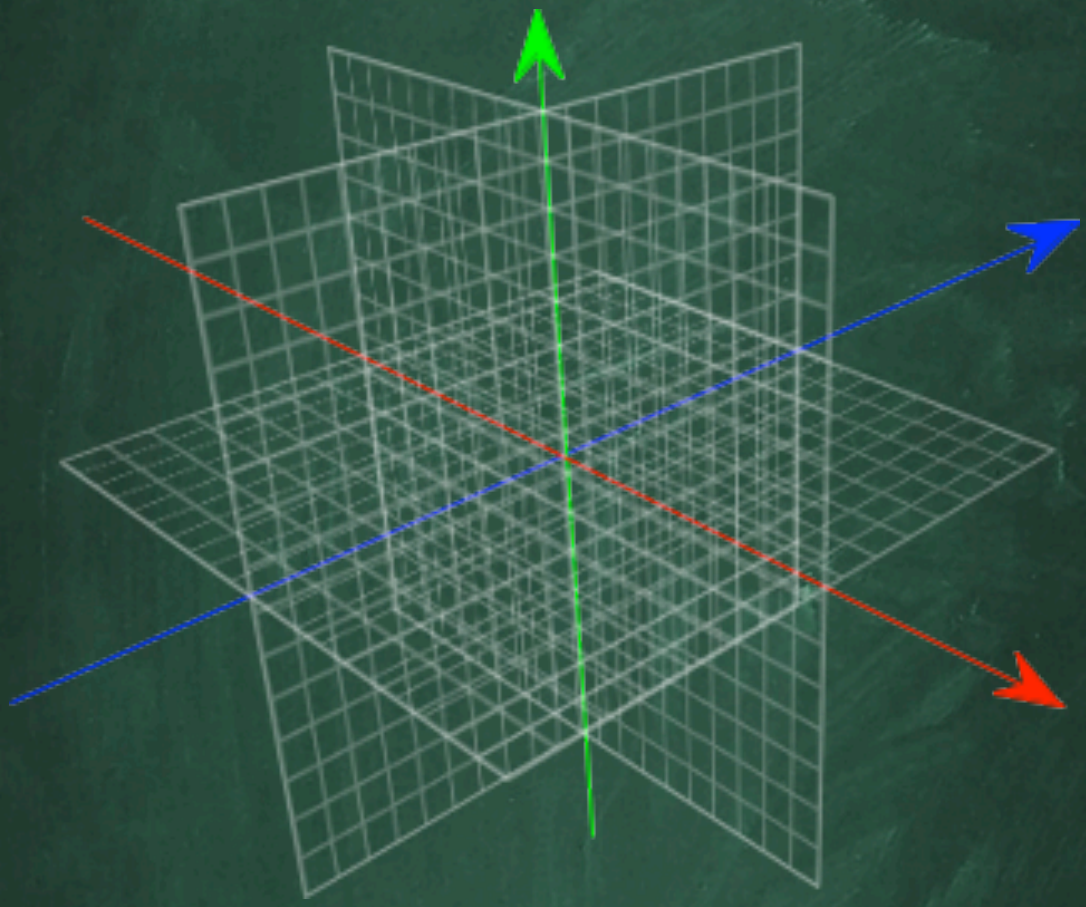
Grid Framework



User Manual

Version 1.3.0

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, infinite in size and can store a set amount of grid vertices as Vector3 inside a matrix for quick reference. 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

What is included

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

The Grid Framework Folder contains the user manual, scripting reference, 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* and *GFHexGrid.cs*, and the folders *Abstract* (containing *GFGrid.cs* and *GFLayeredGrid*), *Extension Methods* (containing *GFTransformExtensions.cs* and *GFVectorThreeExtensions.cs*) and *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)*.

Out of these only *GFRectGrid* and *GFHexGrid* are relevant to end-users, *GFGrid and GFLayerdGrid* are abstract classes and cannot be added to Objects (more on that later) and the rest just contain supplementary 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* and *GFHexGridEditor*, 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.

