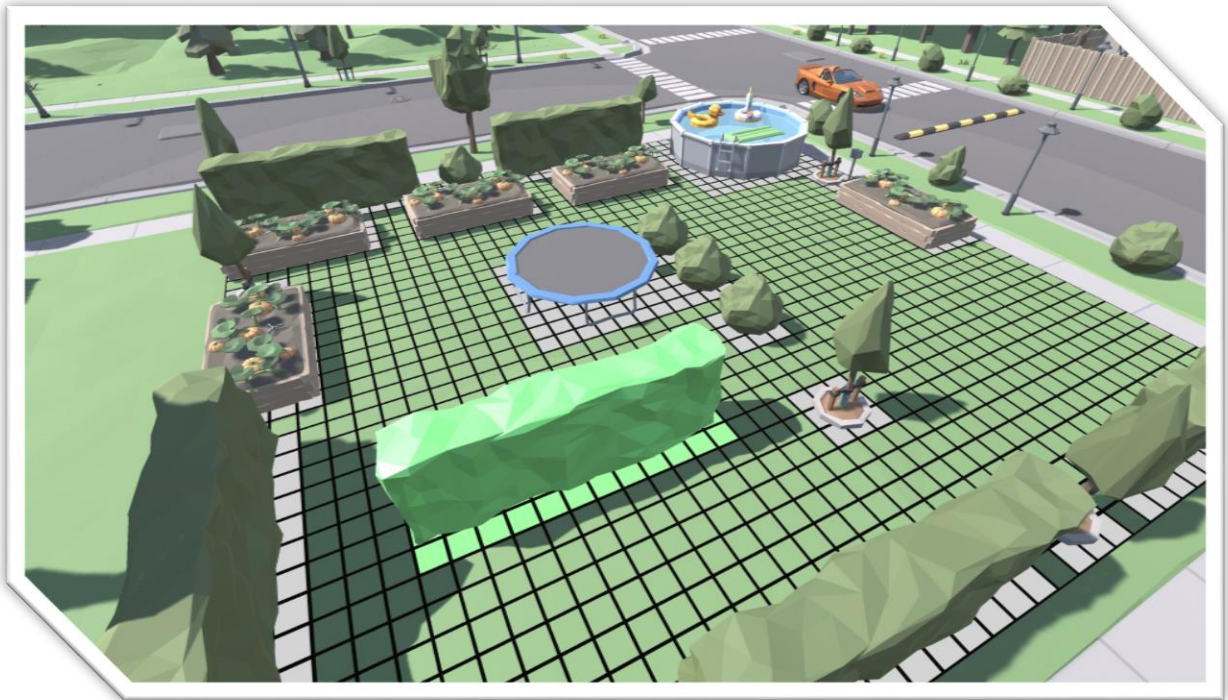


Hypertonic Games- Grid Placement System



Join the Discord server: <https://discord.gg/58Wx3yCAYS>

View the Github: [Github](#)

Table Of Contents

Table Of Contents	2
Overview	4
Compatibility.....	4
Render Pipelines	4
URP.....	4
HDRP	4
Getting Started	5
Installation	5
Configuring Objects for the Grid.....	9
How to override the default sizing	9
How to set the grid height of an object.....	10
Grid Settings.....	14
Grid Rotation.....	14
Parent To Grid.....	14
Cell Image.....	14
Occupied Cell Image	14
Cell Colour Available	14
Cell Colour Not Available	14
Cell Colour Placed	14
Hide Placement Cells If Outside of Grid.....	14
Show Occupied Cells	15
Grid Canvas Camera Name	15
Change Object Materials	15
Object Placeable Material.....	15
Object Unplaceable Material.....	15
Default Alignment.....	15
Initial Placement Cell Index	15
Platform Grid Input Definition Mappings	15
Prevent Input Through UI	16
Width to Height Ratio	16
Grid Size	17

Horizontal Cell Count.....	17
Vertical Vell Count	17
Grid Position	18
Core Features.....	19
Placement mode.....	19
Move	19
Rotation	19
Change Alignment.....	20
Cancel Placement.....	21
Delete Object	21
Modifying the placement of an object.	22
Remove object	22
Clear Grid	22
Saving and Loading	23
Add Programmatically	24
Paint Mode	25
Multiple Grids	27
Custom Input Definitions.....	28
Change Input Settings.....	31
Custom Position Offset	32
Custom Validation.....	33
Custom Placement.....	35
Grid Settings.....	35
Placement Settings	35
Move the Grid at Runtime	36
Rotate the Grid at Runtime	37
Object Grid Position.....	38
Supports 100 million Grid Cells.....	39
Support	40

Overview

The grid placement system was developed to allow developers to quickly and easily drop a grid placement system into their Unity project. The asset is specifically targeted for allowing players to place and modify objects onto a grid at runtime.

Compatibility

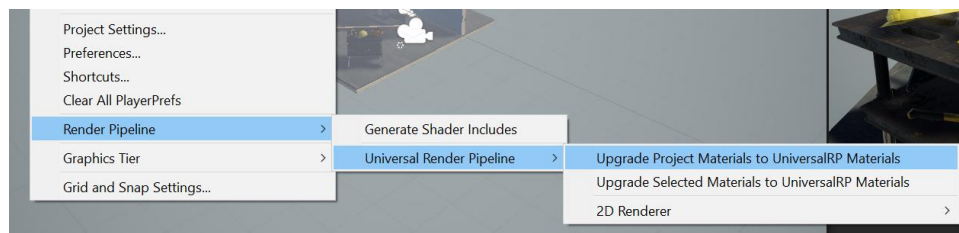
The supported unity version for this asset is 2019 LTS or later.

Render Pipelines

The only materials in the asset are purely for the sample scenes. These materials use the built-in render pipeline. If your project is using the URP or HDRP all you can simply upgrade the materials. Please see the relevant section below for instructions on how to upgrade the materials.

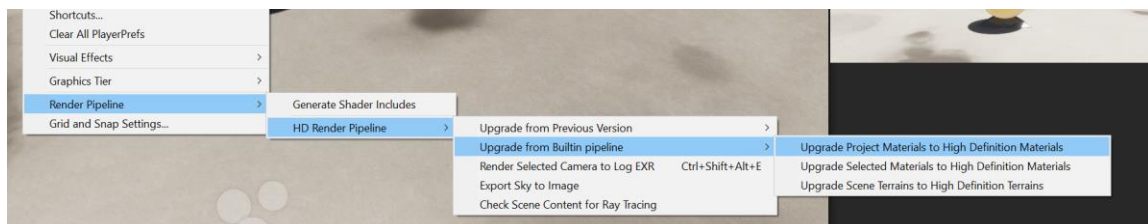
URP

If your project is using Unity's URP and you import the Grid Placement System samples into your project, you can upgrade the materials by navigating to the Unity's edit context menu and select: Render Pipeline > Universal Render Pipeline > Upgrade Project Materials to UniversalRP Materials



HDRP

If your project is using Unity's HDRP and you import the Grid Placement System samples into your project, you can upgrade the materials by navigating to the Unity's edit context menu and select: Render Pipeline > HD Render Pipeline > Upgrade from Builtin pipeline > Upgrade Project Materials to High Definition Materials.



Getting Started

Installation

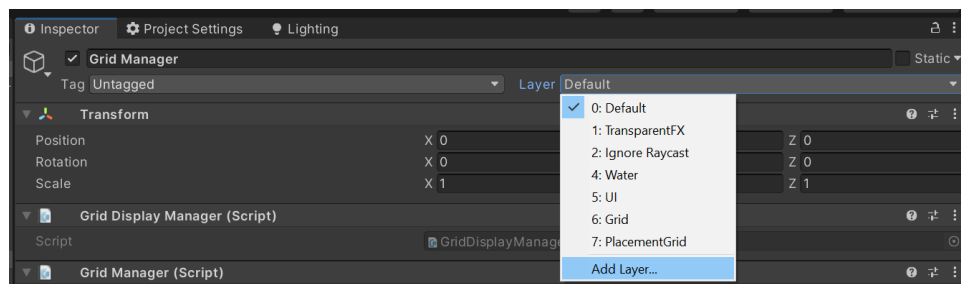
1. Open your project in Unity.
2. Import the .unitypackage file into the project.

Important

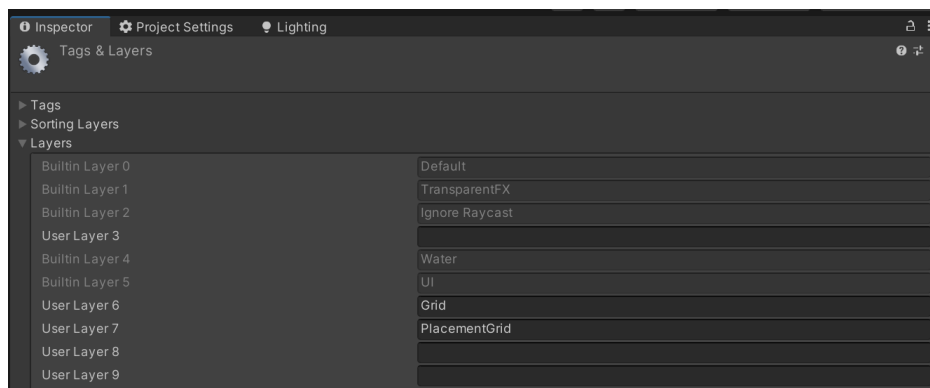
If using **Unity 2019** the creation of the required Unity layers for the grid placement system does not happen automatically. So, you will need to create the layers manually in the Unity Editor. The names of the layers are:

- Grid
- PlacementGrid

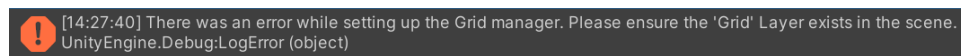
To add new layers in your project, select the Layer dropdown within the inspector tab. Select Add Layer...



