

Inventory System

Inventory Grid System



Inventory Grid Creator : This script creates a grid-based inventory system. It's part of a namespace called "InventoryGridSystem".

Breakdowns of the scripts :

1. It defines a few public variables for setting up the grid:
 - slotGrid: A 2D GameObject array representing the grid of slots in the inventory.
 - slotPrefab: A prefab GameObject for the individual inventory slots.
 - slotSize: A float that represents the size of each slot in the grid.
 - edgePadding: A float that represents the padding between the edge of the inventory panel and the slots.

```
public GameObject[,] slotGrid;  
public GameObject slotPrefab;  SlotPrefab  
public IntVector2 gridSize;  Serializable  
public float slotSize;  "80"  
public float edgePadding;  "7"
```

2. The script has an "Awake" method that initializes the slotGrid array, resizes the panel, creates the slots, and sets the grid size for the InventoryGridManager component.

```
 Event function  PignataroLucas  
private void Awake()  
{  
    slotGrid = new GameObject[gridSize.x, gridSize.y];  
    ResizePanel();  
    CreateSlots();  
    GetComponent<InventoryGridManager>().GridSize = gridSize;  
}
```

3. The "CreateSlots" method creates the inventory slots as follows:
 - It iterates through the gridSize in a nested for loop.
 - Instantiates a new slot using the slotPrefab.
 - Sets the slot name, parent, and position according to the current grid coordinates.
 - Sets the slot size based on the slotSize variable.

- Sets the local scale of the RectTransform component to Vector3.one.
- Sets the GridPos of the Slots component using the current grid coordinates.
- Adds the newly created slot to the slotGrid array.
- Sets the SlotGrid property of the InventoryGridManager component with the filled slotGrid array.

```

1 usage  PignataroLucas
private void CreateSlots()
{
    for (int y = 0; y < gridSize.y; y++)
    {
        for (int x = 0; x < gridSize.x; x++)
        {
            GameObject obj = Instantiate(slotPrefab);

            obj.transform.name = "slot[" + x + "," + y + "]";
            obj.transform.SetParent(transform);
            RectTransform rectTransform = obj.transform.GetComponent<RectTransform>();
            rectTransform.localPosition =
                new Vector3(x * slotSize + edgePadding, y * slotSize + edgePadding, 0);
            rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, slotSize);
            rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Vertical, slotSize);
            obj.GetComponent<RectTransform>().localScale = Vector3.one;
            obj.GetComponent<Slots>().GridPos = new IntVector2(x, y);
            slotGrid[x, y] = obj;
        }
    }

    GetComponent<InventoryGridManager>().SlotGrid = slotGrid;
}

```

4. The "ResizePanel" method resizes the inventory panel based on the gridSize, slotSize, and edgePadding variables. It updates the width and height of the RectTransform component accordingly and sets its local scale to Vector3.one.

```

1 usage  PignataroLucas
private void ResizePanel()
{
    float width, height;
    width = (gridSize.x * slotSize) + (edgePadding * 2);
    height = (gridSize.y * slotSize) + (edgePadding * 2);

    RectTransform rectTransform = GetComponent<RectTransform>();
    rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, width);
    rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Vertical, height);
    rectTransform.localScale = Vector3.one;
}
}

```

This script creates a customizable grid-based inventory system by instantiating inventory slot prefabs, organizing them in a grid, and resizing the panel to fit the slots.