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 above. You can also delete the global *Grid Framework* folder or any of its contents without breaking anything if you don't need it.

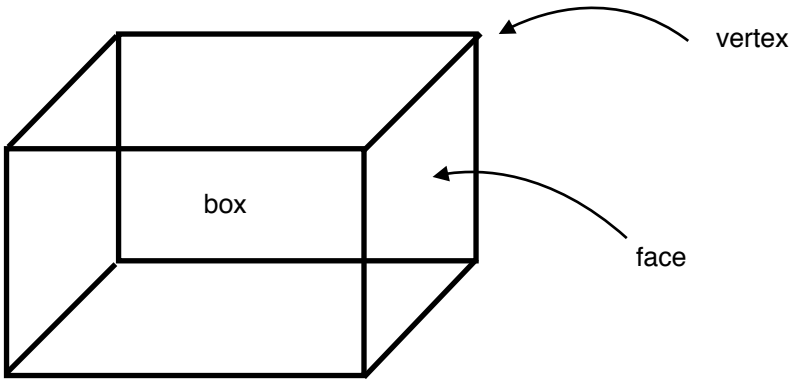
All features of Grid Framework can be accessed through the menu bar, see the *Getting Started* section for more information, there is no reason to attach scripts manually.

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 GridPlane

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.

The Grids

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

```
var myGrid: GFRectGrid
```

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 this code:

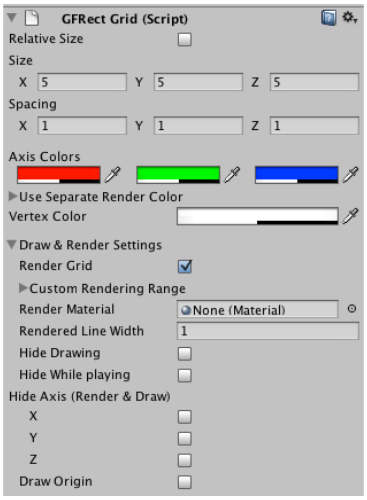
```
var myGrid: GFGrid;
var myVec: Vector3 = myGrid.NearestVertexW(transform.position);
```

will return the appropriate vector for any grid type. See the scripting reference for more details.

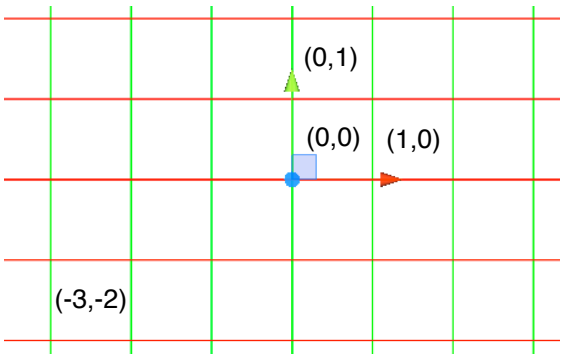
The GFRectGrid Class

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

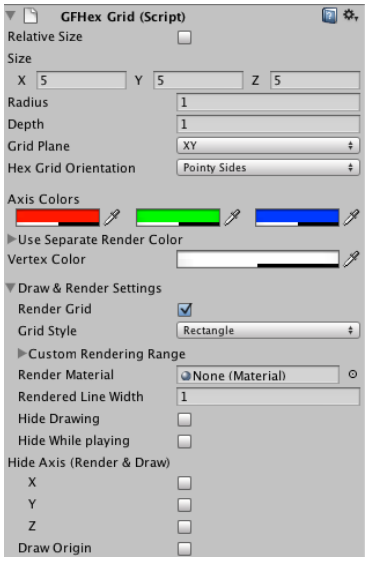
The central vertex has the grid coordinates (0, 0, 0) and each coordinate goes in the direction of the world’s corresponding axis. Practically speaking, the vertex (1, 2, 3) would be one unit to the right, two units above and three units in front of the central vertex (all values in local space). Thus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



The coordinates for faces are different, we need to shift them half a unit in each axis’s direction, since they are identified by their central point. The only coordinate not shifted is the one which corresponds to the plane we are on. So a face on the XY-plane would only have its X and Y coordinate shifted, but not its Z coordinate. Boxes are similar, except all three of their coordinates are shifted.