Then in the next empty space after the built-in layers input the layer names: **Grid** and **PlacementGrid**



If you see the below error. The layers need to be created.

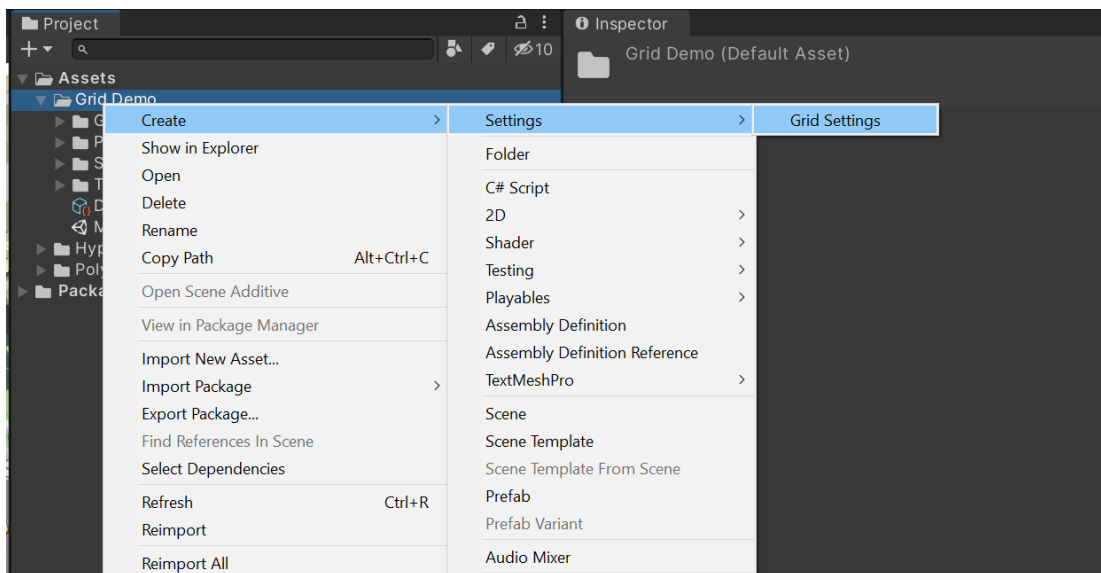


Quick Start

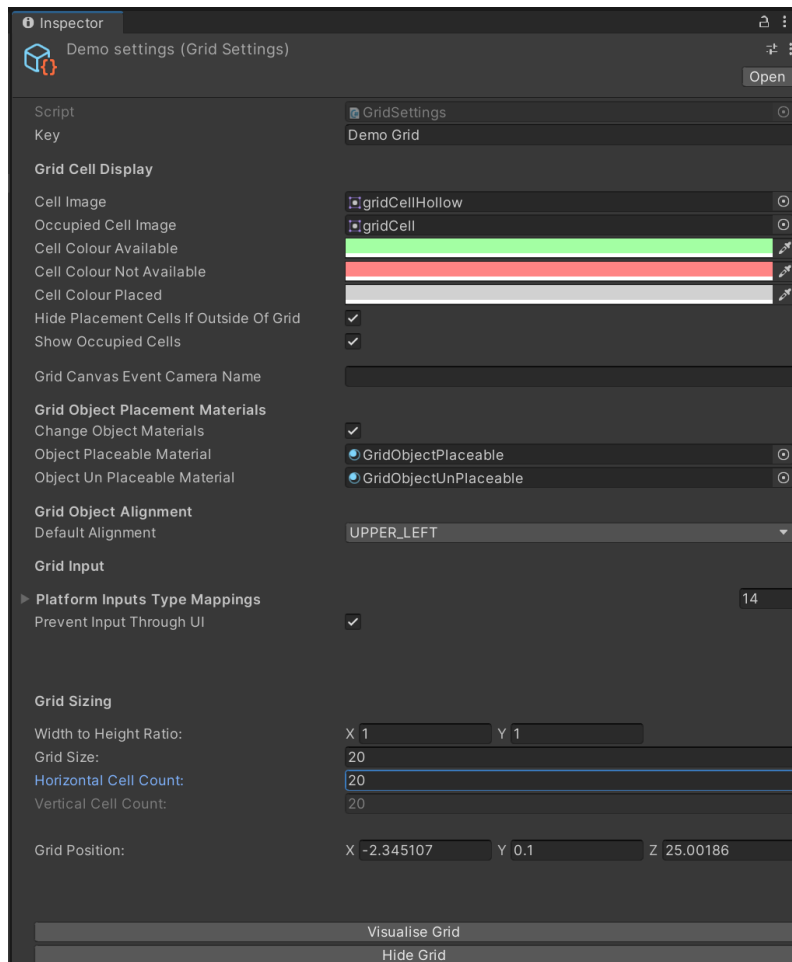
The Basic Grid Demo example scene will best demonstrate how to interface and interact with the Grid System.

To implement the grid placement system into a scene follow these steps:

1. Create a new GridSettings asset in your Project folder. You can add a settings object by right clicking and selecting: Create > Settings > Grid Settings



2. Populate the settings object. (See the Grid Settings section for more details).



3. Create a Grid Manager. You can either drop the GridManager component onto a game object or create the component via code.
4. In another script you'll need a reference to the GridManager component and the settings object. Once you have those 2 references call the Setup function on the GridManager and pass in the Grid Settings.

```

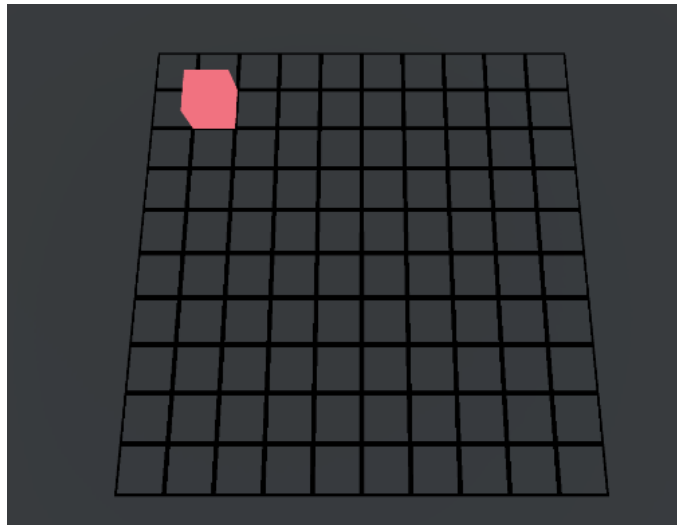
1  private void CreateGridManager()
2  {
3      GameObject gridManagerObject = new GameObject("Grid Manager");
4      GridManager gridManager = gridManagerObject.AddComponent<GridManager>();
5      gridManager.Setup(_gridSettings);
6  }

```

5. Pass an object to the Grid Manager to place. If you don't want to store a reference to the GridManager in the class you can use the **GridManagerAccessor** to access it instead (See the Grid Manager Accessor section for more details).

```
1  GameObject objectToPlace = Instantiate(_gridObjectToSpawnPrefab, GridManagerAccessor.GridManager.GetGridPosition(), new Quaternion());  
2  GridManagerAccessor.GridManager.EnterPlacementMode(objectToPlace);
```

6. Move the object into the desired position on the grid. Once in placement mode you can manipulate the object in several ways, (see the Placement Mode section for more details).



7. Confirm the placement. The confirm placement function will return a bool to inform you if the object was able to be placed or not. If it's an invalid placement it may be due to the fact the object is overlapping another object placed on the grid, or it may be partly off the grid.

```
1  bool placed = GridManagerAccessor.GridManager.ConfirmPlacement();
```


Configuring Objects for the Grid

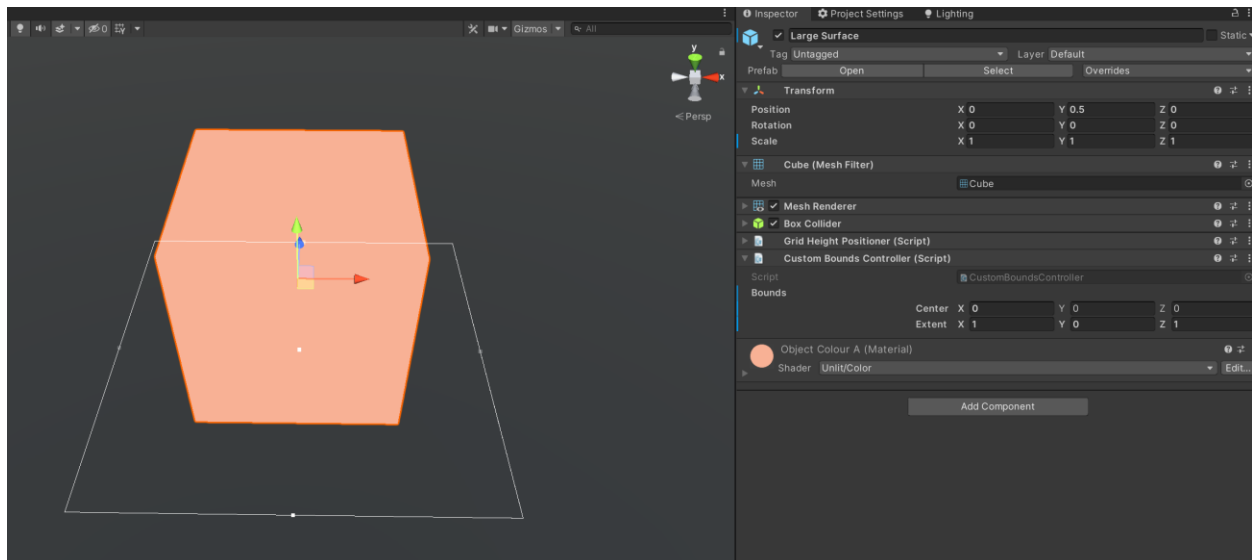
The grid system is designed to not require objects to have any configuration settings added to them before being passed to the grid system. When an object is passed to the grid system it'll calculate the size of the object and how many grid cells that object occupies based on the colliders.

