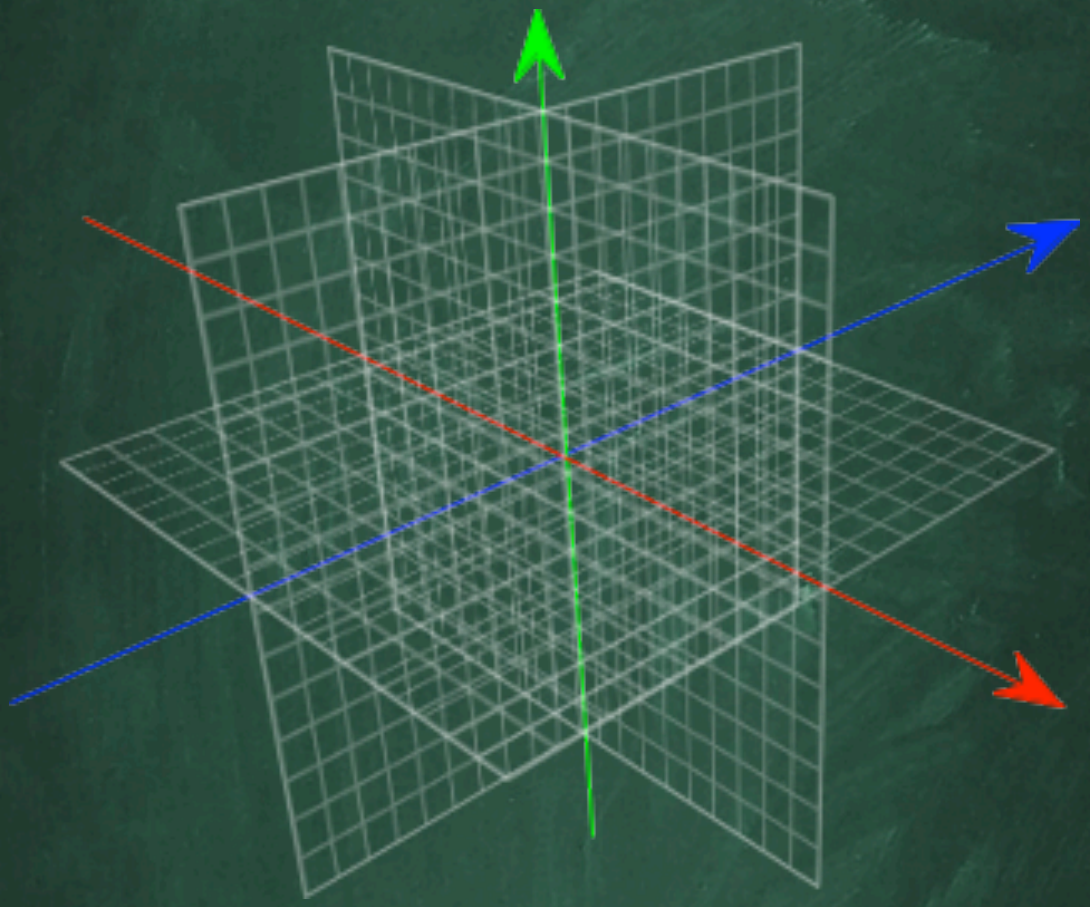
Grid Framework



Scripting Reference

Version 1.3.0

Table of Contents

Grid Framework Scripting Reference

Each grid type has its own type and inherits from the abstract GFGrid class, which in return inherits from MonoBehaviour. I am not going to list any variables and functions inherited from MonoBehaviour since that would blow the size of this document out of proportions. Also, it would be redundant to list the same things for each grid when they all inherit from the same class, which in return only inherits from MonoBehaviour. See Unity's script reference for information on that class.

GFGrid (abstract)

Inherits from MonoBehaviour

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.

Variables

relativeSize	whether the drawing/rendering will scale with spacing
size	the size of the visual representation of the grid
useCustomRenderRange	use you own values for the range of the rendering
renderFrom	custom lower limit for the rendering
renderTo	custom upper limit for the rendering
renderGrid	render the grid at runtime
renderLineWidth	the width of the lines used when rendering the grid
renderMaterial	the material for rendering, if none is given it uses a default
axisColors	colour of the axes when drawing and rendering
useSeparateRenderColor	whether to use the same colours for rendering as for drawing
renderAxisColors	separate colour of the axes when rendering (optional)
vertexColor	colour of vertices when drawing and rendering
hideGrid	don't draw the grid at all
hideAxis	hide just individual axes
hideOnPlay	hide the grid in play mode
drawOrigin	draw a little sphere at the origin of the grid

