Part 1: Setting Up and Using Level Data

Step 1: Creating a Level Data Handler (LevelDataHandler)

The `LevelDataHandler` is a `ScriptableObject` responsible for generating level configurations, saving them to JSON, and loading them back. It acts as the data manager for your levels.

1. **Create a New Level Data Handler**:
    - Go to **Assets > Create > Configurations > LevelDataHandler** to create a new `LevelDataHandler` asset.
    - Rename it as you like, e.g., `LevelDataHandler_Example`.
2. **Assign Configuration for Container Generation**:
    - In the inspector, you'll see fields that control container and platform configurations, such as `minContainers`, `maxContainers`, `minPlatformsPerContainer`, etc.
    - Customize these values to define the range of containers and platforms in each level.
    - **Platform Configurations**: Assign different `SavedPlatformConfigSO` assets (containing platform settings like movement and rotation) to the `platformConfigs` list, giving you variability within generated levels.
3. **Generate Container Data for a New Level**:
    - By using the `GenerateContainerData()` method in the code or through a custom editor tool, `LevelDataHandler` can create randomized containers and platforms within the specified configuration ranges.
4. **Save and Load Levels**:
    - You can save the generated container data to a JSON file by calling `SaveToJson("fileName")` in the code or editor script. The file will be saved with the specified `fileName.json` in the `Application.persistentDataPath`.
    - Load saved levels using `LoadFromJson("fileName")`, which populates the `containerData` list in `LevelDataHandler` with the data from the JSON file.

Step 2: Creating Levels in Unity Using Level JSON Files

To easily manage multiple levels, you can save each level configuration as a JSON file and load it directly in Unity.

1. **Generate and Save JSON Files**:

    - Use `LevelDataHandler` to generate container data and save each configuration as a separate JSON file.
    - Name your files in a series (e.g., `level1.json`, `level2.json`) for easy access.
2. **Add JSON Files to Unity as TextAssets**:
    - Place the JSON files in your **Assets** folder.
    - In the **LevelBuilder** inspector (explained below), you can add each JSON file as a `TextAsset` to the `levelJsonFiles` list.
    - This allows `LevelBuilder` to cycle through each level in the JSON list during gameplay or testing.

Part 2: Using LevelBuilder to Build and Manage Levels

The `LevelBuilder` script is responsible for taking level data (either generated from scratch or loaded from JSON files) and building the actual level in Unity. It also adds a final container to the end of each level.

LevelBuilder Inspector Setup

1. **Prefab References**:
    - **Container Prefab**: Assign the prefab for regular containers in the `containerPrefab` field.
    - **Platform Prefabs**: Assign platform prefabs for each difficulty (`easyPlatform`, `mediumPlatform`, `hardPlatform`).
    - **Final Container Prefab**: Assign a prefab for the final container (`finalContainerPrefab`), which will be added at the end of each level.
2. **Level Settings**:
    - **Use JSON List**: A toggle (`useJsonList`) to specify whether to load levels from JSON files or generate new levels based on `LevelDataHandler`.
    - **Max Active Containers**: Set the maximum number of containers that can be active simultaneously.
    - **Activate All at Start**: If enabled, all containers are activated at the start.
3. **Level JSON Files**:
    - Add all JSON files to the `levelJsonFiles` list by dragging them into the inspector. This is only needed if `useJsonList` is enabled.
4. **Data Manager**:
    - Assign the `LevelDataHandler` asset (created in Part 1) to the `levelDataHandler` field.

LevelBuilder Code Overview

Here's a detailed breakdown of how each part of the `LevelBuilder` script works.

Core Methods

1. Start

csharp
Copy code

```
void Start()
{
    if (useJsonList)
    {
        BuildLevelFromJson(currentLevelIndex);
    }
    else
    {
        BuildNewLevel();
    }
}
```

- **Function**: Determines whether to build a level from JSON or generate a new one from scratch based on the `useJsonList` toggle.
- **Execution**:
  - **If `useJsonList` is true**: Loads and builds the first level in `levelJsonFiles`.
  - **If `useJsonList` is false**: Generates a new level using data from `LevelDataHandler`.

