Ultimate Isometric Toolkit v1.2.1

Marvin Neurath, *CodeBeans*

This document is an introduction for the Ultimate Isometric Toolkit. It references version 1.2.1 and above.

Problems with Unity and Isometric perspective

When starting to create a game in isometric perspective in Unity you have basically two options:

First Setting up a 3d scene and putting the camera into the correct perspective.

Second Using sprites in Unity and aligning them correctly in the Scene.

Unity as a 3d engine is quite good using the first approach. Of course it comes with the overhead of creating and rendering 3d objects.
But when it comes to the second approach you need a lot of additional code to solve common problems like sorting orders, intersecting sprites and continuous movements. The advantages for the second method are obvious. You safe time since you do not have to model/texture/rig/... 3d Objects, you eliminate the overhead of rendering 3d scenes, you safe money by taking one of the hundreds of spritesheets out there, etc.
This toolkit is a solution for these problems. It is a all-in-one solution to make isometric games inside Unity.

IsoObject

IsoObject is the base component in this Asset. Everything, not just sprites, that appears in the scene should have this component attached.
Sorting is done not only in scene view but in the editor window as well to get you immediate feedback.

Custom Editor

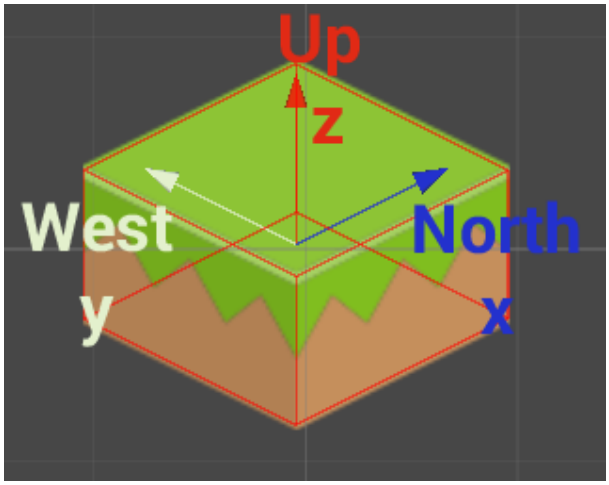

Figure 1: *Isometric directions*

IsoObjects have custom handles for all three dimensions and a screen overlay for their bounds.
IsoObjects have three properties ***Depth***, ***Size*** and ***Position*** which can easily be accessed and modified either by script or in the inspector. The IsoObjects projection is calculated once via a transformation matrix. If the position is changed via accessing the property its isometric position is updated automatically. However you can always update the projection by calling *updateIsoProjection()*.

```
IsoObject myIsoObject = ..;
myIsoObject.Position = Vector3 ...
myIsoObject.Size = Vector3 ...
float depth = myIsoObject.Depth;
```

Isometric Sorting

Now that we know each Object has a `position`, `depth` and `size` we need a sorting order for our sprites. While up to version 1.2 many different approached were tested and updated constantly, the best results could be accomplished by using a z-buffer. At first the sorting order of the SpriteRenderer was recalculated to guarantee a total order per frame. This has three major disadvantages. Firstly the **massive overhead**. With 10000 sprites in a scene you could end up with frame rates down to 10fps. Secondly, while the position of an object is stored in continuous float values its depth is stored as a discrete *int* value which can lead to strange effects at neighbouring sprites. And thirdly, the constraint to have a running instance of the IsoSorting component in each scene.

With version 1.2.1 this is now obsolete. The sorting is done by Unity which, as far as I know, uses an optimization on the graphic card. There is nothing you have to do to achieve a proper sorting order in each frame. No additional overhead, no limitations.