ownVertexMatrix	three-dimensional matrix for storing a list of grid vertices

Functions

NearestVertexW	returns the world position of the nearest vertex
NearestFaceW	returns the world position of the nearest face
NearestBoxW	returns the world position of the nearest box
BuildVertexMatrix	returns a Vector3[,,] containing the world position of grid vertices within a certain range of the origin
ReadVertexMatrix	returns the world position of a specified vertex in the vertex matrix
RenderGrid	renders the grid at runtime
DrawGrid	draws the grid using gizmos
DrawVertices	draws the vertex matrix entries using gizmos
GetVectrosityPoints	returns an array of Vector3 ready for use with Vectrosity
GetVectrosityPointsSeparate	same as above, except all lines of the same direction are grouped together

Overridable Functions

WorldToGrid	converts world coordinates to grid coordinates
GridToWorld	converts grid coordinates to world coordinates
NearestVertexW	returns the world position of the nearest vertex
NearestFaceW	returns the world position of the nearest face
NearestBoxW	returns the world position of the nearest box
NearestVertexG	returns the grid position of the nearest vertex
NearestFaceG	returns the grid position of the nearest face
NearestBoxG	returns the grid position of the nearest box
AlignTransform	fits a Transform inside the grid, but does not scale it
AlignVector3	similar to the above, except only for Vectors
ScaleTransform	scales a Transform to fit the grid but does not move it
ScaleVector3	similar to the above, except only for Vectors

Classes

GridPlane: enum	specifies on of three grid planes (XY, XZ and YZ)

GFGrid Variables

GFGrid.relativeSize

var **relativeSize** : boolean

If you disable this flag (default) then the values of `size`, `renderFrom` and `renderTo` will be interpreted as world unit length. If enabled they will be interpreted as grid unit lengths and the size of your drawing/rendering as well as the points returned by `GetVectrosityPoints` will depend directly on the "spacing" of the grid.

For example rectangular a grid with a `spacing` of (2, 0.5, 1), a `renderFrom` of (0, 0, 0) and a `renderTo` of (3, 4, 5) will be 3 x 4 x 5 world units large when the flag is disabled and 6 x 2 x 5 world units (3 x 4 x 5 grid units) large when the flag is enabled.

This affects only the drawing, coordinates in grid space and world space are still the same.

GFGrid.size

var **size** : Vector3

You can use this to set a limit for the grid. All grids are infinitely large, so use this to set limits. It also affects how much of the grid will be drawn. Note that none of the components can be less than 0.

GFGrid.useCustomRenderRange

var **useCustomRenderRange** : boolean

If false the grid will be rendered within its `size` limits, it true it will use custom limits.

GFGrid.renderFrom

var **renderFrom** : Vector3

Lower limit for the custom rendering range, can only be less or equal than `renderTo`.

GFGrid.renderTo

var **renderTo** : Vector3

Upper limit for the custom rendering range, can only be greater or equal than `renderFrom`.

GFGrid.renderGrid

var **renderGrid** : boolean

When set the grid will be rendered at runtime.

GFGrid.renderLineWidth

var **renderLineWidth** : int

The width of the rendered lines in pixels. If the width is one, then simple lines will be drawn, for higher numbers quads (rectangles) are used

GFGrid.renderMaterial

var **renderMaterial** : Material = null

The material used for rendering. If you don't specify any the system will use a default material:

```
defaultRenderMaterial = 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 }" +
"} } }" );
```

GFGrid.axisColors

var **axisColors** : GFColorVector3

The colours for the gizmos when drawing and rendering the grid. You can set the colour for each axis individually.

GFGrid.useSeparateRenderColor

var **useSeparateRenderColor** : boolean

By default the same colours are used for drawing and rendering, however, if you prefer to have separate colours you can set this flag. This option is useful if you want the grid to be barely visible in the game but still clearly visible in the editor. Or, if you have several grids per level, you could have all grids looks the same in the game but different in the editor so you can distinguish them more easily.

GFGrid.renderAxisColors

var **renderAxisColors** : GFColorVector3

If the above flag is set these colours will be used for rendering, otherwise this does nothing.

GFGrid.vertexColor

var **vertexColor** : Color

The colour used by the function `DrawVertices`

GFGrid.hideGrid

var **hideGrid** : boolean

When this flag is set the grid will not be drawn or rendered. This will prevent all the for-loops from being fired, saving performance.

GFGrid.hideAxis

var **hideAxis** : GFBoolVector3

Same as above, but only for individual axes. Applies both to drawing and rendering.

