Lumen

SECTION

Level Linker

Level Linker

Requirements

In order to use Level Linker in your project you need Unity version 2021.3 on newer. Compatibility with older versions has not been tested.

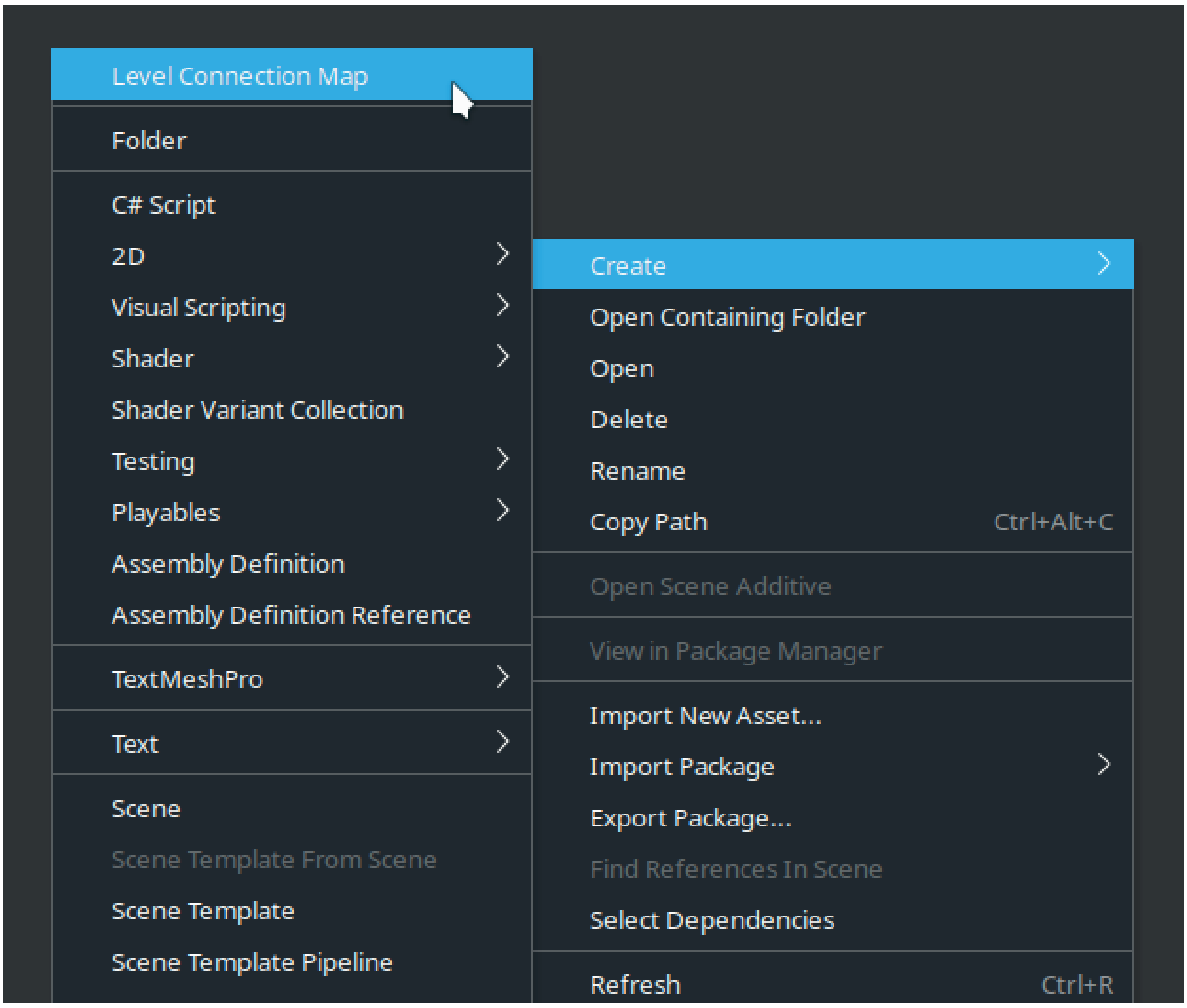
How it works

Level Linker assumes you are going to use scenes to build your levels (one scene for each level). Your levels should contain «doors»: specific objects responsible for spawning the player and loading other levels. Level Linker keeps track of all your levels and the doors inside them. It allows you to create a graph connecting the doors to each others. At runtime, your scripts can query Level Linker to know where each door leads to.