The grid placement system will recursively search each child object of the object and calculate the size required based on each collider on the parent and child objects.

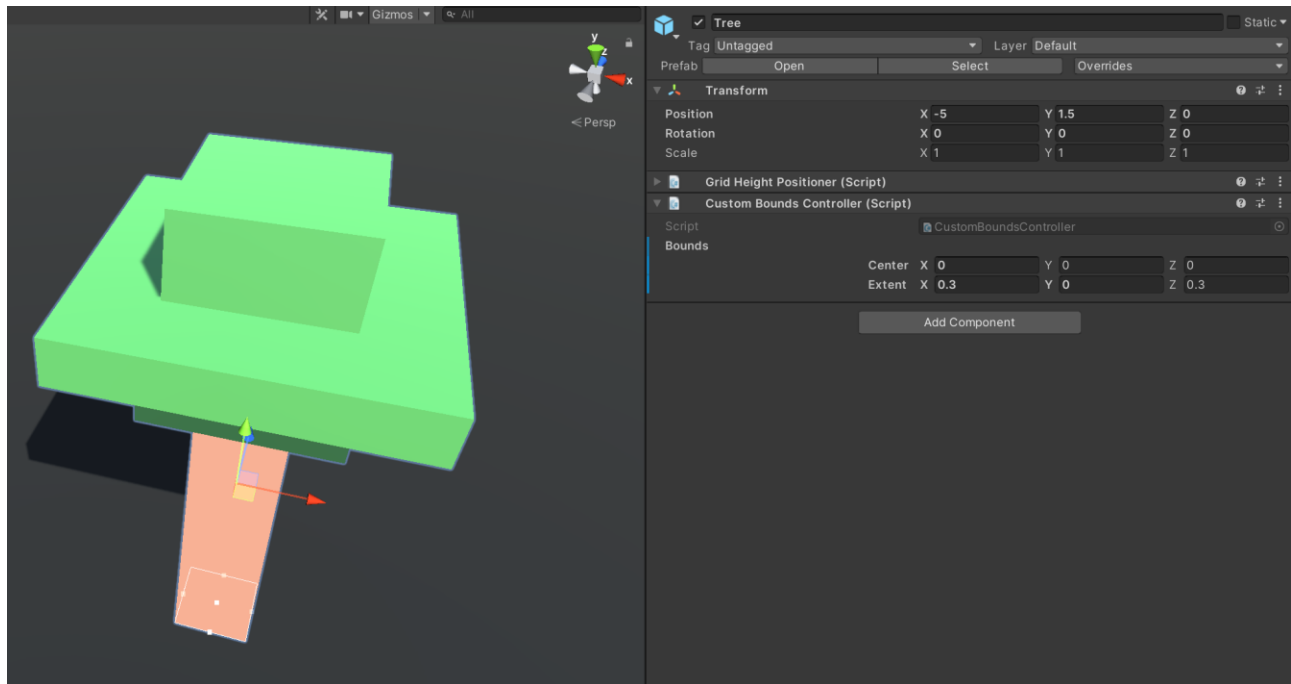
How to override the default sizing

You can override the automatic size calculated for an object by specifying it yourself. To do this you'll need to add a **Custom Bounds Controller** component to the object. The component will expose a Bounds property. The Bounds property will control the size the object occupies on the grid.

You can adjust the bounds from the inspector or by using the bounds handles in the scene view. Please note the Y properties of the bounds values will be ignored.



A great use case for using custom bounds on an object is a tree. The default sizing calculations would find the colliders on a tree model and include the size of the branches for the space it would occupy on the grid. However, the trunk of the tree is much narrower and takes up less space. Therefore you can add the Custom Bounds Controller to a tree object and set the bounds to be the size of the trunk, therefore allowing other objects to be placed snug to the trunk and underneath the branches. The risk of setting a narrower bounds size is that you could face overlapping meshes.



As you can see in the image above a **Custom Bounds Controller** has been added to this tree object. At the base of the trunk you can see the white bounds gizmo showing that the bounds are set to the size of the trunk.

Checkout the **Object Sizing Demo** to see examples of this feature.

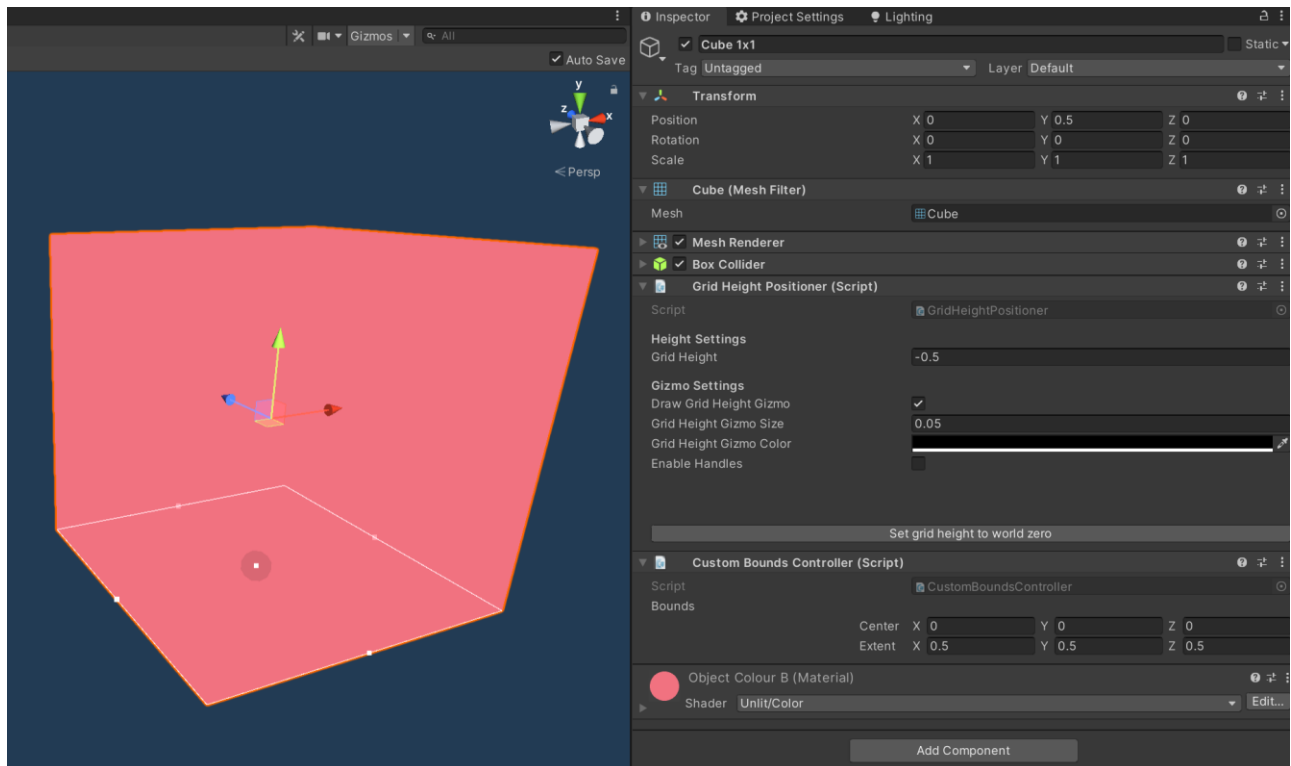
How to set the grid height of an object.

By default when an object is added to the grid its height is set to the pivot of the object. If the pivot of the object is at the base of an object, then it will automatically be aligned to the grid correctly. If, however, the pivot is not at the base you can control which point along the y-axis the object is snapped to the base of the grid. This can be achieved by adding the **Grid Height Positioner** component to the object.