Slots : This script defines the "Slots" class for the inventory system within the "InventoryGridSystem" namespace. Each slot in the inventory grid represents an individual cell in the grid where an item can be placed. The script defines the following properties and methods:

1. Public properties for the slot:

- GridPos: An IntVector2 (a custom struct with two integer components, x and y) representing the slot's position in the grid.
- text: A Text component to display the position of the slot within the grid.
- storedItemObject: A GameObject that represents the item stored in the slot.
- storedItemSize: An IntVector2 that represents the size of the stored item.
- storedItemStartPos: An IntVector2 that represents the starting position of the stored item.
- storedItemClass: A GameItem class that represents the item's properties.
- isOccupied: A boolean that indicates if the slot is occupied by an item.

```
public IntVector2 GridPos;  ⚙️ Serializable
public Text text;  ⚙️ Changed in 2 assets

public GameObject storedItemObject;  ⚙️ Unchanged
[FormerlySerializedAs( oldName: "StoredItemSize")] public IntVector2 storedItemSize;  ⚙️ Serializable
[FormerlySerializedAs( oldName: "StoredItemStartPos")] public IntVector2 storedItemStartPos;  ⚙️ Serializable
public GameItem storedItemClass;  ⚙️ Serializable
public bool isOccupied;  ⚙️ Unchanged
```

2. A constructor for the Slots class that takes an IntVector2 argument (gridPos) and assigns it to the GridPos property.

```
⚙️ PignataroLucas
public Slots(IntVector2 gridPos)
{
    GridPos = gridPos;
}
```

3. A Start method that updates the "text" component's text to display the slot's position (GridPos) in the grid.

```
⚙️ Event function ⚙️ PignataroLucas
private void Start()
{
    text.text = GridPos.x + "," + GridPos.y;
}
```

This script defines the structure of an individual slot in the inventory grid system, stores its position in the grid, and displays its position using a Text component. The slot also keeps track of the stored item and its properties, such as size and starting position, and indicates whether the slot is occupied or not.

InventoryGridManager : This script manages the inventory grid, handles item placement and swapping, and updates the grid's visual appearance based on different conditions.

The script contains the following key properties and methods:

1. Public properties:

- **SlotGrid**: A 2D array of GameObjects representing the slots in the inventory grid.
- **highlightedSlot**: The currently highlighted slot in the grid.
- **dropParent**: A Transform object that serves as a parent object for items placed in the grid.
- **GridSize**: The size of the grid in the form of an IntVector2 (a custom struct with two integer components, x and y).
- **listManager**: A reference to the ItemListManager class that manages the list of items available for the inventory.

```
public class InventoryGridManager : MonoBehaviour
{
    public GameObject[,] SlotGrid;
    public GameObject highlightedSlot;  ⚙️ Unchanged
    public Transform dropParent;  ⚙️ DropParent (Transform)

    [HideInInspector] public IntVector2 GridSize;  ⚙️ Serializable

    public ItemListManager listManager;  ⚙️ Unchanged
}
```

2. Private properties and fields for internal calculations and temporary storage of data.

```
private IntVector2 _totalOffset, _checkSize, _checkStartPos;
private IntVector2 _otherItemPos, _otherItemSize;

private int _checkState;
private bool _isOverEdge;
```

3. **Update()** method: Handles the user's interaction with the grid by checking the highlighted slot and the selected item. It determines the action to take based on the current state, such as storing an item, swapping items, or retrieving an item from the grid.

```

Eventfunction & Pignatarolucas *
private void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        if (highlightedSlot != null && InventoryItemInteraction.selectedItem != null && !_isOverEdge )
        {
            switch (_checkState)
            {
                case 0 :
                    StoreItem(InventoryItemInteraction.selectedItem);
                    ColorChangeLoop(SlotColorHighlights.Blue, InventoryItemInteraction.selectedItemSize,
                        startPos, totalOffset);
                    InventoryItemInteraction.ResetSelectedItem();

                    break;

                case 1 :
                    InventoryItemInteraction.SetSelectableItem(obj SwapItem(InventoryItemInteraction.selectedItem));
                    InventorySlotHighlighter.selector.PossOffset();
                    ColorChangeLoop(SlotColorHighlights.Gray , _otherItemSize , _otherItemPos);
                    RefreshColor(enter: true);

                    break;
            }
        }
        else if (highlightedSlot != null && InventoryItemInteraction.selectedItem == null && highlightedSlot.GetComponent<Slots>().isOccupied == true)
        {
            ColorChangeLoop(SlotColorHighlights.Gray , highlightedSlot.GetComponent<Slots>().storedItemSize ,
                highlightedSlot.GetComponent<Slots>().storedItemStartPos);
            InventoryItemInteraction.SetSelectableItem(obj GetItem(highlightedSlot));
            InventorySlotHighlighter.selector.PossOffset();
            RefreshColor(enter: true);
        }
    }
}