(0,1)
(0,0)
(1,0)
(-3,-2)

The GFHexGrid class

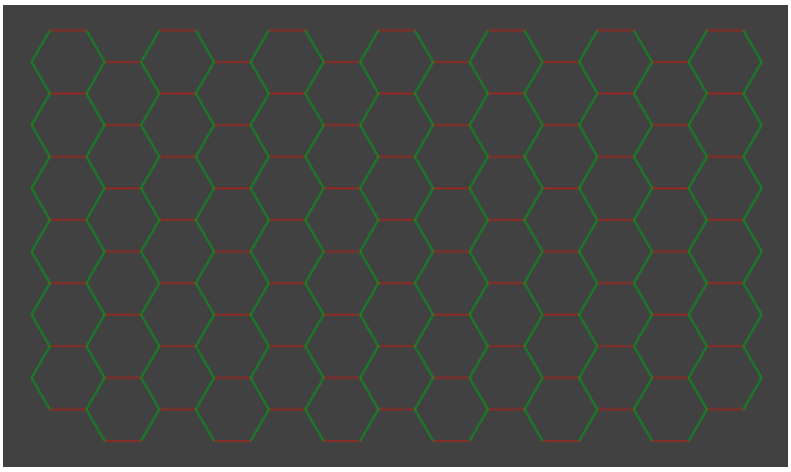


Hex grids resemble honeycomb patterns, they are composed of regular hexagons arranged in such a way that each hex shares an edge with another hex. The distance between two neighbouring hexes is always the same, unlike rectangular grids where two rectangles can be diagonal to each other.

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 `grid plane` 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.

Different kinds of hex grids

There are several ways to draw 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. When it comes to drawing you can also set either all columns to have the same height or have every second one (the shifted ones) reduced by one. 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.

