Dynamic Loading Kit

This guide was last updated on 12/15/14 and applies to v 3.0.3 of the Terrain Slicing & Dynamic Loading Kit– Author: Kyle Gillen

Please check http://deepspacelabs.net/dynamic_loading/dynamic_loading.html for the latest version.

Availability: Part of the Terrain Slicing & Dynamic Loading Kit, which can be found on the Unity Asset Store

Table of Contents

General Overview

The Dynamic Loading Kit is a component based API that allows for the dynamic loading of Unity GameObjects during game play. Notice the key word ***GameObject***. While the kit was primarily created with the intention of loading Unity Terrains, any Unity GameObject can be dynamically loaded. In addition, there is no restriction that loading take place solely on the X/Z Axis (the axis Unity Terrains sit on). The kit can work on the X/Y Axis (for 2D side scrolling games) or even on all three axes (for a fully three dimensional world).

The utilization of a component based system ensures the kit integrates seamlessly with Unity's design flow. The behavior of the Dynamic Loading Kit is adjusted the same way other Unity Game Object behaviors are; by removing and adding components, or by tweaking exposed variables in the inspector.

Special Note*

1) Do not create/destroy Dynamic Loading Kit components at runtime via GameObject.Instantiate and GameObject.Destroy. You should only ever need to create/destroy Worlds and Active Grids, which should be

done through the Component Manager (CreateNonPersistentActiveGrid, DestroyWorld, etc.). Other components, such as the Boundary Monitor, Primary Cell Object Sub Controller, etc., should never be created/destroyed at runtime.

2) While disabling/enabling components at runtime is a common and useful Unity practice, never do so with World's/Active Grids, as it will screw with the Component Manager. Either create/destroy these components when they are needed/not needed (see item 1 above), or let them sit idle while not in use.

Chapter 1: Inner Workings

The Dynamic Loading Kit utilizes a grid based system, where each cell in the grid is identified by a row and column number. At any given time, a number of cells are "active," which means the objects associated with those cells are loaded into the scene or enabled. This active grid can move up, down, right, or left, and doing so activates cells (in the direction of the move) and deactivates cells (in the opposite direction). This chapter will examine these concepts in detail, and will hopefully provide you with a thorough understanding of how the kit works.

Section 1: The World Grid

Conceptually, the World Grid represents the terrains or other objects that make up your world. Have 8 rows and 8 columns of terrain in your world? Then your world grid is also an 8 x 8 grid.

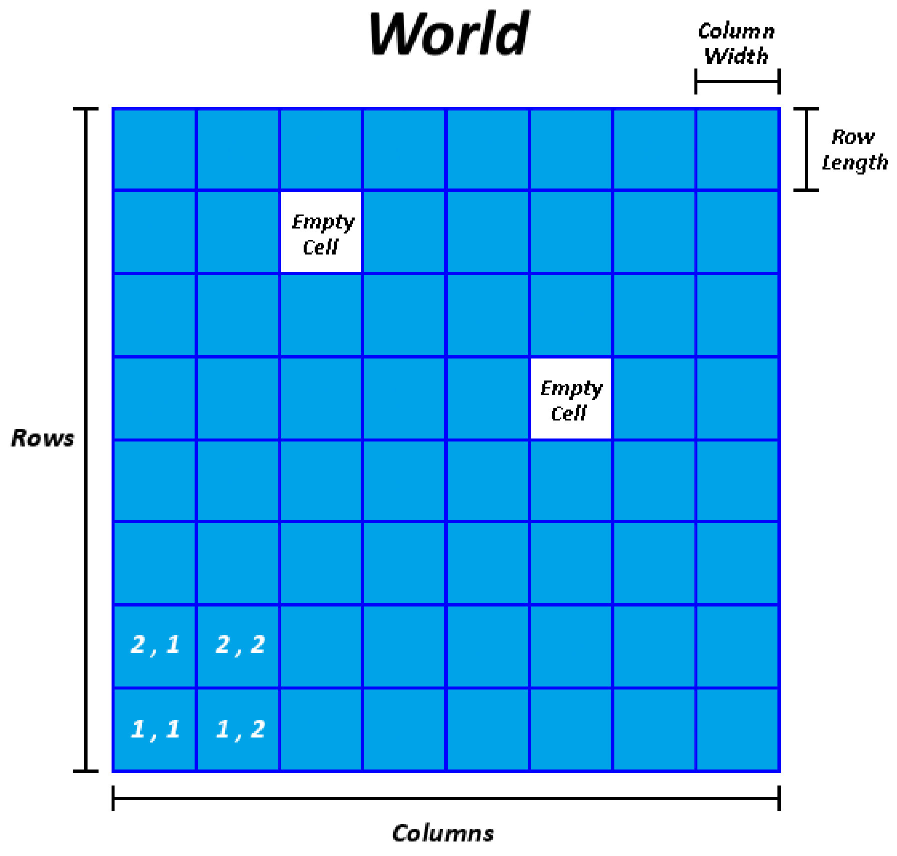


Figure 1: The World Grid (for 2D World)

There are six key pieces of information relating to the world grid that the dynamic loading kit cares about. These are:

1) The base name of your cell objects (terrain or other objects). This name should be the name each of your cell objects start with (it's very important that all objects in your world grid share a common name). Note that this does not mean all objects need to be parented under a single game object. Each terrain/object in your world should be its own game object (with its own children, if applicable).

2) Whether the objects associated with the World Grid are Unity Terrains.

3) The World Type of your World Grid. Is it a 2D grid on the XZ or XY plane, or a 3D world that uses all three Axes?

4) The number of rows, columns, and layers (for 3D Worlds) in your World Grid.

5) The length of each row, width of each column, and height of each layer (for 3D worlds) in your World Grid.

6) Whether each cell in the grid has a terrain (or other object) associated with it.

Special Note Optimizations are made when all data of a specific type are the same. For instance, if all column widths are 200, then a single value of 200 is stored rather than an array of values. When creating world grids with a massive amount of cells, it may be necessary to ensure values of a particular type are the same, so that this optimization can take place.*

Managing the Required Data

Within Unity, the required information about the world grid is stored in a Scriptable Object Asset. You create this asset via a button on the Dynamic Loading Configuration Form, or via the menu option under Assets -> Dynamic Loading Kit -> Create World Grid. The asset is created in whatever folder you have selected when you press "Create World Grid," or in the Assets folder if no folder is selected. You can also right click a folder and find the same option under "Dynamic Loading Kit."

Once the asset is created, you can select it in your project hierarchy. You are free to rename the Asset to whatever you like; it has no bearing on the Dynamic Loading Kit.

The info for items 1-4 is set directly in the inspector. Items 5-6 is calculated or retrieved based on the data from 1-4 (plus some additional info). For more info on the World Grid, including how to Set its Data, see the World Grid section under the chapter "Configuring the Dynamic Loading Kit."

Important Note About Naming Your Objects
Your main objects/terrains should adhere to the following naming convention:

BaseName_Row#_Column#, where BaseName is the same name as item 1 in the list above. Rows and columns start at 1, NOT 0, and it's important that you don't include any leading 0's before the row or column numbers.

Ex:

Good: WorldTerrain_2_2

Bad: WorldTerrain_02_2

The reason this naming convention is necessary is because I use a custom class designed specifically around this naming convention (the CellString class). The specifics of this class are not important, only that it was designed to reduce garbage generation and improve performance when running string comparisons (primarily, when looking for new objects loaded via the SceneLoader classes).

Please see the section on the World Grid in the next chapter for more info.

Section 2: The World

Conceptually, a World is a representation of your Terrain/Object group in the Scene.

With version 2.0 of the TS&DLK, the World is now fully represented via its own component, aptly named the "World" component.

The distinction between the World and World Grid is very important.

The World Grid is pure data. While it contains some data retrieval functions, it does not actually "do" anything in game. The World, on the other hand, is the driving force behind representing your terrain/object group in game. If you think of the World Grid as a sort of blueprint for your terrain/object group, the World is the actual implementation of that blueprint. This leads to a very important point:

A single World Grid can be implemented multiple times via multiple Worlds (even in the same scene).

Each World implementation has two key pieces of information separate from the World Grid; the World's *position* in the scene, and whether the World is endless along one or ore more of its axes. This means you could have two Worlds representing the same World Grid in two different locations, or even one world which is endless on the X axis, and one that is fixed.

A World can also be setup to stay centered about its origin. This is useful when using endless worlds or when a non-endless world is so large that floating point errors become an issue.

Cells and Cell Users

Like the World Grid, the World is made up of a collection of cells. Unlike the World Grid's cells, these cells can extend endlessly on any axis, and also have position (along with some other data). For example, let's say you have a simple 2x2 World Grid, with four terrains. An endless World implementation of this World Grid would have these standard four cells, and each cell would be associated with one of the four terrains. Cell_1_1 would be associated with Terrain_1_1; Cell_1_2 would be associated with Terrain_1_2, and so on. Because the World is endless, however, Cell_3_1 would also exist, even though Terrain_3_1 does not. In this case, Cell_3_1 is also associated with Terrain_1_1, since the world repeats.

Each cell in the World exists only as long as it has *users*. A user is simply an entity that makes use of the cell in question (such as an Active Grid). A cell can have multiple users at a given time, and users can be added or removed at any time. If a cell does not exist, but some user specifies it needs to use it, the cell is created and the terrain/object associated with that cell is loaded. Note that cells only exist for positions on your World Grid that actually have terrains/objects. Empty grid locations are simply ignored, and no cells are created for them.

When a user no longer is using a cell, it tells the World, and if the cell has no additional users, the terrain/object associated with the cell is removed from the scene.

The term "Users" is perhaps a poor choice of words, as the entities don't actually use the cells of the World. That is, they do not receive a reference to the cells. Rather, it is more like they are staking an interest in the cell, saying "I need that cell to exist."

Section 3: The Active Grid

An Active Grid is simply a collection of cells. Like the World and World Grid cells, each active grid cell is defined by an index; it has a row, column, and (if using a 3D world) layer (it also has a level of detail, which is unused at this point in time). Unlike the World and World Grid cells, this index is the only information the active grid cell is concerned with.

The number of cells in the Active Grid is defined by the grids dimensions, which can be adjusted via the inspector or programmatically during the game.

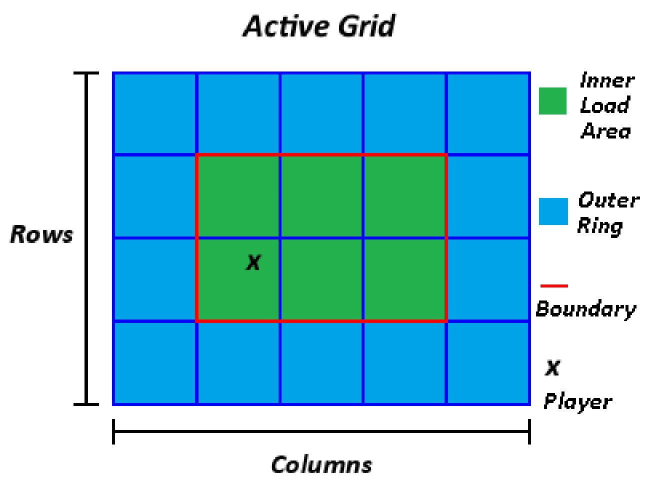


Figure 2: Active Grid for 2D World

This grid can shift during the game, either manually (via function calls) or automatically (based on the player's position). As it shifts, cells that are no longer used are removed, and new cells are added. In conjunction with this update, the Active Grid tells the World component which cells it is no longer using and which new cells it is now using. Note that the Active Grid does not care (in most cases) what the World does with that information.

While each Active Grid cell only contains information about its index, the Active Grid as a whole cares about other data. For instance, the grid needs to be able to retrieve the dimension of a cell in order to set up and maintain the inner area boundary (see Figure 2). This data is retrieved from the World (which may in turn retrieve it from the World Grid), and as such, an Active Grid must always be "synced" to a World in order to function.

The Inner Load Area

The inner load area is the portion of the active grid which is "safe for travel," when using a Boundary Monitor and Player. As long as the player remains in this zone and does not cross one of the boundaries, the Active Grid remains fixed. As soon as a boundary is crossed, however, the grid shifts according to which boundary was crossed.

You define the dimensions of the inner area load area on the Active Grid Component in the inspector, by specifying the number of Inner Rows and

Inner Columns (as well as Inner Layers for 3D worlds). In figure 2, there are 2 inner rows and 3 inner columns in the inner load area.

Outer Ring

The outer ring is the area outside of the inner load area. When using a Boundary Monitor, a grid shift is initiated when the player crosses a boundary into the outer ring area. You specify the amount of objects that make up the outer ring by specifying the Outer Ring Width on the Dynamic Loading Configuration Form. Note that for every increment to the outer ring width, you add two rows/columns/layers of cells to the total number of rows/columns/layers.

Calculating Total Number of Cells

The total number of cells in your active grid can be calculated with this formula:

2D World:

cells = [(2 * outer ring width) + inner rows] * [(2 * outer ring width) + inner columns]

3DWorld:

Cells = [(2 * outer ring width) + inner rows] * [(2 * outer ring width) + inner columns] * [(2 * outer ring width) + inner layers]

Section 4: Component Manager

The Component Manager has many responsibilities and allows for some awesome functionality.

First, it maintains a list of all Active Grids and Worlds in the scene. This provides, among other things, a central repository from which you can retrieve these components during runtime.

Second, it allows for the creation and removal of Active Grids and Worlds in game at any time. You can destroy Worlds/Active Grids that you added to your scene in the Editor (via the inspector), and these components will be destroyed at the beginning of the scene on each successive play session. You can create entirely new Worlds and Active Grids using prototypes, and have these new components loaded automatically at the beginning of each play session.

Third, it handles the initialization and startup of all Active Grids and Worlds in the scene when the Scene is loaded. While you can do this manually, allowing the Component Manager to handle things is simplicity itself.

Important Note
While the Component Manager is not a Singleton, it’s imperative that there is only a single Component Manager in any given scene.

Section 5: Other Components

Note: This will be a brief overview of these components. See later chapters for more information on them

<u>**Persistent Data Controller**</u>

A component which controls saving and retrieving persistent data used by Active Grids and Component Managers. You can have multiple Persistent Data Controllers in a scene, but you should ensure each one has a unique Scene ID (actually, all Persistent Data Controllers in your project need to have unique ID's). The controller also contains extensive options for wiping persistent data.

The Persistent Data Controller is key based, which may or may not work for your particular saving/loading solution. If you need to use a non-key based solution, check the "Use Custom Save/Load Solution" on your Component Manager and disregard the Persistent Data Controller.

<u>**Primary Cell Object Sub Controller**</u>

A component which controls the cell objects (most likely terrains) associated with your World. For instance, it controls whether the cell objects are pooled or destroyed when they are no longer needed. The World (via the Cell Object Master Controller) requests cell objects from the primary cell object sub controller when it needs them, and also returns them to the controller when they are no longer needed.

Note that a single Cell Object Sub Controller can be used by multiple Worlds.

<u>**Cell Object Loader**</u>

A component which controls how cell objects are loaded into the scene. For instance, are they loaded via Instantiate, or Application.LoadLevelAdditive? This component is used exclusively by the Primary Cell Object Sub Controller.

Note that a single Cell Object Loader can be used by multiple Primary Cell

Object Sub Controllers.

Cell Object Destroyer

A component which can be used to fine tune the manner in which objects are destroyed. This is an optional component which can be supplied to the Primary Cell Object Sub Controller component.

Boundary Monitor

A component which monitors the dynamic or static boundary defined by an Active Grid, checking to see if the player has crossed one or the other. When the player does cross a boundary, monitor notifies the Active Grid.

Chapter 2: Configuring the Dynamic Loading Kit

Note: I will assume you are starting fresh, with no terrain/object group created. If you have your group created and named correctly, you can skip to Section 2. Also, even if you've already configured the kit, I suggest reading this chapter, as there are many new components and features in v2.0!

Section 1: Creating your Terrain/Object Group

The first order of business is to create your Terrain/Object Group. These are the objects that will be associated with the cells in your World Grid. If using the Terrain Slicing tool, you simply need to slice your base terrain into however many slices you desire. The slices will be outputted in the correct format required by the World Grid.

Otherwise, you will need to manually name your objects so they adhere to the correct format.

Naming Convention

The naming convention to follow is GroupName_Row_Column for 2D groups, and GroupName_Layer_Row_Column for 3D groups. Terrain groups are 2D groups and must follow the 2D convention.

GroupName is the base name that all of your objects share, while Row/Column/Layer refers to the placement of that object within the group. The following sections offer helpful guides on how to properly name your objects.

2D Worlds

Note: For these examples, let's assume the group name for your group is Slice

For a 2D world, start by aligning your scene camera so it is facing directly at your world, so that the vertical (Z axis for an XZ World, and Y axis for a XY World) scene gizmo is pointing towards the top of your screen, and the horizontal (always the X axis) scene gizmo is pointing to the right of your screen. It may be helpful to place the scene camera into an Isometric View for this.

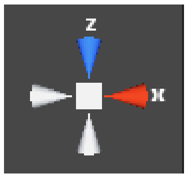


Figure 3: How Gizmo should look when looking at 2D XZ World

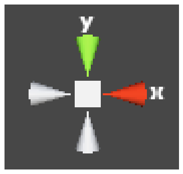


Figure 4: How Gizmo should look when looking at 2D XY World

When you have the view correct, naming your objects becomes quite easy. Simply start with bottom left most object; this object is the first object in your group and will have a row of 1 and column of 1 (Slice _1_1). The object to the right of this object is Slice_1_2, the next one to the right is Slice _1_3, then Slice _1_4, and so on. Heading back to Slice _1_1, now look to the object above the object on the vertical axis. This object is Slice _2_1. The object to the right of Slice _2_1 is Slice _2_2, and so on.