```

4. CheckArea(), SlotCheck(), RefreshColor(), ColorChangeLoop(), SecondColorChangeLoop(): These methods are responsible for checking the available space in the grid, verifying whether the item can be placed, updating the colors of the slots based on the item's position, and changing the colors to indicate valid or invalid placements.
5. StoreItem(), GetItem(), SwapItem(): These methods handle the placement, retrieval, and swapping of items in the inventory grid.
6. SlotColorHighlights struct: A collection of static color properties used to represent the different colors used for highlighting slots in the inventory grid.

This script manages the inventory grid by handling item placement, swapping, and retrieval. It also updates the visual appearance of the grid based on the position and status of items within the grid.

InventoryItemInteraction : This script is responsible for managing the interaction of inventory items in an inventory grid system.

The script contains the following key properties and methods:

1. Public properties:
 - item: A reference to the GameItem class that represents the item being managed by this script

```
public GameItem item;
```

2. Private properties & Statics:
 - _inventoryPanel: A reference to the inventory panel UI object.

- selectedItem: A static reference to the currently selected item in the inventory grid.
- selectedItemSize: A static IntVector2 representing the size of the selected item.
- isDragging: A static bool indicating if the user is currently dragging an item.
- _slotSize: A float representing the size of a slot in the inventory grid.

```
private GameObject _inventoryPanel;
private float _slotSize;

public static GameObject selectedItem;
public static IntVector2 selectedItemSize;
public static bool isDragging = false;
```

3. Awake() method: Initializes the _slotSize property by getting a reference to the "InvenPanel" GameObject and retrieving its slot size from the InventoryGridCreator component.

```
Event function PignataroLucas
private void Awake()
{
    _slotSize = GameObject.FindGameObjectWithTag("InvenPanel").GetComponent<InventoryGridCreator>().slotSize;
}
```

4. Update() method: If an item is being dragged, it updates the item's position to follow the mouse cursor.

```
Event function PignataroLucas
private void Update()
{
    if (isDragging) selectedItem.transform.position = Input.mousePosition;
}
```

5. SetItemObject() method: Sets up the item object in the inventory grid by resizing the RectTransform to fit the item's size and setting the item's sprite to the passed GameItem's icon.

```
1 usage PignataroLucas
public void SetItemObject(GameItem passedItem)
{
    RectTransform rectTransform = GetComponent<RectTransform>();
    rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, size: passedItem.size.x * _slotSize);
    rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Vertical, size: passedItem.size.y * _slotSize);
    item = passedItem;
    GetComponent<Image>().sprite = passedItem.icon;
}
```

6. SetSelectableItem() method: A static method that sets the selected item to the provided GameObject, updates the selectedItemSize, sets isDragging to true, and sets the item's parent to the "DragParent" GameObject.

```

Frequently called 3 usages PignataroLucas
public static void SetSelectableItem(GameObject obj)
{
    selectedItem = obj;
    selectedItemSize = obj.GetComponent<InventoryItemInteraction>().item.size;
    isDragging = true;
    obj.transform.SetParent(GameObject.FindGameObjectWithTag("DragParent").transform);
    obj.GetComponent<RectTransform>().localScale = Vector3.one;
}

```

7. ResetSelectedItem() method: A static method that resets the selectedItem, selectedItemSize, and isDragging properties to their default values.

```

Frequently called 1 usage PignataroLucas
public static void ResetSelectedItem()
{
    selectedItem = null;
    selectedItemSize = IntVector2.Zero;
    isDragging = false;
}

```

This script manages the interaction of individual items in an inventory grid system, such as dragging and positioning the items within the grid, as well as updating the items' appearance and properties based on user interactions.