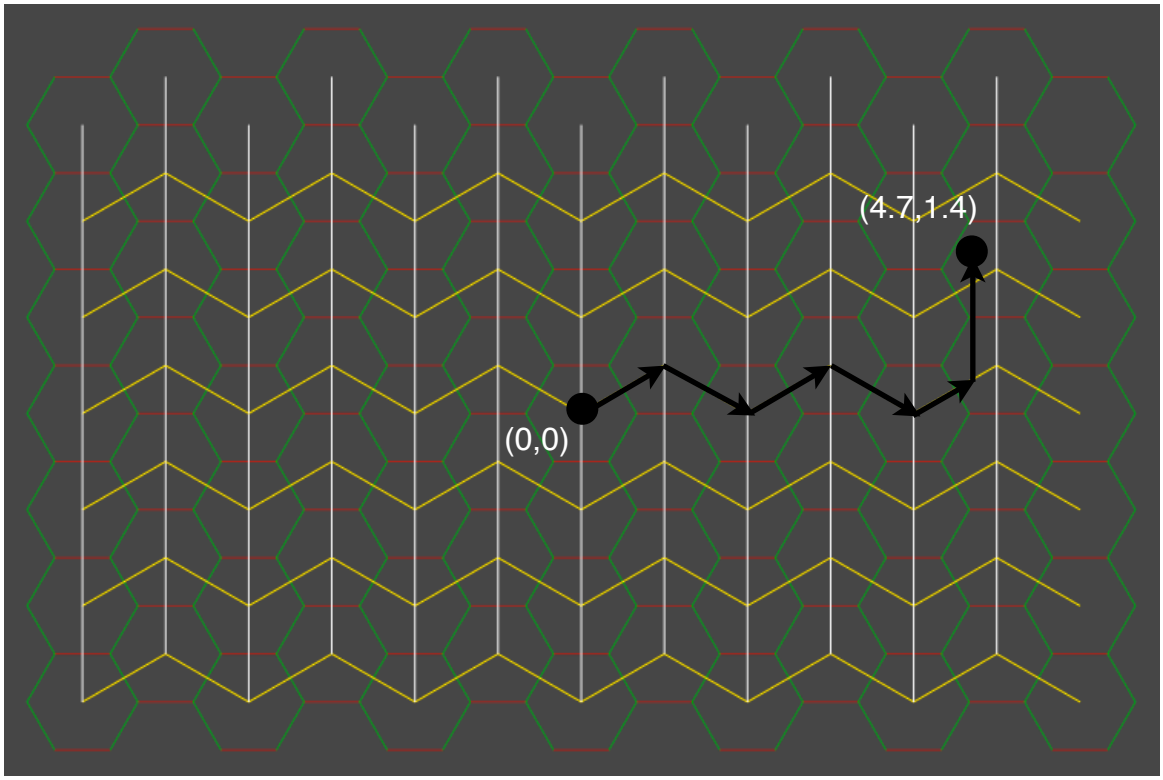


a honeycomb pattern with pointy sides

Coordinate system 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. The centres of these faces are then connected to form a herringbone pattern. Imagine a cartesian coordinate system where every second column is pulled upwards. The centre of the central face is our origin. To get the world position of a point inside the grid one would first move along the X-axis into the strip of the point, then add to that the fraction of the X-axis vector and finally add the remaining difference in

height as the Y-coordinate. The Z-coordinate is the distance from the central plane of the grid. Here is an example of how it works:



This coordinate system makes movement from one hex to another very easy, but vertex-based movement is a tricky matter. Let's say we want the herring-coordinates of the north-eastern vertex of the central hex. That would be (1/3,1/3), i.e. we add 1/3 to both coordinates of the face. This rule applies to all hexes with an even X-coordinate. However, if we take a look at the hex (1,0) we see that it is necessary to add (1/3, 2/3) to its coordinates. The reason for this is that in the even case the X-axis is pointing upwards and in the odd case it's pointing downwards, hence you need to add an extra 1/3.

Other coordinate systems are possible as well and will be added in the future.

Nearest coordinates in hex grids

The functions `NearestFaceG`, `NearestBoxG` and `NearestVertexG` work similar to their counterparts in rectangular grids. When dealing with faces and vertices the coordinates are exactly the herring coordinates of the centre of the face or the vertex in question. `NearestBoxG` gets you similar results, except the Z-coordinate is shifted half a unit forward, similar to how it is done for rectangular grids.

AlignToGrid and ScaleToGrid in hex grids

`AlignToGrid` places the pivot point of an object at the centre of the nearest vertex, *regardless* of the object's scale, unlike rectangular grids where the object's scale is used to place it so it fits as closely inside the grid as possible. i made this decision because no matter how I tried to align a cuboid (which is what the Vector3 scale represents), none of it felt intuitive. I also looked at other hex-based games and how they handled it and they used the same pivot point approach. You can either adjust the pivot point manually in you

3D modeling application or parent your object to an empty GameObject that serves as your pivot point.

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 coordiante systems, a standard cylindrical one and a grid coordinate system similar to what the other grids use.

What are polar and cylindrical coordinates?

Polar grids are based on polar coordinates. In the usual cartesian coordinate system we indentify 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 indentify a point by its absolut 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 *alititude*) 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, going from 0° to 360° and as radians reaching from 0 to 2π. 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 (2π) and degree (360°) there need to be some limitations in place.

If the number is larger than the range it gets wrapped around, meaning 750° becmes 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).