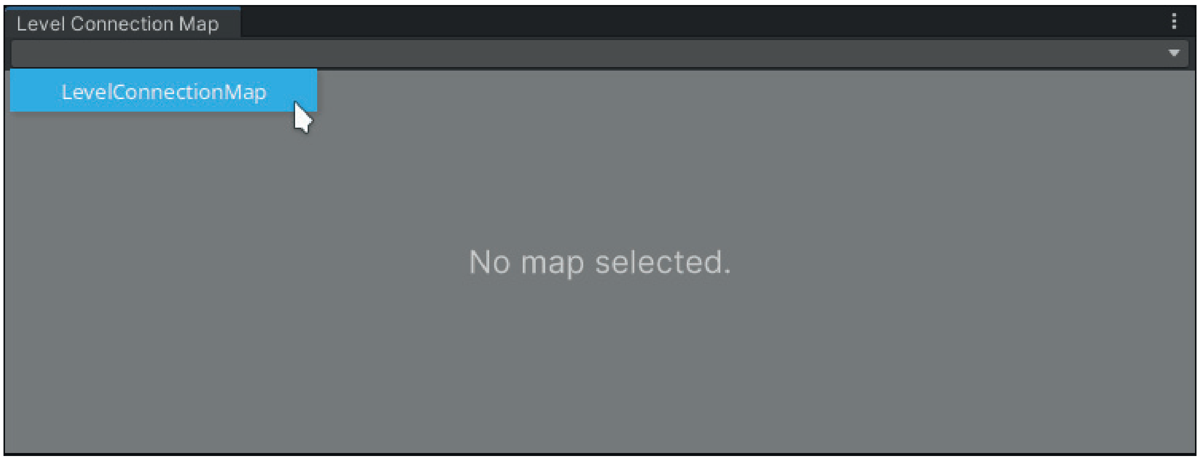
Level Linker does not implement the doors for you because these will act differently depending on the requirements of your game. Level Linker only need your door scripts to implement an interface so it can keep track of them.

How to use

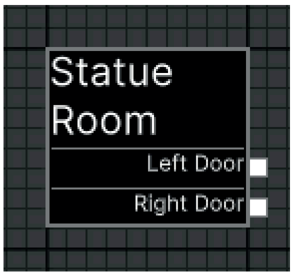
Before you can use Level Linker you must create a **Level Connection Map** asset. This asset is where the graph will be saved. You can store this asset anywhere in your project. To create it, right click into the **Project** tab and choose **Create** > **Level Connection Map**.



To display the graph, in the menu bar choose **Tool** > **Level Connection Map**. Then select your asset in the drop down menu at the top of the window.

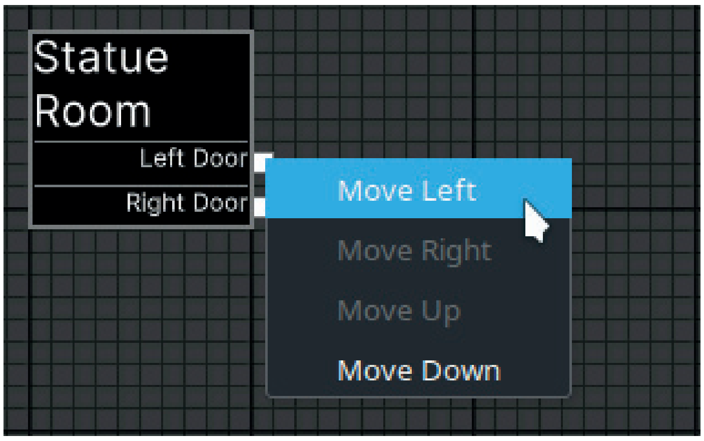


For now the graph is empty. Next time you will save a scene containing at least one door, a node representing that scene will appear on the graph. The plugs on the right side of the node represent the doors in the scene.