InventorySlotHighlighter : This script is responsible for highlighting the inventory slots when the user interacts with them using the mouse.

The script contains the following key properties and methods:

1. Inherits from MonoBehaviour and implements the IPointerEnterHandler and IPointerExitHandler interfaces, which are used for handling mouse pointer events.
2. Public properties:
 - slotParent: A reference to the parent GameObject of the inventory slot.
 - posOffset: A static IntVector2 representing the offset position when an item is dragged.
 - selector: A static reference to the InventorySlotHighlighter instance that the pointer is currently over.
 - quadNum: An integer representing the quadrant number of the inventory slot.

```

public GameObject slotParent;  Changed in 2 assets
public static IntVector2 posOffset;
public static InventorySlotHighlighter selector;
public int quadNum;  Changed in 2 assets

```

3. Private properties:
 - _gridManager: A reference to the InventoryGridManager component attached to the parent GameObject.

- `_parentSlot`: A reference to the Slots component attached to the slotParent GameObject.

```
private InventoryGridManager _gridManager;
private Slots _parentSlot;
```

4. `Start()` method: Initializes the `_gridManager` and `_parentSlot` properties.

```
Event function PignataroLucas
private void Start()
{
    _gridManager = this.gameObject.transform.parent.parent.GetComponent<InventoryGridManager>();
    _parentSlot = slotParent.GetComponent<Slots>();
}
```

5. `OnPointerEnter()` method: Triggers when the mouse pointer enters the inventory slot. Sets the selector to the current instance, updates the `_gridManager`'s `highlightedSlot`, calls the `PossOffset()` method, and refreshes the slot colors depending on the current state of the `selectedItem` and `storedItemObject`.

```
PignataroLucas *
public void OnPointerEnter(PointerEventData eventData)
{
    selector = this;
    _gridManager.highlightedSlot = slotParent;
    PossOffset();

    if (InventoryItemInteraction.selectedItem != null) _gridManager.RefreshColor(enter: true);
    if (_parentSlot.storedItemObject != null && InventoryItemInteraction.selectedItem == null)
        _gridManager.ColorChangeLoop(SlotColorHighlights.Blue , _parentSlot.storedItemSize, _parentSlot.storedItemStartPos);
}
```

6. `PossOffset()` method: Updates the `posOffset` based on the `selectedItemSize` and the current quadrant number.

Frequently called 3 usages PignataroLucas

```
public void PossOffset()
{
    if (InventoryItemInteraction.selectedItemSize.x != 0 &&
        InventoryItemInteraction.selectedItemSize.x % 2 == 0)
    {
        switch (quadNum)
        {
            case 1:
                posOffset.x = 0; break;
            case 2:
                posOffset.x = -1; break;
            case 3:
                posOffset.x = 0; break;
            case 4:
                posOffset.x = -1; break;
        }
    }

    if (InventoryItemInteraction.selectedItemSize.y != 0
        && InventoryItemInteraction.selectedItemSize.y % 2 == 0)
    {
        switch (quadNum)
        {
            case 1:
                posOffset.y = -1; break;
            case 2:
                posOffset.y = -1; break;
            case 3:
                posOffset.y = 0; break;
            case 4:
                posOffset.y = 0; break;
        }
    }
}
```

7. OnPointerExit() method: Triggers when the mouse pointer exits the inventory slot. Resets the selector, highlightedSlot, and posOffset properties, and refreshes the slot colors based on the current state of the selectedItem and storedItemObject.

```

PignataroLucas
public void OnPointerExit(PointerEventData eventData)
{
    selector = null;
    _gridManager.highlightedSlot = null;
    if(InventoryItemInteraction.selectedItem != null) _gridManager.RefreshColor(enter: false);
    posOffset = IntVector2.Zero;
    if(_parentSlot.storedItemObject != null && InventoryItemInteraction.selectedItem == null)
        _gridManager.ColorChangeLoop(SlotColorHighlights.Blue2, _parentSlot.storedItemSize, _parentSlot.storedItemStartPos);
}

```

Resume :

The InventoryItemInteraction script manages the behavior of individual items within the inventory. It allows users to select, drag, and drop items in the inventory. It also sets the size and appearance of the item based on the Gameltem data passed to it.