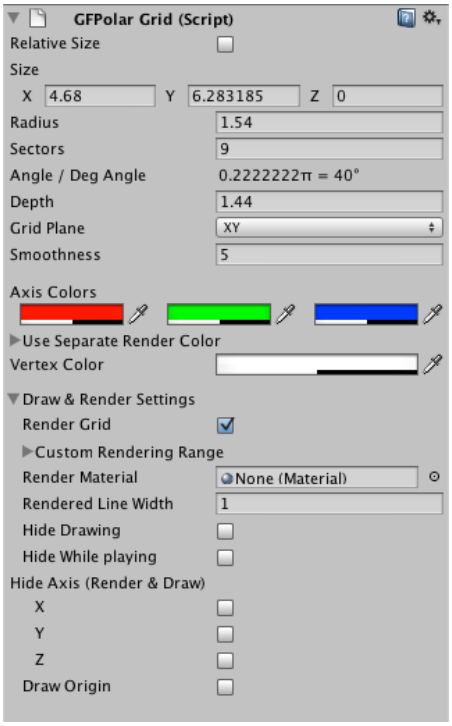
These limitations are always applied internally when you are using a polar grid’s methods and accessors, so you won’t have to worry about anything. Just be aware of what’s happening in case your circles start going the other way around or something like that.

The polar grid

In Grid Framework the defining characteristics of a polar grid are its `radius`, the amount of `sectors` and the `depth`. The `radius` is the radius of the innermost circle around the origin and the radii of all the other circls are multiples of it.

The next value, `sectors`, tells us into how many sectors we divide the circles. It's an iteger value of at least one (the circle itself is one huge sector).We derive the angle between sectors by dividing 2π (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	YZ-grid	XZ-grid
X-component	radius	height	radius
Y-component	polar angle	polar angle	height
Z-component	height	radius	polar angle

The polar coordinate system

This is a standard cylindrical coordinate system and internally it is the dafault 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.

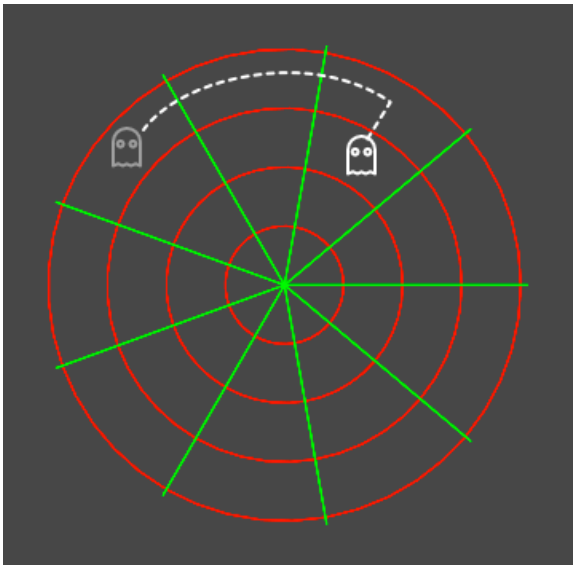
The grid coordinate system

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 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

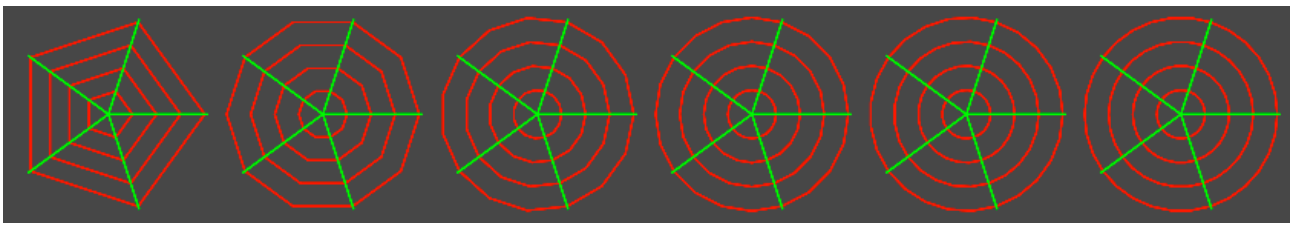
```
var myPlayerPosition: Vector3 = myGrid.WorldToGrid(player.transform.position);
myPlayerPosition += Vector3(1, 2, 0);
player.transform.position = myGrid.GridToWorld(myPlayerPosition);
```

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 2π. If you decide to use a custom range the angle of the from-value can be larger than the to-value, in this case the grid will be drawn counter-clock wise.