GFGrid.hideOnPlay

var **hideOnPlay** : boolean

Same as `hideGrid`, but hides the grid only in play mode. That way you can judge performance without interference from the grid drawing.

GFGrid.drawOrigin

var **drawOrigin** : boolean

When set draws a small sphere at the centre of the grid.

GFGrid.ownVertexMatrix

var **ownVertexMatrix** : Vector3[,,]

A three-dimensional Vector3-array, intended for storing the vertex matrix. You can technically store the value returned by `BuildVertexMatrix` in any three-dimensional Vector3 array, this variable is just for your convenience.

GFGrid functions

GFGrid.NearestVertexW

function **NearestVertexW** (**fromPoint**: Vector3,
doDebug: boolean = false) : Vector3

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. That drawing is not affected by the variable `DrawVertices`.

GFGrid.NearestFaceW

function **NearestFaceW** (**fromPoint**: Vector3,
thePlane: GFGrid.GridPlane,
doDebug: boolean = false) : Vector3

function **NearestFaceW** (**fromPoint**: Vector3,
doDebug: boolean = false) : Vector3

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 there.

For hex- and polar grids you can also ommit the plane, in that case the grid's `gridPlane` will be used.

GFGrid.NearestBoxW

function **NearestBoxW** (**fromPoint**: Vector3,
doDebug: boolean = false) : Vector3

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 there.

GFGrid.BuildVertexMatrix

function **BuildVertexMatrix** (**width**: float,
height: float,
depth: float) : Vector3[,,]

Builds the vertex matrix, a three-dimensional native .NET array of Vector3 values. The size of the matrix is specified as float, but the parameters get rounded to the nearest integers. The entries contain the world position of grid vertices within the specified range. The matrix starts in the upper left front corner (front in the sense of positive Z-coordinate).

Since .NET arrays cannot be resized, this function builds the matrix from scratch every time, so be aware of what you are doing if you decide to build this matrix every frame. Personally though, in that case I would recommend using the other functions which calculate positions on the fly.

GFGrid.ReadVertexMatrix

function **ReadVertexMatrix** (**x**: int,
y: int,
z: int,
vertexMatrix: Vector3[,,],
warning: boolean = false) : Vector3

Reads the vertex matrix in a cartesian way, i. e. the central entry has coordinates (0, 0, 0). You can pass any Vector3[,,], but the result makes most sense if you use a matrix created by `BuildVertexMatrix`. Setting warning will print out a warning to the console if you try to read something beyond the size of the matrix (like trying to read (7, -2, 1) in a 2x3x2 matrix). In any case the returned value will default to (0, 0, 0)

GFGrid.RenderGrid

function **RenderGrid** (**width**: int = 0,
cam: Camera = null) : void

function **RenderGrid** (**from**: Vector3,
to: Vector3,
width: int = 0,
cam: Camera = null) : void

Renders the grid at runtime based either on its `size` or the two points `renderFrom` and `renderTo` in local space, not affected by scale. Keep in mind that the grid is three-dimensional, to render it there are three for-loops nested into each other, so rendering a huge grid can slow down the editor. You shouldn't call the function manually, the framework makes sure it gets called at the rights places. If you still want to call it yourself, I recommend the `OnPostRender()` function of a camera.

If no width or camera are given or the width is one, the lines will be one pixel wide. The camera passed will be used to calculate the screen position for the points of the rectangles.

GFGrid.DrawGrid

function **DrawGrid** () : void

function **DrawGrid** (**from**: Vector3
to: Vector3) : void

Simply draws the grid using gizmos, the size is based on the `size` variable of the grid or the two points `renderFrom` and `renderTo` in local space, not affected by scale. Keep in mind that the grid is three-dimensional, to draw it there are three for-loops nested into each other, so drawing a huge grid can slow down the editor.

The grid you see is just a visual representation of an infinitely large grid, it is fine just to draw a small part, no matter how much of the grid you need.

GFGrid.DrawVertices

function **DrawVertices** (**vertexMatrix**: Vector3[,,],
drawOnPlay: boolean = false) : void

Draws the entries from the vertex matrix. The same warning applies as for DrawGrid. Usually the vertices won't be drawn while playing, so set drawOnPlay to true if you want to override this.

GFGrid.GetVectrosityPoints

function **GetVectrosityPoints** () : Vector3[]

function **GetVectrosityPoints** (**from**: Vector3,
to: Vector3) : Vector3[]

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. The returned points represent the grid's size.

If no arguments are passed the grid's `size` is used, otherwise the two points are. These points are in local space but not affected by scale.