The InventorySlotHighlighter script manages the behavior of the inventory slots. It highlights the slots when the user interacts with them using the mouse. It also updates the appearance of the slots based on the current state of the selected item and the items stored in the slots.

- When the user clicks on an item, the InventoryItemInteraction.SetSelectableItem() method is called, setting the selectedItem and enabling dragging by setting isDragging to true.
- While dragging the item, the InventoryItemInteraction.Update() method continuously updates the item's position to match the mouse position.
- When the user hovers over an inventory slot with a selected item, the InventorySlotHighlighter.OnPointerEnter() method is called. This method updates the InventoryGridManager's highlightedSlot, and calls the PosOffset() method to update the position offset of the selected item.
- Based on the current state of the selected item and the items stored in the slots, the OnPointerEnter() method also calls _gridManager.RefreshColor() and _gridManager.ColorChangeLoop() to update the appearance of the slots.
- When the user stops hovering over a slot, the InventorySlotHighlighter.OnPointerExit() method is called. It resets the selector, highlightedSlot, and posOffset properties, and updates the slot colors based on the current state of the selected item and the items stored in the slots.
- If the user drops an item into a valid slot, the item's position will be updated, and the InventoryItemInteraction.ResetSelectedItem() method will be called to reset the selectedItem, selectedItemSize, and isDragging properties.

These scripts create a functional inventory grid system where users can click, drag, and drop items in the inventory, and the inventory slots are highlighted based on user interactions and the current state of the items.

Item

Game item : This script contains information about the item's properties, such as its ID, type, category, level, quality, and appearance. The class also provides methods to set and retrieve these values.

The script contains the following key properties and methods:

1. `globalID`: A unique identifier for the item.
2. `typeName`: The name of the item's type.
3. `categoryID`, `typeID`, `categoryName`, `serialID`: Hidden attributes for the item's category and type, used internally for organizing items.
4. `icon`: A sprite representing the item's appearance in the inventory or UI.
5. `size`: The size of the item in the inventory, represented as an `IntVector2`.
6. `level`: The item's level, which can be a value between 1 and 100.
7. `qualityInt`: An integer representing the item's quality, ranging from 0 (Common) to 4 (Legendary).
8. `Quality`: An internal enumeration for item qualities, used for mapping `qualityInt` values to their corresponding names.
9. `GetQuality()`: A method that returns the string representation of the item's quality.
10. `SetItemValues()`: Two overloaded methods that set item values based on the given parameters or the default values from the `ItemDatabaseLoader` component.
11. `GamelItem()`: Two constructors for the `GamelItem` class, one of which creates a new `GamelItem` based on an existing one, and the other initializes an empty `GamelItem`.

```

public class GameItem
{
    public int globalID;  ⚡ Serializable
    public string typeName;  ⚡ Serializable

    [HideInInspector] public int categoryID;  ⚡ Serializable
    [HideInInspector] public int typeID;  ⚡ Serializable
    [HideInInspector] public string categoryName;  ⚡ Serializable
    [HideInInspector] public string serialID;  ⚡ Serializable
    [HideInInspector] public Sprite icon;  ⚡ Serializable
    [HideInInspector] public IntVector2 size;  ⚡ Serializable

    [Range(1, 100)] public int level;  ⚡ Serializable
    [Range(0, 4)] public int qualityInt;  ⚡ Serializable

    1 usage  ⚡ PignataroLucas  5 exposing APIs
    private enum Quality {Common, UnCommon, Rare, Epic, Legendary}

    ⚡ PignataroLucas
    public string GetQuality()
    {
        return Enum.GetName(typeof(Quality), qualityInt);
    }

    1 usage  ⚡ PignataroLucas
    public static void SetItemValues(GameItem item, int id, int lvl, int quality)
    {
        item.globalID = id;
        item.level = lvl;
        item.qualityInt = quality;
        GameObject.FindGameObjectWithTag("ItemDatabase").GetComponent<ItemDatabaseLoader>().PassItemData(ref item);
    }

    ⚡ PignataroLucas
    public static void SetItemValues(GameItem item)
    {
        GameObject.FindGameObjectWithTag("ItemDatabase").GetComponent<ItemDatabaseLoader>().PassItemData(ref item);
    }

    ⚡ PignataroLucas
    public GameItem(GameItem passedItem)
    {
        globalID = passedItem.globalID;
        level = passedItem.level;
        qualityInt = passedItem.qualityInt;
    }

    1 usage  ⚡ PignataroLucas
    public GameItem(){}