Aligning and scaling

When aligning a transform only its Z-scale is taken in accound, just as with rectangular grids, but the X- and Y value is ignoreg, 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.

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 `GFGridRenderCamera` 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

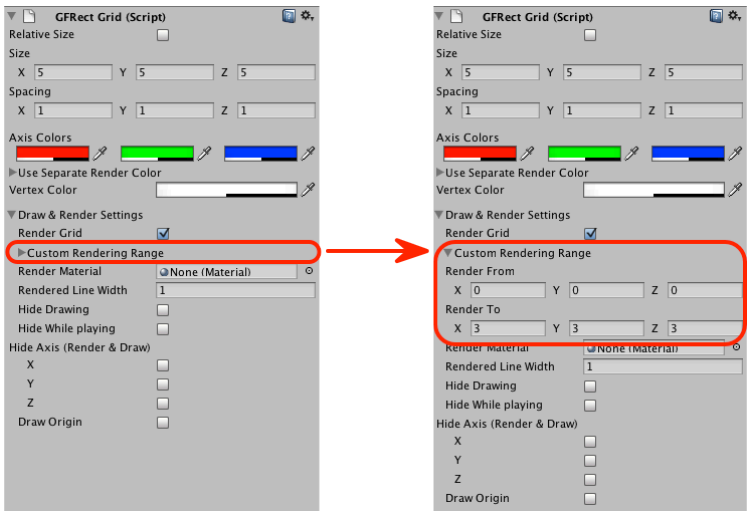
By default the `size` (and `renderFrom`/`renderTo`) is interpreted as absolute lengths in world units. This means a grid with an X-size of 5 will always be five wold units wide to both sides, regardless of its other values like *spacing* or *radius*. Setting the `useRelativeSize` flag will interpret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

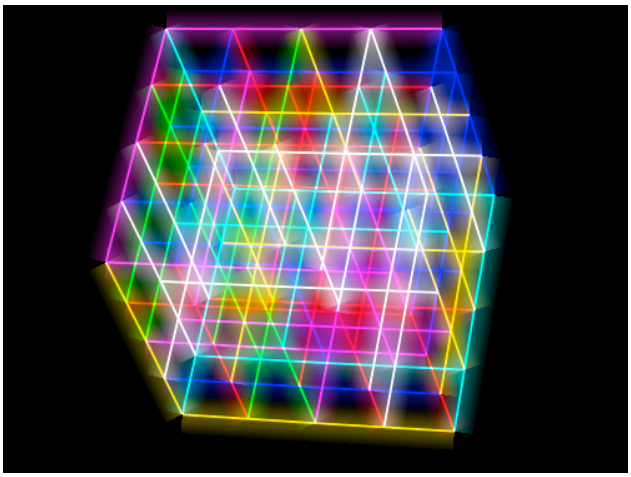


Shared Settings

The rendering shares some options with the drawing. They have the same colours and you can toggle the axes individually. `hideGrid` and hideOnPlay are only for drawing, you can toggle rendering through the `renderGrid` flag.

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.



an example of what is possible with Grid Framework and Vectrosity

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.