2. BuildLevelFromJson(int levelIndex)

csharp
Copy code

```
private void BuildLevelFromJson(int levelIndex)
{
    // Logic to load and build a level from a JSON file
}
```

- **Function**: Loads the specified JSON file from `levelJsonFiles`, parses it, and builds the level.
- **Key Steps**:

    - Retrieves and parses JSON data using `JsonUtility.FromJson`.
    - Calls `BuildContainers` to instantiate containers based on the loaded data.
    - **Additional Detail**: This method allows cycling through JSON-defined levels by changing `levelIndex`.

3. BuildNewLevel

csharp
Copy code

```
private void BuildNewLevel()
{
    // Generates a new level from scratch based on LevelDataHandler
configuration
}
```

- **Function**: Generates a new level by creating random containers and platforms based on the ranges defined in `LevelDataHandler`.
- **Key Steps**:
    - Calls `GenerateContainerData()` from `LevelDataHandler` to create randomized container data.
    - Calls `BuildContainers` to instantiate containers based on this generated data.

4. BuildContainers(List<SavedContainer> containerData)

csharp
Copy code

```
private void BuildContainers(List<SavedContainer> containerData)
{
    // Logic to build containers from provided data
}
```

- **Function**: Instantiates containers and platforms from a given container data list.
- **Key Steps**:
    - Loops through `containerData`, calling `CreateContainer` for each container.
    - At the end, it calls `PlaceFinalContainer` to add the final container to the level.

5. CreateContainer(List<SavedContainer> levelData, int index)

csharp
Copy code

```
private void CreateContainer(List<SavedContainer> levelData, int
index)
```

```
{
    // Logic to instantiate a container and its platforms
}
```

- **Function**: Creates a container based on data from `levelData`.
- **Key Steps**:
    - Instantiates a `containerPrefab` at the specified position.
    - Loops through each platform in the container and instantiates the appropriate prefab (`easyPlatform`, `mediumPlatform`, or `hardPlatform`) based on difficulty.
    - Positions and configures each platform within the container.

6. PlaceFinalContainer

csharp
Copy code

```
private void PlaceFinalContainer()
{
    // Instantiates the final container at the end of the level
}
```

- **Function**: Places a final container at the end of each level.
- **Positioning**: It calculates the position based on the last container's Z-axis position, offsetting it by a set amount (10 units in this example).
- **Addition to Active Containers**: Adds the final container to the `activeContainers` list, ensuring it's destroyed when cycling levels.

7. CycleToNextLevel

csharp
Copy code

```
private void CycleToNextLevel()
{
    // Logic to destroy current level and move to the next level in
JSON list
}
```

- **Function**: Destroys the current level and builds the next level in `levelJsonFiles`.
- **Key Steps**:
    - Calls `DestroyCurrentLevel` to clear all active containers.
    - Updates `currentLevelIndex` to load the next level in the list.

    - Calls `BuildLevelFromJson` with the updated index.

8. SaveLevel(string fileName)
csharp
Copy code
```
public void SaveLevel(string fileName)
{
    // Saves the current level's container data to a JSON file with a
custom name
}
```

- **Function**: Saves the current level's container data to a JSON file.
- **Usage**: Call `SaveLevel("CustomFileName")` to save the current layout of containers and platforms with the specified filename.

Key Helper Methods

- **DestroyCurrentLevel**: Destroys all containers in the `activeContainers` list to prepare for building a new level.
- **GetPlatformPrefab**: Selects the correct platform prefab based on the difficulty of each platform (`easyPlatform`, `mediumPlatform`, or `hardPlatform`).
- **PositionNeonSignSpawnPoints**: Adjusts the positions of neon sign spawn points within each container.

Summary

- **LevelDataHandler**: Manages the generation, saving, and loading of level data.
- **LevelBuilder**: Constructs levels based on the data, either from JSON files or dynamically generated configurations. It also includes functionality to add a final container at the end of each level.
- **Usage Flow**:
    - Use `LevelDataHandler` to configure, generate, save, and load level data.
    - Use `LevelBuilder` to load levels from JSON or generate new ones, cycle through JSON levels with the "N" key, and save levels with custom names.

This setup gives you complete flexibility to manage, test, and save complex level layouts in Unity without needing to modify individual assets manually. Let me know if there are any additional details you'd like!