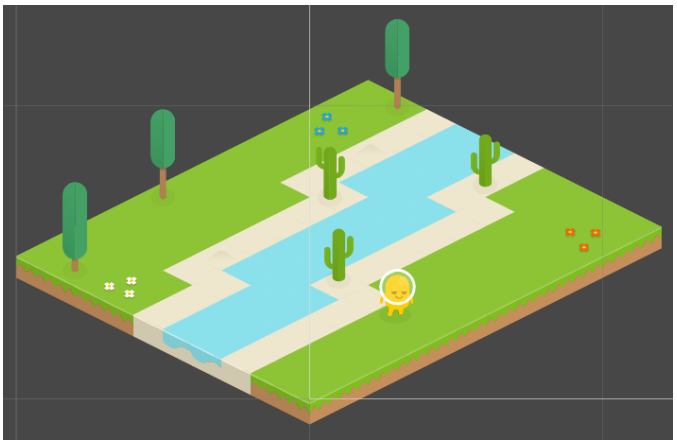

Figure 2: *Orthographic front view*

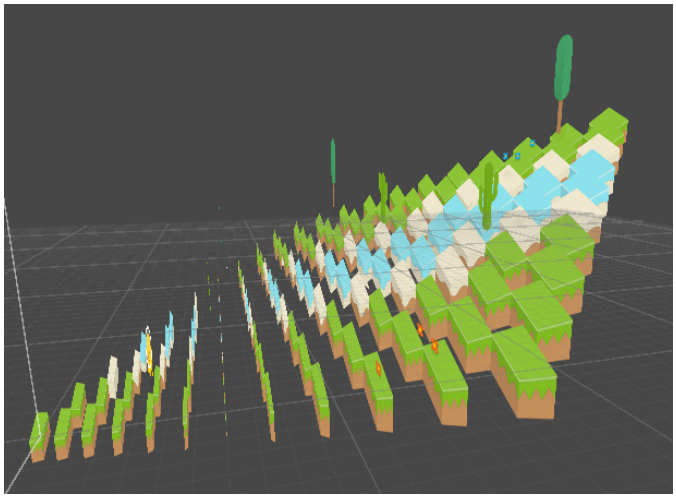

Figure 3: *Perspective Side view*

Editor Tools

I provide you with two helpful editor tools.

Isometric Snapping

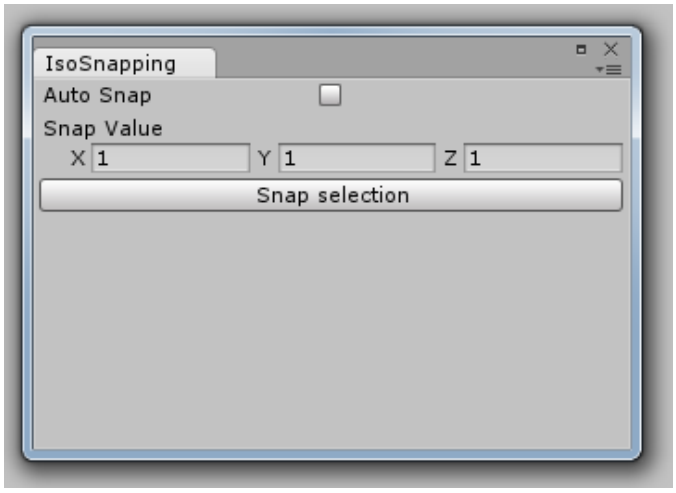

Figure 4: *Isometric Snapping window*

Isometric Snapping can be opened either by clicking in the upper menu under *IsoTools* or by pressing *CTRL-L*.
The Isometric Snapping can be used to align sprites in a grid system to avoid overlapping areas. Enter the sprite size as the *Snap Value* and activate Auto Snap. Now drag an IsoObject in the Scene along its isometric Axis. The IsoObject will snap to the closest multiple of that vector. To align a selection of sprites just click *Snap selection*.

How to get started

So you might bought the package and want to know how to get started. I provide 5 different sample scenes for different use cases. All you have to do is adding sprites into your scene and adding the IsoObject component on each of those sprites. Now you can drag the custom handles by clicking one or more sprites in the scene view. Set up your level, implement a game logic, add either your own or one of my controllers to move objects around and you are good to go.

Advanced Precision

Click on an IsoObject and drag it along one of the iso axes. By pressing *shift*, the Input is scaled down to 10% to get precise positioning option.

IsoObjectControllers

For moving objects in your scene you need controllers. You get 5 Controllers ready to use.

AbstractIsoObjectController

This is the base class of all controllers. If you need additional behaviour just inherit from this class and add your custom code. Standard implementations to move around can be overridden.

SimpleIsoObjectController

This is a simple controller to move around. It uses Unitys standard *transform.Translate(...)* code. It works without collision detection, ghost objects, etc.