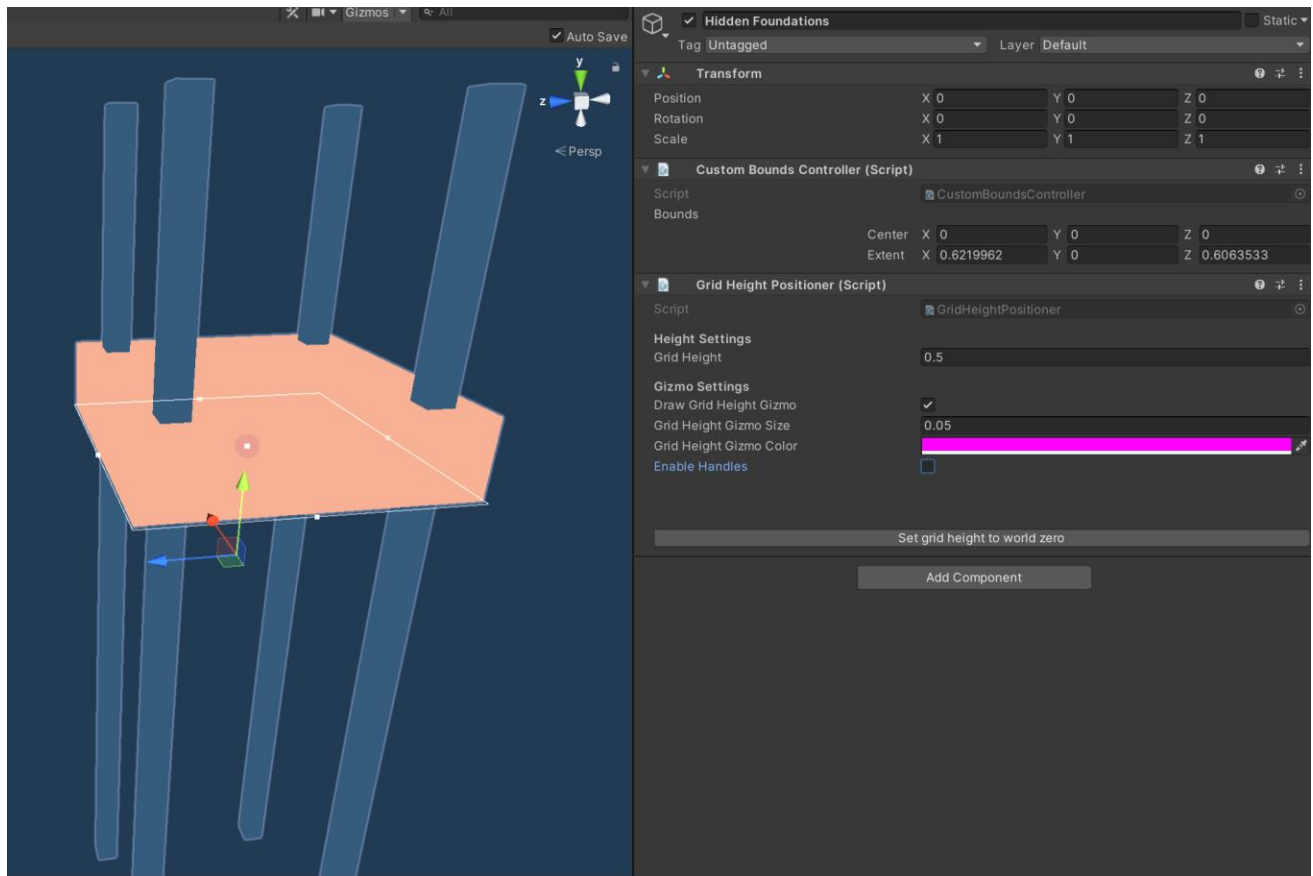
When the Grid Height Positioner component has been added a sphere gizmo will show up on the object to display where the current height is set to. If you also have a custom bounds component attached, you will see the bounds gizmo also aligns with the height set.

For most use cases you will most likely want to set the grid height to the base of the object, that way when the object is added to the grid it'll sit perfectly on top of the grid. You can adjust the grid height property in the inspector, or to make this process quicker and easier use the **Set grid height to world**

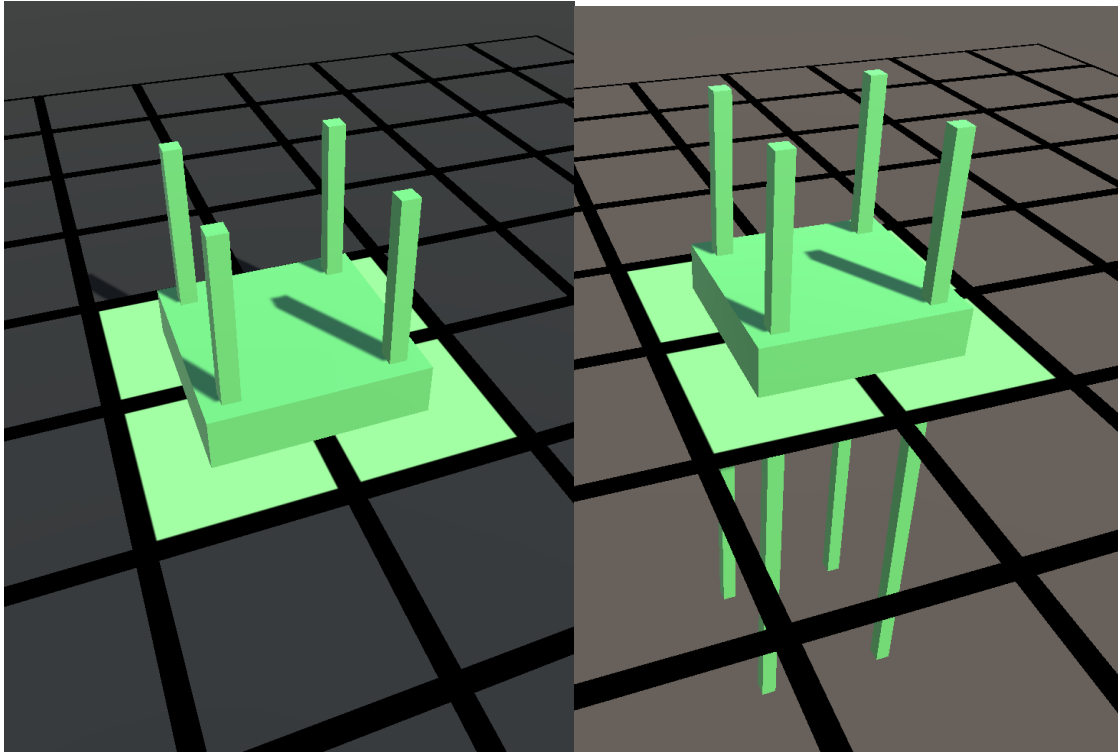
zero button on the component to automatically set the position to the base of the object. (This shortcut will not work if the pivot is not in the center of the object).



There may be a use case where there is an object that has some form of foundations that should be situated below the grid. In this case you can adjust the grid height position to determine how much of the object sits above the grid.



In the above example there is an object that is comprised of a tan cuboid with 4 blue elongated cuboids intersecting with the tan cuboid. If for example we only wanted the object to be visible from the tan cuboid upwards, we can set the grid height position to be at the bottom of the tan cuboid.



This would result in the object being snapped to the grid at the base of the tan cuboid and the foundations hidden below. The image on the right has a transparent grid to allow us to see the hidden foundations that are not visible in the left image.

Grid Settings

The grid is generated using a settings object that allows the user to define many aspects of the grid and how the placement of an object behaves.

Grid Rotation

Sets the initial rotation for the grid. If you need to update the rotation at runtime see the Update Grid Rotation.

Parent To Grid

When enabled objects placed on the grid will be parented to a game object called “Grid Object Container” which mirrors the position and rotation of the grid.

Cell Image

The grid settings take in a Unity sprite to use as the visualisation of the grid. The grid is generated by repeating this spite to create the grid cells. The sample scenes contain examples of a simple square and circle grid cell.

Occupied Cell Image

This sprite is used to display the currently unavailable grid cells due to an object occupying those cells. It's also used to show the grid cells that the item being placed will occupy if placed at that position.

Cell Colour Available

The colour of the cells when the item is being placed is valid.

Cell Colour Not Available

The colour of the cells when the item is being placed is not valid.

Cell Colour Placed

The colour of the cells of occupied cells.

Hide Placement Cells If Outside of Grid

If part of the object being placed is outside the grid hide or show those cells.

Show Occupied Cells

Defines whether the cells that are occupied are displayed differently to the normal grid cells.

Grid Canvas Camera Name

The Grid Manager generates a UI Canvas to display the grid image. This Unity canvas can have an event camera assigned to it. To do this, provide the name of the game object that the desired camera is attached to. If left blank, Camera.Main is used.

Change Object Materials

When an object is being placed the materials of the object are swapped to either the **Object Placeable Material** or the **Object Unplaceable Material** depending on if the placement is valid. To prevent the object's material being changed while in placement mode, turn this setting off.

Object Placeable Material

The material applied to an object when the placement of the object is valid.

Object Unplaceable Material

The material applied to an object when the placement of the object is invalid.

Default Alignment

Defines the alignment of the grid object when being placed onto the grid. The alignment of the object can be changed when being placed. (See the alignment section in Placement Mode for more details).

Initial Placement Cell Index

Defines the grid cell to place the object at when it's passed to the grid. This can be overridden by the placement settings parameter. See the **Custom Placement** settings for more details.

Platform Grid Input Definition Mappings

Allows you to set an **Input Definition** for each runtime platform. An input definition asset defines what input the grid manager uses to position the grid object on the grid. It also controls when the grid object should follow the users input.

To find out more and create your own input definitions, see the **Custom Input Definitions** section.

Prevent Input Through UI

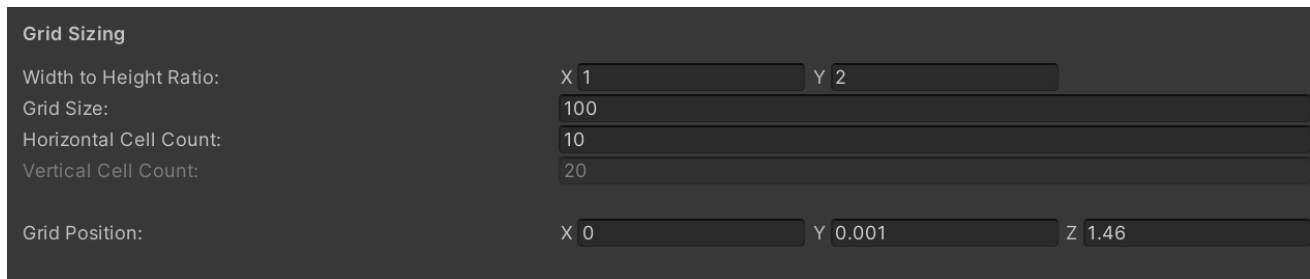
When this setting is on it'll prevent the raycast that detects interactions with the grid. This may be needed when you have a UI menu on top of the grid and don't want to be moving the selected grid object behind the menu.