GFGrid.GetVectrosityPointsSeparate

function **GetVectrosityPointsSeparate** () : Vector3[3][]

function **GetVectrosityPointsSeparate** (**from**: Vector3
to Vector3) : Vector3[3][]

Similar to above, except you get a jagged array. Each of the three arrays contains the points of lines for the same direction, i. e. the first entry contains only the lines along the grid's X-axis, the second one the Y-axis, the third one the Z-axis. Example:

```
var myLines: Vector3[][] = myGrid.GetVectrosityPointsSeparate();
myLine: VectorLine = new VectorLine("Y-Lines", myLines[1], Color.green, null, 3.0);
```

GFGrid Classes

GFGrid.GridPlane

enum **GridPlane** {YZ, XZ, XY}

This represents a grid plane. The C# documentation contains detailed information about enumerations; the important part is that you can read the values of this class either as their values or integers. You can use this type in C# as well as Unity's Javascript.

The integer value corresponds to the plane's missing axis (X=0, Y=1, Z=2), i. e. the YZ plane has the number 0, since the X-axis (the first axis) is the missing one. For instance, to refer the XZ plane you could write

```
var myPlane: GFGrid.GridPlane = GFGridPlane.XZ;
var planeIndex: int = (int)myPlane; // sets the variable to 1
```

GFRectGrid

Inherits from GFGrid

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

Variables

spacing	how large the grid boxes are
right	direction along the X-axis of the grid in world space
up	direction along the Y-axis of the grid in world space
forward	direction along the Z-axis of the grid in world space

Functions

WorldToGrid	converts world coordinates to grid coordinates
GridToWorld	converts grid coordinates to world coordinates
NearestVertexG	returns the grid position of the nearest vertex
NearestFaceG	returns the grid position of the nearest face
NearestBoxG	returns the grid position of the nearest box
AlignTransform	fits a Transform inside the grid, but does not scale it
AlignVector3	similar to the above, except only for Vectors
ScaleTransform	scales a Transform to fit the grid but does not move it
ScaleVector3	similar to the above, except only for Vectors

GFRectGrid Variables

GFRectGrid.spacing

var **spacing** : Vector3

How far apart the lines of the grid are. You can set each axis separately, but none may be less than 0.1 (please contact me if you *really* need lower values).

GFRectGrid.right

var **right** : Vector3

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

GFRectGrid.up

var **up** : Vector3

Same as above, except for the Y-axis.

GFRectGrid.forward

var **forward** : Vector3

Same as above, except for the Z-axis.

GFRectGrid Functions

GFRectGrid.WorldToGrid

function **WorldToGrid** (**worldPoint**: Vector3) : Vector3

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. Rotation is taken into account for this operation.

GFRectGrid.GridToWorld

function **GridToWorld** (**gridPoint**: Vector3) : Vector3

The opposite of `WorldToGrid`, this returns the world position of a point in the grid. They cancel each other out.

GFRectGrid.NearestVertexG

function **NearestVertexG** (**fromPoint**: Vector3) : Vector3

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

GFRectGrid.NearestFaceG

function **NearestFaceG** (**fromPoint**: Vector3,
thePlane: GFGrid.GridPlane) : Vector3

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

GFRectGrid.NearestBoxG

function **NearestBoxG** (**fromPoint**: Vector3) : Vector3

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

GFRectGrid.AlignTransform

function **AlignTransform** (**theTransform**: Transform,
rotate: boolean = true
lockAxis: GFBoolVector3 = new GFBoolVector3) : void

Tries to fit an object inside the grid by using the object's transform. If the object's scale is an even multiple (or close to one) the object's centre will be place on an edge between faces, else on a face. Setting `doRotate` makes the object take on the grid's rotation. The parameter lockAxis makes the function not touch the corresponding coordinate.

GFRectGrid.AlignVector3

function **AlignVector3** (**position**: Vector3
scale: Vector3 = Vector3.one
lockAxis: GFBoolVector3 = new GFBoolVector3(false)) : Vector3

Works similar to `AlignTransform` but instead 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. The `lockAxis` parameter works just like above.

GFRectGrid.ScaleTransform

function **ScaleTransform** (**theTransform**: Transform,

lockAxis: GFBoolVector3 = new GFBoolVector3) : void

Scales an object's transform to the nearest multiple of the grid's `spacing`, but does not change its position. The parameter `lockAxis` makes the function not touch the corresponding coordinate.

GFRectGrid.ScaleVector3

function **ScaleVector3** (**scale**: Vector3

lockAxis: GFBoolVector3 = new GFBoolVector3(false)) : Vector3

Like the above, but takes a Vector3 instead and returns the scaled vector. The `lockAxis` parameter works just like above.