AdvancedIsoObjectController

This controller can be used to implement games with collision detection. To get the best results and compatibility with Unity it uses a *Ghost Object*. Ghost Objects are 3d cubes, relative to its associated IsoObject. They are spawned at runtime. The IsoObjects position is mounted to its Ghost Object position. To change the position, input forces, etc. just apply these changes to the Ghost Object rather than to the IsoObject itself. Look at figure 5 and 6.

TurnbasedIsoObjectController

The TurnbasedIsoObjectController should be used for turn based games. The Controller has an action queue. It slowly dequeues and invokes these actions with a given speed. You can enqueue new actions by pressing WSAD/Space.

NodeIsoObjectController

This is an implementation for continuous movements. You define a path by setting up nodes in your scene and link them to this controller instance. You can alter the transition method between nodes. See http://easings.net/de to get a full list of them. The IsoObject this controller is attached to will now walk along this path. To enable looping just activate the flag via the inspector GUI.

Isometric Collision and Rigidbodys

The collision detection uses the same Ghost Object approach. Add an *IsoCollider* to all objects in the scene that your IsoObject could collide with.

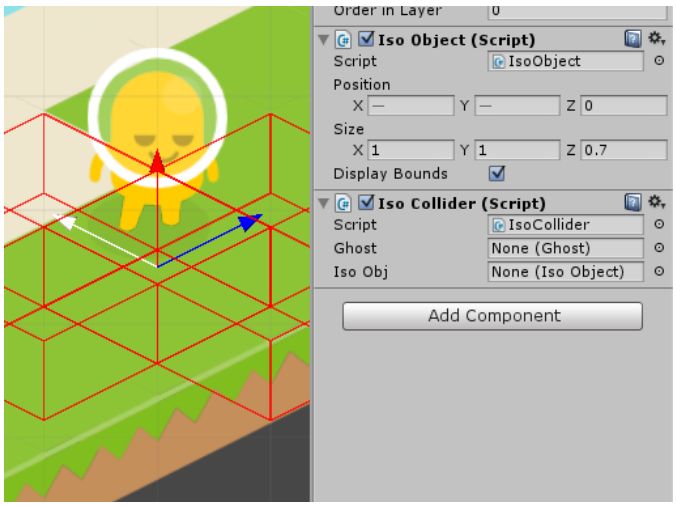

Figure 5: *IsoColliders on ground tiles*

Now add the *IsoRigidbody* to your IsoObject and it should work properly.

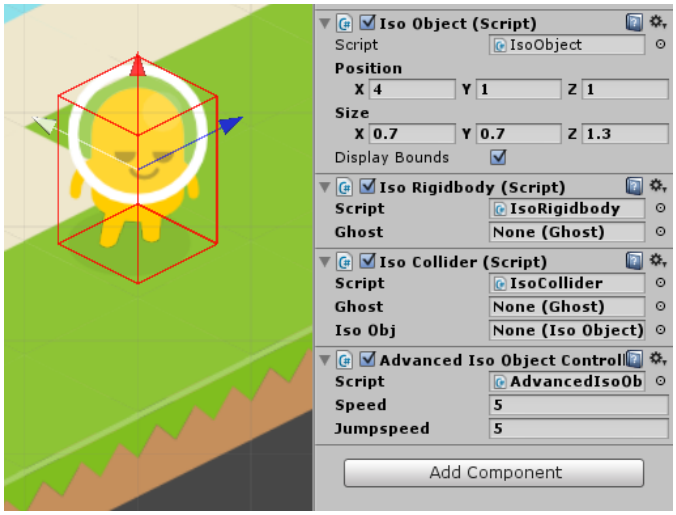

Figure 6: *IsoRigidbody & IsoCollider on player*

Take a look into the *collsionSampleScene* for more help.

GridMap

The *GridMap* is an exemplary implementation for tile maps with discrete values. It stores sprites in a flattened 3d array and can be accessed via *[x,y,z]* coordinates. The GridMap has barely any restrictions. To create a Map either 2d or 3d you need to provide a proper *delegate* which is just a function that maps

a position [x,y,z] to a sprite to spawn at that area (which can be *null* as well). You can watch a more complex use case with PerlinNoise and procedural level generation online.