Level Linker automatically keeps track of scene and doors renaming and deletion.

The position of the plugs can be modified by right clicking on them an using the contextual menu. This can help keeping the graph readable.



You can create a connections by dragging a door plug on another. You can pan the view by pressing the right mouse button anywhere and dragging. Connections and nodes can be deleted by selecting them and pressing the **Suppr** key.

Implementing the Door interface

Your door scripts must implement the IDoor interface which resides in the LumenSection.LevelLinker namespace.

```
public interface IDoor
{
  string Guid {get;}
  string Name {get;}
}
```

The interface only contains two members.

The Name property must return the name that will be displayed on the graph. Level Linker does not use this name internally so you can use anything. The name does not have to be unique.

The Guid property must return an identifier that is unique to the door across your entire project (not just the scene the door is in) and not change if the door is renamed. You can generate this identifier yourself in you want, but Level Linker can do it for you if you prefer.

To let Level Linker take care of the Guid, simply inherit the UniqueId class (which inherits MonoBehaviour). This class is designed to handles corner cases such as object duplication.

Here is the easiest way to implement your door script:

```
public class MyDoor : UniqueId, IDoor
{
  public string Name => gameObject.name;
}
```

Querying Level Linker at runtime

The whole point of Level Linker is to let your game know where each door leads to. To get the destination of a door at runtime you must call the GetDestination() method of the LevelConnectionMap class.

LevelConnectionMap inherits from ScriptableObject. It is the class that code the map asset you edit with the graph view. The easiest way to access the map it to keep a reference to it into a public variable in your script. But you can also load it with the Resources or the Addressable API if you prefer.

The GetDestination() method takes the Guid of the door as parameter. It returns a tuple containing the name of the destination scene and the name of the destination door.

Here is an example of how to implement a trigger door for a 2D game:

```
public class MyDoor : UniqueId, IDoor
{
  // Must be assigned in the inspector
  public LevelConnectionMap Map;
  public string Name => gameObject.name;

  private void OnTriggerEnter2D(Collider2D col)
  {
    if (!col.gameObject.CompareTag(«Player»))
      return;
    (string nextSceneName, string nextDoorName) = Map.
GetDestination(Guid);
    ... your code to load the scene and spawn the player...
  }
}
```

Using more than one map

You can create as many Level Connection Map assets as you want. This can be useful in some cases. For example, if you game is split into several worlds, you may want to have one graph for each world.

By default, every scene containing at least one door will be registered into all Level Connection Map assets upon saving. If you use more than one map you will likely want to filter which scene is registered into which map. The easiest way to do that it to use the Prefix and Suffix parameters of the Level Connection Map asset. If Prefix is not empty then the map will only register scenes with a name that starts with it. If Suffix is not empty then the map will only register scenes with a name that ends with it. You also have the option of coding your own filter, see next section.