GFLayeredGrid (abstract)

inherits from GFGrid

Another abstract class, it contains members shared between hex grids and polar grids. Unlike rectangular grids, these grids have a defining plane and the grid itself is two-dimensional. These grids then get stacked on top of each other to add a third coordinate (like how polar coordinates become cylindrical coordinates). The `gridPlane` determines the orientation of the 2D layers.

Variables

depth	distance between two grid layers
gridPlane	whether it's an XY-, XZ- or YZ-grid

GFLayeredGrid variables

GFLayeredGrid.depth

var **depth** : float

As mentioned in the user manual all grids are three-dimensional. A honeycomb or polar pattern on its own is just two-dimensional, so they get stacked on top of each other to form a three-dimensional grid. The depth is the distance between two such patterns and a lower values means a more dense grid. The value may not be less than 0.1 (please contact me if you really need lower values).

GFLayeredGrid.gridPlane

var **gridPlane** : GFGrid.GridPlane

This tells the grid on which of the three planes (XY, XZ or YZ) the 2D pattern lies. If for example you were to make a top-down game you could technically just take an XY-grid and rotate it, but it is more intuitive to use an XZ-grid, because the coordinates will be along the XZ-plane. This means that one unit into the Z-direction really changes the Z coordinate inside the grid's coordinate system instead of the Y-coordinate.

GFHexGrid

Inherits from GFGrid

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

Variables

radius	distance from the centre of a hex to a vertex
hexSideMode	pointy sides or flat sides
gridStyle	the shape of the overall grid, affects only drawing and rendering, not the calculations
side	1.5 times the radius
height	distance between opposite edges
width	distance between vertices on opposite sides

Functions

WorldToGrid	converts world coordinates to grid coordinates
GridToWorld	converts grid coordinates to world coordinates
NearestVertexG	returns the grid position of the nearest vertex
NearestFaceG	returns the grid position of the nearest face
NearestBoxG	returns the grid position of the nearest box
AlignTransform	fits a Transform inside the grid, but does not scale it
AlignVector3	similar to the above, except only for Vectors
ScaleTransform	scales a Transform to fit the grid but does not move it
ScaleVector3	similar to the above, except only for Vectors
GetDirection	returns a vector in the specified direction

Classes

HexOrientation: enum	pointy sides or flat sides
HexGridShape: enum	rectangular or compact rectangular
HexDirection: enum	enum for cardinal directions

GFHexGrid variables

GFHexGrid.radius

var **radius** : float

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).

GFHexGrid.hexSideMode

var **hexSideMode** : GFHexGrid.HexOrientation

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

GFHexGrid.gridStyle

var **gridStyle** : GFHexGrid.HexGridShape

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

GFHexGrid.side

var **side** : float

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

GFHexGrid.height

var **radius** : float

Shorthand writing for `Mathf.Sqrt(3.0f) * radius` (read-only).

GFHexGrid.width

var **width** : float

Shorthand writing for `2.0f * radius` (read-only).

GFHexGrid Functions

GFHexGrid.WorldToGrid

function **WorldToGrid** (**worldPoint**: Vector3) : Vector3

Takes in a position in world space and calculates where in the grid that position is. The origin of the grid is the centre of the central face. Rotation is taken into account for this operation. The coordinate system used is the “herringbone pattern”, please refer to the user manual to learn how the herringbone pattern works.

GFHexGrid.GridToWorld

function **GridToWorld** (**gridPoint**: Vector3) : Vector3

The opposite of `WorldToGrid`, this returns the world position of a point in the grid. They cancel each other out.

GFHexGrid.NearestVertexG

function **NearestVertexG** (**fromPoint**: Vector3) : Vector3

Similar to `NearestVertexW`, except you get grid coordinates instead of world coordinates. Uses the herringbone pattern as the coordinate system.

GFHexGrid.NearestFaceG

function **NearestFaceG** (**fromPoint**: Vector3) : Vector3

Similar to `NearestFaceW`, except you get grid coordinates instead of world coordinates. Uses the herringbone pattern as the coordinate system.

GFHexGrid.NearestBoxG

function **NearestBoxG** (**fromPoint**: Vector3) : Vector3

Similar to `NearestBoxW`, except you get grid coordinates instead of world coordinates. Uses the herringbone pattern as the coordinate system.

GFHexGrid.AlignTransform

function **AlignTransform** (**theTransform**: Transform,

rotate: boolean = true,

lockAxis: GFBoolVector3 = new GFBoolVector3) : void

Places on abject onto the grid by positioning its pivot point on the centre of the nearest face. Please refer to the user manual for more information. Setting `doRotate` makes the object take on the grid’s rotation. The parameter `lockAxis` makes the function not touch the corresponding coordinate.