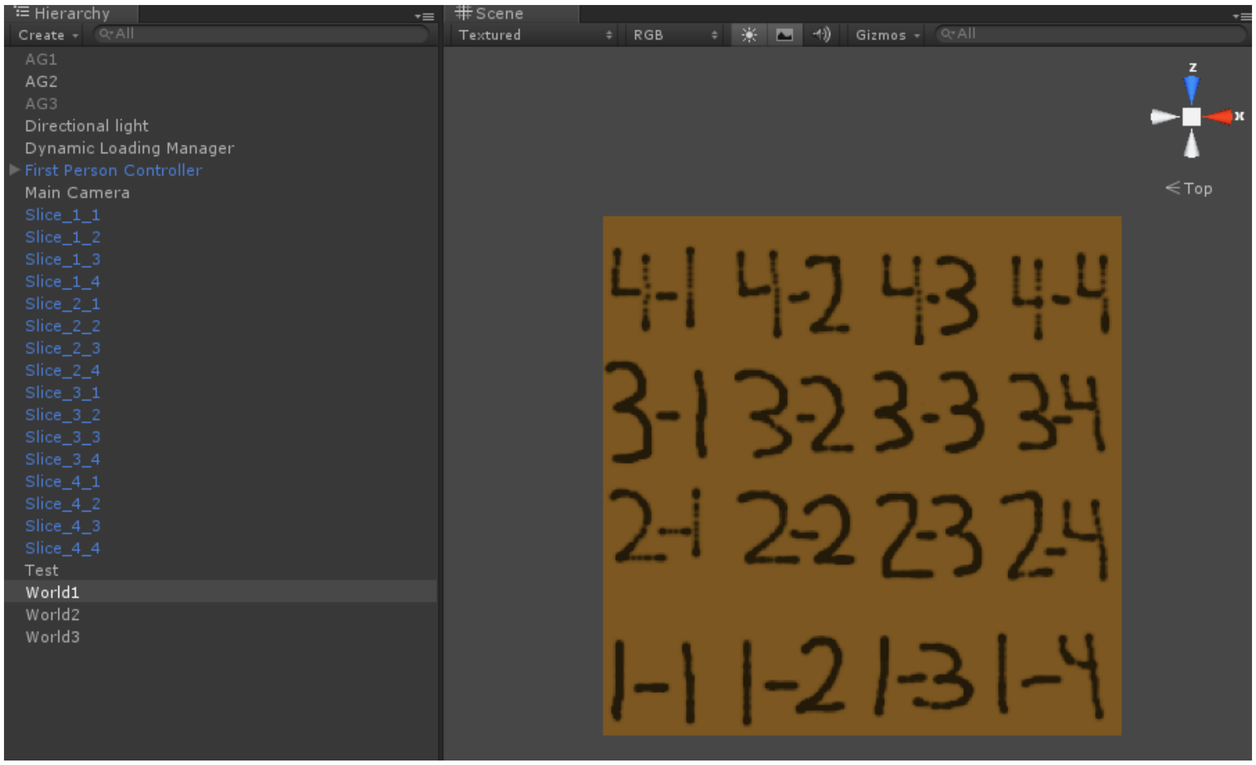


Figure 5: Visual Representation of an Object Group

3D Worlds

When it 3D worlds, the same logic that is used to assign the row and column indexes to XZ 2D object groups can be used to assign the row and column indexes to 3D worlds. This is because 3D worlds always have their rows on the Z axis and Columns on the X Axis. The third index, layers, is always on the Y axis. So simply use the instructions for 2D worlds and Figure 5 to name the row and column indexes. Imagine that Figure 5 is simply showing a single layer in your object group.

For layer naming, align your scene camera so you are facing the side of your group (so that the Y Gizmo is facing towards the top of your screen and the X Gizmo is facing the right side of your screen). In this view, the objects at the bottom of your group are your first layer (layer 1), the layer above that is layer 2, and so on.

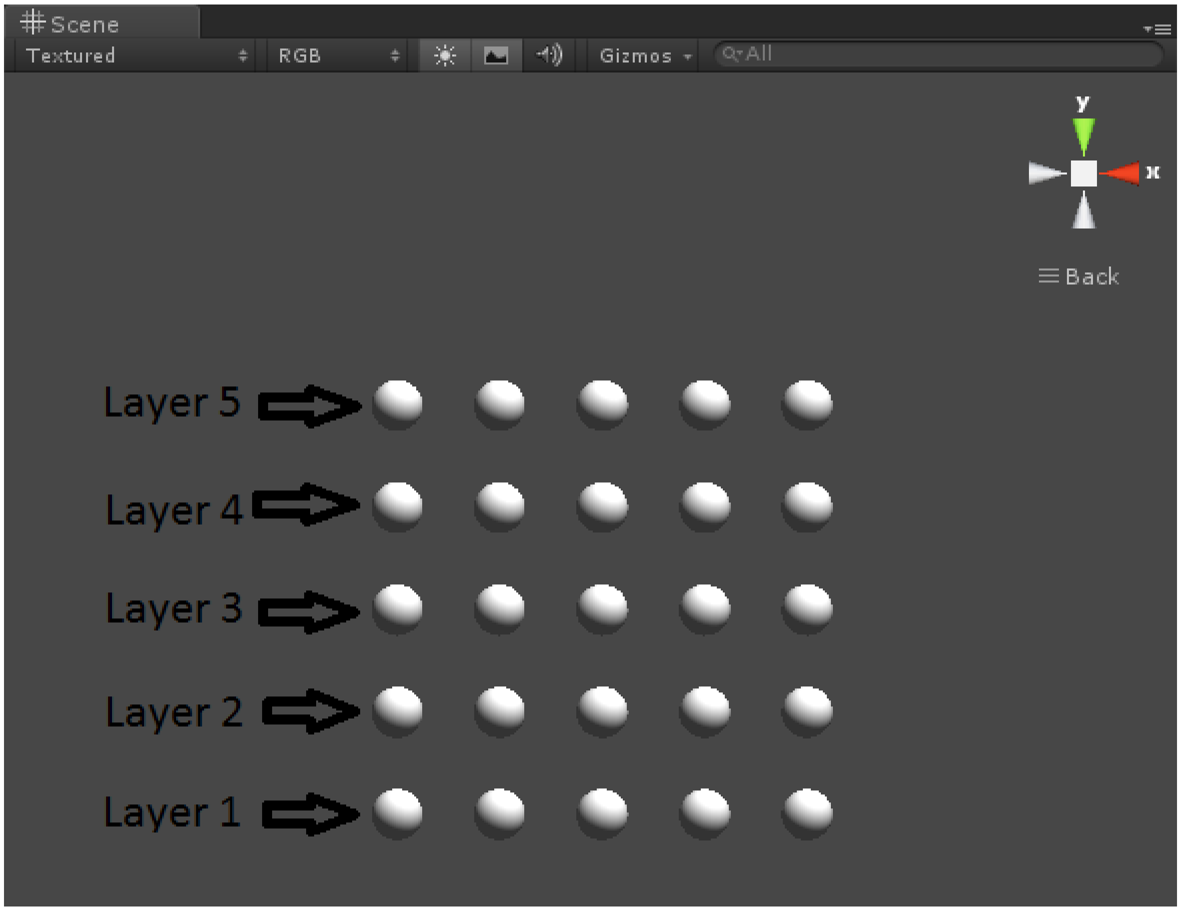


Figure 6: Layers of a 3D World

In Figure 6, (assuming the Z axis is running away from the camera), the sphere at the bottom left of the screen would be (assuming a group of Sphere) Sphere_1_1_1 (remember, it's GroupName_Layer_Row_Column).

Leading Zeroes and the Group Name

It is very important that your row/column/layer indexes are not preceded by a 0. A specialized class is used internally to load and find cell objects, and this class works under the assumption that there are no leading 0's in your indexes. Also, while I have not tested every combination of naming, it shouldn't matter what your group name is. You can have spaces, underscores, and any other combination of numbers/letters in it.

Wokdjowdk_1_1 = valid name
Ter_01_1 = invalid name (leading 0 before row index)
Slice_1_0 = invalid name (index cannot be 0)

Creating Prefabs for your Group

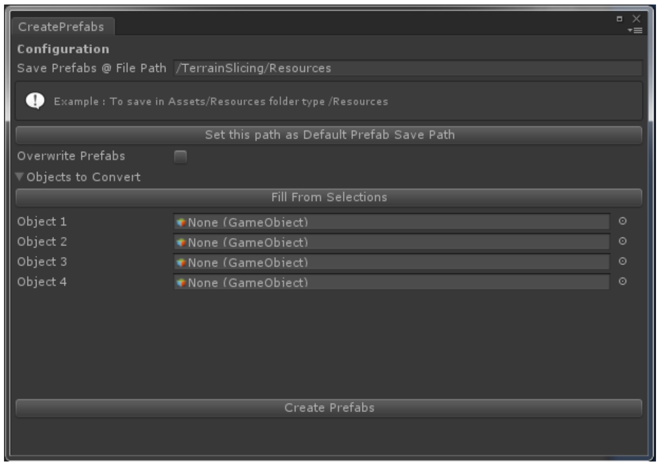
Once your terrain/object group is created and named properly, you'll need to convert the game objects that make up your group into prefabs. This is required for several reasons:

1) If using a Prefab Instantiator Cell Object Loader component (explained later), you will need to use the prefabs directly during your game.

2) If using a Scene Loader Cell Object Loader component, the scenes you will use during your game are generated using the prefabs. While you are free to delete your prefabs once these scenes are generated, I suggest not doing so. It's always good to keep a backup of your objects for future use.

3) In Section 2, you will see that for one of the steps, prefabs can be used to set some data on the World Grid. While other options exist for setting this data, in some instances prefabs MUST be used (as will be explained).

I have included a handy little tool called the Create Prefabs tool which makes it very easy to convert your objects into prefabs. The tool can be found on the top Menu Bar under Assets -> Dynamic Loading Kit -> Create Prefabs.



Configuration
Save Prefabs @ File Path /TerrainSlicing/Resources
Example : To save in Assets/Resources folder type /Resources
Overwrite Prefabs
Objects to Convert
Object 1 None (GameObject)
Object 2 None (GameObject)
Object 3 None (GameObject)
Object 4 None (GameObject)

Figure 7: Create Prefabs Tool

To use it, simply select the game objects you want to turn into prefabs in your scene hierarchy and click the "Fill From Selection" button. You should see a list of all the game objects you selected. For the “Save Prefabs @ File Path,” this is a folder path relative to the main Assets folder where you want your prefabs to be stored. For example, “/TerrainSlicing/Resources” saves the prefabs in the folder “Assets/TerrainSlicing/Resources.” You can simply right click your folder and select “Dynamic Loading Kit -> Copy Relative Folder Path,” and then Ctrl-V or Edit->Paste the folder into the appropriate field.

Prior to v2.0, the World Grid required prefabs to be stored in a folder called Resources in order to use them to set the grids data. With 2.0, you can still use this same functionality, but you also have the choice of specifying a folder path instead.

If you are not planning on using a Prefab Instantiator component, this means you should stick your prefabs in some non-Resources folder, as assets in the Resources folder will always be included with your project build, which you don’t want if not using the Prefab Instantiator.

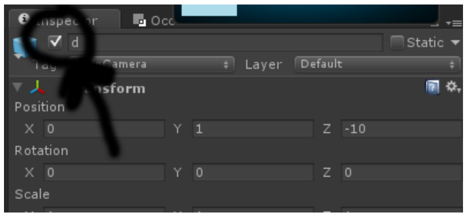
If you are planning on using a Prefab Instantiator, then you must place the prefabs in a folder called Resources!

Finally, you can select the “Overwrite Prefabs” option to have the tool automatically overwrite any existing prefabs in the specified folder automatically. If this option is left unchecked, you will be asked if you want to overwrite the prefab (if one exists with the same name in the folder).

Deactivating Prefabs (Deprecated)

Note: This step is not required if using v3.0 of the kit.

There is one final step that only applies when using the Prefab Instantiator component. The prefabs in your Resources folder must be set to an inactive state (uncheck the checkbox in the Inspector). This ensures that when the prefab is loaded into the scene, it is not visible until it has been positioned correctly. It is also required for Tree colliders to work properly.



Tag Camera Layer Default

Position X 0 Y 1 Z -10

Rotation X 0 Y 0 Z 0

Scale

Figure 8: Check Box to Deactivate Prefab

Section 2: Creating Your World Grid

Once your terrain/object group is created and your prefabs are created, you can either create your World Grid, or start adding the components you'll need to make the Dynamic Loading Kit function to the scene. Logically however, it makes sense to transition from the creation of your terrain/object group to the creation of your World Grid, so we will concentrate on this stage of the process first.

Select a folder where you want your WorldGrid asset to be created, and perform one of the following equivalent operations:

1) From the top Unity Menu Bar, select Assets -> Dynamic Loading Kit -> Create World Grid.
2) Right click the folder you've selected and choose Dynamic Loading Kit -> Create World Grid.

Option 2 is obviously easier, but you are free to use whatever method you prefer. Once created, you can rename the asset to whatever you'd like. The actual asset name has no bearing on the Dynamic Loading Kit functionality.

Click on the World Grid asset and configure it (please note, all of the following options have tooltips. Hover over the Option Name in the inspector to view the tooltip).

Default Group Name

The same group name you used when naming your terrain/object group in Section 1.

Example: If an object in your group is named IAm_an_object what?_1_2, the Group name would be IAm_an_object what?

Any World that uses this World Grid will use this Default Group Name as its starting group name, however it is possible to change the World's group name (not the World Grid's) during the game. This allows you to load alternate versions of your objects/terrains (such as ones with alternate textures or lower/higher resolutions). The alternate objects must be of the same type as the default (If the defaults are terrains, the alternates must also be terrains) and size.

For more information, take a look at the API found on my website. The methods that allow you to modify the group name are found on the World page and all have GroupName in the method name somewhere. You can also find more information in Chapter 4: Advanced Topics.

Cell Object Type

The type of the objects your group is made up of. For Dynamic Loading functionality, the kit is only concerned with whether the objects are Unity Terrains or not (because Terrain Neighboring needs to be performed on Unity Terrains).

The distinction between "Non Terrain With Renderer" and "Non Terrain Without Renderer" only matters to the World Grid, as it effects the method used to retrieve data when using the "Set Using Prefabs" data set method. Please see the sub section "Method to Set Data" below for more information.

World Type

The axes that your terrains/objects sit on. When your Cell Object Type is "Unity Terrains," only one option is displayed:

Two Dimensional using XZ Axes

Selecting this option tells the program that the world grid is two dimensional and objects should be loaded on the X and Z axes. The Y axis is fixed (to the World's origin), which means two things:

1) Every object has the same Y value, and
2) The Y position of the player has no effect on dynamic loading

The two other options are available whenever the Cell Object Type is set to either of the Non-Terrain options.

Two Dimensional using XY Axes
Selecting this option tells the program that the world grid is two dimensional and objects should be loaded on the X and Y axes. The Z axis is fixed (to the World's origin), which means two things:

1) Every object has the same Z value, and
2) The Z position of the player has no effect on dynamic loading

Three Dimensional
Selecting this option tells the program that the world grid is three dimensional and objects should be loaded on all three axes. No axis is fixed, and the X, Y, and Z position of the player all effect dynamic loading.

Layers

The number of layers on your grid. It is only shown when the World Type is "Three Dimensional." More than likely, this is the largest Layer Index value in your object names (ex: Slice_8_1_1, if 8 is the largest layer index, then you have 8 layers).

Rows

The number of rows on your grid.

Columns

The number of columns on your grid.

Underneath the Columns option, you will see a helpful message showing what the expected prefab format of your objects is. If you have any doubts about your names, this should clear things up.

Method to Set Data

The method you want to use to set the World Grid's data. This data includes the length of each row, width of each column, and height of each layer (for 3D worlds), as well whether each grid on your World Grid has an object associated with it. The three methods are as follows:

Set Using Default Values

This is the most straightforward method for setting your data. Every row length in your grid will be set to the "Default Row Length" value, every column width will be set to "Default Column Width," and every layer height will be set to "Default Layer Height." In addition, all grid locations will be marked as NOT empty.

This method should only be used when the following two conditions are true:

1) All of your grid locations have a terrain/object associated with them (so if you've deleted objects in your group, don't use this method, as it will mark a grid location as not empty when really it *is* empty).
2) Your objects all have the same dimensions. This means that every object has the same height (if obj is 3D), length, and width. If using the slicing tool, for example, your terrain slices will all have the same dimensions (width = 200, length = 400 for example).

Set Using Prefabs

In general, this method will try to set the World Grid's data using the prefabs you created earlier. When selected, an additional option will appear, "Load From Resources Folder." If you're prefabs are in a folder called Resources, you can leave this option enabled. If not, disable the option and enter the relative folder path of where your prefabs are stored into the field "Relative Folder Path Where Prefabs Are Stored."

Again, this is a folder path relative to the main Assets folder, so if you're prefabs are in Assets/MyPrefabs, simply type /MyPrefabs. You can right click your folder and select Dynamic Loading Kit -> Copy Relative Folder Path, and then Edit -> Paste (or Ctrl-v) the path into the field. The prefabs must be in the correct format in order to be found.

Before pressing the “Set Data” button, you should understand the differences that exist in how the “Set Using Prefabs” method works, depending upon the Cell Object Type and World Type you selected previously.

When the Cell Object Type is Unity Terrains

The program will look for every prefab, starting with GroupName_1_1 (or GroupName_1_1_1 for 3D worlds), and ending with GroupName_Rows_Columns (or GroupName_Layers_Rows_Columns for 3D worlds).

If a prefab is not found, the corresponding grid location is marked as empty. If a prefab is found, the following actions are taken:

1) The corresponding grid location is marked as not empty.
2) If the row length of the row the terrain is on has not been set yet, the program retrieves the length of the terrain using terrainData.size.z and sets the row length equal to it.
3) If the column width of the column the terrain is on has not been set yet, the program retrieves the width of the terrain using terrainData.size.x and sets the column width to it.
4) After the program has attempted to retrieve every prefab, it checks to make sure every row length and column width has been set. If a row length has not been set (in the case where an entire row of terrains is missing), the row length is set to “Default Row Length.” If a column width has not been set, the column width is set to “Default Column Width.”