```

The GameItem class serves as a blueprint for creating and managing items in the game. It works together with other scripts, like the inventory system and item database, to provide a complete item management solution.

CreateItemManager : This script is responsible for creating random items in the game. It can either spawn the item directly in the game world or add it to a list.

Here's a breakdown of the class and its members:

1. `willAddToList`: A boolean variable that determines if the created item will be added to a list or spawned directly in the game world.
2. `poolManager`: A reference to an `ObjectPoolManager` component, which is responsible for managing the item objects pool.
3. `RandomItem()`: A method that creates a random `GameItem` if there is no currently selected item in the inventory. It sets the item's values by calling `GameItem.SetItemValues()` with random parameters for the item's ID, level, and quality. Then, it calls `SpawnOrAddItem()` to either spawn the item in the game world or add it to a list.
4. `SpawnOrAddItem(GameItem item)`: A method that takes a `GameItem` as an input parameter. If `willAddToList` is set to false, it gets an object from the object pool using `poolManager.GetObject()` and sets the item's properties using `itemObject.GetComponent<InventoryItemInteraction>().SetItemObject(item)`. Then, it calls `InventoryItemInteraction.SetSelectableItem(itemObject)` to make the item selectable. If `willAddToList` is set to true, it logs a warning that the item has been added to a list (although no actual list implementation is present in the script).

```
C# CreateItemManager.cs
1  using InventoryGridSystem;
2  using UnityEngine;
3  using Utility;
4
5  namespace Item
6  {
7      1 asset usage PignataroLucas
      public class CreateItemManager : MonoBehaviour
8      {
9          public bool willAddToList = false; 1 asset usage PignataroLucas
10         public ObjectPoolManager poolManager; 1 asset usage PignataroLucas
11
12
13         1 asset usage PignataroLucas
14         public void RandomItem()
15         {
16             if (InventoryItemInteraction.selectedItem == null)
17             {
18                 GameItem newItem = new GameItem();
19                 GameItem.SetItemValues(newItem, id: Random.Range(0, 4), lvl: Random.Range(1, 101), quality: Random.Range(1, 5));
20                 SpawnOrAddItem(newItem);
21             }
22         }
23
24         1 usage PignataroLucas
25         private void SpawnOrAddItem(GameItem item)
26         {
27             if (willAddToList == false)
28             {
29                 GameObject itemObject = poolManager.GetObject();
30                 itemObject.GetComponent<InventoryItemInteraction>().SetItemObject(item);
31                 InventoryItemInteraction.SetSelectableItem(itemObject);
32             }
33             else
34             {
35                 Debug.LogWarning(message: "Item Added to List");
36             }
37         }
38     }
39 }
```

DataBase

This script is responsible for loading item data from a CSV file and storing it in a database-like structure in the game.