Width to Height Ratio

This defines the ratio of the width to the ratio of the height of the grid. For any square grid this can be left as a 1:1 ratio. You can adjust the width to height ratio to create an elongated shaped grid. Because adjusting the ratio of width to height would stretch the grid cells in either the x or y dimension the vertical cell count must be updated to ensure each grid cell is still a square within a non-square grid.

To help with this, the vertical cell count is automatically updated based on the new ratio of width to height and the horizontal cell count. Although the vertical cell count is calculated for you, you'll still need to ensure that the dimensions are valid.

For example, look at the image below.



Grid Sizing			
Width to Height Ratio:	X 1	Y 2	
Grid Size:	100		
Horizontal Cell Count:	10		
Vertical Cell Count:	20		
Grid Position:	X 0	Y 0.001	Z 1.46

You can see the width to height ratio is 1:2. This means that because the grid size is 100, the width of the grid will be 100 and the height will be 200. As there is a horizontal cell count of 10, the vertical cell count is adjusted to 20 (matching the width to height ratio). If the vertical cell count remained at 10 then each grid cell would be twice tall as it is wide.

Important

When adjusting the ratio or horizontal cell count you must ensure the values are valid. As the vertical cell count is adjusted automatically it could result in a non-integer number. When this happens, an error will be shown in the inspector to inform you the values are invalid. If the error is ignored in the settings object, the grid manager will throw an error when setting up. See before for an example of invalid values.

Grid Sizing

Width to Height Ratio:

X 2Y 1

Grid Size:

10

Horizontal Cell Count:

3

Vertical Cell Count:

1.5

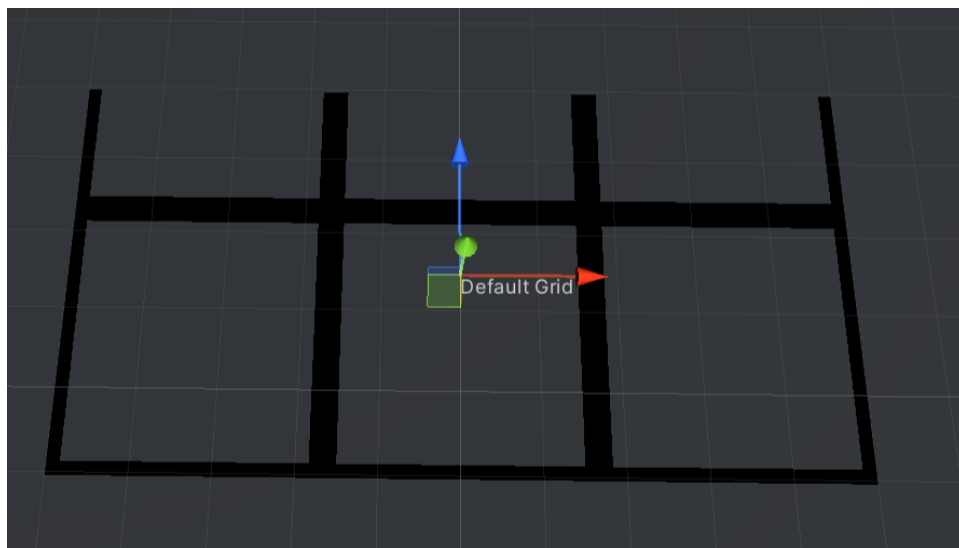
Grid Position:

X 0Y 0.001Z 1.46

!

Invalid Grid Cell Count. Please adjust the cell count or the width to height ratio

As you can see above the ratio is 2:1 and there are only 3 horizontal cells. The ratio of 2:1 == 0.5. $3 * 0.5 = 1.5$. As you cannot have half a cell the grid is invalid.



Grid Size

This determines the width and height of the grid in Unity world space. When the width to height ratio is adjusted the grid size will control the width and the height will be calculated based on the ratio.

Horizontal Cell Count

This determines how many cells the grid will contain in its x axis.

Vertical Vell Count

This property cannot be changed manually. Instead this value is driven by the width to height ratio setting in combination with the horizontal cell count. Please see the Width to Height property listed above for more details on this.

Grid Position

This sets the position of the grid in Unity world space. When the grid settings are selected you can use the transform handles in the scene view to move the grid into the desired location.

Core Features

Placement mode

Placement mode is where an object is passed to the Grid Manager to place onto the grid. Once in placement mode, a number of options are available.

Call **EnterPlacementMode** to add a new object to the grid. Once called the grid will be shown and the object will follow the mouse around the grid when held down.

A screenshot of a code editor window with a dark background and light blue borders. It contains a single line of C# code: `1 GridManagerAccessor.GridManager.EnterPlacementMode(objectToPlace);`. The code is written in a light blue font, and the line number '1' is in a lighter blue. The window has three colored dots (red, yellow, green) in the top left corner, typical of a macOS-style window.

Important

If you want to have UI on top of the grid while the grid is being displayed you'll most likely not want to move the grid object on the grid if interacting with the UI. To block the raycasts from going through the UI when in placement mode see the **Prevent Input Through UI** section in the grid settings above.

Move

You can move the object around the grid by the input definition defined for the runtime platform. Most sample scenes use the mouse position and left mouse click as the way to move the object around the grid. However, you can define your own definitions that allow you to move it however you desire. The sample scenes contain examples of different input definitions such as touch controls for mobile support.

Rotation

When the object is being placed on the grid the user may need to rotate the object to achieve the desired position. The object should be rotated independently using the normal Unity rotation techniques. It is the user's responsibility to handle the rotation so they can apply any animations or extra styling to the object when it rotates. However, once rotated the "HandleGridObjectRotated" function should be called to let the Grid Manager know it needs to recalculate if the new position of the object is valid.

```

1  private void HandleRotateRightPressed()
2  {
3      _selectedGridObject.transform.Rotate(new Vector3(0, 90, 0));
4
5      GridManagerAccessor.GridManager.HandleGridObjectRotated();
6  }

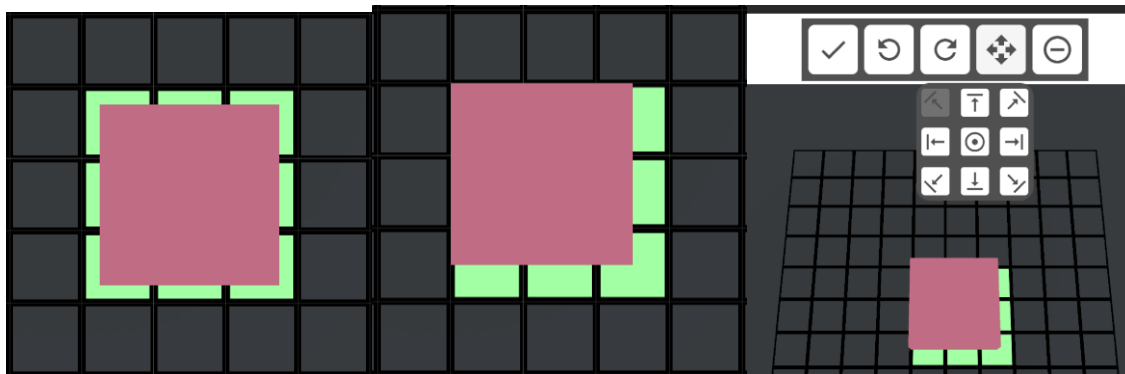
```

Important

For the grid placement system to work correctly with object rotations. The object must be rotated in increments of 90 degrees on the y axis.

Change Alignment

If the object being placed does not fit perfectly within the grid cells it occupies, it will have padding. Changing the alignment of the object will change which edge(s) of the occupied cells the object is snapped to. You can change the alignment of the object so that the object is snug against any of the edges or corners of the cells the object occupies. It can also be aligned to the center of the cells it occupies to create even padding surrounding the object.



The default alignment is set in the grid setting object. However, to change the alignment when in placement mode call the “ChangeAlignment” function on the Grid Manager.

```
1 private void HandleChangeAlignmentPressed(ObjectAlignment objectAlignment)
2 {
3     GridManagerAccessor.GridManager.ChangeAlignment(objectAlignment);
4 }
```

Cancel Placement

If the object is in placement mode and has not already been placed, then you can cancel the current placement. This will destroy the object currently being placed and hide the grid. To cancel the placement of an object call the “CancelPlacement()” function on the Grid Manager.

```
1 private void HandleCancelPlacementPressed()
2 {
3     GridManagerAccessor.GridManager.CancelPlacement();
4 }
```

Delete Object

If you wish to delete an object that has already been placed on the grid you can achieve this by calling the DeleteObject function on the Grid Manager and pass in the object to delete.

```
1 private void HandleDeleteObjectPressed()
2 {
3     GridManagerAccessor.GridManager.DeleteObject(_selectedGridObject);
4     _selectedGridObject = null;
5 }
```

If you wish for the grid to remain showing after an object is deleted, you can pass **false** as a second parameter to the delete function.

```
1 GridManagerAccessor.GridManager.DeleteObject(gameObjectToDelete, false);
```

Modifying the placement of an object.

If an object has already been placed on the grid, you can re-enter placement mode to achieve the desired placement. To do this, call “ModifyPlacementOfGridObject”.

```
1 private void HandleExampleGridObjectSelected(GameObject gridObject)
2 {
3     GridManagerAccessor.GridManager.ModifyPlacementOfGridObject(gridObject);
4 }
```

Remove object

To remove all references to an object and convert the occupied cells back to unoccupied without destroying the grid object you can call “RemoveObjectFromGrid”.