When the Cell Object Type is Non Terrain With Renderer

In this case, the same procedure used in “When the Cell Object Type is Unity Terrains” is used, with the following differences:

1) When the World Type is set to “Two Dimensional using XZ Axes,” renderer.bounds.size.x is used instead of terrainData.size.x, and renderer.bounds.size.z is used instead of terrainData.size.z.
2) When the World Type is set to “Three Dimensional,” in addition to the method used in 1), renderer.bounds.size.y is used to set a layers height.

3) When the World Type is set to “Two Dimensional using XY Axes,” row lengths are set with renderer.bounds.size.y instead of renderer.bounds.size.z. Column widths are still set using renderer.bounds.size.x.

When the Cell Object Type is Non Terrain without Renderer

In this case, the program still searches the folder you’ve specified (or the Resources folder) for your prefabs, and uses them to set the corresponding grid locations as empty or not empty.

Notice however, that there is no way to infer the size of the objects, as they are not terrains and do not have a Renderer component. In this case, every row length is set to the “Default Row Length,” every column width is set to “Default Column Width,” and every layer height is set to “Default Layer Height.”

This functionality leads to an important part:

You do not need to set the default values to the exact dimensions of your objects.

Take the following figure, for example:

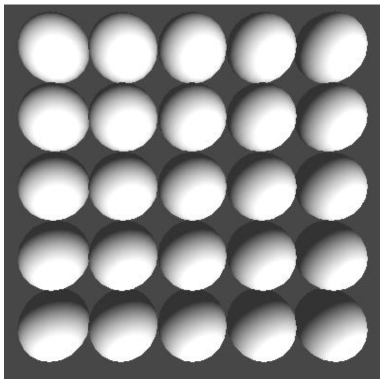


Figure 9: Default Values Equal to Sphere Dimensions

This World Grid is made up of a group of 5 x 5 x 5 spheres (aka, the spheres diameter is 5). The default values for this group were set to the same dimensions of the sphere (Default Row Length = 5, Default Column Width =5). The result is a world where the spheres sit side by side with no space between them. On the other hand, if we bump up the default values, to 10 and 10, the result is this world:

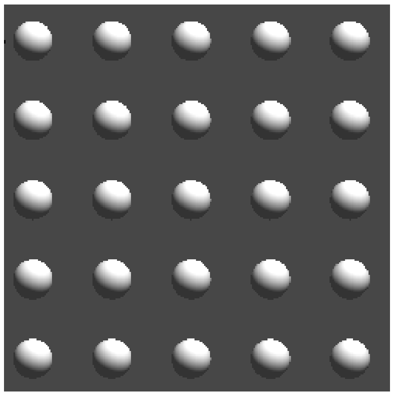


Figure 10: Default Values Larger than Sphere Dimensions

This is an incredibly powerful feature that allows you to fine tune exactly how your world will appear in the scene. Also note, even if your objects are terrains or non-terrains with a renderer component, you can set the Cell Object Type to Non Terrain without Renderer to get this functionality. Keep in mind, however, that doing so when using Terrains will disable runtime Terrain Neighboring, so only follow this path if you intend to add empty space between your terrains (in which case, the terrains do not need to be neighbored).

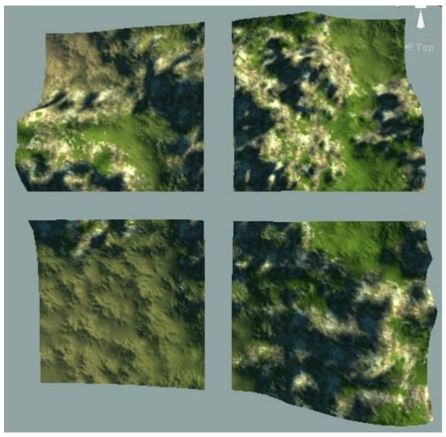


Figure 11: Terrain treated as non terrain without renderer, with default values set to terrain dimensions + 100

Once you have set your data, you will see one or two new Inspector Elements.

Offsets

Note: This option is not displayed when your Cell Object Type is Unity Terrain.

Each cell on your World Component's grid (which remember, can be endless), has a set position in world space. This position is measured at the bottom left position of the cell. When an object associated with that cell is loaded into the scene, by default it will be loaded at this position. For Unity Terrains, this is not a problem since a Terrain's position is measured at the bottom left most point on the terrain, so the terrain fit snugly into the cell.

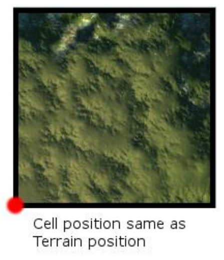


Figure 12

Non Terrain objects on the other hand often times have their position measured at the *center* of the object, which can result in unexpected behavior when using the Dynamic Loading Kit.

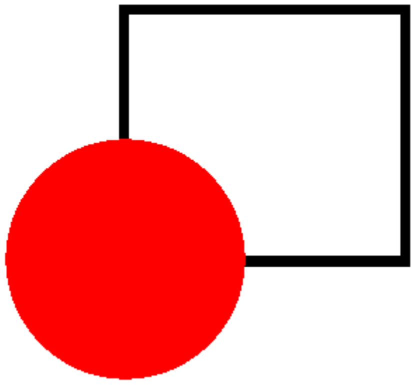


Figure 13: Object not completely in cell

In figure 11, you can see that the red object is not completely in the cell, because its position is measured at the center, while the cell's position is measured at its bottom left corner. In some cases, this behavior might be what you are looking for, in which case you can leave the position offsets at

0. On the other hand, if you want the object to be centered in the cell, you will need to adjust the position offset.

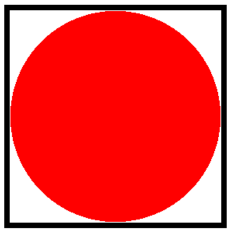


Figure 14: Object centered in cell

This is where Cell Offset % enters the picture. There are three cell offset percentages, one for each axis, though it's important to note that the offset for an axis will only show if the world type is using that axis. If using a 2D XZ world type, for example, the Y Cell Offset % will not be shown.

Each offset basically specifies at what position in the cell your objects should be loaded at. For instance, if using a 2D XZ world, an X offset of 0 specifies that no offset exist, and the object's position should be set to the cell's normal x position. An offset of 1 (100%), on the other hand, would specify that the object's position should be set to the cell's right most boundary. In other words:

$$Object\ X\ Position = cellPosition.x + (cellWidth * xOffset)$$

$$Object\ Z\ Position = cellPosition.z + (cellLength * zOffset)$$

$$Object\ Y\ Position = cellPosition.y + (cellLength * yOffset)\ for\ 2D\ XY\ Worlds$$

$$Object\ Y\ Position = cellPosition.y + (cellHeight * yOffset) for\ 3D\ Worlds$$

If the pivot point of your objects is in the center and you are looking to center the object in the cell, you will want to use a value of .5 (50%) for your offsets.

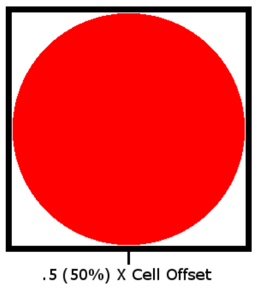


Figure 15: Object offset by .5 on the X axis

The reason a percentage is used rather than a fixed value is so that you can create a grid in which the rows have different lengths, columns have different widths, and/or layers have different heights.

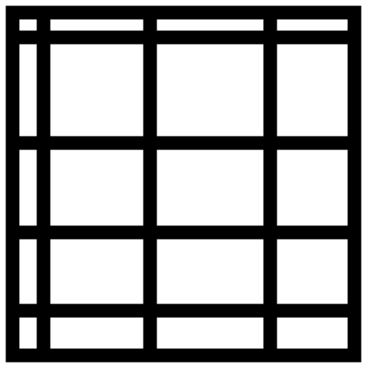


Figure 16: Valid World Grid

Notice in this grid how each row has a different value than every other row, and every column has a different width than every other width. This is a perfectly valid World Grid. The only requirement when creating your world grid is that all cells on a row have the same length, all cells on the same column have the same width, and all cells on the same layer have the same height (for 3D worlds).

Using a fixed offset for such a world, however, would cause issues. For instance, if we wanted to center an object with a centered pivot point in row 2, which we'll say has a length of 150, we'd need to provide an offset of 75. While this would work for centering objects in row 2, it would not work for row 1 (which is much smaller). In fact, it would not work for any of the other rows.

A percentage on the other hand, will work for every row/column/layer, regardless of their measurement.

Centering objects whose pivot point is at the bottom left point on the object is a bit trickier (at least, it is when the object is smaller than your cell; if the cell and object have the same dimensions, then you can use offsets of 0).

To do so, use the following formula (or equivalent formula for the other dimensions):

$$xOffset = \frac{\left(\frac{(cellWidth - objectWidth)}{2}\right)}{cellWidth}$$

While calculating the correct offset for other pivot points should be possible, it is beyond the scope of this document. I suggest using either a bottom left or centered pivot point, for simplicities sake.

Display Set Data in Log

After you set your data, you can press this button to have a formatted version of your data displayed in the console. You can use this functionality for debugging purposes, or if you just want to check and make sure your data was set/calculated correctly.

At this point, your World Grid should be fully operational and ready to go!

Section 3: Adding the Scene Components

The easiest way to get started with adding the necessary components required to get the Dynamic Loading Kit up and running is to create a preconfigured “Dynamic Loading Manager” Game Object.

There are several varieties of Managers available, all of which can be found under the following menu:

GameObject -> Create Other -> Dynamic Loading Kit

Each of these flavors of managers comes with the following components:

1. Component Manager
2. World
3. Active Grid
4. Player Prefs Persistent Data Controller
5. Boundary Monitor
6. A Primary cell Object Sub Controller
7. A Cell Object Loader

Each manager consists of a different combination of the last two components, which you can identify by the manager’s name.

For instance, the “Non-Pooling Prefab Instantiator” Manager comes with a Non Pooling Primary Cell Object Sub Controller and Prefab Instantiator Cell Object Loader, in addition to components 1-5 listed above.

Currently there are two varieties of Primary Cell Object Sub Controllers and three varieties of Cell Object Loaders (though one is a Unity Pro only feature), which makes for 6 different Managers.

Regardless of which manager you choose, you can always swap out one or both of these two components for another type later down the road.

At least one of each of the seven components is required for the dynamic loading kit to function properly (actually, the Boundary Monitor is not technically required, though you will most likely always need to use it).

You can create custom versions of the Primary Cell Object Sub Controller, Cell Object Loader, and Persistent Data Controller, and use the custom version rather than the one of the default ones. See chapter **Chapter 4: Advanced Topics** for more information.

If you need to swap out components, or wish to add a component manually, all components can be found under the following menu:

Component -> Dynamic Loading Kit

Keep in mind that each component usually has one or more *dependencies* on other components. For instance, the Component Manager component *depends on* the Persistent Data Controller component. That is, the Component Manager makes use of data or functions on the Persistent Data Controller. As such, there is a Persistent Data Controller field on the Component Manager's inspector that cannot be null.

If you created a Manager with all components already attached, these dependencies are automatically set up. If you did not do this, or if you remove/add components, you will see a warning message below the dependency fields in the inspector.

You can find a detailed listing of the various dependencies at the end of this section.

Component Manager

The Component Manager manages the creation, destruction, and setup of all Active Grid and World components in the scene. It also handles the saving of Active Grid persistent data by interacting with the Persistent Data Controller. While this component is not technically required, not having it would make it much more difficult to set up your Dynamic Loading, and there really is no reason not to use it.

Component Manager ID

An integer ID that uniquely identifies this Component Manager. Because there should only be a single Component Manager in each scene, you should not need to change this value from the default (0), though you can if you want to.

Initialize On Awake
When enabled, the component manager will be initialized during Awake/Start. Initialization will cause all Active Grids/Worlds to be initialized, all persistent data to be automatically found/incorporated (if not done manually), and all cell objects that need to be loaded to do so. Generally speaking, you should leave this option enabled. You should only need to disable it in the following situations:

1) If you need to load persistent save data via LoadSaveData and you're not able to do so before the Component Manager's Awake method is called.
2) If you need to ensure initialization occurs over a series of frames (using InitializeGradually) rather than two frames (default method) for performance reasons.

Suppress Warnings
If enabled, various warning messages on the Component Manager, Worlds, and Active Grids will be printed to the console. It's a good idea to leave this enabled until you have a thorough understanding of how the kit works.

Use Custom Save/Load Solution
When enabled, the persistent data controller will not be used by Component Manager. If you wish to save/load persistent data, you will need to use the GetSaveData and LoadSaveData methods on the ComponentManager class.

Auto Save Data On Game Exit
When enabled, this option will cause all persistent data in the scene (which includes the Component Manager's data and every Active Grid in the scene's data) to be saved when the game ends. This is only displayed and used when "Use Custom Save/Load Solution" is disabled.

This functionality is really meant for debugging purposes, and should not be used in your actual game. Instead, you will probably want the player to control when saving occurs, and/or have some sort of periodic auto save system set up.

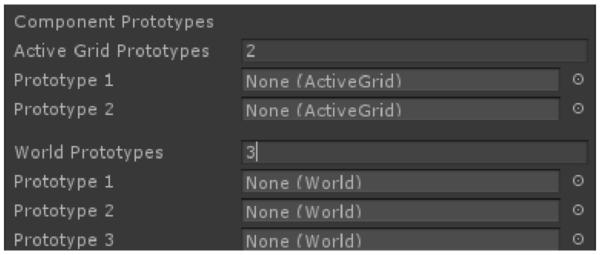
Persistent Data Controller
The Persistent Data Controller component that should be used to save and load persistent data. This is only displayed and used when "Use Custom Save/Load Solution" is disabled. If this is the case, this field cannot be left blank.

Component Prototypes
Component Prototypes can be used to create Active Grids and Worlds during your game. They are necessary in order to allow runtime created Active Grids and Worlds to persistent between game sessions.

To understand why, think of a World Component. As mentioned previously, each World Component requires a World Grid Scriptable Object Asset (as well as some other components). Creating a World in game would be easy; we'd just need to set the required World Grid and other components after creating it, as well as adjust its settings to our liking.

From one session to another, however, there is a problem. If we have some script which automatically recreates Worlds from the previous session (as our Component Manager does), it won't know which World Grid (or other components) the World was using, or how to get them. Prototypes take care of this problem, because they define the base settings and dependencies that all Worlds (and Active Grids) created in game use. By storing which prototype each World/Active Grid was created from, we can recreate these components between sessions.

To add prototypes, simply change the default of the prototype you wish to add (World or Active Grid) from 0 to however many prototypes you need. Then drag the prototypes onto the fields which now appear in the inspector.



Component Prototypes	
Active Grid Prototypes	2
Prototype 1	None (ActiveGrid)
Prototype 2	None (ActiveGrid)
World Prototypes	3
Prototype 1	None (World)
Prototype 2	None (World)
Prototype 3	None (World)

Figure 17: Component Prototypes in the Inspector

Prototypes must exist in the scene and must be attached to a deactivated game object.

World

The World component controls how your terrain/object group (represented by a World Grid) appears in game. As such, it contains some options which control positioning and loading functionality, as well as some asset/component dependency fields.

World ID

An integer ID that unique identifies this World component. Every World in a single scene should have a unique ID, so the first thing you'll want to do if using multiple Worlds is to change this ID.

Load Cushion

The World works by taking in request to add or remove users from a specific cell on the World. These request are processed so that duplicate request are combined and/or competing request are removed (add and remove requests on the same cell cancel each other out).

If load cushion is set to 0, the World will automatically move on to refreshing the World (aka, adding/removing cell objects associated with new/defunct cells) once it finishes processing add/remove requests. If any additional requests come in during the refresh phase, they are processed after the refresh is completed (which can take a couple of seconds).

A load cushion of greater than 0 will tell the World to wait x seconds (where x = load cushion) after processing add/remove request, and then check to make sure there are no additional add/remove request before moving on to the refresh phase. To see why this is useful, imagine this scenario:

1. Player crosses right boundary of Active Grid, initiating an East Grid Shift. Cell_1_1 is now obsolete and the object associated with it is removed from the scene.
2. Immediately after crossing the right boundary, the player moves west, crossing the West Boundary (which is the same as the previous East Boundary). Cell_1_1 is now being used, so immediately after unloading the object associated with it, the object is reloaded into the scene.

In this scenario, the Cell_1_1 remove request is processed, and while the world is being refreshed, a Cell_1_1 add request comes in, and so the world must be refreshed again after it is refreshed the first time. The world is refreshed twice (which impacts the frame rate) even though the end result is that there is no change in the world.

In this and other similar scenarios, a load cushion can be used to limit the number of world refreshes, by creating a window in which other sources can add their request to the request queue. Since these requests are compared with existing requests, this allows similar requests to be combined and competing requests to be removed. For instance, in the previous scenario, the Cell_1_1 remove request and Cell_1_1 add request would cancel each other out, and no world refresh would take place.

In general, larger load cushion values are better, though obviously it will depend on how quickly you need your objects to be loaded. If you're player moves very fast, for instance, you may not be able to use a large load cushion, since it would slow the loading of your objects too much.

Are Layers Endless

If enabled, the layers of the world will repeat endlessly. Note this this option is only used when the World Grid associated with the World has a World Type of Three Dimensional.

Are Rows Endless
If enabled, the rows of the world will repeat endlessly.

Are Columns Endless
If enabled, the columns of the world will repeat endlessly.

If any of your axes are endless, it's a good idea to set the world so it stays centered about its origin. Otherwise, floating point inaccuracies will probably become an issue!

Use this Game Objects Position as World Origin
If enabled, the World Origin will be set to the position of the game object this component is attached to. Note that the world origin is fixed mid-game, i.e. changing the position of the game object will not change the world's origin mid game.

World Origin
This option is only shown when "Use this Game Objects Position as World Origin" is disabled. The World Origin is the position of the first cell (1_1 or 1_1_1) in your World.

Keep World Centered Around Origin
When checked, this option ensures the World will stay centered around its origin. It does so by having one or more Active Grid's synced to the World monitor a set of boundaries, and when the player crosses one of those boundaries, shifting the entire World back towards the origin.