Customizing map behavior

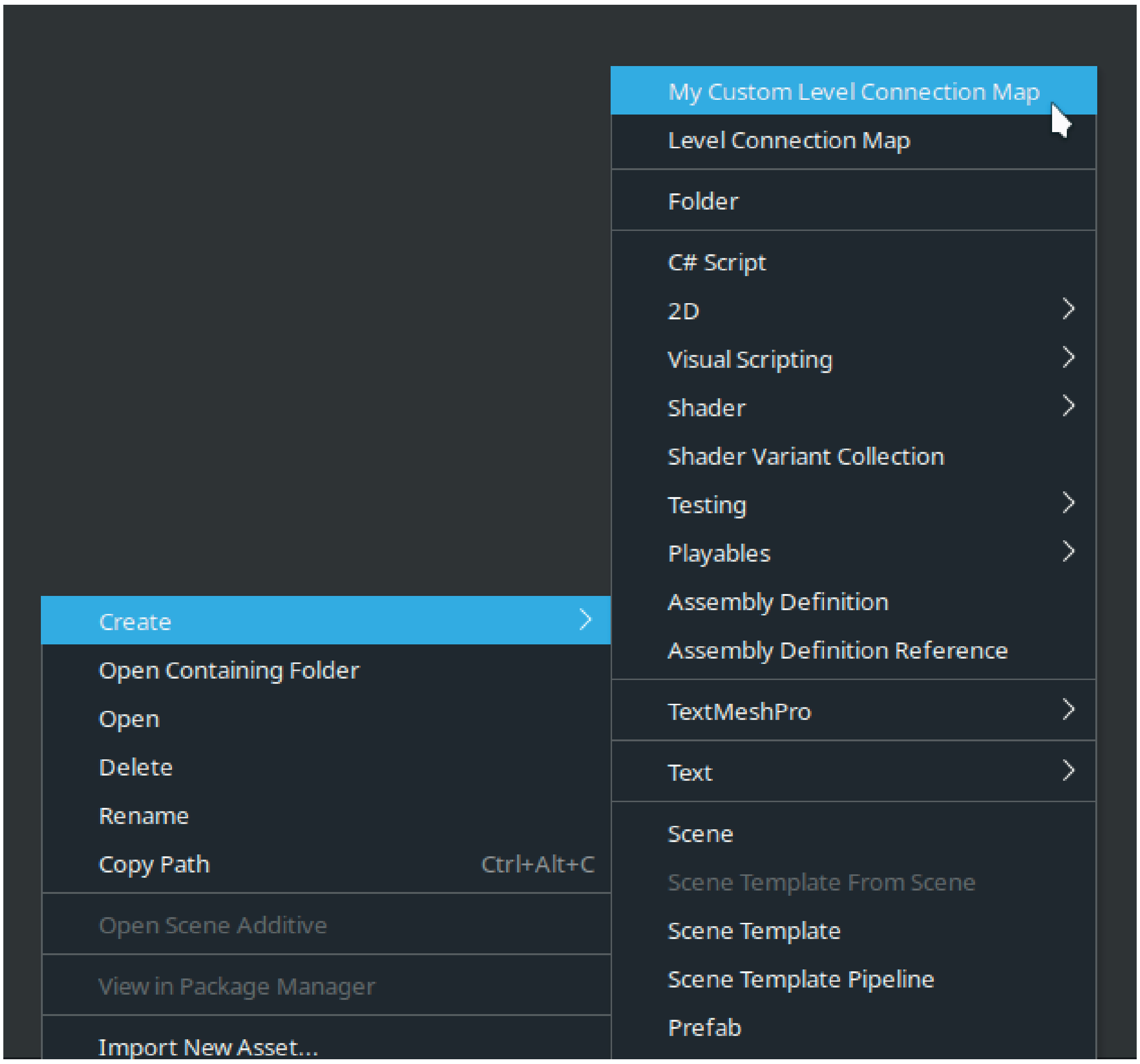
You can change how the map works by inheriting the LevelConnectionMap class and re implementing its virtual methods. Here is an example of how to do it:

```
[CreateAssetMenu(menuName = «My Custom Level Connection Map»)]
public class MyCustomLevelConnectionMap : LevelConnectionMap
{
  protected override bool MustRegisterScene(Scene scene, string
scenePath, string sceneName)
  {
    return base.MustRegisterScene(scene, scenePath, sceneName);
  }

  public override string FormatSceneName(string sceneName)
  {
    return base.FormatSceneName(sceneName);
  }

  public override string FormatDoorName(string doorName)
  {
    return base.FormatDoorName(doorName);
  }
}
```

Then you will have to create instances of your own class instead of LevelConnectionMap.



The **MustRegisterScene()** method is the filter telling which scenes must be registered into the map. The base method implements the prefix/suffix mechanism discussed in the previous section. If it returns true the scene will be registered into the map.

The **FormatSceneName()** and **FormatDoorName()** methods format the scenes and doors name for display on the graph. Name formatting is cosmetic only and has no influence on the working or the map.

The base method for **FormatDoorName()** remove underscores and add spaces between capital letters. The base method for **FormatSceneName()** does the same and also remove the prefix and suffix if present.