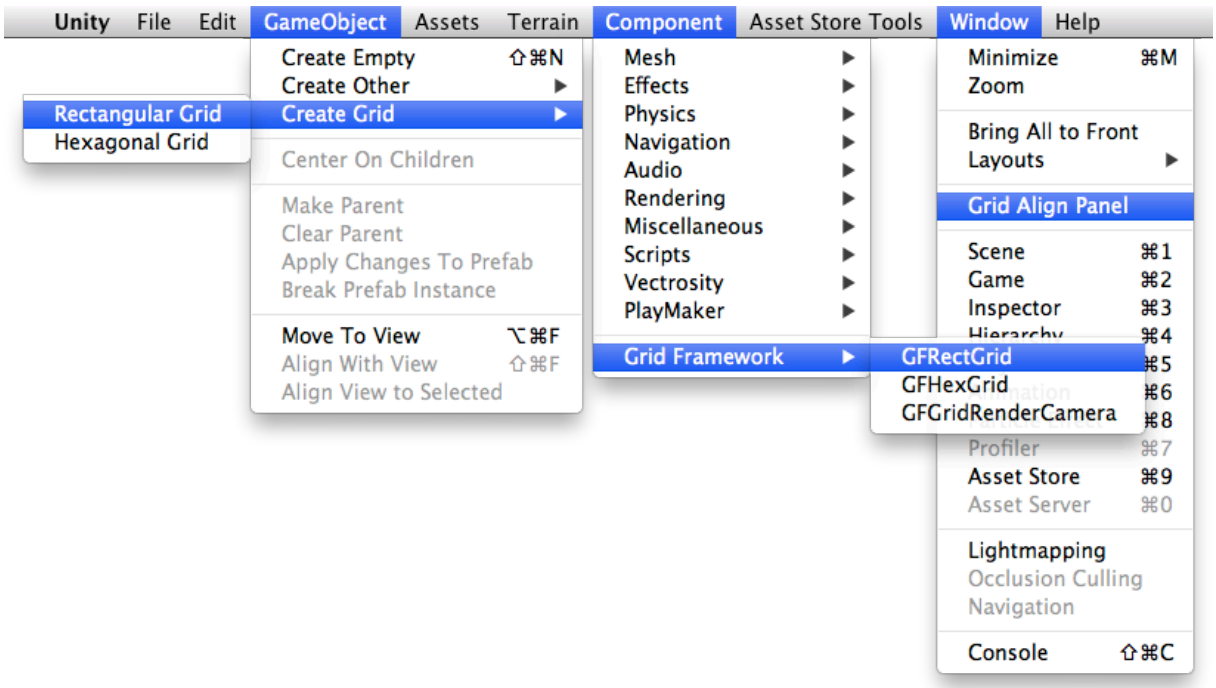
Hex grids on the other hand consist of many line segments, basically every hex adds three lines to the count (lines shared between hexes are drawn only once), outer hexes add even more. This makes a hex grid more expensive than a rectangular grid of the same size.

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 smoothign 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.

Getting started



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.

Referencing a grid

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

```
var myGrid: GFGrid = myGameObject.GetComponent.<GFGrid>();// (Javascript)
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

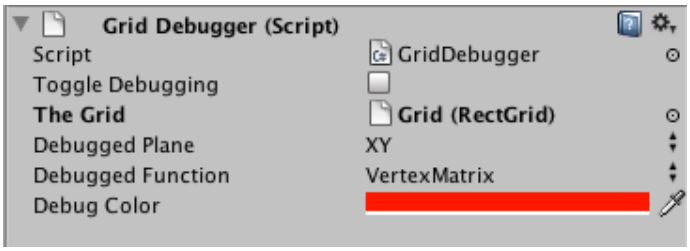
You can find this window under Window → Grid Align Panel. To use it, simply drag & drop an object with a grid from the hierarchy into the Grid field. The buttons are pretty self-explanatory. You can also set which layers will be affected, this is especially useful if you want to manipulate large groups 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.

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 → Grid Framework → Debug you'll find the script use for the sphere prefab as well as a script for debuggin various conversions in polar grids.