Distance of East and West Boundaries from Origin
The distance of the X Axis boundaries from the World's origin.

Distance of North and South Boundaries from Origin
The distance of the Z Axis (when using a 2D XZ or 3D world) or Y Axis (when using a 2D XY world) boundaries from the World's origin.

Distance of Top and Bottom Boundaries from Origin
The distance of the Y Axis boundaries from the World's origin. This is only used when the World is using a 3D World Grid.

Primary Cell Object Sub Controller
The primary cell object sub controller component that this World should use. This component controls some behavior relating to the objects associated with the cells on your World (currently, this behavior only relates to whether the objects are pooled). This field cannot be null.

World Grid
The World Grid that should be used for this World. Simply drag the World Grid asset you created earlier from your Project Hierarchy onto this field. This field cannot be null.

Special Note*
A World or game object it is attached to should only be disabled if it is being used as a prototype. If you need a World that will only be used at specific times, instead of enabling/disabling it when needed/not needed, you should create a prototype for that World and create/destroy it instead (via the Component Manager).

Active Grid

The active grid represents the collection of cells that are currently "active" in the scene. It sends this information, as well as changes to its makeup, to whatever world it is synced to.

Grid ID
An integer ID that is unique to this Active Grid component. Every Active Grid in a single scene should have a unique Grid ID.

Player
The Transform component of the player you want to associate with this Active Grid. While this field is optional, to make use of the actual "Dynamic Loading," you will need to provide a player. Otherwise you can manipulate the grid manually via function calls.

This transform is the transform whose position will be tracked by the Boundary Monitor to determine when a boundary of the Active Grid is crossed, so make sure you set it to the game object whose position is actually moving when you move your player.

Boundary Monitor
The Boundary Monitor which will track the Player's position to determine if a boundary of the Active Grid is crossed.

A single Boundary Monitor can be used with multiple Active Grids, so you should only ever need one Boundary Monitor in each scene (though you are free to add more if you wish: for instance, if you need monitors with different detection frequencies).

Player Mover
A component which is responsible for moving the player when moving the player is required. The reason this field exist is that in some cases, moving the player needs to be done in a specific way to avoid the player falling through the world.

Keep in mind; **you only need to fiddle with any of the following if you know the player will need to be moved by the Active Grid.** When the grid is shifted as a result of a World Shift, for instance, the player needs to be moved along with the grid. If you don't plan on using world shifting or any of the manual method calls that require player movement (see **API**), you can leave the Player Mover field blank and skip the next section.

If no Player Mover is provided, the player is moved via its transforms position. If the player needs to be moved 100 meters to the right, for example, it is done like so:

$Player.transform.position += \ new\ Vector3(100f, 0f, 0f);$

Or if the player's position needs to be set to some new position, it is done like this:

$Player.transform.position = new\ Vector3(20f, 0f, 320f);$

Depending on your character movement system, this method may or may not cause issues. When using Rigidbodies, for instance, I have not run into any issues, though you should do your own testing to verify this. CharacterController's, on the other hand, will not play nicely with this method. You will need to experiment or research to find out if your system is compatible with transform positional changes.

If you find yourself needing a Player Mover component, create a new script or modify an existing one to derive from PlayerMover. Implement the following two methods to handle the two types of player movements.

public sealed override IEnumerator<YieldInstruction> MovePlayerByAmount (Transform player, Vector3 amountToMovePlayer)

public sealed override IEnumerator<YieldInstruction> MovePlayerToPosition (Transform player, Vector3 position)

Note that because this is using the generic enumerator, you will need to add a using or import statement for System.Collections.Generic. These methods are coroutines because sometimes in order to move the player without having him fall through the world, you must do so over several frames.

You must also override the PlayerMoved property, which should return a bool value indicating whether the player's position has been changed yet by either of your two methods. This property is queried by the Active Grid using the Player Mover while either of the two coroutines are running, and is needed to ensure the player's position is saved accurately.

There are two special cases.

a) You change the player’s position immediately when the coroutines starts, i.e., before the first yield return statement. In this case, have the property always return true.
b) You immediately exit the coroutine after changing the player’s position (ex: Player.transform.position = position; yield break;). In this case, have the property always return false.

Otherwise, the PlayerMover property should initially return false, and only after you modify the player’s position should it return true.

I have provided some sample scripts which show how to implement the PlayerMover class.

They can be found in the following folder: *TerrainSlicing/OtherScripts/DynamicLoadingScripts/PlayerMoverControllers*

These scripts replace the default “FPSInputController” and “CharacterMotor” scripts provided by Unity (note that the PlayerMoverCharacterMotor script is exactly the same as the original CharacterMotor script, except is has been set to make use of PlayerMoverFPSInputController rather than FPSInputController).

The PlayerMoverCharacterMotor script is the one that derives from PlayerMover. Here is a sample of one of the method implementations outlined above:

```
public sealed override IEnumerator<YieldInstruction> MovePlayerByAmount
(Transform player, Vector3 amountToMovePlayer)
{
	player.position += amountToMovePlayer;
	freezePlayer = true;
	if (waitTillFixedUpdate == null)
	   waitTillFixedUpdate = new WaitForFixedUpdate();

	yield return waitTillFixedUpdate;
	freezePlayer = false;
}
```

In this example, the player’s position is immediately adjusted by the set amount, but then freezePlayer is set to true. In the UpdateFunction of this same script, there is this bit of code:

```
void UpdateFunction ()
{
	if(freezePlayer)
		return;

	//Rest of code ...
}
```

So by freezing the player, we effectively disable player movement. We then wait for one Fixed Update frame to pass, and then re-enable player movement by setting freezePlayer to false. Note that this script meets the criteria of special case a listed above, so we can have the property always return true.

If you are currently using the FPSInputController script, you can replace it (along with the CharacterMotor script) with the provided PlayerMoverFPSInputController and PlayerMoverCharacterMotor scripts.

If using a Character Controller but not the FPSInputController script, use the PlayerMoverFPSInputMotor script as a guide in implementing your own solution. Remember, the key is to change the player’s position, then freeze movement until a fixed update.

If using some other method for controlling your character, you will need to figure out how best to implement the PlayerMover methods.

Defaults
The rest of the Active Grids fields make up a set of default values that will only be used when no persistent data exist for the Active Grid in question. This means that if persistent data exists for the Active Grid, changing these values between sessions in the editor will have ***no*** effect, until you clear the persistent data via the Persistent Data Controller.

World
The world this Active Grid should begin synced to. You can leave this blank and sync to a world during runtime (see API)

Grid Dimensions
The dimensions of the active grid.

Inner Layers
The number of layers in the inner load are of the grid (refer to **Chapter1 Section 3** for an explanation of inner vs outer load area). Note that this field is only used when the World this Active Grid is synced to uses a World Grid with a World Type of Three Dimensional.

Inner Rows
The number of rows in the inner load area of the grid.

Inner Columns
The number of columns in the inner load area of the grid.

Outer Ring Width
The number of cells to pad around the inner load area.

Index of First Cell in Grid
The index of the first cell in the Active Grid (aka, the bottom left most cell). Note that these values are only used when no Player is present on the grid. If a player is present, then the Active Grid is constructed so that the player

is in the bottom left most cell of the inner portion of the Active Grid. Also note that Layer Index is only used when using a 3D world.

Cell Objects Enabled
Should cell objects be enabled for this Active Grid? If disabled, the Active Grid will not send add/remove request to the World it is synced to when it shifts or is reset. You will most likely want to leave this option enabled.

Monitor Inner Area Boundaries When Synced
Should the inner area boundaries of the active grid be monitored while the grid is synced to a World?

Allow Grid to Force World Shifts
If checked, when this Active Grid is synced to a World set to stay centered around its origin, the player associated with this grid will be monitored and if they cross the World's shift boundary (defined by the World), the World will be notified and a world shift will occur. It is not recommended to have multiple Active Grids able to shift the same World at the same time.

Special Note*
A World or game object it is attached to should only be disabled if it is being used as a prototype. If you need a World that will only be used at specific times, instead of enabling/disabling it when needed/not needed, you should create a prototype for that World and create/destroy it instead (via the Component Manager).

Player Prefs Persistent Data Controller

This component is default implementation of the Persistent Data Controller class (though you are free to make your own). In general, the Persistent Data Controller controls the manipulation (saving, loading, and deletion) of persistent data.

The Player Prefs version does this by saving persistent data to Player Prefs.

Scene ID
This is a unique string that identifies this Persistent Data Controller. It's important that every Persistent Data Controller in your project has a unique ID (note, this behavior differs from the other IDs, which only needed to be unique from other IDs in the same scene).

The scene ID is used to form part of every Component Manager and Active Grids Player Prefs key, like so:

Component Manager Key = SceneID_CM_ComponentManagerID

Active Grid Key = SceneID_AG_ActiveGridID

"CM" and "AG" are used as identifies which allow Active Grids and Component Managers to share the same ID.

Data Clearing
There are several data clearing methods which allow you to remove various persistent data. Select an option in the "Clear Method" field and then hover over it with your mouse to see more information about each option.

Boundary Monitor
The Boundary Monitor is responsible for monitoring the boundaries of the Active Grid (both reset boundaries and inner area boundaries). It has only one option, **Detection Frequency**, which simply specifies how often (in seconds) the boundary monitor will check the player's position to see if a boundary was crossed.

Primary Cell Object Sub Controller
Every primary cell object sub controller component has three fixed fields.

Cell Object Loader
The cell object loader component that should be used to load new cell objects into the scene. This field cannot be null.

Cell Object Destroyer
This is an optional component which can be added to gain more control over how cell objects are destroyed. If left blank, cell objects and any children on the objects will be destroyed in a single frame.

Post Destroy Yield Time
The amount of time (in seconds) to yield for after destroying any object. You will need to play around with this to find the optimal value for your game. A value of 0 will result in multiple Destroy calls in a single frame, and thus is not recommended.

Use Cell Actions
Cell Actions are an **Advanced Topic**, but in short, they are special MonoBehaviour's that can be attached to your cell objects, which contain methods that can be executed after a cell object is loaded and/or before a cell object is deactivated (note that deactivated does not necessarily mean destroyed. A pooling primary cell object sub controller does not destroy cell objects in some cases, for instance, but the object is still "deactivated").

If you are not using Cell Actions, please leave this option disabled. Enabling the option will create some garbage when no cell actions are actually present on the objects.

Max Pool Objects
This option is only available on the Pooling Primary Cell Object Sub Controller. It specifies the maximum number of objects that can be stored in the pool for each cell. If an object is added to the pool for a cell that already has reached its maximum limit, that object is destroyed instead of pooled.

Note that this option is only useful when using an endless world; otherwise there will always only be one object in the pool for each cell.

A higher value may improve performance, but will also increase memory usage.

Prefab Instantiator

The prefab Instantiator component loads new cell objects into the scene via Instantiate(Resources.Load(prefab)); It has a single option, **Instantiate Frequency**, which specifies the amount of time (in seconds) to wait between Instantiate calls. Larger values will improve performance at the cost of loading speed, while a value of 0 will result in all Instantiates occurring in a single frame (not recommended).

Scene Loader

The scene Loader uses the Application.LoadLevelAdditive method to load scenes non-asynchronously in an additive manner (existing scenes are not destroyed).

Unfortunately, the root object in each scene must be in an activated state in order to efficiently be found by the Scene Loader component. This can cause issues, since when the root object is first loaded it may not be in the correct position, and the player might see the object(s) out of position. In order to avoid this problem, layer hiding can be used. More information can be found in Chapter 4, Section 3.

Time to Wait Between Loads

This is similar to the Prefab Instantiator's "Instantiate Frequency" option, and specifies the amount of time (in seconds) to wait between scene loads.

Unbound Object Tag

A tag you should have set up when using the Scene Creation tool. All of your root cell objects in each scene should have this tag by default. Once they are loaded and assigned to a cell, they are set to "Untagged."

Are Scenes Fixed
Are the scenes the scene loader component will be loaded fixed/static? Only check this option if you know the root objects in the scenes will never need to be moved from their default positions. Refer to Fixed/Static Scenes for more information.

Use Layer Hiding (Legacy)
Enable if using the legacy layer hiding system. Note that with update 3.0.3, this method is not recommended. Better performance can be achieved by utilizing a deactivated node. See Chapter 4, Section 3 for more info.

Visible Layer
This is the layer the root object will be set to (after being loaded) once it has been found and positioned correctly. The Default Layer is usually sufficient, though you are free to use whatever layer you wish.

Async Scene Loader (Pro Only prior to Unity 5)

The Async Scene Loader is very similar to the Scene Loader, except it loads the scenes asynchronously via the Application.LoadLevelAddtiveAsync method. Like the Scene Loader, it contains the "Unbound Object Tag" and "Visible Layer" options, which act in the same manner as the Scene Loader component. In addition, it has the **Load Priority** option, which specifies how Unity should priorities the loading of the scenes. A value of "High" will tell Unity to devote more resources to loading the scenes, resulting in quicker loads at the cost of performance, while a value of "Low" will do the opposite.

For more information on setting up your scenes properly for the Scene Loader components, please refer Chapter 4, Section 3.

Simple Cell Object Destroyer

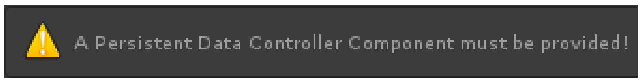
This is a simple implementation of the Cell Object Destroyer abstract class. This destroyer destroys a specific number of child objects on a cell object each frame, and then finally destroys the root cell object after all child objects have been destroyed. This destroyer only goes down one level on the child-parent hierarchy. Any children of the first class of children are destroyed when the first class child is destroyed, so if you have a large number of sub children, it will probably be a good idea to create a custom cell object destroyer.

Children To Destroy Per Frame

The number of children to destroy each frame. A lower number will result in better performance, but will increase the time it takes for objects to be destroyed (thus slowing the dynamic loading speed).

Dependencies

This sub-section contains a listing of the various dependencies that exist between the various components. Look for warning messages on your components for missing dependencies.

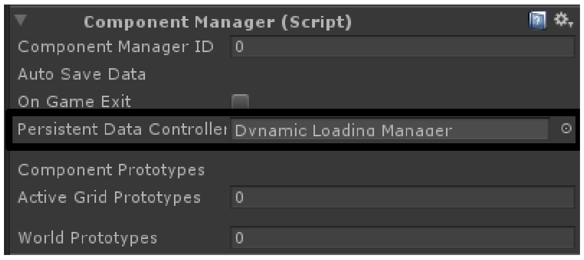


A Persistent Data Controller Component must be provided!

Figure 18: Example Missing Dependency Warning Message

Component Manager

The Component Manager has a single dependency on a Persistent Data Controller component.



Component Manager (Script)
Component Manager ID 0
Auto Save Data
On Game Exit
Persistent Data Controller Dynamic Loading Manager
Component Prototypes
Active Grid Prototypes 0
World Prototypes 0

Figure 19: Component Manager Dependencies

World

The World has two dependencies, one on a Primary Cell Object Sub Controller and one on a World Grid Scriptable Asset (not a component, you should have created this in your project hierarchy).

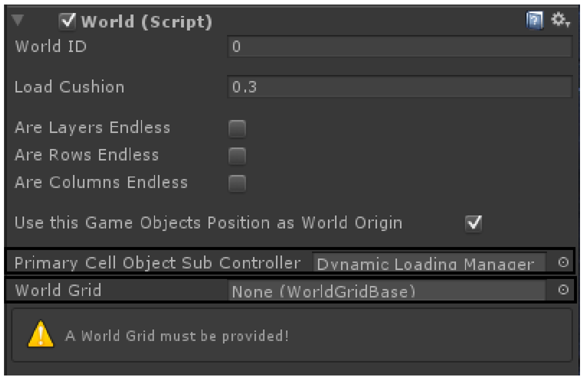

World (Script)
World ID 0
Load Cushion 0.3
Are Layers Endless
Are Rows Endless
Are Columns Endless
Use this Game Objects Position as World Origin ✓
Primary Cell Object Sub Controller Dynamic Loading Manager
World Grid None (WorldGridBase)
A World Grid must be provided!

Figure 20: World Dependencies

Active Grid

The Active Grid has several dependencies, though not all of them are technically required to be filled.

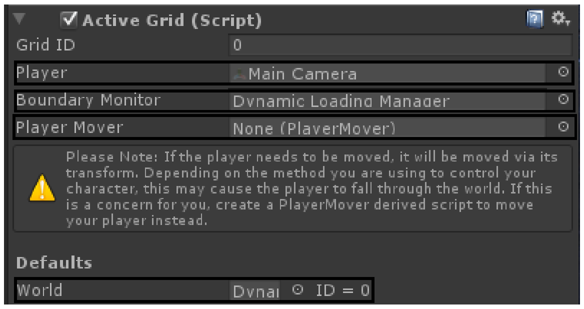

Active Grid (Script)
Grid ID 0
Player Main Camera
Boundary Monitor Dynamic Loading Manager
Player Mover None (PlayerMover)
Please Note: If the player needs to be moved, it will be moved via its transform. Depending on the method you are using to control your character, this may cause the player to fall through the world. If this is a concern for you, create a PlayerMover derived script to move your player instead.
Defaults
World Dynai ID = 0

Figure 21: Active Grid Dependencies

Player

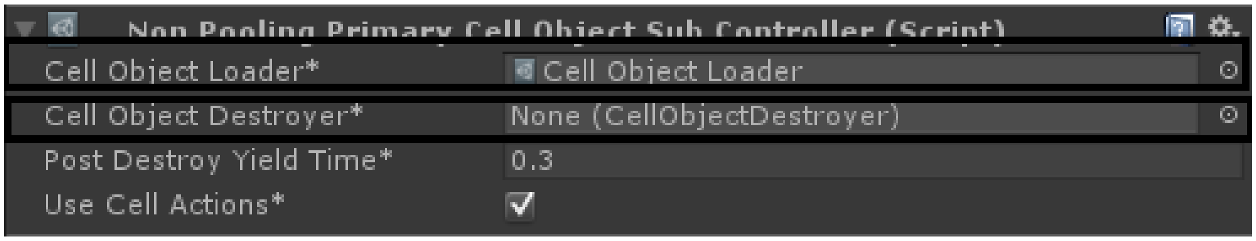
This is the transform of the player you wish the Active Grid to be associated with. The Active Grid is not required to be associated with a player, though if you wish to make use of true dynamic loading, you will need to fill this dependency.