Here's a breakdown of the class and its members:

1. `dbFile`: A `TextAsset` that represents the CSV file containing the item data.
2. `typeNameList`: A list of strings storing the type names of the items.
3. `dbList`: A list of `ItemData` objects representing the item database.
4. `Awake()`: A Unity lifecycle method that is called when the script is initialized. It calls the `LoadDB()` method to load the item data from the CSV file.
5. `ItemData` class: A nested class representing a single row of item data. It has properties such as `globalID`, `categoryID`, `categoryName`, `typeID`, `typeName`, `size`, and `icon`.
6. `LoadDB(TextAsset textAsset)`: A method that takes a `TextAsset` as an input parameter and loads item data from the CSV file using `CsvFileHandler.LoadTextFile(textAsset)`. It creates an `ItemData` object for each row in the CSV file and fills it with the parsed data. The method then adds the `typeName` to the `typeNameList` and the `ItemData` object to the `dbList`.
7. `PassItemData(ref GameItem item)`: A method that takes a reference to a `GameItem` object and fills its properties using the data from the `dbList`. It first retrieves the item data from the `dbList` using the item's `globalID` and then sets the item's `categoryID`, `categoryName`, `typeID`, `typeName`, `size`, and `icon`.

1 asset usage 2 usages PignataroLucas

```
public class ItemDatabaseLoader : MonoBehaviour
```

```
{
```

```
    public TextAsset dbFile; 1 Serializable
```

```
    public List<string> typeNameList = new List<string>(); 1 Serializable
```

```
    public List<ItemData> dbList = new List<ItemData>();
```

Event function PignataroLucas

```
    private void Awake()
```

```
    {
```

```
        LoadDB(dbFile);
```

```
    }
```

4 usages PignataroLucas

```
    public class ItemData
```

```
    {
```

```
        public int globalID;
```

```
        public int categoryID;
```

```
        public string categoryName;
```

```
        public int typeID;
```

```
        public string typeName;
```

```
        public IntVector2 size;
```

```
        public Sprite icon;
```

```
    }
```

1 usage PignataroLucas

```
    private void LoadDB(TextAsset textAsset)
```

```
    {
```

```
        string[][] grid = CsvFileHandler.LoadTextFile(textAsset);
```

```
        for (int i = 1; i < grid.Length; i++)
```

```
        {
```

```
            ItemData row = new ItemData();
```

```
            row.globalID = Int32.Parse(grid[i][0]);
```

```
            row.categoryID = Int32.Parse(grid[i][1]);
```

```
            row.categoryName = grid[i][2];
```

```
            row.typeID = Int32.Parse(grid[i][3]);
```

```
            row.typeName = grid[i][4];
```

```
            typeNameList.Add(row.typeName);
```

```
            row.size = new IntVector2(Int32.Parse(grid[i][5]), Int32.Parse(grid[i][6]));
```

```
            row.icon = Resources.Load<Sprite>("itemIcons/" + grid[i][4]);
```

```
            dbList.Add(row);
```

```
        }
```

```
    }
```

2 usages PignataroLucas

```
    public void PassItemData(ref GameItem item)
```

```
    {
```

```
        int id = item.globalID;
```

```
        item.categoryID = dbList[id].categoryID;
```

```
        item.categoryName = dbList[id].categoryName;
```

```
        item.typeID = dbList[id].typeID;
```

```
        item.typeName = dbList[id].typeName;
```

```
        item.size = dbList[id].size;
```

```
        item.icon = dbList[id].icon;
```

```
    }
```

```
}
```

```
}
```

Utility

IntVector2 : This script is a representation of a two-dimensional (2D) vector that uses integers instead of floating-point numbers as in Unity's Vector2 structure.