This is useful for scenarios where the user wishes to place an object at a certain grid position that once placed will move on its own. (Essentially using the grid placement system to snap the game object to the desired grid location to spawn an object in a controlled fashion).

```
1 private void HandleRemoveObjectPressed()
2 {
3     GridManagerAccessor.GridManager.RemoveObjectFromGrid(_selectedGridObject);
4     _selectedGridObject = null;
5 }
```

Clear Grid

To delete all the current items in the grid, call the “ClearGrid(bool destroyGridObjects = true)” function on the Grid Manager. To just remove all references of the objects from the grid without deleting the

objects you can pass in false as a parameter.

```
1 private void DeleteAllGridObjects()
2 {
3     GridManagerAccessor.GridManager.ClearGrid();
4 }
```

Saving and Loading

There may be scenarios where the objects in the grid need to be persisted between levels/sessions. The Grid Manager offers functionality to easily meet these requirements. (See the “Grid Save Manager” class in the sample scene for an example of how this may be implemented.

The Grid Manager exposes two functions to achieve the desired functionality.

Accessing Grid Data

The current grid data can be accessed via the “GridData” Property on the Grid Manager.

```
1 private void HandleSaveGridObjectsPressed()
2 {
3     GridData gridData = GridManagerAccessor.GridManager.GridData;
4
5     SaveData saveData = new SaveData();
6
7     for(int i = 0; i < gridData.GridObjectPositionDatas.Count; i++)
8     {
9         GridObjectPositionData gridObjectPositionData = gridData.GridObjectPositionDatas[i];
10
11         GridObjectSaveData gridObjectSaveData = new GridObjectSaveData(
12             gridObjectPositionData.GridObject.name,
13             gridObjectPositionData.GridCellIndex,
14             gridObjectPositionData.ObjectAlignment,
15             gridObjectPositionData.GridObject.transform.rotation);
16
17         saveData.GridObjectSaveDatas.Add(gridObjectSaveData);
18     }
19
20     string saveDataAsJson = JsonUtility.ToJson(saveData);
21
22     PlayerPrefs.SetString(_playPrefsSaveDataKey, saveDataAsJson);
23 }
```

Populating with Grid Data

Once you have retrieved the grid data that has been persisted between sessions you can load it into the grid. To populate the grid with the grid data object pass it to the “PopulateWithGridData” function on. This is an async method and may need to be awaited in certain scenarios.

```

1 private async Task LoadGridData()
2 {
3     if (!PlayerPrefs.HasKey(_playPrefsSaveDataKey))
4     {
5         Debug.LogWarning("There is no save data stored yet. You must save the grid data before being able to load it.");
6         return;
7     }
8
9     string saveDataAsJson = PlayerPrefs.GetString(_playPrefsSaveDataKey);
10
11     SaveData saveData = JsonUtility.FromJson<SaveData>(saveDataAsJson);
12
13     List<GridObjectPositionData> gridObjectPositionDatas = new List<GridObjectPositionData>();
14
15     foreach (GridObjectSaveData gridObjectSaveData in saveData.GridObjectSaveDatas)
16     {
17         GameObject prefab = _gridObjectPrefabs.Where(x => x.name.Equals(gridObjectSaveData.PrefabName))
18             .FirstOrDefault();
19         GameObject gridObject = Instantiate(prefab);
20         gridObject.transform.rotation = gridObjectSaveData.ObjectRotation;
21
22         // Remove the "(Clone)" from instantiated name.
23         gridObject.name = prefab.name;
24
25
26         if (!gridObject.TryGetComponent(out ExampleGridObject exampleGridObjectComponent))
27         {
28             gridObject.AddComponent<ExampleGridObject>();
29         }
30
31         GridObjectPositionData gridObjectPositionData = new GridObjectPositionData(
32             gridObject,
33             gridObjectSaveData.GridCellIndex,
34             gridObjectSaveData.ObjectAlignment);
35
36         gridObjectPositionDatas.Add(gridObjectPositionData);
37     }
38
39     GridData gridData = new GridData(gridObjectPositionDatas);
40
41     await GridManagerAccessor.GridManager.PopulateWithGridData(gridData, true);
42 }

```

There is an optional parameter that provides the option to clear the grid of its current objects prior to populating it with the grid data. It is recommended to clear the grid prior to populating the data as it will mitigate the risk of invalid placements due to overlapping objects. However, there may be use cases where the grid needs to have the grid data combine with the existing grid objects.

Add Programmatically

As well as adding objects to the Grid in bulk with the “PopulateWithGridData” function mentioned above. There is also the option to add a single object to the grid without entering placement mode.

To achieve this call “AddObjectToGrid” on the Grid Manager. You will need to pass in the object to place on the grid, the grid cell index to place the object at, and the desired alignment of the object.


```
1  await GridManagerAccessor.GridManager.AddObjectToGrid(  
2      gridObject,  
3      gridObjectPositionData.GridCellIndex,  
4      gridObjectPositionData.ObjectAlignment);
```

Important

If you need to call `AddObjectToGrid` multiple times in short succession, E.G., in a loop. Each call must be awaited.

Paint Mode

You are able to add objects continuously to the grid. This is sometimes referred to as “paint mode”. When in the paint mode state, the grid will spawn a new grid item as soon as the previous grid item has been placed. This allows users to quickly add lots of the same item quickly.

To use this feature call “`StartPaintMode`” on the grid manager and pass in the object you wish to place continuously.

```
1  public void StartPaintMode()  
2  {  
3      _gridManager.StartPaintMode(_gridObjectPrefab);  
4  }  
5
```

When you no longer wish to be in paint mode call “`EndPaintMode`”.

You can check out the `PaintMode` sample scene. This will demonstrate the paint mode functionality as well as how to quickly delete many objects from the grid.

Grid Manager Accessor

The Grid Manager Accessor provides an effortless way to access the Grid Manager class without the need for storing a reference to it. Call “GridManagerAccessor.GridManager” to access the default Grid Manager.

```
1 GridManagerAccessor.GridManager
```

The Grid Manager class will register itself to the Grid Manager Accessor when it's set up. The key used to register itself is the key in the Grid settings object provided to the Grid Manager when it's set up. If you have multiple Grid Managers, you can access them individually through the Grid Manager Accessor class by calling “GridManagerAccessor.GetGridManagerByKey” and passing in the Grid Manager's Key.

```
1 GridManagerAccessor.GetGridManagerByKey("MY_GRID_KEY");
```

If switching between two Grid Manager's frequently you can set the selected Grid Manager in the class by calling “SetSelectedGridManager”. This means that when you call “GridManagerAccessor.GridManager” the selected Grid Manager will be returned.

```
1 GridManagerAccessor.SetSelectedGridManager("MY_PRIMARY_GRID_KEY");
```

This class has been created to provide an easy way to reference the Grid Manager. However, there is no reason you cannot reference the Grid Manager directly in a script if you choose.

Important

If you have multiple Grid Managers, you must ensure that the key set in the grid settings object is unique.

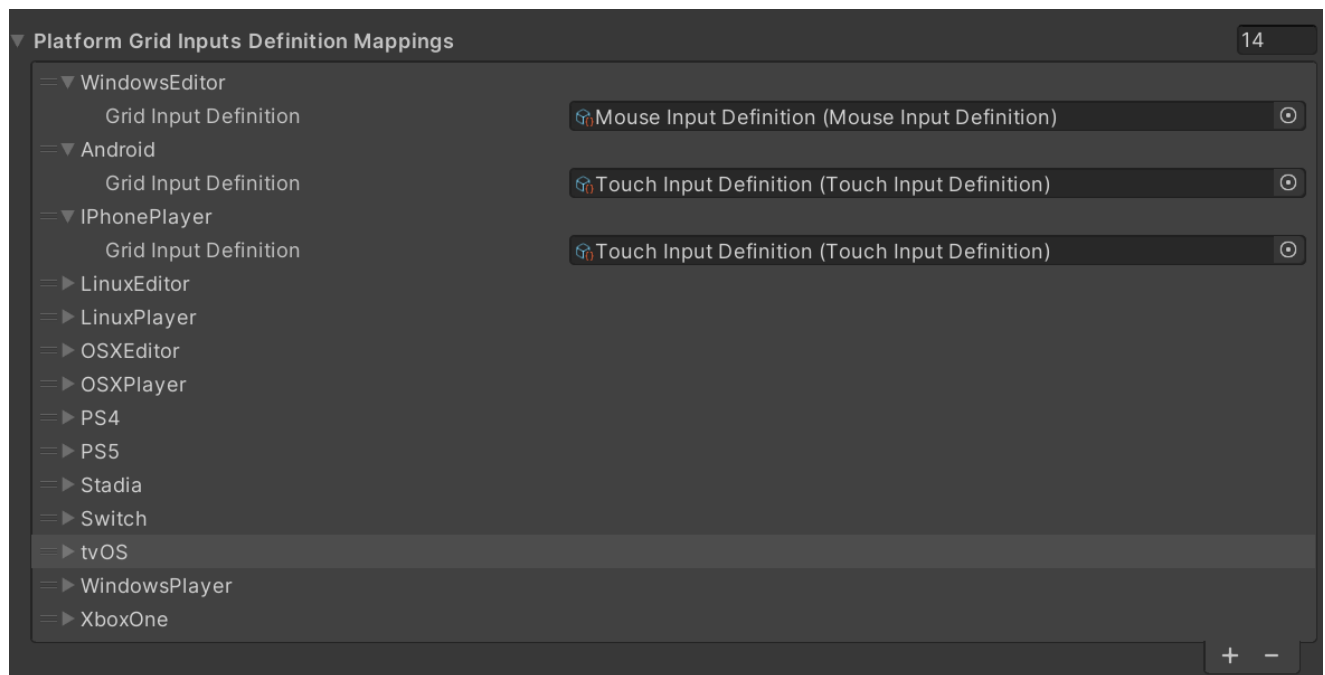
Multiple Grids

This asset supports multiple Grid Managers. If you use the Grid Manager Accessor class to obtain a reference for the Grid Manager, you can specify which Grid Manager you wish to access. See the “Grid Manager Accessor” section above to learn how you can access different Grid Managers.