Boundary Monitor
The Boundary Monitor that will be used to track the transform of the player (provided in the "Player" field). This field is only displayed when a player has been provided. If you wish to have the Active Grid dynamically shift/reset depending on the player's location, you will need to fill this dependency.

Player Mover
A Player Mover that will be used to move the provided player when necessary (when resetting the world and in other instances). A default PlayerMover that simply moves the player by adjusting its Transform.position will be used if this dependency is not filled. Consult Chapter 2, Section 3, Sub-Section Player Mover for more information.

World
Default World the Active Grid will be synced to on Play. This dependency should usually be filled, though in some instances it may not be required.

Primary Cell Object Sub Controller
The sub controller has two dependencies, one on a Cell Object Loader componentwhich must always be filled, and one on a Cell Object Destroyer component which is optional.



Cell Object Loader*	Cell Object Loader
Cell Object Destroyer*	None (CellObjectDestroyer)
Post Destroy Yield Time*	0.3
Use Cell Actions*	✓

Figure 22: Primary Cell Object Sub Controller Dependency

If a component has not been listed in this sub-section, than it has no dependencies.

Chapter 3: Upgrading to 2.0

Quick Note: If your version of the kit is greater than 2.0, you can skip this chapter.

Section 1: Cha-Cha-Cha-Changes

Bye-Bye 3.5

Perhaps the biggest change to v2.0 of the kit is the elimination of Unity 3.5 as a viable platform. The reason for this change was simple. Unity 3.5 was never fully compatible with the kit in the first place, which I only recently discovered while working on v2.0.

This incompatibility stems from the Cell Object Loader components. The Prefab Instantiator component requires the prefabs used to be disabled in the project hierarchy, which in Unity 4.x is not a problem. Unfortunately, I didn't realize enabling/disabling prefabs in the project hierarchy was a feature added in 4.0, and thus in Unity 3.5 the Prefab Instantiator component is useless.

The alternative is to use one of the Scene Loader components; however these components also have their own issues in 3.5. They each make use of the Application.LoadLevelAdditive method to load objects in Awake/Start so that the objects are visible before the first Update is called. In Unity 4.x, this works fine, as this method always takes exactly one frame to work. In Unity 3.5, however, this method takes several frames, making it less than ideal.

With that in mind, I suggest 3.5 users simply disregard update 2.0. If you MUST have the upgrade (which I can't blame you for, it contains a host of awesome new features), please contact me. While the prefab Instantiator component will never be useable in 3.5, I can construct a different version of the kit that uses a modified Scene Loader component. It won't be a great solution, but it can be done (with some work on each of our parts).

Other Changes

Version 2.0 of the Terrain Slicing & Dynamic Loading Kit introduces several new components, all designed around increasing the flexibility and usefulness of the kit. These components include the World, Active Grid, and Component Manager.

Dynamic Loading Configuration Form Removal

The introductions of these components have an unfortunate side effect, however; the Dynamic Loading Configuration Form (DLCF) Component is now obsolete. 99% of the functionality inherent in the DLCF can now be found in the World and Active Grid components.

World Grid Reset

In addition, the code for the World Grid class was moved to the DynamicLoadignKit.dll, and this change has the unintended side effect of clearing any data you have set. To be clear, your existing World Grid assets will still be useable, HOWEVER, you will need to reset the data before the grid can be used.

Persistent Data Obsolete

A change to how persistent data is saved has made old persistent data redundant. If this causes issues for you, contact me, as there should be a way to create a converter that takes the old data and converts it to the appropriate format.

Secondary Cell Object Sub Controller Removal

With v2.0, Secondary Cell Object Sub Controllers have been removed. I very much doubt that anyone is actually using these advanced components (they require you to create them yourself via scripting), but if you are using them, you should be able to recreate the functionality via other methods (described below).

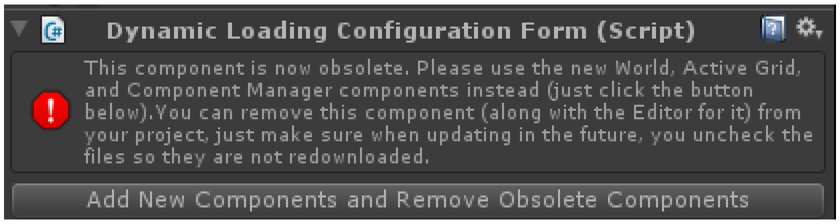
Non Endless World Resetting Removed
Finally, some changes have also removed the ability to keep your non-endless world centered about the origin (by resetting it). You can still use a similar option when using an endless world, and for those users, refer to the **Active Grid** portion of **Chapter 2, Section 3**.

The Active Grid Resetter component has also been removed. Again, see **Chapter 2, Section 3 – the portion on configuring the Active Grid** for details on how to get your rest behavior back up and running.

For those who require there non-endless world to stay centered about the origin, I am currently looking into a solution. For now, I suggest avoiding this update!!

Step-By-Step Upgrade

1) The first thing you will notice is that your Dynamic Loading Configuration Form component is still in your scene, but its Inspector Editor has been replaced with the following:


Dynamic Loading Configuration Form (Script)

This component is now obsolete. Please use the new World, Active Grid, and Component Manager components instead (just click the button below).You can remove this component (along with the Editor for it) from your project, just make sure when updating in the future, you uncheck the files so they are not redownloaded.

Figure 23: Configuration Form After Upgrading to v2.0

2) Clicking the button "Add New Components and Remove Obsolete Components" will add a World, Active Grid, and Component Manager component to the same GameObject the DLKF is on, as well as remove the DLCF and Active Grid Component if one is present (this component is no longer used).

3) Pre 2.0, component dependencies (which components use other components) were automatically set up internally. This made things simple, but also required all components to be on the same Game Object. In 2.0, dependencies are explicitly defined via fields on the Components. This allows you to put components on separate game objects, but also means you need to ensure all dependencies are taken care of.

If your secondary components (Boundary Monitor, Cell Object Loader, etc.) were on the same Game Object as your DLCF, then these dependencies will be set up automatically when you click the button in Step 2. Otherwise, you will need to set them up yourself.

a. Component Managers requires a reference to a Persistent Data Controller.
b. World requires a reference to a World Grid Scriptable Object Asset.
c. World requires a reference to a Primary Cell Object Sub Controller.
d. Primary Cell Object Sub Controller requires a reference to a Cell Object Loader.
e. Active Grid requires a reference to a World (in 99% of cases).

Optional dependencies also exist.

a. Active Grid can take a reference to your Player's Transform component. While this is optional, in order to use the "Dynamic" functionality, you'll need to set this, as well (b) below.
b. Active Grid can take a reference to a Boundary Monitor component. Required for Dynamic Loading functionality. Field only shows up if a Transform is provided (1).
c. Active Grid can take a reference to a Player Mover component. See the section on "Player Mover" for more info. Field only shows up if a Transform is provided (1).
d. Active Grid can take a reference to an Active Grid Cells Creator component (advanced, see chapter on Advanced Topics).
e. World can take a reference to a Cell Object Master Controller component (advanced, see chapter on Advanced Topics).

4) When the button in Step 2 is pressed, most of the info from your DLCF (aka, the inspector values), will be interpreted and translated to new inspector values on the World and Active Grid components. For example, your grid dimensions (Inner Rows, Inner Columns, etc.) will be supplied to the Active Grid. At this point, it's a good idea to look through the new components and make sure you understand what each field means. Potentially confusing fields should have tooltips (place your mouse over the field name). If you need more info about a field, please look back to the chapter on Configuring the Dynamic Loading Kit.

5) Find your Player Prefs Persistent Data Controller. Change the Scene ID if it is set to the default value of "Make me Unique." This Scene ID needs to be unique (any other Persistent Data Controllers in your project should use a different ID).

   Also note that the DoNotSaveSession option has been removed from the Player Prefs Persistent Data Controller. End of Session saving is now disabled by default. You can enable it via the "Auto Save Data On Game Exit" option on the Component Manager component.

6) Find your World Grid asset in your project hierarchy. As noted earlier, you will need to reset the data. Please note however that the World Grid Inspector has changed. The new "World Type" field is self-explanatory, and you shouldn't need to fiddle with it. In fact, if using Unity Terrains, you cannot change it from the default "Two Dimensional using XZ Axes," since the X-Z plane is the only plane Unity Terrains can operate on. If not using Unity Terrains, and you'd like to look into created a 3D world, please see the Chapter titled "3D Worlds."

   There is also a new field, "Method to Set Data." By default, it is set to "Set Using Default Values." This method sets all of your row lengths to the default row length value, column widths to the default column width value, and layer heights to the default layer height (if using 3D worlds). It also sets the grid so all grid locations are not empty. If all of your terrain/objects have the same width/length, and you haven't removed any terrains/objects from your group, you can leave the method as "Set Using Default Values," and click "Set Data."

   If your terrain/objects have differing dimensions and/or you have missing terrains/objects from your group, you can use one of the other two methods.

   1) Set Using Prefabs works as it did before for the most part. If you're prefabs are in a folder called "Resources," leave "Load From Resources Folder" checked and press "Set Data." Otherwise uncheck this option and type the root Assets folder relative folder path where your prefabs are stored (ex: if in Assets/Terrains, type /Terrains).

2) If you don’t have access to your prefabs, but cannot use the Set Using Default Values method, you can also create a text asset with the necessary data. See the example.txt and example3DWorld.txt files in the project hierarchy (TerrainSlicing/OtherScripts/DynamicLoadingScripts) for examples on how to format this text document.

7) Once your data is reset, you should be good to go (assuming you’ve set up your scripts to your liking).

Chapter 4: Advanced Topics

The previous chapters provide enough information to get the dynamic loading kit up and running, and in many cases, that is all you will require. In some instances, however, you may have need for additional functionality.

Section 1: Scripting API

The scripting API provides an overview of all the public types in the Dynamic Loading Kit. There are a number of methods available that allow you to achieve more interesting results when using the kit. Create or destroy Worlds and Active Grids mid-game, manipulate an Active Grid's dimensions and settings at any time, or transport the player across worlds; all this and more is unlocked through the API.

You can find the API on my website, @ http://deepspacelabs.net/dynamic_loading/api_documentation/documentation_main.html.

Special Note* If you navigate to Edit -> Project Settings -> Script Execution Order in the Unity Editor, you'll see that the Component Manager's order is set to 500. It's important that any scripts which use the API are set to run **before** the Component Manager!

Understanding Internal Procedures

Before diving into the API, a firm understanding of the inner workings of the Dynamic Loading Kit is required.

Persistent Data Incorporation

Persistent data is data that persist after a gaming session has ended, and which can be reloaded during a future gaming session. Any generic save/load system uses persistent data. With the Dynamic Loading kit, there are three categories of persistent data.

1) Component Manager Data – Information about what persistent Worlds/Active Grids were created mid-game, and what inspector created Worlds/Active Grids were destroyed mid-game.
2) Active Grid Data – Information about a specific Active Grid component, such as the World it was synced to when the persistent data was saved.
3) World Data – Information about a specific World component, such as the Word's origin or Group Name.

You will primarily be dealing with persistent data via the Component Manager.

How

When using a Persistent Data Controller, all three categories of data are saved separately under different keys, and the data is loaded automatically when the component manager decides it needs to load the persistent data.

When using a custom Save/Load solution, you can use GetSaveData to get a string which contains all the persistent data, and then save that string using whatever method you prefer (to a file, for example). When you load your game save, you can then pass in this string via LoadSaveData. If no save data has been loaded via LoadSaveData and the component manager needs to incorporate persistent data, the component manager will assume there is no persistent data.

When

Understanding when persistent data is incorporated by the Component Manager is even more important than understanding how it is saved/loaded.

Data is incorporated when,

1) The Component Manage is initialized, either automatically if "Initialize On Awake" is checked in the inspector, or manually via the Initialize or InitializeGradually methods.
2) An Active Grid is created via the CreatePersistentActiveGrid or CreateNonPersistentActiveGrid method.

3) A World is created via the CreatePersistentWorld or CreateNonPersistentWorld method.
4) Data is loaded via the LoadSaveData method.

You may be wondering why persistent data is incorporated in situations 2 and 3. The reason is simple. When you create an Active Grid or World mid-game, a unique ID must be assigned to it. In order to keep this ID from clashing with the ID of a World/Active Grid created in a previous session, the Component Manager must first load all previously created Worlds/Active Grids (the information for which is contained in the persistent data).

This leads to a very important rule:
When using a Custom Save/Load Solution, always load existing save data before creating a World/Active Grid via one of the Component Manager's "Create" methods.

The fact that persistent data is incorporated when the Component Manager is initialized may also become an issue for you. In some situations, it may be impossible to load save data before the component manager's Awake method is called. In these situations, you will need to uncheck "Initialize On Awake" in the component manager's inspector, and manually initialize the component manager via the Initialize or InitializeGradually method.

Component Manager Initialization

Component Manager Initialization will occur in the following situations:

1) When "Initialize On Awake" is called, during the Awake/Start method of the component manager.
2) When Initialize is called manually.
3) When InitializeGradually is called manually.

In situations 1 and 2, the initialization process will take place over two frames (Awake/Start if possible). Depending upon your situation, this may cause a performance issue, in which case you should use InitializeGradually instead.

What does Initialization Entail?
During initialization, 1) any existing persistent data is incorporated (if not already), 2) all Worlds and Active Grids in the scene are initialized, 3) starting cell objects are loaded for inspector created Active Grids or Active Grids created during a previous session, and 4) Active Grids which are set to monitor their inner area have their Boundary Monitors started.

Initialization does not automatically load the starting cell objects for Active Grids creating during the current game session. You must load the objects manually!

Active Grid Initialization
An Active Grid must be initialized before most of its methods can be called. One notable exception, however, is with the PreInitialize_SetWorld method. This method must be used before a grid has been initialized.

An Active Grid is initialized when

a) it is created via CreatePersistentActiveGrid or CreateNonPersistentActiveGrid or
b) the Component Manager is initialized.

World Initialization
A world must be initialized before most of its methods can be called. Two notable exceptions include the PreInitialize_SetWorldOrigin and PreInitialize_SetGroupName methods, which must be used before the World has been initialization.

A world is initialized when

a) it is created via CreatePersistentWorld or CreateNonPersistentWorld,
b) an Active Grid which is synced to the World is initialized, or
c) the Component Manager is initialized.

Cell Object Loading
When you create an Active Grid via CreatePersistentActiveGrid or CreateNonPersistentActiveGrid, the cell objects for the grid will not be loaded

automatically. Instead, you must manually load them via TryLoadCellObjectsASAP, TryLoadCellObjects, or TryLoadCellObjectsAndWaitForLoadToComplete (all of which are methods of the ActiveGrid class).

TryLoadCellObjectsAndWaitForLoadToComplete is a Coroutine, and as such cannot be used from the initial Awake phase at the start of a new scene. This method is ideal when you need to load the cell objects in a non-performance heavy way, and when you need to wait on the objects to load.

TryLoadCellObjects also has a low performance penalty, but using this method will not allow you to be notified when the objects are fully loaded.

TryLoadCellObjectsASAP is meant to be used from another scripts Awake method before the Component Manager's Awake method has run. It's provided as a means of ensuring cell objects are loaded at the start of a scene before the first Update cycle runs (i.e. before the Payer sees the game view for the first time). In any other situation, calling this method will act as if you called *TryLoadCellObjects* instead.

On the other hand, once you manually load the cell objects, the cell objects will be automatically loaded at the start of the next gaming session (assuming there is persistent data and component manager finds it/you load it). The cell objects for inspector created Active Grids will also be loaded automatically by the Component Manager. Automatic loading occurs when the Component Manager is initialized.

Creating Active Grids/Worlds at Runtime

One of the most powerful features of the API is the ability to create Active Grids and Worlds mid-game. These components can be divided into two categories: non persistent and persistent.

Persistent Active Grids

Persistent Active Grids are grids which are saved with the Dynamic Loading Kit's persistent data. That is, when the Component Manager loads persistent data containing information on persistent Active Grid's, the grids are automatically recreated and their objects are automatically loaded when the Component Manager is initialized (assuming their cell objects are enabled).

Drawbacks

Persistent Active Grids are extremely powerful and useful, but they do have some limitations.

1) They cannot be synced to non-persistent Worlds. This ensures that when the Active Grid is recreated during a future session, the World it is synced to is still around.
2) You cannot assign a player to them at runtime.

The latter limitation is due to the fact that the Component Manager would not be able to reassign the same player to the grid when it is recreated.

Non-Persistent Active Grids

Non persistent Active Grids are not saved with the Dynamic Loading Kit's persistent Data. That is, when persistent data is loaded by the Component Manager, it's as if these grids never existed.

Advantages

The advantages to using non persistent active grids are the exact opposite of the drawbacks to using persistent Active Grids. That is, you can sync these grids to non-persistent Worlds (as well as persistent Worlds), and you can assign players to them at runtime (by passing the player into the CreateNonPersistentActiveGrid method).

The most common situation in which you'd need to use a non-persistent active grid is when you need to create your player during your game. This is very common when dealing with networking architecture, where you're required to use a special creation method which creates the player on each client.

Saving Data for Non-Persistent Active Grids

While it is not possible for the Component Manager to automatically save/load data for non-persistent active grids, you **can** do so manually. When your game is saved, simply get the Active Grid's persistent data via its GetPersistentStringSaveData method. When your game is loaded, recreate the non-persistent Active Grid via CreateNonPersistentActiveGrid, and pass in the string data retrieved from GetPersistentStringSaveData for the persistentDataToSetStateFrom argument.

Persistent and Non-Persistent Worlds

Like persistent Active Grids, persistent worlds are saved with the Dynamic Loading Kit's persistent data and recreated by the Component Manager when that persistent data is loaded. Non-Persistent Worlds are not. Other than that, both types of World's are effectively the same, except for the fact that persistent Active Grids cannot be synced to non-persistent Worlds.