Here is a breakdown of the IntVector2 structure and its members:

1. Fields x and y: These two integer fields store the x and y components of the IntVector2.
2. Constructors: There are three constructors provided to create IntVector2 instances from different input types, such as two integers, two floats, or a Vector2.
3. ToString(): A method that returns a string representation of the IntVector2.
4. Static properties: One, OneNeg, Zero, Up, Down, Left, Right are pre-defined IntVector2 instances representing common values like (1,1), (0,0), (0,1), etc.
5. Operator overloads: The IntVector2 structure provides overloaded operators for addition, subtraction, multiplication, and division with other IntVector2 instances or integers. There are also equality and inequality operators.
6. Utility methods: The Area(), Slope(), Swap(), Vector2(), Vector3(), OrGreater() and OrLesser() methods are provided to perform common operations or comparisons on IntVector2 instances.

CsvFileHandler : This script read and writing CSV files. The class provides two static methods for handling CSV files represented as TextAsset objects: LoadTextFile and WriteToFile.

1. LoadTextFile(TextAsset textFile): This method takes a TextAsset as input, which represents the CSV file to read. It reads the content of the file and splits it into rows and columns based on newline (\n) and comma (,) characters. The method returns a jagged string array (an array of string arrays), where the first index represents the row, and the second index represents the column.
2. If the input TextAsset is null, the method prints a warning message "File Does Not Exist." and returns null.
3. WriteToFile(string[][] strList, TextAsset textFile): This method takes a jagged string array (strList) and a TextAsset object (textFile) as inputs. The jagged string array represents the content to write to the CSV file, with the first index representing the row and the second index representing the column.
4. The method first converts the jagged string array into an array of strings, where each string represents a row in the CSV file. Then, it writes the string array to the file represented by the TextAsset object. If the input TextAss


```

using System.Linq;
using System.Net;
using UnityEditor;
using UnityEngine;

namespace Utility
{
    No asset usages 1 usage Pignatarolucas
    public class CsvFileHandler : MonoBehaviour
    {
        1 usage Pignatarolucas
        public static string[][] LoadTextFile(TextAsset textFile)
        {
            if (textFile != null)
            {
                string[] row = textFile.text.Split(separator: '\n');
                string[][] grid = new string[row.Length][];
                for (int i = 0; i < row.Length; i++)
                {
                    grid[i] = row[i].Split(separator: ',');
                }

                return grid.ToArray();
            }
            else
            {
                Debug.LogWarning(message: "File Does Not Exist.");
                return null;
            }
        }

        Pignatarolucas
        public static void WriteToFile(string[][] strList, TextAsset textFile)
        {
            if (textFile != null)
            {
                string[] lines = new string[strList.Length];
                for (int i = 0; i < strList.Length; i++)
                {
                    for (int j = 0; j < strList[i].Length; j++)
                    {
                        lines[i] += strList[i][j];
                    }
                }
                File.WriteAllLines(AssetDatabase.GetAssetPath(textFile), lines);
            }
            else
            {
                Debug.LogWarning(message: "File Does Not Exist.");
            }
        }
    }
}

```

ObjectPoolManager :

The script contains two classes:

1. **ObjectPoolManager**: This class is responsible for managing the pool of objects. It has the following properties and methods:
 - **prefab**: A `GameObject` reference to the prefab that the object pool will create and manage.
 - **_inactiveInstances**: A stack of `GameObjects` containing the inactive instances of the prefab.
 - **GetObject()**: A method that returns an available `GameObject` from the pool. If there are inactive instances, it pops one from the stack and sets it active. If there are no inactive instances, it instantiates a new `GameObject` from the prefab, adds a `PooledObject` component to it, and sets it active.
 - **ReturnObject(GameObject toReturn)**: A method that returns a `GameObject` to the object pool. It first checks if the `GameObject` has a `PooledObject` component and if it belongs to the same pool. If the checks pass, it deactivates the `GameObject`, resets its transform, and pushes it back onto the stack of inactive instances. If the checks fail, it prints a warning and destroys the `GameObject`.
2. **PooledObject**: This is a simple class with a single property `pool` of type `ObjectPoolManager`. It is added as a component to each pooled `GameObject` to keep a reference to the pool it belongs to. This helps the `ReturnObject` method identify if the returned object belongs to the correct pool.