GFHexGrid.AlignVector3

function **AlignVector3** (**position**: Vector3,

scale: Vector3 = Vector3.one,

lockAxis: GFBoolVector3 = new GFBoolVector3) : Vector3

Works similar to `AlignTransform` but instead aligns a point to the grid. The `lockAxis` parameter works just like above.

GFHexGrid.ScaleTransform

function **ScaleTransform** (**theTransform**: Transform,
lockAxis: GFBoolVector3 = new GFBoolVector3) : void

Scales an object's transform to the nearest multiple of the grid's `radius` and `depth`, but does not change its position. The parameter `lockAxis` makes the function not touch the corresponding coordinate.

GFHexGrid.ScaleVector3

function **ScaleVector3** (**scale**: Vector3,
lockAxis: GFBoolVector3 = new GFBoolVector3) : Vector3

Like the above, but takes a Vector3 instead and returns the scaled vector. The `lockAxis` parameter works just like above.

GFHexGrid.GetDirection

function **GetDirection** (**dir**: GFHexGrid.HexDirection,
mode: HexOrientation = hexSideMode) : Vector3

Returns a Vector in the specified cardian direction. The result is in world coordinates, dependent on the specified orientation and by default it uses the grid's own orientation. This function is a handy way of moving from hex to hex without having to convert between grid- and world coordiantes.

GFHexGrid classes

GFHexGrid.HexOrientation

enum **HexOrientation** { PointySides, FlatSides }

Information whether the grid has pointy sides or flat sides

GFHexGrid.HexGridShape

enum **HexGridShape** { Rectangle, CompactRectangle }

The shape of the drawn hex grid. `Rectangle` draws the hex grid as a rectangle where every second column (relative to the centre of the grid) is pulled upwards. `CompactRectangle` is similar, except that the upwards shifted columns have their topmost hexagon cut off.

GFHexGrid.HexDirection

enum **HexGridShape** { N, NE, E, SE, S, SW, W, NW }

A way of specifying the cardinal direction. What exactly it does depends on the function it is used in.

GFPolarGrid

Inherits from GFGrid

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 basd 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 not 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.

Variables

radius	the radius of the inner-most circle of the grid
sectors	the amount of sectors per circle
smoothness	divides the sectors to create a smoother look
angle	the angle of a sector in radians (read only)
angleDeg	the angle of a sector in degrees (read only)

Functions

WorldToGrid	converts world coordinates to grid coordinates
GridToWorld	converts grid coordinates to world coordinates
WorldToPolar	converts world coordinates to polar coordinates
PolarToWorld	converts polar coordinates to world coordinates
GridToPolar	converts grid coordinates to polar coordinates
PolarToGrid	converts polar coordinates to grid coordinates
Angle2Sector	convert an angle coordinate to a sector coordiante
Sector2Angle	convert a sector coordinate to an angle coordiante
Angle2Rotation	returns the Quaternion for rotationg around the origin
Sector2Rotation	returns the Quaternion for a sector around the origin
World2Rotation	returns the Quaternion for a point around the origin
World2Angle	returns the angle of a world point in the grid

World2Sector	returns the sector of a world point in the grid
World2Radius	returns the distance of a world point from the origin
NearestVertexG	returns the grid position of the nearest vertex
NearestFaceG	returns the grid position of the nearest face
NearestBoxG	returns the grid position of the nearest box
NearestVertexP	returns the polar position of the nearest vertex
NearestFaceP	returns the polar position of the nearest face
NearestBoxP	returns the polar position of the nearest box
AlignTransform	fits a Transform inside the grid, but does not scale it
AlignVector3	similar to the above, except only for Vectors
ScaleTransform	scales a Transform to fit the grid but does not move it
ScaleVector3	similar to the above, except only for Vectors

GFPolarGrid variables

GFPolarGrid.radius

var **radius** : float

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

GFPolarGrid.sectors

var **sectors** : int

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

GFPolarGrid.smoothness

var **smoothness** : int

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

GFPolarGrid.angle

var **angle** : float

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
```

GFPolarGrid.angleDeg

var **angleDeg** : float

The same as above except in degrees, it's a shorthand writing for