Prototypes

In order to recreate a persistent World and/or Active Grid, there must be a mechanism in place to reassign the correct components to the World/Active Grid upon recreation (such as the Player Transform assigned to an Active Grid). Prototypes serve this purpose, in addition to providing a repository of base settings to use for the runtime created components (ex: is a World endless on its rows axis?).

A prototype is simply a World or Active Grid component attached to an in-activate game object in the scene. You assign prototypes in the Component Manager's inspector, and then pass in the prototype's number (in the inspector) when creating a World/Active Grid at runtime.

Destroying Active Grids/Worlds at Runtime

In addition to creating Active Grids and Worlds at runtime, it is also possible to destroy them. When you destroy a component, that change is reflected across gaming sessions (assuming you save/load the kit's persistent data).

In order for these changes to be tracked, however, you must destroy these components **exclusively** through the Component Manager. Manually calling GameObject.Destroy on an Active Grid and/or World is not allowed! This ensures cell objects are properly removed/destroyed, and with regards to Worlds, that any Active Grids synced to the World are properly de-synced when the World is destroyed.

Use DestroyWorld or DestroyWorldAndWaitForCellObjectsToBeRemoved to destroy a World, and DestroyActiveGrid or DestroyActiveGridAfterRemovingCellUsers to destroy an Active Grid.

Changing the World's Group Name

By default, a World will use the Default Group Name of the World Grid it is linked to. You can change this group name pre-game or mid-game however, which is very useful if you have alternate versions of your terrains/objects. If the World is persistent, any changes to this group name will also persistent between game sessions, so you don't have to set it every time. There are two methods which can be used to change the group name.

PreInitialize_SetGroupName

This can be used to change the group name of the world before the world has been initialized. This should be used if you want to make sure the starting objects for a world are reflective of this new group name.

ChangeGroupName

Changes the group name post initialization (during the game). You can optionally choose to force the World to refresh the objects it's using in order to load the objects using the new name. Keep in mind, however, that this process will not look pretty, so it should be used with a loading screen or some other technique that hides the game view from the user. Note: this is a Coroutine.

Changing a World’s Origin

You can change a world’s origin, but only before it has been initialized. This change will persistent between sessions on persistent World’s.

Section 2: Extending the Dynamic Loading Kit

If you find that you cannot achieve the results you wish through the core functionality of the Dynamic Loading Kit, you also have the option of extending the kits functionality through the creation of custom components and classes.

Cell Actions

Cell Actions are special components which can be attached to the objects that make up your World Grid. They can be used to perform an action or actions over one or more frames at two key times:

1. Immediately *after* a cell is “activated.” That is, after the object associated with a cell has been loaded and set to an activated state.
2. Immediately *before* a cell is “deactivated.” That is, before the object associated with a cell has been set to a deactivated state and sent to the primary cell object sub controller for processing.

An object can have 0 or more Cell Action components attached to it (it is perfectly fine to have a Cell Action on one object in your group but not another).

Creating a Cell Action

In order to create a Cell Action, you will need to create a custom class that derives from CellAction, and then implement one or both methods from the CellAction class depending on when you want your actions to occur. These two methods are DoStuffBeforeCellIsDeactivated and DoStuffAfterCellIsActivated.

In UnityScript, your class will look like this:

```
import DynamicLoadingKit;

public class CustomCellAction extends CellAction
{
	function DoStuffBeforeCellIsDeactivated(worldCell : IWorldCell)
	{
		Debug.Log("Goodbye " + worldCell.CellOnWorldGrid.ToString());
		yield;
	}

	function DoStuffAfterCellIsActivated(worldCell : IWorldCell)
	{
		Debug.Log("Hello " + worldCell.CellOnWorldGrid.ToString());
		yield;
	}
}
```

Note that in UnityScript, at least one yield statement is required in your function, even if you don't need it. In C#, you can use yield break if your action only takes a single frame, but the equivalent "return" statement in UnityScript will not work with my kit.

Here is the equivalent code in C#:

```
using DynamicLoadingKit;
using System.Collections.Generic;
using UnityEngine;

public class CustomCellAction : CellAction
{
	public override IEnumerator<YieldInstruction>
	DoStuffBeforeCellIsDeactivated(IWorldCell worldCell)
	{
		Debug.Log("Goodbye " + worldCell.CellOnWorldGrid.ToString());
		yield break;
	}

	public override IEnumerator<YieldInstruction>
	DoStuffAfterCellIsActivated(IWorldCell worldCell)
	{
		 Debug.Log("Hello " + worldCell.CellOnWorldGrid.ToString());
```

```
            yield break;
        }
}
```

Note that you do not need to implement both methods. If you only want to do something before a cell is deactivated, then don't include the DoStuffAfterCellIsActivated method.

IWorldCell

As you can see from the two examples above, you are passed an IWorldCell object in your method implementations. This object provides a host of useful information relating to the cell which was activated or is about to be deactivated. This includes:

1. A Cell object (contains a layer index, row index, and column index, though note the layer index is valid only for 3D worlds) which describes the cells index on your World Grid. This is implemented as a property called CellOnWorldGrid.
2. The position of the cell in world space (note that depending on your object offset, this may or may not be the same position of your object). This is implemented as a Vector3 property called Position.
3. The height of the cell. If the world is not three dimensional, this will always be 0f (property is called Height).
4. The Length of the cell.
5. The Width of the cell.

Enabling Cell Actions

By default, Cell Actions are disabled. This is because trying to access Cell Actions on objects that don't have Cell Actions (using GetComponent) creates unnecessary garbage. Cell Actions can be enabled via whatever Primary Cell Object Sub Controller component you are using, by checking the "Use Cell Actions" option.

<u>Custom Cell Object Loaders</u>

By default, the Dynamic Loading Kit comes packed with support for three different load methods. The Prefab Instantiator component makes use of the

Instantiate method, the Scene Loader makes uses of Application.LoadLevelAdditive, and the Async Scene Loader makes use of Application. LoadLevelAdditiveAsync.

If you wish to create a custom loading strategy, or add additional functionality to an existing loading strategy, you will need to create a custom Cell Object Loader component that derives from the CellObjectLoader class.

Note that the source code is provided as a zip file (if using 3.0.3), and you can find examples of how to implement existing loading strategies in the BaseSceneLoader.cs, SceneLoader.cs, AsyncSceneLoader.cs and PrefabInstantiator.cs files.

Game Object State

The Dynamic Loading Kit will automatically activate the primary cell object which you attach to the cell via the AttachCellObjectToCell method if it is not already activated. It will not mess with the state of any child objects, so make sure to activate whatever you need to before exiting the load/attach method. For instance, the included Scene Loader components activate the "Deactivated Node" object if found.

Cell Positioning

Previously, cell positioning was handled by internal classes. With update 3.0.0 the responsibility was shifted to the Cell Object Loader classes. When using prefab instantiation, ensure that you instantiate each object at its correct position (IAttachableWorldCell.CellObjectPosition).

If using scene loading or some other method, you can simply pass in false to the second argument of the IAttachableWorldCell.AttachCellObjectToCell method. This tells the cell that the object has not been positioned yet, so that it does so itself.

Example Cell Object Loader (this is the exact implementation of the Prefab Instantiator component)

```
using UnityEngine;
using System.Collections.Generic;

public sealed class PrefabInstantiator : CellObjectLoader
{
	[SerializeField]
	internal float timeToYieldBetweenInstantiates = .3f;

	YieldInstruction yieldForTime;

	public sealed override bool IsSingleFrameAttachmentPreloadRequired
	{
		get { return false; }
	}

	void Awake()
	{
		yieldForTime = timeToYieldBetweenInstantiates > 0f ? new
		WaitForSeconds(timeToYieldBetweenInstantiates) : null;
	}

	public sealed override void AttachCellObjectsToCellsInSingleFrame<T>
	(List<T> cells, loaderID)
	{
		CellString cellString = RegisteredUsers[loaderID].CellString;
		foreach(T cell in cells)
		{
			cellString.MatchStringToCell(cell.CellOnWorldGrid);
			string objName = cellString.ToString();
			GameObject obj = (GameObject)GameObject.Instantiate (
						Resources.Load(objName),
						cell.CellObjectPosition,
						Quaternion.identity);

			obj.name = objName;
			cell.AttachCellObjectToCell(obj, true);
		}
	}
	//Continued below



	public sealed override IEnumerator<YieldInstruction>
	LoadAndAttachCellObjectsToCells<T>(List<T> cells, int loaderID)
	{
		CellString cellString = RegisteredUsers[loaderID].CellString;
		foreach(T cell in cells)
		{
```

```
                        cellString.MatchStringToCell(cell.CellOnWorldGrid);
                        string objName = cellString.ToString();
                        GameObject obj = = (GameObject)GameObject.Instantiate (
                                                Resources.Load(objName),
                                                cell.CellObjectPosition,
                                                Quaternion.identity);

                        obj.name = objName;
                        cell.AttachCellObjectToCell(obj, true);
                        if (yieldForTime != null)
                                yield return yieldForTime;
                }
        }
}
```

IsSingleFrameAttachmentPreloadRequired
There is one very important piece of information you are required to provide when creating a custom Cell Object Loader. This information is provided by overriding the IsSingleFrameAttachmentPreloadRequired properties getter, as is done in the previous example.

Why is this Necessary?
Each Cell Object Loader has two responsibilities:

1. Load the cell objects into the scene (and position it).
2. Once loaded, attach the cell object to its cell via the AttachCellObjectToCell method.

Mid-game (i.e., during the normal Update cycle), only the IEnumerator Coroutine version of the cell object loader is used. This allows the cell object loader to take as much time as it needs to load and attach the cell objects to the cells, which is fine since we already have other objects in the scene the player can walk on.

At the start of the game or new scene, however, there are no pre-existing objects, so if we don’t completely load and attach the starting objects to the cells (attachment is needed to position the objects correctly) before the first Update is called, the player will spawn into an empty world. To avoid this, the AttachCellObjectsToCellsInSingleFrame method is used instead of the LoadAndAttachCellObjectsToCells method.

Ideally, this method should load and attach all the cell objects for the cells passed in (which will be the starting objects for the beginning of the game/scene) in a single frame. We would call this method in Awake or Start when performance isn't really a concern, and our objects would be guaranteed to be loaded when the player first "sees" the scene.

Unfortunately, the reality is that some load methods are incapable of this feat. The Application.LoadLevelAdditive method, for instance, which is used by the Scene Loader and Async Scene Loader components to "try" and load objects in a single frame, only adds the scene to the current scene at the end of the frame. This means that objects in the newly loaded scene can only be accessed and referenced (and thus attached to a cell) in the following frame.

If your custom Cell Object Loader requires two frames to complete the attachment and loading process (**two frames is the max**), follow these steps:

1. Override the IsSingleFrameAttachmentPreloadRequired property getter to return true (This tells the primary cell object sub controller using this loader that pre loading is required).
2. Add an implementation for the PerformSingleFrameAttachmentPreload method (example below). This is where you will actually load your objects into the scene. All objects should be loaded in a single frame.
3. Override the AttachCellObjectsToCellsInSingleFrame method to find and attach the loaded cell objects to the cells.

Note that in the example above, the PrefabInstantiator component does not follow these steps. This is because the Instantiate method immediately loads a game object and returns a reference to it, and thus single frame loading/attachment is possible. It may be that your loading strategy does not need pre-loading, in which case you will set IsSingleFrameAttachmentPreloadRequired to return false, and only override the AttachCellObjectsToCellsInSingleFrame method (where you will both load and attach the objects to the cells). The PerformSingleFrameAttachmentPreload method can be avoided.

Here is an example of an implementation for the PerformSingleFrameAttachmentPreload method:

```
public sealed override void PerformSingleFrameAttachmentPreload<T>(
List<T> cells, int loaderID)
{
	CellString cellString = RegisteredUsers[loaderID].CellString;

	foreach(T cell in cells)
	{
		cellString.MatchStringToCell(cell.CellOnWorldGrid);
		Application.LoadLevelAdditive(cellString.ToString());
	}
}
```

Type of T
T will be of type IAttachableWorldCell, which means you will have access to the properties of IWorldCell (see IWorldCell subsection in previous section) in addition to the AttachCellObjectToCell(GameObject) method.

CellString
CellString is a custom class I've designed around reducing the garbage generation inherent in using strings. Just think of it as a representation of the name of your cell objects (GroupName_Row_Column, for instance). Each Cell String stores the group name internally, and you can adjust the row, column, and layer (if using a 3D world) via the MatchStringToCell(Cell).

For instance, say your group name is "Terrain" and you call cellString.MatchStringToCell(new Cell(layer : 1, row : 1, column : 3)). This will set your cellString to represent Terrain_1_3 (for a 2D world) or Terrain_1_1_3 (for a 3D world). When you call cellString.ToString(), "Terrain_1_3" or "Terrain_1_1_3" will be returned.

Cell String Benefits
The benefits of using CellString may not be apparent in the PrefabInstantiator example, since we are just converting the cell string to a string. In this case, we could probably just create a normal string and we would see the same amount of garbage generation.

Some load methods, however, may require you to run comparisons between the names of objects in the scene and the expected name of your cell object. For instance, the built in scene loader components load objects into the scene, and then must find the newly loaded objects in order to get a reference to them.

In these instances, it's useful to be able to run a string comparison without actually creating a new string object. The cellString.IsEqualTo(string) method is ideally suited for this task. It takes in a string, which will likely be the name of some game object in the scene, and compares it to the name represented by the Cell String object. If the names match, true is returned; otherwise, false is returned.

Important: The cell string will match the last cell passed in via MatchStringToCell, so make sure to always match the correct cell before using the IsEqualTo method.

Of course, there is no requirement that you use the cell string. You can create a string of the correct name for a cell by using this line of code:

```
string cellName = cellString.baseName + "_" + cell.Row + "_" + cell.Column;
```

Never store and increment the row and/or column values yourself, and then use that to create your string. There is no guarantee that the input cells are in sequential order (i.e. cell_1_1, cell_1_2, cell_1_3, etc.).

Registered Users

You will notice this line of code in the example above, which was used to retrieve the Cell String described in the previous subsection:

```
CellString cellString = RegisteredUsers[loaderID].CellString;
```

The RegisteredUsers property can be used to get the set of users registered with the Cell Object Loader. It is a RegistrationHandler<CellObjectLoaderUser> object whose purpose is to allow for multiple "users" to make use of a single cell object loader. Each user registers via the base CellObjectLoader's "Register" method. The only requirement of the user is that they pass in a World object, though in the Dynamic Loading Kit only Primary Cell Object Sub Controllers are Cell Object Loader users.

Each user receives an ID when they register, and when they invoke a method on the Cell Object Loader they pass in this ID, which allows the loader to access specific data and/or objects tied to the user, and then use these data/objects to execute the method.

For instance, in the example above each user has its own Cell String object, which is retrieved at the beginning of the method and used to execute it.

Creating a New User Type

By default, the default CellObjectLoaderUser class only stores a Cell String object. For your custom cell object loader, you may need to store additional information or objects for each user, in which case you can implement your own user class.

To do so, create a custom class inside your Cell Object Loader class that derives from CellObjectLoaderUser. The makeup of this derived class is up to you; there are no methods or properties from the base CellObjectLoaderUser class that need to be implemented, though you will need need to call the base classes constructor. You will also gain access to the cell string from the base class, which you can access via the CellString property.

Here is an example of a custom user class (this is straight from the BaseSceneLoader class):

```
protected class SceneLoaderUser : CellObjectLoaderUser
{
        public List<string> UnboundCellObjectNames { get; private set; }
        public List<int> UnboundCellObjectIndexes { get; private set; }

        public SceneLoaderUser(World worldAssociatedWithUser)
           : base(worldAssociatedWithUser)
        {
           UnboundCellObjectNames = new List<string>();
           UnboundCellObjectIndexes = new List<int>();
        }
}
```

With your custom class created, you now need to tell the base CellObjectLoader class to use your custom type rather than the default one. To do this, override the CreateNewUser method.

Here is an example:

```
protected sealed override CellObjectLoaderUser CreateNewUser(
World worldAssociatedWithUser)
{
	return new SceneLoaderUser (worldAssociatedWithUser);
}
```

Using your custom Cell Object Loader User

You can access each user using their loaderID like so:

```
registeredUsers[loaderID]
```

This returns a CellObjectLoaderUser object which you can then cast to your custom type, as in the following example.

```
SceneLoaderUser user = (SceneLoaderUser)(RegisteredUsers[loaderID]);
```

Custom Primary Cell Object Sub Controllers

There are two primary cell object sub controller components included with the default kit. The Pooling Primary Cell Object Sub Controller can be used to pool objects when they are not being used, while the Non-Pooling version simply destroys objects when they are no longer needed.

The pooling version should offer better performance, but will also increase memory usage (sometimes dramatically). You can control the maximum number of objects that will be pooled for a given cell. For instance, a value of two would result in two instances of the same terrain/object being pooled; when a third instance is sent to the sub controller, it is destroyed rather than being pooled.

Custom Primary Cell Object Sub Controllers can also be created. There are two main reasons you may wish to create a custom sub controller.

1) Implement a custom strategy for handling cell objects when they are not being used. My pooling strategy is very simple; you may wish to create a more advanced version of the pooling sub controller.

2) Cell Actions can only act after an object is activated and/or before an object is deactivated. A custom sub controller, on the other hand, could perform some operation on each object before it is activated and/or after it is deactivated. One such possible use could be to modify

a terrain's heightmap before it is activated.

Game Object State
As noted in the Custom Cell Object Loaders sub-section, all game objects must be in a deactivated state once control leaves one of the cell object loaders load methods. In addition, when the World passes objects to the primary cell object sub controller for disposal/storage, they are also in a deactivated state. This means the Primary Cell Object Sub Controller will always be dealing with game objects that are in a deactivated state, and it should stay that way (do not activate these game objects yourself).

Awake
The base PrimaryCellObjectSubController class uses Awake to do some initialization, so it's imperative that your derived class does not include an Awake method. You can override the AwakeExtended method instead and it will be called in the base class's Awake method.