Mobile Support

The Grid Placement System supports mobile touch controls on the grid. The grid placement system comes with a mouse and touch input definition asset. The touch input definition assets are set as the default input definition for mobile runtime platforms such as Android and iPhonePlayer.

If you desire more control over how users interact with the grid you can create your own input definitions. See Custom Input Definitions below for more information.



Custom Input Definitions

An input definition asset defines what input the grid manager uses to position the grid object on the grid. It also controls when the grid object should follow the user's input. The grid placement system comes bundled with 2 pre-defined input definitions. One is used for mouse input and the other is used for touch input.

You can easily create your own input definition assets to customise how the user interacts with the grid. These assets are scriptable objects which means they can be easily referenced by the grid settings.

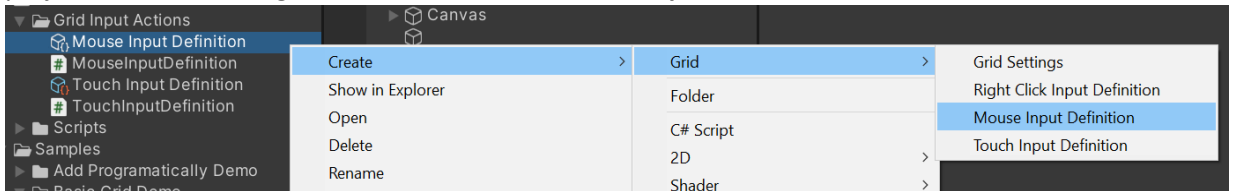
1. Create a new script for your input definition.
2. Inherit from GridInputDefinition.
3. Override the GetInput and ShouldInteract functions.

4. Add the CreateAssetMenu attribute to enable creation of the asset in the editor.



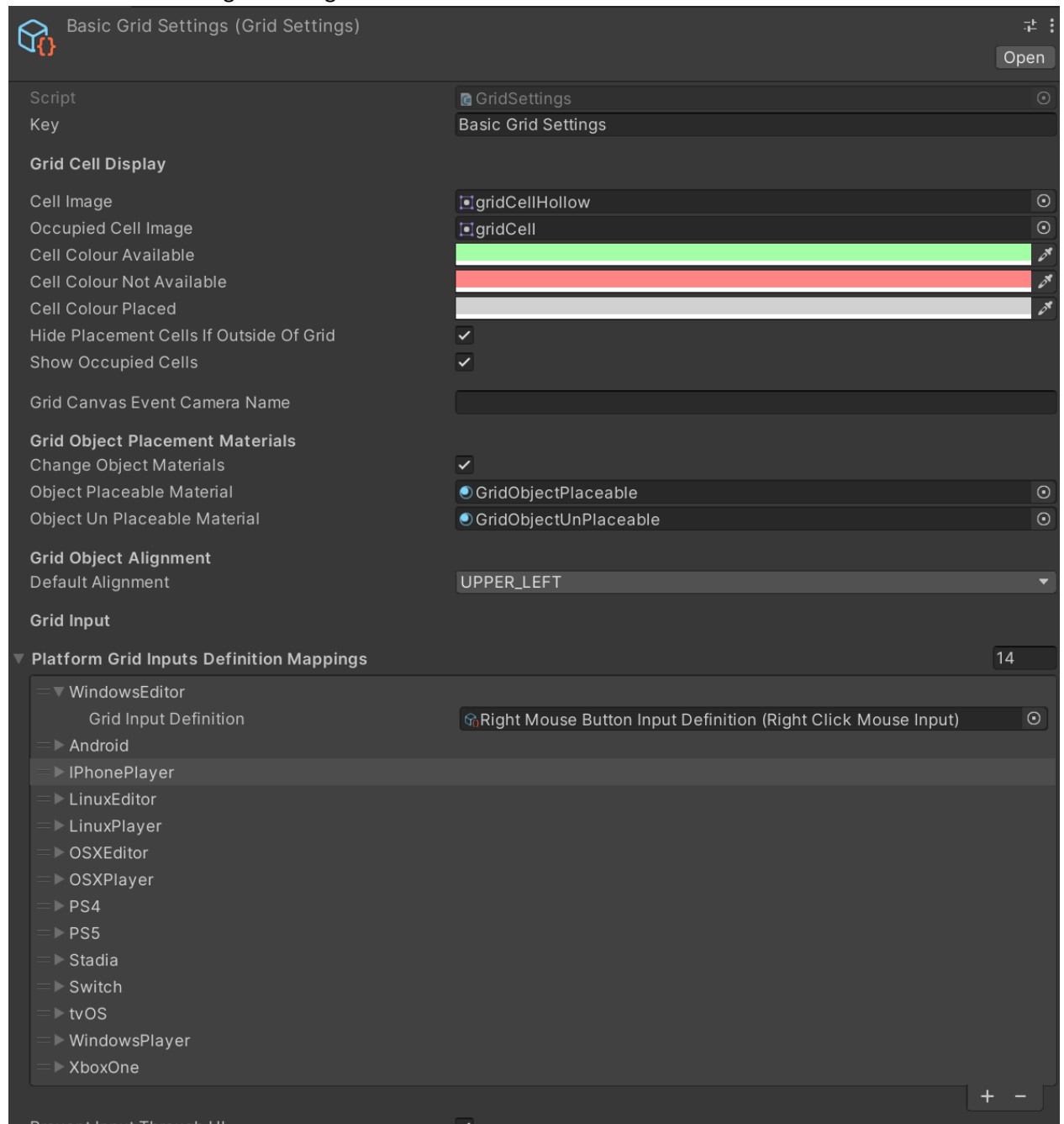
An example of script for the MouseInputDefinition for reference. You can also view the scripts in the package.

5. In the Unity editor create your custom input definition. This will depend on the menu name you defined in the script. Using the MouseInputDefinition as an example you would right click in the project folder and navigate to **Create > Grid > Mouse Input Definition**.



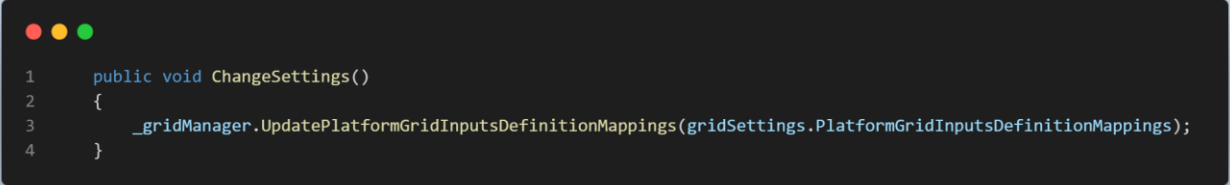
6. Finally, you will need to assign your custom asset to your desired runtime platform in your grid settings. For example, if you wanted to use the right mouse button to interact with the grid on Windows platforms. You would need to assign your custom input definition to the Windows

Editor runtime in the grid settings.



Change Input Settings

There is the functionality to switch the Input definitions at runtime. This can be achieved by calling the **UpdatePlatformGridInputsDefinitionMappings** function on the grid manager.

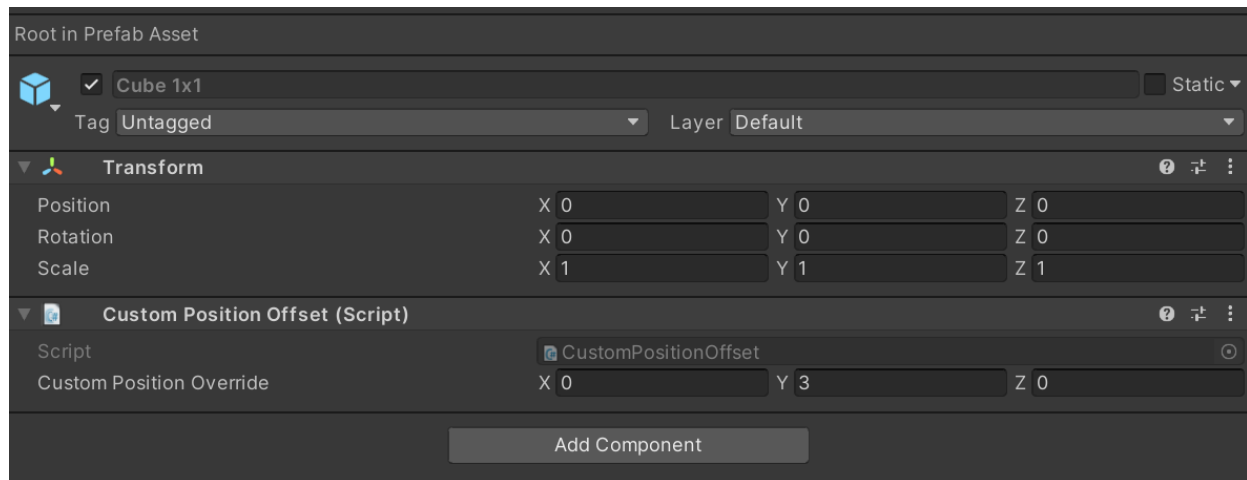


```
1  public void ChangeSettings()
2  {
3      _gridManager.UpdatePlatformGridInputsDefinitionMappings(gridSettings.PlatformGridInputsDefinitionMappings);
4  }
```

Custom Position Offset

When in placement mode the object being placed is snapped to each cell by the Grid Manager. To provide a way to have control over the position of the object there is a component you can attach to the object being placed. The component is called “CustomPositionOffset”.