```
260.0f / sectors
```

GFPolarGrid functions

GFPolarGrid.WorldToGrid

function **WorldToGrid** (**world**: Vector3) : Vector3

Takes in a position in world space and calculates where in the grid that position is. The origin of the grid is the centre of the grid. Rotation is taken into account for this operation. The first coordinate is hom many multiples of `radius` we are away from the origin, the second coordinate is the current sector (bounded from above by the value of `sectors`) plus any fraction. the third coordinate is the height of the cylindrical coordinates.

GFPolarGrid.GridToWorld

function **GridToWorld** (**gridPoint**: Vector3) : Vector3

The opposite of the above

GFPolarGrid.WorldToPolar

function **WorldToPolar** (**worldPoint**: Vector3) : Vector3

Converts from world coordinates to cylindrical coordinates.

GFPolarGrid.PolarToWorld

function **PolarToWorld** (**polarPoint**: Vector3) : Vector3

The opposite of the above.

GFPolarGrid.GridToPolar

function **GridToPolar** (**gridPoint**: Vector3) : Vector3

Converts from grid coordinates to cylindrical coordinates.

GFPolarGrid.PolarToGrid

function **PolarToGrid** (**polarPoint**: Vector3) : Vector3

Converts from cylindrical coordinates to grid coordinates, the opposite of the above.

GFPolarGrid.Angle2Sector

function **Angle2Sector** (**angle**: float,
    mode: GFAngleMode = GFAngleMode.radians) : float

Converts an angle in the grid to a sector in the grid. For example in a 5-sector grid the ange 72 becomes 1.

GFPolarGrid.Sector2Angle

function **Sector2Angle** (**sector**: float,
    mode: GFAngleMode = GFAngleMode.radians) : float

The inverse of the above.

GFPolarGrid.Angle2Rotation

function **Angle2Rotation** (**angle**: float,
mode: GFAngleMode = GFAngleMode.radians) : Quaternion

The resolt is the rotation if you were to rotate an object around the origin of the grid, assuming it had standard rotation when the angle is 0.

GFPolarGrid.Sector2Rotation

function **Sector2Rotation** (**sector**: float) : Quaternion

Same as above, except with sectors.

GFPolarGrid.World2Rotation

function **World2Rotation** (**world**: Vector3) : Quaternion

Takes world coordinates and then returns a rotation based on how much the point is rotated around the grid's origin.

GFPolarGrid.World2Angle

function **World2Angle** (**world**: Vector3,
mode: GFAngleMode = GFAngleMode.radians) : float

Same as above, except it retuns only an angle.

GFPolarGrid.World2Sector

function **World2Sector** (**world**: Vector3) : float

Same as above, except with sectors.

GFPolarGrid.World2Radius

function **World2Radius** (**world**: Vector3) : float

Returns the radial coordinate of the point's polar coordinates.

GFPolarGrid.NearestVertexG

function **Func** (**world**: Vector3) : Vector3

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

GFPolarGrid.NearestFaceG

function **Func** (**world**: Vector3, **thePlane**: GFGrid.GridPlane) : Vector3

function **Func** (**world**: Vector3) : Vector3

Returns the grid coordinates of the nearest grid face from a given point in world space. The coordinates represent the centre of the face. Unlike rectangular grids you don't need to specify a plane, the grid's own plane is used.

GFPolarGrid.NearestBoxG

function **Func** (**world**: Vector3) : Vector3

Returns the grid coordinates of the nearest grid box from a given point in world space. The coordinates represent the centre of the box.

GFPolarGrid.NearestVertexP

function **Func** (**world**: Vector3) : Vector3

Returns the polar coordinates of the nearest grid vertex from a given point in world space.

GFPolarGrid.NearestFaceP

function **Func** (**world**: Vector3, **thePlane**: GFGrid.GridPlane) : Vector3

function **Func** (**world**: Vector3) : Vector3

Returns the polar coordinates of the nearest grid face from a given point in world space. The coordinates represent the centre of the face. Unlike rectangular grids you don't need to specify a plane, the grid's own plane is used.

GFPolarGrid.NearestBoxP

function **Func** (**world**: Vector3) : Vector3

Returns the polar coordinates of the nearest grid box from a given point in world space. The coordinates represent the centre of the box.

GFPolarGrid.AlignTransform

function **Func** (**theTransform**: Transform,

rotate: boolean = true,

lockAxis: GFBoolVector3 = GFBoolVector3.False) : void

Places on abject onto the grid by positioning its pivot point on the centre of the nearest face, the transform's height is ignored. Setting `doRotate` makes the object take on the grid's rotation. The parameter `lockAxis` makes the function not touch the corresponding coordinate.

GFPolarGrid.AlignVector3

function **Func** (**position**: Vector3,

scale: Vector3 = Vector3.one,

lockAxis: GFBoolVector3 = GFBoolVector3.False) : Vector3

Works similar to `AlignTransform` but instead aligns a point to the grid. The `lockAxis` parameter works just like above.

GFPolarGrid.ScaleTransform

function **Func** (**theTransform**: Transform,

lockAxis: GFBoolVector3 = GFBoolVector3.False) : void

Scales an object's transform to the nearest multiple of the grid's `radius` and `depth`, but does not change its position. The parameter `lockAxis` makes the function not touch the corresponding coordinate.

GFPolarGrid.ScaleVector3

function **Func** (**scale**: Vector3,

lockAxis: GFBoolVector3 = GFBoolVector3.False) : Vector3

Like the above, but takes a Vector3 instead and returns the scaled vector. The `lockAxis` parameter works just like above.

GFColorVector3

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.

Variables

x	X component of the colour vector
y	Y component of the colour vector
z	Z component of the colour vector
this[int index]	Access the X, Y or Z components using [0], [1], [2] respectively

Constructors

GFColorVector3	Creates a new colour vector with given X, Y and Z components

Class Variables

RGB	same as `GFColorVector3()`
CMY	same as `GFColorVector3 (Color (0, 1, 1, 0.5), Color (1, 0, 1, 0.5), Color (1, 1, 0, 0.5))`
BGW	same as `GFColorVector3 (Color (0, 0, 0, 0.5), Color (0.5, 0.5, 0.5, 0.5), Color(1, 1, 1, 0.5))`

GFColorVector3 Variables

GFColorVector3.x

var **x** : Color

X component of the colour vector.

GFColorVector3.y

var **y** : Color

Y component of the colour vector.

GFColorVector3.z

var **z** : Color

Z component of the colour vector.

GFColorVector3.this [int index]

var **this[index** : int**]** : Color

Access the x, y, z components using [0], [1], [2] respectively. Example:

```
var c : GFColorVector3;
c[1] = Color.green; // the same as c.y = Color.green
```

GFColorVector3 Constructors

GFColorVector3.GFColorVector3

static function **GFColorVector3(x**: Color**, y**: Color**, z**: Color**)** : GFColorVector3

Creates a new colour vector with given x, y, z components.

static function **GFColorVector3(color**: Color**)** : GFColorVector3

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

static function **GFColorVector3()** : GFColorVector3

Creates a new standard RGB colour vector where all three colours have their alpha set to 0.5

GFColorVector3 Class Variables

GFColorVector3.RGB

static var **RGB**: GFColorVector3

Shorthand writing for `GFColorVector3()`

GFColorVector3.CMY

static var **CMY**: GFColorVector3

Shorthand writing for `GFColorVector3(Color(0,1,1,0.5), Color(1,0,1,0.5), Color(1,1,0,0.5))`

GFColorVector3.BGW

static var **BGW**: GFColorVector3

Shorthand writing for `GFColorVector3(Color(0,0,0,0.5), Color(0.5,0.5,0.5,0.5), Color(1,1,1,0.5))`

GFBoolVector3

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.

Variables

x	X component of the bool vector
y	Y component of the bool vector
z	Z component of the bool vector
this[int index]	Access the X, Y or Z components using [0], [1], [2] respectively

Constructors

GFBoolVector3	Creates a new bool vector with given X, Y and Z components

Class Variables

True	same as `GFBoolVector3 (true, true, true)`
False	same as `GFBoolVector3 (false, false, false)`

GFBoolVector3 Variables

GFBoolVector3.x

var **x** : boolean

X component of the bool vector.

GFBoolVector3.y

var **y** : boolean

Y component of the bool vector.

GFBoolVector3.z

var **z** : boolean

Z component of the bool vector.

GFBoolVector3.this [int index]

var **this[index** : int**]** : boolean

Access the x, y, z components using [0], [1], [2] respectively. Example:

```
var b : GFBoolVector3;
b[1] = true; // the same as b.y = true
```

GFBoolVector3 Constructors

GFBoolVector3.GFBoolVector3

static function **GFBoolVector3(x**: boolean**, y**: boolean**, z**: boolean**)** : GFBoolVector3

Creates a new bool vector with given x, y, z components.

static function**GFBoolVector3(condition**: boolean**)** : GFBoolVector3

Creates a new bool vector with all components set to *condition*.

static function **GFBoolVector3()** : GFBoolVector3

Creates a new standard all false vector.

GFBoolVector3 Class Variables

GFBoolVector3.True

static var **True**: GFBoolVector3

Shorthand writing for `GFBoolVector3(true)`

GFBoolVector3.False

static var **False**: GFBoolVector3

Shorthand writing for `GFBoolVector3(false)`

GFAngleMode

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.

GFAngleMode

enum **GFAngleMode** {radians, degrees}