Property YieldInstruction YieldForTime
You will have access to the yieldForTime property in your derived class, which will either be null or a WaitForSeconds object. It's null if you specify "Post Destruction Yield Time" to be <= 0 in the inspector and a WaitForSeconds object otherwise. You can use this yield instruction in your DetatchAndProcessCellObjectsFromDeactivatedCells method, which will be described below.

Property CellObjectLoader CellObjectLoader
You will have access to the CellObjectLoader property in your derived class, which you can use to get the Cell Object Loader associated with your sub controller (via the inspector).

Property RegistrationHandler<PrimaryCellObjectSubControllerUser> registeredUsers
You will have access to the RegisteredUsers property, which you can use to get the set of users registered with the primary cell object sub controller. Each user can be accessed via its primaryCellObjectSubControllerID like so:

```
PrimaryCellObjectSubControllerUser user =
RegisteredUsers[primaryCellObjectSubControllerID];
```

For information on users, please refer to the Registered Users portion in the Custom Cell Object Loader sub-section, as the information contained there can be applied to this section as well.

By default, each primary cell object sub controller user only has a LoaderID associated with it, which is an ID used with the Cell Object Loader. If you need additional data to be associated with each user, create a custom type in your class which derives from PrimaryCellObjectSubControllerUser, and then override the PrimaryCellObjectSubControllerUser CreateNewUser(World, int) method (in your custom sub controller class, not the custom user class).

Note that when accessing the user, you will need to cast it to your custom user type, like so:

```
CustomUserType user =
(CustomUserType)RegisteredUsers[primaryCellObjectSubControllerID];
```

Property bool ObjectsHaveCellActions
The ObjectsHaveCellActions property simply returns whether cell actions have been enabled on the sub controller via the inspector option "Use Cell Actions." You shouldn't need to use this.

Method CreateNewUser
Signature

```
protected virtual PrimaryCellObjectSubControllerUser CreateNewUser(World
worldAssociatedWithUser, int loaderID)
```

About

This method is called when a new user registers with the sub controller. You only need to override this method when you need to use a custom user type in order to associate additional data with each user.

Example Override

```
protected sealed override PrimaryCellObjectSubControllerUser
CreateNewUser(World worldAssociatedWithUser, int loaderID)
{
        int poolSize = 1;

        if(worldAssociatedWithUser.AreRowsEndless ||
        worldAssociatedWithUser.AreColumnsEndless ||
        (worldAssociatedWithUser.worldGrid.worldType ==
        WorldType.Three_Dimensional &&
        worldAssociatedWithUser.AreLayersEndless))
        {
                //maxObjectsInPool is set in the inspector
                poolSize = maxObjectsInPool;
        }

         return new PoolingPrimaryCellObjectSubControllerUser
                (worldAssociatedWithUser, loaderID, poolSize);
}
```

PoolingPrimaryCellObjectSubControllerUser is a custom class implemented within the PoolingPrimaryCellObjectSubController class.

```
class PoolingPrimaryCellObjectSubControllerUser :
PrimaryCellObjectSubControllerUser
{
          public IGridPool<GameObject> Pool { get; private set; }

          public PoolingPrimaryCellObjectSubControllerUser(World
                worldAssociatedWithUser, int loaderID, int poolSize)
                        : base(worldAssociatedWithUser, loaderID)
          {
                WorldGridBase worldGrid = worldAssociatedWithUser.worldGrid;
                if (worldGrid.WorldType == WorldType.Three_Dimensional)
                {
                        Pool = new GridPool3D<GameObject>(worldGrid.Layers,
                                        worldGrid.Rows, worldGrid.Columns, poolSize);
```

```
            }
            else
            {
                    Pool = new GridPool2D<GameObject>(worldGrid.Rows,
                            worldGrid.Columns, poolSize);
            }
        }
}
```

Don’t worry about what’s happening in the constructor of this class, as it has nothing to do with creating your own user type. The important thing to note here is that the user type derives from PrimaryCellObjectSubControllerUser, and calls the base classes constructor via base(worldAssociatedWithUser, loaderID).

As long as your custom type does these two things, you can do whatever you want with the make-up of your custom user type.

Method AttachCellObjectsToCellsInSingleFrame

Signature

```
public abstract void AttachCellObjectsToCellsInSingleFrame<T>(List<T> cells, int
primaryCellObjectSubControllerID) where T : IAttachableWorldCell;
```

About

This method is called when cell objects must be attached to the passed in cells in a single frame. This is used at the beginning of the scene in Awake, in order to assure objects are loaded before the first Update cycle begins. List<T> cells is the list of cells which need and object attached to them, while primaryCellObjectSubControllerID is the ID assigned to the user who is calling this method (the ID is assigned when they registered).

This method must be overridden in your custom class.

Example Override

```
public sealed override void AttachCellObjectsToCellsInSingleFrame<T>(List<T>
cells, int primaryCellObjectSubControllerID)
{
        cellObjectLoader.AttachCellObjectsToCellsInSingleFrame<T>(cells,
                RegisteredUsers[primaryCellObjectSubControllerID].LoaderID);
}
```

In this example, since the non-pooling sub controller does not have any of the required cell objects (and thus they need to be loaded into the scene), the responsibility for loading and attaching the cell objects to the cells is passed off to the cell object loader.

The LoaderID associated with the user who called this method is retrieved from RegisteredUsers and passed (along with the cells) to the cell object loader.

Method AttachCellObjectsToCells

Signature

```
public abstract IEnumerator<YieldInstruction>
AttachCellObjectsToCells<T>(List<T> cells, int primaryCellObjectSubControllerID)
where T : IAttachableWorldCell;
```

About

This method is called during the normal Update cycle, when cell objects cannot be attached to the cells in a single frame (this would cause a performance bottleneck). Instead, the process can be carried out over a period of time. This method must be overridden in your custom class.

Example Override (taken from NonPoolingPrimaryCellObjectSubController)

```
public sealed override IEnumerator<YieldInstruction>
AttachCellObjectsToCells<T>(List<T> cells, int primaryCellObjectSubControllerID)
{
        IEnumerator<YieldInstruction> e =
                cellObjectLoader.LoadAndAttachCellObjectsToCells<T>(cells,
                RegisteredUsers[primaryCellObjectSubControllerID].LoaderID);

        while (e.MoveNext())
                yield return e.Current;
}
```

In this example, the non-pooling sub controller does not have any of the objects stored, so it passes the responsibility of loading and attaching the objects to the cells off to the cell object loader.

Note that e.Current in the method above is a YieldInstruction. When implementing this method, use the same strategy I've implemented here (iterate over the enumerator manually using MoveNext, and return the YieldInstruction from e.Current).

Another Override Example (taken from pooling sub controller)

```
public sealed override IEnumerator<YieldInstruction>
AttachCellObjectsToCells<T>(List<T> cells, int primaryCellObjectSubControllerID)
{
        PoolingPrimaryCellObjectSubControllerUser user =
             (PoolingPrimaryCellObjectSubControllerUser)
                    RegisteredUsers[primaryCellObjectSubControllerID];

        for (int i = cells.Count - 1; i >= 0; i--)
        {
             GameObject gameObject;
             if (user.Pool[cells[i].CellOnWorldGrid].TryGetObjectFromPool(out
                    gameObject))
             {
                    cells[i].AttachCellObjectToCell(gameObject);
                    cells.RemoveAt(i);
             }
        }




        if (cells.Count > 0)
        {
             IEnumerator<YieldInstruction> e =
                    cellObjectLoader.LoadAndAttachCellObjectsToCells<T>
                           (cells, user.LoaderID);

             while (e.MoveNext())
                    yield return e.Current;
        }
}
```

In this example, the cells passed in as an argument are iterated over (backwards), and pool associated with the user who called the method is checked to see if the object for that cell is present. If the object is found, it is attached to the cell, and the cell is removed from the list (this step is why the backwards iteration is required).

If at the end of this process there are any cells which still need objects (because the pool did not have objects for them), then these cells are passed to the cell object loader, which will load and attach the objects needed for the cells.

Method DetatchAndProcessCellObjectsFromDeactivatedCells

Signature

```
public abstract IEnumerator<YieldInstruction>
DetatchAndProcessCellObjectsFromDeactivatedCells<T>(List<T> deactivatedCells,
int primaryCellObjectSubControllerID) where T : IDetatchableWorldCell;
```

About

This method is called when cells on the World are deactivated and the cell objects associated with those cells are deactivated and need to be processed. Processing can consist of anything you want. You can pool the objects or simply destroy them. This method must be overridden in your custom class.

Example Override

```
public sealed override IEnumerator<YieldInstruction>
DetatchAndProcessCellObjectsFromDeactivatedCells<T>(List<T> deactivatedCells,
int primaryCellObjectSubControllerID)
{
        int count = deactivatedCells.Count;
        foreach(T cell in deactivatedCells)
        {
                Destroy(cell.DetatchCellObjectFromCell());
                if (YieldForTime != null)
                        yield return YieldForTime;
        }
}
```

In this example, the cell objects (which remember, are just game objects) are retrieved from the passed in cells using the DetatchCellObjectFromCell method, and then destroyed using Unity's Destroy method. Some time is yielded after each destroy, which helps with performance.

Note that the DetatchCellObjectFromCell method is a method from the IDetatchableWorldCell interface. If you need more information about this interface, or any of the other public types found in the Dynamic Loading Kit, you can consult the API found at http://deepspacelabs.net/dynamic_loading/api_documentation/documentation_main.html.

Editor Considerations

When creating a custom Primary Cell Object Sub Controller, you will need to create a custom editor and implement some default logic in order to get some of the base inspector options to display. To do so, follow these steps:

1) In a folder named "Editor" create a script using whatever name you please (though 'CustomSubControllerTypeEditor' is a good naming convention to follow).

2) Make sure the class derives from Editor (you will need to add a using UnityEngine statement).

3) Add a CustomEditor attribute above the class declaration, such as: [CustomEditor(typeof(NonPoolingPrimaryCellObjectSubController))]

4) Add an override for OnInspectorGUI(). This is where you will call the base primary cell object sub controller inspector GUI, and also implement your own inspector options.

Example

```
using UnityEditor;
using DynamicLoadingKit;
using DynamicLoadingKitEditors;

[CustomEditor(typeof(NonPoolingPrimaryCellObjectSubController))]
class NonPoolingPrimaryCellObjectSubControllerEditor : Editor
{
	NonPoolingPrimaryCellObjectSubController targetScript;
	PrimaryCellObjectSubControllerEditor baseEditor;
```

```
        public override void OnInspectorGUI()
        {
                targetScript = (NonPoolingPrimaryCellObjectSubController)target;
                if (baseEditor == null)
                {
                        baseEditor = new
                                PrimaryCellObjectSubControllerEditor(targetScript);
                }

                //This will tell you if any of the options
                //were changed in the base editor. If they
                //were, you will need to set your script to dirty
                bool wasOptionChanged;

                //This displays the base options in the inspector
                baseEditor.OnInspectorGUI(out wasOptionChanged);

                //At this point you can display any
                //other options related to your custom sub controller

                if (wasOptionChanged)
                        EditorUtility.SetDirty(targetScript);
        }
}
```

Custom Persistent Data Controller

The Persistent Data Controller controls how persistent data is saved between game sessions. By default, only a Player Prefs version is provided, but this component is not recommended for a production quality game. It doesn't support multiple game saves and Player Prefs is generally considered a poor way of handling saves. Most likely, you will want/need to use some other method of saving your data.

To do so, create a class which derives from PersistentDataController (make sure to add a using DynamicLoadingKit to your script), or modify an existing class to derive from it. Then provide implementations for the following methods.

Method SaveData

Signature

```
public abstract void SaveData(string key, string data);
```

About

Saves the data using the key specified.

Example Override

```
public sealed override void SaveData(string key, string data)
{
	PlayerPrefs.SetString(key, data);
	PlayerPrefs.Save();
}
```

Method TryGetData

Signature

```
public abstract bool TryGetData(string key, out string data);
```

About

Tries to retrieve the data associated with key. Returns true if the data exist; otherwise false.

Example Override

```
public sealed override bool TryGetData(string key, out string data)
{
	if (DoesDataExist(key))
	{
		data = PlayerPrefs.GetString(key);
		return true;
	}
	else
	{
		data = null;
		return false;
	}
}
```

Method TryDeleteData

Signature

```
public abstract bool TryDeleteData(string key);
```

About

Tries to remove the data associated with key. If the removal is successful, returns true; otherwise false.

Example Override

```
public sealed override bool TryDeleteData(string key)
{
	if (DoesDataExist(key))
	{
		PlayerPrefs.DeleteKey(key);
		PlayerPrefs.Save();
		return true;
	}
	else
		return false;
}
```

Editor Considerations

Like the Primary Cell Object Sub Controller, you will need to create a custom editor in order to gain access to the inspector options and functionality of the default Persistent Data Controller class. Refer to Chapter 4, Section 2, Custom Primary Cell Object Sub Controller, Editor Considerations for more information.

Example

```
using UnityEditor;
using UnityEngine;
using DynamicLoadingKit;

[CustomEditor(typeof(PlayerPrefsPersistentDataController))]
class PlayerPrefsPersistentDataControllerEditor : Editor
{
	PlayerPrefsPersistentDataController targetScript;
	PersistentDataControllerEditor baseEditor;

	bool needToSave;

	public sealed override void OnInspectorGUI()
	{
		targetScript = (PlayerPrefsPersistentDataController)target;

		if (baseEditor == null)
			baseEditor = new PersistentDataControllerEditor(targetScript);

		baseEditor.OnInspectorGUI(out needToSave);
		if (needToSave)
			targetScript.SaveChanges();
	}
}
```

Custom Cell Object Destroyer

The Cell Object Destroyer component controls how cell objects are destroyed (when required). By creating a custom destroyer, you can implement destruction logic designed around your object hierarchy structure. For instance, you can create logic which destroys some set of child objects 5 sub levels deep one frame, then another set 4 levels deep in the next, and so on.

Method DestroyCellObject

Signature

```
public abstract IEnumerator<YieldInstruction> DestroyCellObject(GameObject
cellObject);
```

About

This is the only method which needs to be implemented in a custom cell object destroyer. At the very least, this method should destroy the root cell object (though, if that's all you're doing, it's better to leave the Cell Object Destroyer field empty on the Primary Cell Object Sub Controller you are using, and not even mess with creating a custom destroyer). More than likely, you will want to destroy a specific amount of children each frame.

You can yield return any valid YieldInstruction, such as WaitForSeconds or null.

Example Override

```
public sealed override IEnumerator<YieldInstruction>
DestroyCellObject(GameObject cellObject)
{
	Transform cellObjectTransform = cellObject.transform;

	int children = cellObjectTransform.childCount;

	while (true)
	{
		int max = children < childrenToDestroyPerFrame ? children :
			childrenToDestroyPerFrame;

		for (int i = 0; i < max; i++, children--)
			Destroy(cellObjectTransform.GetChild(i).gameObject);

		if(children == 0)
		{
			Destroy(cellObject);
			yield break;
		}
		else
			yield return null;
	}
}
```

Custom IOriginCenteredWorldUser

Though it is not recommended, it is possible to create custom class which can register with an origin centered World. This class must implement the IOriginCenteredWorldUser interface and adhere to a specific set of rules, and as a result it will be notified when a World shifts so it can respond appropriately. The rules are as follows:

1) Implement a property called IsReadyForWorldShiftPreparation. This property should only return true when the user is ready for the other methods of the interface to be called. Returning false will force the World to skip shifting in the current frame, giving your class time to finish whatever action it is currently executing.
2) BlockOtherActionsUntilWorldShiftCompletes is called immediately by the World after IsReadyForWorldShiftPreparation. This method should stop other actions that would keep the final two methods from running successfully.
3) WorldShiftCommencing is called right before a World Shift operation is carried out by the World. At this point, the World is treated as if the shift was already completed, so that if the game is saved, the World is loaded at the shifted position. You're custom class should take appropriate action so that if it saved, it can handle the World being at the shifted position.
4) WorldShiftComplete is a Coroutine that is called immediately following the World being shifted. You should take action in response to the World shift in this method (such as moving objects by the amount specified when WorldShiftCommencing was called).
5) Register with the origin centered world using the World's Register method. This method will also set an ID that you must use when deregistering with the World.

Section 3: Configuring the Dynamic Loading Kit to Use Scene Loader Components

If you wish to use one of the Scene Loader Cell Object Loader Components, you will need to create scenes first, and then properly configure your Scene Loader Components. This section covers the steps to do this in detail.

Update 3.0.3

Prior to Update 3.0.3, Layer Hiding was the strategy used to hide objects in a scene before they were positioned correctly. This system was overly complicated and did not result in the best performance, since physics calculations would need to be run possibly twice, once as objects in the scene were initially loaded, and once after they were moved.

Update 3.0.3 introduced a new strategy, which allows for simpler setup and better performance. I recommend updating to 3.0.3 so you can take advantage of this new system, which shall be henceforth referred to as the **Deactivated Node System**.

A Little Background

Before getting into how to set up your scenes, it's important to understand how the scene loader components work behind the scenes.

Finding Scene Objects

Unlike the Instantiate method, the methods used to load new scenes on top of the existing scene (Application.LoadLevelAdditive and Application.LoadLevelAdditiveAsync) do not return a reference to any of the objects from the newly loaded scene.

This makes sense, since a scene can contain many unrelated game objects, but it does make things more complicated for us, since Dynamic Loading Kit needs a reference to the root object in the scene (for moving/unloading/accessing cell actions/etc.).

There are numerous ways to find game objects. The method used by the Dynamic Loading Kit is GameObject.FindGameObjectsWithTag. This method is more efficient than other methods, since only the pool of tagged objects in the scene needs to be searched, rather than all objects in the scene.

It's also nice because with our Scene Loader components, we load multiple scenes at a time (spread out over multiple frames), so we simply call this method once to get all of the newly loaded root objects, rather than using an alternate method that would find each root object individually.