If you want a constant offset, you can simply attach this component and set the offset in the inspector. This may be useful if you want some objects to hover off the base of the grid.



If you need to update the offset programmatically you can call **SetCustomOverride** on the **CustomPositionOffset** component.

```
1 CustomPositionOffset customHeightOverride = objectBeingPlaced.GetComponent<CustomPositionOffset>();  
2 customHeightOverride.SetCustomPositionOverride(new Vector3(xPos, yPos, zPos));
```

There is a demo scene dedicated to demonstrating this feature. The scene has a basic example of how you may add a placement animation by adjusting the height of the object being placed.

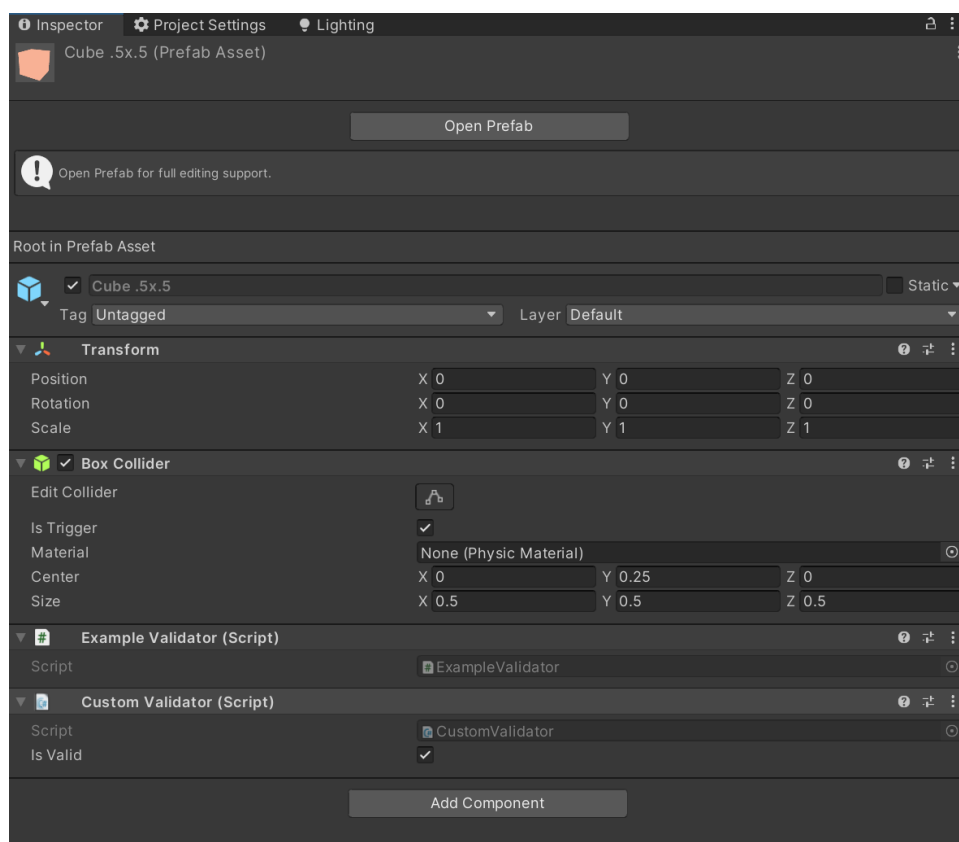
Custom Validation

You can override the grid validation of an object being placed to be invalid, even if the grid determines that all the grid cells required are available.

An example use-case is if you want to have a piece of environment that the grid does not manage but it still sits on top of the grid. In this case you may want to detect if the grid object is intersecting with the environment object (such as a wall) and tell the grid that this placement is not valid.

Because the environment or other external objects are not part of the grid, you will need to tell the grid when the object being placed is no longer intersecting with the external object and mark the custom validation as valid again.

To use the custom validation feature you must attach a CustomValidator component to the gameobject you're placing.



You then need to call the SetValidation function on the CustomValidator component and pass in if the placement is valid.

```
1 private void HandleEnteredWallArea()
2 {
3     _customValidator = GetComponent<CustomValidator>();
4     _customValidator.SetValidation(false);
5 }
6
7 private void HandleExitedWallArea()
8 {
9     _customValidator = GetComponent<CustomValidator>();
10    _customValidator.SetValidation(true);
11 }
```

There is a demo scene dedicated to demonstrating this feature. The scene is called Custom Validation Demo. The scene has 2 walls that are not managed by the grid, but the placement becomes invalid if the object being placed intersects with the wall.

Important

Although you can set the custom validator to be valid, if the grid detects an invalid placement because the required cells are not available then the placement will still be invalid, regardless of the custom validator.

Custom Placement

You can set the initial placement of an object in several ways. There is a demo scene called **Custom Placement** that has a working example of the custom placement functionality.

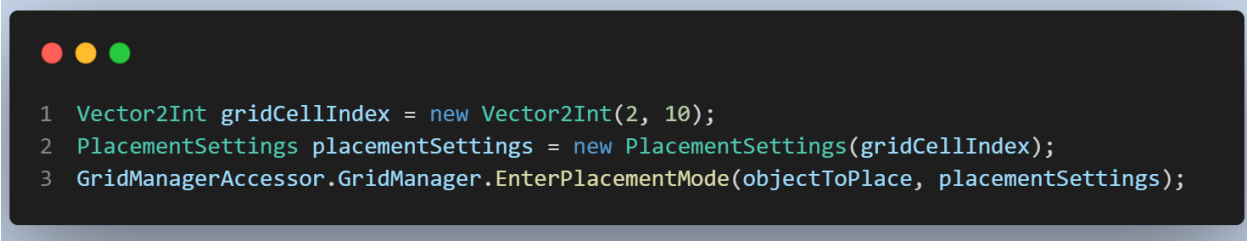
Grid Settings

The grid settings contain a property called **Initial Placement Cell Index**. When you enter placement mode the object will be placed at the grid cell defined in this setting by default.

Placement Settings

You can override the initial placement settings of the grid settings by passing a **Placement Settings** object as a second parameter to the **EnterPlacementMode** function. When you construct the Placement Settings object you can either set a world position or a grid cell position. This is determined by the type you pass in. A **Vector2Int** is used to set a grid cell index, a **Vector3** is used to set the world position. The Y property of the vector 3 is ignored when setting the world position.

Setting a world position will result in the object being placed at the specified world position, regardless of if that position is in the bounds of the grid or not. Setting the grid cell index will cause the object to be spawned at the grid position specified.



```
1 Vector2Int gridCellIndex = new Vector2Int(2, 10);
2 PlacementSettings placementSettings = new PlacementSettings(gridCellIndex);
3 GridManagerAccessor.GridManager.EnterPlacementMode(objectToPlace, placementSettings);
```

Move the Grid at Runtime

If you need to move the grid at runtime along with the objects placed on the grid you can call the **MoveGridTo** function on the **Grid Manager**, passing in the desired position. If you want to find out the current position of the grid you can use the **RuntimeGridPosition** property on the **Grid Manager**.

There is a demo scene called **Move Grid Demo** that has a working example of this functionality.

Important

The **Parent to Grid** setting in the grid settings must be enabled for the placed objects to follow the grid as it moves.

Rotate the Grid at Runtime

If you need to rotate the grid at runtime along with the objects placed on the grid you can call the **RotateGridTo** function on the **Grid Manager**, passing in the desired rotation. If you want to find out the current rotation of the grid you can use the **RuntimeGridRotation** property on the **Grid Manager**.

There is a demo scene called **Move Grid Demo** that has a working example of this functionality.

Important

The **Parent to Grid** setting in the grid settings must be enabled for the placed objects to follow the grid as it rotates.

Object Grid Position

You can obtain the grid cell position of an object while it's being placed by subscribing to the **OnObjectPositionUpdated** event on the grid Manager. To see an example of this, check out the **GridCoordinateManager** class in the Basic Demo Scene.

You can also obtain the grid position of a placed object by reading the **GridCellIndex** property from the **GridObjectInfo** component. The **GridObjectInfo** component is added onto every object placed onto the grid.



```
1 GridObjectInfo gridObjectInfo = gridObject.GetComponent<GridObjectInfo>();  
2 Vector2Int gridCellIndex = gridObjectInfo.GridCellIndex;
```

Supports 100 million Grid Cells

The grid can support millions of grid cells within a single grid. The system has been tested and can generate a grid with 100 million cells. Such a large grid takes time to generate.

However, it's ultimately up to the hardware of the device. The grid is designed to be very performant when generating the grid when "Setup" is called on the Grid Manager.

Grids with a cell count of 100,000 generate in less than 1 second (on an 11th Gen i7 processor). As the grid cell count increases the grid will take longer to generate. With a grid cell of 10,000,000 it takes ~1 minute (on an 11th Gen i7 processor). For a grid cell count of 100,000,000, it takes ~10 minutes on an 11th Gen i7 processor, however it consumes a lot of memory so is not recommended to go as big as 100,000,000. The grid cell generation is threaded so it does not block the game.

Important

For **extremely large** cell counts the grid will take time to generate when it's set up for the first time. (Once set up the grid will show and hide instantly). As the grid set up is threaded to avoid blocking the main thread you need to ensure that the grid has finished generating before trying to place an object on the grid. When grids are so large that they are not generated instantly you can wait for a setup callback to be invoked to ensure the grid is ready to use. (See the large grid cell demo scene which demonstrates using this callback to wait for a grid to generate that contains 10 million cells.

Support

If you experience any issues or need assistance with this asset, please send an email to:
support@hypertonicgames.com.

Join the Discord server: <https://discord.gg/58Wx3yCAYS>

View the Github: [Github](#)