The Root Object

The root object in each scene used by the Scene Loader components is the top level object which all other objects in the scene are parented to. By parenting all objects in the scene to the root object, we ensure that all objects in the scene are properly moved when the root object is moved and removed when no longer needed.

By having a root object, we also open up the possibility for efficient reference finding by utilizing the tag property of the object and GameObject.FindGameObjectsWithTag. Unfortunately, in order for this method to work, the root object must be in an activated state upon being loaded.

Visibility Issues

The fact that the root object must be in an activated state leads to some potential visibility problems when the scene that is loaded must be moved from its default position, as is the case in various scenarios, such as when utilizing an endless world.

If we were to simply leave all objects in the scene in an activated state which is visible to the camera, the player might see the objects from the scene out of place. The objects may even spawn on top of the player!

Previously, this issue was addressed by placing the objects on a layer which the camera did not render. In addition to being a pain to set up, this method was not very efficient, since physics information had to be calculated twice, once at the objects initial positions, and once at their post-move positions.

Deactivated Node

Obviously, a more efficient method is to deactivate the objects in the scene, and only activate them after they are positioned correctly. This way, the physics and other calculations are only performed once, when the objects are first activated. There are two issues with this strategy, however.

1) The root object cannot be deactivated, since doing so would make it impossible to find via GameObject.FindGameObjectsWithTag (or really, any method).
2) We don't want to deactivate each individual game object in the scene. In addition to being inefficient, you may wish to use the activeSelf state of each game object for your own purposes.

We can solve these issues quite simply by using a "deactivated node" (my own term). This node is simply an inactive game object parented directly to the root object in the scene, which all other game objects are parented to. This node can be an empty game object or your main terrain/mesh; it does not matter.

Using a deactivated node, we can:

1) Keep the root object in a deactivated state.
2) Hide all other objects in the scene (since they become deactivated when parented to the deactivated node).
3) Leave the activeSelf, tag, and layer properties of each game object alone. This allows you to use these properties for whatever purpose you wish, unrelated to the Dynamic Loading Kit.

Cell Action Consideration

Note that the root object in each scene need not be an empty game object. Indeed, if using Cell Actions, the cell action components must be attached to the root object in order to work.

Just make sure that the root object does not contain any component which would renderer something to the game!

Fixed/Static Scenes

Fixed/Static scenes are scenes that will never need to be moved. For instance, if you have a root object in a scene at position 0, 0, 0, the scene can be considered fixed if that root object will only ever need to be at position 0, 0, 0. Fixed scenes are more efficient than non-fixed scenes (since they are not moved).

With fixed scenes, the visibility issues discussed in the previous issue are not a concern, since the objects in the scene will only ever appear in their default, correct position. As such, use of a deactivated node is not necessary.

Whether your scenes are fixed depends entirely upon how you configure your World component. The following conditions will make your scenes **NOT** fixed/static.

1) Enabling any of the endless options (Are Layers Endless, Are Rows Endless, Are Columns Endless).
2) Enabling the "Keep World Centered Around Origin" option.
3) Using a World Origin that is different than the position of the root object in your first scene (the one that ends with _1_1 or _1_1_1).

Setting up Your Scenes Manually

If working from a new scene, remove the Main Camera game object from the scene.

Next, add the objects which will comprise the scene to the scene view or scene hierarchy. If you have prefabs that you wish to add to the scene, you can right click them and choose Dynamic Loading Kit -> Add Prefabs To Scene.

Hierarchy

You should have a scene that looks something like this.

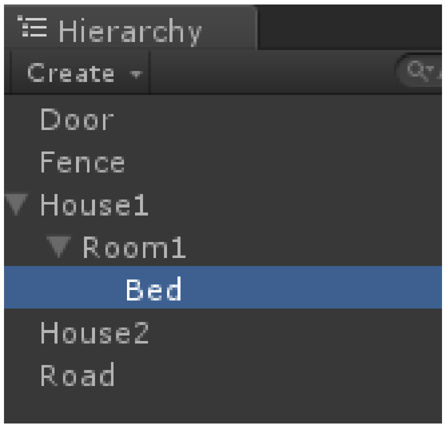
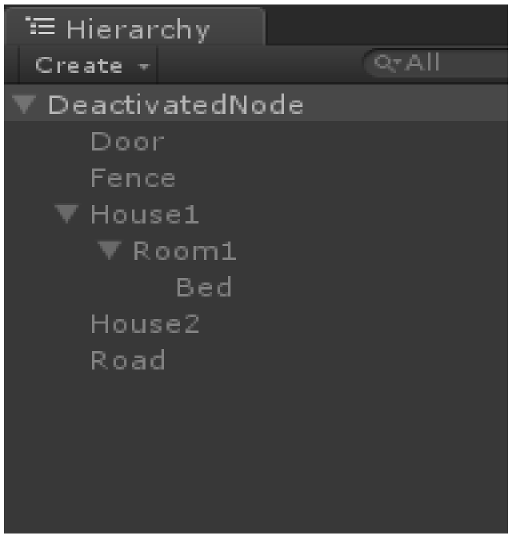


Figure 24: Starting Scene Hierarchy

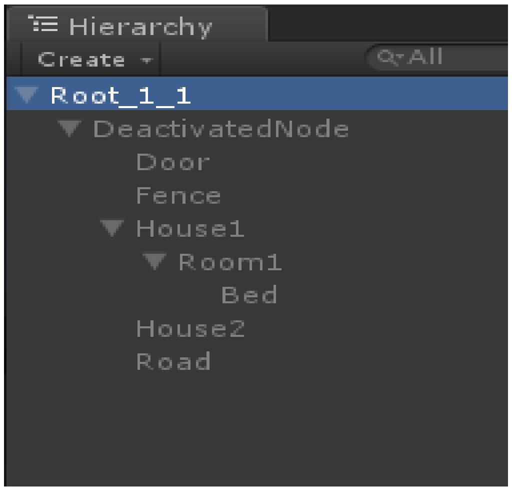
If you know that your scenes are not fixed, the next step is to add a game object to serve as the deactivated node, deactivate it, and parent all other game objects in the scene to it.



```
DeactivatedNode
    Door
    Fence
    House1
        Room1
            Bed
    House2
    Road
```

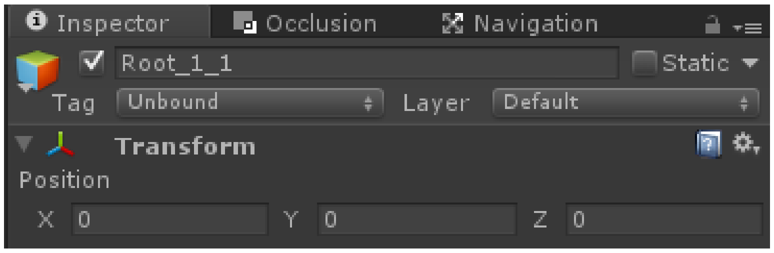
Figure 25: With Deactivated Node

Next, create the root object for your scene and set its tag to the Unbound Object Tag (create this tag in the inspector if you haven't already. You can call it whatever you want, though "Unbound" is a great name)). Parent the deactivated node object to it. Remember, the root object is left in an activated state.



```
Root_1_1
    DeactivatedNode
        Door
        Fence
        House1
            Room1
                Bed
        House2
        Road
```

Figure 26: With Root Object



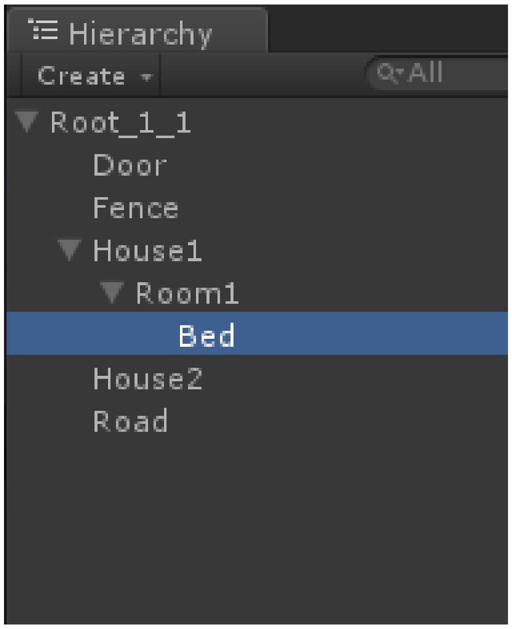
Root_1_1

Tag: Unbound
Layer: Default

Position
X 0 Y 0 Z 0

Figure 27: Root Object Inspector

If using a fixed scene, you do not need to include the deactivate node object, since none of the objects in the scene will need to be deactivated.



```
Root_1_1
  Door
  Fence
  House1
    Room1
      Bed
  House2
  Road
```

Figure 28: Fixed Scene with no Deactivated Node

Saving Your Scene and Adding it to the Build Settings

You must save your scene with the same name as your root object. For example, if the scene represented by the hierarchy in Figure 28 were saved, it would be named "Root_1_1".

You must also add the scenes you create to the build settings, found under File -> Build Settings. If you have the scene open, you can choose "Add Current," otherwise you can drag and drop the scene from the project hierarchy onto the appropriate section of the window after it has been saved.

Root Object Position
To understand root object positioning, I recommend reading the sections on the World Grid here and here first.

Each scene that you will create is associated with a cell on the World Grid. For instance, the Root_1_1 scene represented by Figure 26 and 28 would be associated with cell 1_1 on the World Grid.

Ideally, the position of the root object should match the position of its associated cell (each cell's position is measured at its bottom left most point). This will ensure that you don't need to mess with the Offset values in the World Grid's inspector.

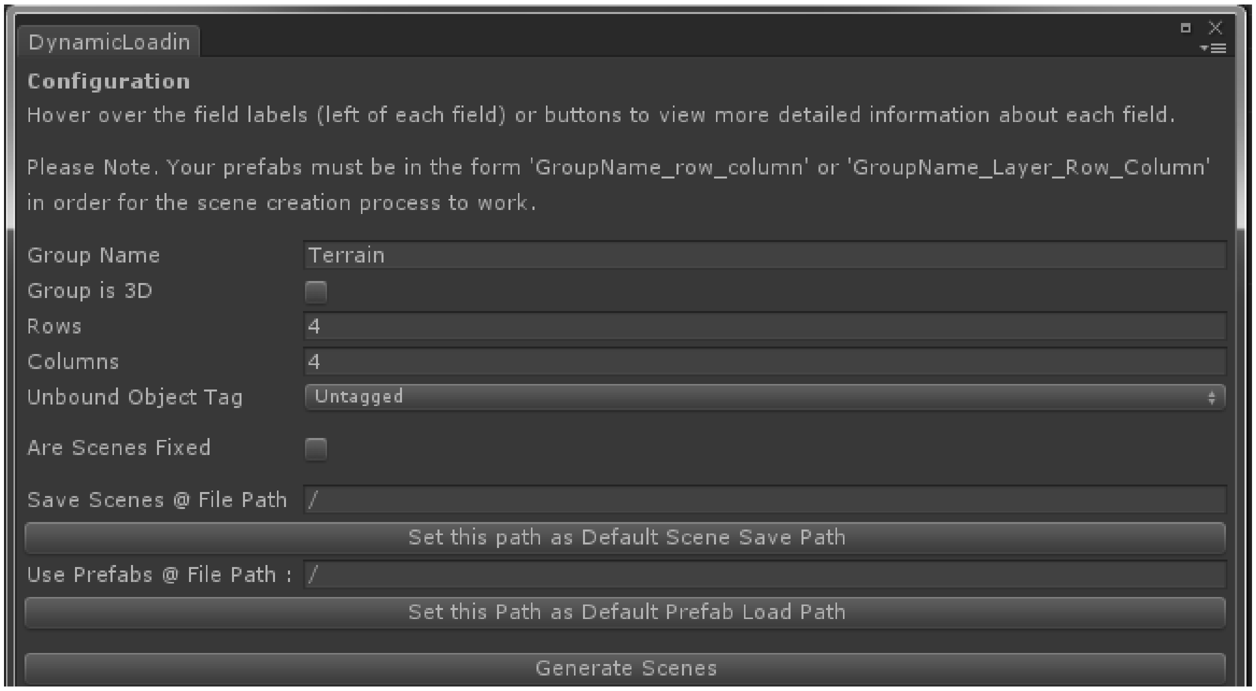
You can have positions that differ from their associated cell's position, as long as the position is within the bounds of the cell. In these cases, however, you will need to adjust the offset values in the World Grid's Inspector. More information can be found here.

The actual position of where your scenes' root objects will be placed depends on your World Origin and Are Scenes Fixed option on your Scene Loader component, as well as whether the scene is being used on an endless world or world set to stay centered about its origin.

If you are trying to create fixed scenes, make sure the position of the root object in your first scene (1_1), matches the origin of your World. Then position the remainder of your root objects based on this initial one.

Using the Scene Generation Tool

If you have a set of prefabs that represent your "World" already created (such as those created using the Terrain Slicing Tool), the easiest way to generate your scenes is to use the Scene Generation Tool found under Assets -> Dynamic Loading Kit -> Generate Scenes. This tool generates a group of scenes from a group of prefabs, and is very simple to use.


DynamicLoadin

Configuration

Hover over the field labels (left of each field) or buttons to view more detailed information about each field.

Please Note. Your prefabs must be in the form 'GroupName_row_column' or 'GroupName_Layer_Row_Column' in order for the scene creation process to work.

Group Name	Terrain
Group is 3D	☐
Rows	4
Columns	4
Unbound Object Tag	Untagged
Are Scenes Fixed	☐
Save Scenes @ File Path	/
Use Prefabs @ File Path :	/

Set this path as Default Scene Save Path
Set this Path as Default Prefab Load Path
Generate Scenes

Figure 29: Scene Generation Tool Window

Each options label can be hovered over for additional information about that option.

The “Are Scenes Fixed” option controls how the generated scenes’ hierarchies are laid out. If enabled, each scene will simply contain a copy of the prefab found in the folder specified by “Use Prefabs @ File Path,” with only the root object in the scene changed so that it’s tag is set to the defined “Unbound Object Tag.”

If disabled, the prefab object will serve as the deactivated node, and a new root object with the same name as the prefab will be created. Don’t be concerned that this deactivated node has the same name as the root object!

Cell Actions

For efficiency reasons, only Cell Action components on the root object will be detected by the Dynamic Loading Kit. When using the Scene Generation Tool, any existing Cell Actions on your prefabs will not be copied to the new root object (if one is created). In these cases, you will need to manually move the Cell Actions from the old root object (which should now be the deactivated node) to the new root object. Better yet, only add Cell Actions after you create your scenes.

Build Settings
The Scene Generation Tool automatically adds the generated scenes to the build settings, however if for some reason you wish to regenerate your scenes, you will need to manually go into the build settings and remove the scenes before regeneration.

Scene Loader Inspector

The Scene Loader and Async Scene Loader should be configured if they haven't already at this point. Refer to the Scene Loader section for more info.

Frequently Asked Questions

Question: What is the maximum size of the world?
Answer: When using a non-pooling primary sub controller and a scene or asset bundle cell object loader (the asset bundle loader has not been added yet), the maximum size of the world is virtually endless.

Scenes are stored as separate files when building your project, and as such they should not cause out of memory issues. The Prefab Instantiator, on the other hand, uses prefabs stored in a Resources folder, and as such ***is*** subject to the Unity memory limit (3.5 GB's or so).

You might also need to setup your World so it stays centered about the origin. This is usually required when using an endless world, or when the world you're dealing with is so big that floating point inaccuracies become an issue.

Note that the worlds are not limitless. Even on an origin centered world or endless world, if the player were to move continuously in a given direction, eventually a limit would be reached.

Question: Is the Dynamic Loading Kit compatible with Time of Day/Night and/or Weather Asset Store Packages such as UniStorm and Time of Day - Dynamic Sky Dome?
Answer: At this time, the kit does not boast compatibility with any other asset store package. This does not mean the kit will not work with them, only that I have not tested it with them. If you are experiencing any issues, please contact me and I will attempt to help (though keep in mind, I most likely don't own the package you are having trouble, so my support may be limited). If you own an Asset Store package and are interested in making it compatible with my kit, feel free to contact me and we can work something out.

Question: Does the kit work in 2 dimensions or 3?
Answer: The kit was primarily designed around the use of Unity Terrains, which operate on the 2D X/Z plane. With that said, 2D X/Y and 3D worlds are also now possible with version 2.0!

Question: Does the kit support varying Levels of Detail?
Answer: While I was previously planning on supporting this, it is no longer in my plans.

Question: What platforms are supported by the Terrain Slicing & Dynamic Loading Kit?
I know of no compatibility issues, but I have also not been able to test the kit on most platforms. If any issues arise, please contact me immediately. If a solution cannot be found, a full refund will be provided (the order invoice must be provided).

Do keep in mind that this kit primarily is designed around loading/unloading objects/terrains, so it has not been heavily optimized in terms of garbage allocations (that is not to say I haven't reduced allocations where possible), so this kit might not be suited for mobile platforms. Also keep in mind that Unity Terrain's, in general, are not optimized for mobile platforms.

Question: Will using the Dynamic Loading Kit improve the performance of my game?
Prior to Unity 5, the short answer would have been "Probably not."

If you have a one or just a few terrains in your scene, switching to this kit will probably not improve performance, as all you'll be doing is increasing the number of terrains in the scene.

If you have a bunch of terrains in your scene, switching to a dynamic loading solution should help improve performance by reducing the number of terrains in the scene at one time.

Generally speaking, if you are achieving good/acceptable performance with your current solution, it's probably a good idea to avoid this kit.

With Unity 5 and the possibility of all users being able to use the Async Scene Loader component, the performance of this kit should be improved dramatically. Further testing is still required before saying for sure.

Question: If performance will not be improved, what is the point of this kit?
The main advantage of this kit is the ability to create incredibly large worlds that would not otherwise be possible due to memory limitations. As long as you use one of the Scene Loader Cell Object Loaders, memory should not be a concern when using the Dynamic Loading Kit.

Using the Dynamic Loading Kit also allows you to create endless repeating worlds!

In addition, as stated previously, the performance has not been fully evaluated with Unity 5, which should improve things dramatically!