Connectopia

Connectopia is a powerful Unity package offering essential functionalities for efficient variable management, event triggering, and data persistence within your Unity projects.

Initially built upon the foundation of GUI Pro Casual Game

Version History

v1.0.0 (Initial Release): Basic functionality for storing, managing, and triggering variables, values, and events.

Scriptable Objects

Overview

The SerializableScriptableObject is a utility class in the Zanoshky namespace, extending Unity's ScriptableObject. This asset introduces serialization support by storing a unique identifier (GUID) for each instance, facilitating efficient asset tracking and management.

Features

- Serialization Support: Automatically generates and stores a GUID for each instance during asset validation.
- Unity Editor Integration: Utilizes Unity Editor directives to enhance functionality exclusively in editor mode.

Unity Editor Integration

The SerializableScriptableObject class provides enhanced functionality when used within the Unity Editor.

GUID Generation

The GUID is automatically generated and assigned during the validation process, ensuring a unique identifier for each instance. This identifier is useful for tracking and managing assets.

OnValidate Method

The OnValidate method is invoked in the Unity Editor during the validation process. It retrieves the asset's path and converts it into a GUID, ensuring consistency between the asset and its serialized form.

Use Case Example

```
using UnityEngine;

namespace YourNamespace
{
    public class YourScriptableObject :
```

```
Zanoshky.SerializableScriptableObject
    {
        // Your scriptable object properties and methods
    }
}
```

```
YourScriptableObject scriptableObject = default;
// Assign via Unity Editor
// Instantiate or reference your scriptable object;
string guid = scriptableObject.Guid;
```

```
#if UNITY_EDITOR
void OnValidate()
{
    var path = AssetDatabase.GetAssetPath(this);
    _guid = AssetDatabase.AssetPathToGUID(path);
}
#endif
```

```
using UnityEngine;

#if UNITY_EDITOR
using UnityEditor;
#endif

namespace Zanoshky
{
    public class ExampleScriptableObject : SerializableScriptableObject
    {
        [SerializeField] private int exampleValue;

        public int ExampleValue => exampleValue;

        #if UNITY_EDITOR
        // Additional editor-specific functionality
        #endif
    }
}
```

Unity Editor Tips

- Ensure the SerializableScriptableObject instances are validated in the Unity Editor for the proper generation of GUIDs.
- Leverage the GUIDs for asset tracking and management within your Unity project.

Variables

Overview

The xxxVariable is a Unity ScriptableObject designed to store and manage various primitive and unity based variables. This asset provides a convenient way to create, modify, and reference various values within your Unity project.

The DefaultValue field is non-serialized and is used internally to reset the Current value when deserializing. You can set the default value in the Unity Editor.

Use Case Example

BoolEventChannelSO class is particularly useful for managing events related to boolean states, such as toggling a UI interface, open and close panel for example. By using this class, you can create a centralized communication channel for boolean events across different parts of your Unity project.

Asset Creation

This class can be created as an asset in the Unity Editor using the CreateAssetMenu attribute. In the Unity Editor, right-click in the Project window. Navigate to "Create" > "Zanoshky" > "Variables"

Example Usage

```
public class ExampleUsage : MonoBehaviour
{
    public DoubleVariable myDoubleVariable;

    void Start()
    {
        // Accessing the current value
        double currentValue = myDoubleVariable.Current;

        // Modifying the current value
        myDoubleVariable.Current = 42.0;

        // Resetting to the default value
        myDoubleVariable.Current = myDoubleVariable.DefaultValue;
    }
}
```

GameState

Overview

The GameStateSO Unity ScriptableObject is a versatile asset designed to manage and persistently store game-related data such as scores, time, settings, and authorization information. This asset provides a centralized location for managing the state of your game, making it easy to save and load important data.

Features

- Modular Design: Organizes game-related data into categories like Debug, Game, Settings, and Authorization.
- Variable Integration: Utilizes various custom ScriptableObjects (IntVariable, LongVariable, FloatVariable, BoolVariable, StringVariable) to represent different types of game data.
- Debug Logging: Includes a debug log to record events during development and testing.
- CreateAssetMenu Integration: Allows for easy creation of instances through Unity's Asset menu.

Ensure that you have assigned appropriate default values to the variables in the Unity Editor. The use of custom ScriptableObjects (IntVariable, LongVariable, FloatVariable, BoolVariable, StringVariable) allows for easy integration with other systems and promotes a modular design.

Use Case Example

The GameStateSO class is extreamly useful to store all various game tracking values, economy, settings, user info, mobile information, in a single, or partitioned way to refence in various scripts only as a one entry field.

Asset Creation

This class can be created as an asset in the Unity Editor using the CreateAssetMenu attribute. In the Unity Editor, right-click in the Project window. Navigate to "Create" > "Zanoshky" > "GameState"

Example Usage

```
public class ExampleUsage : MonoBehaviour
{
    public GameStateSO gameState;

    void Start()
    {
        int currentScore = gameState.score.Current;
        float musicVolume = gameState.settingMusic.Current;

        // Accessing and modifying game data
        int currentScore = gameState.score.Current;
        gameState.score.Current += 10;

        // Logging debug events
        gameState.logEvents.Add("Game started");

        // Accessing authorization information
        string googlePlayAuthCode = gameState.googlePlayAuthCode.Current;

        // Access and modify other variables as needed.
    }
}
```

Events

Overview

The xxxEventChannelSO class facilitates the implementation of events in Unity, specifically designed for scenarios where events involve primite/unity values. It extends SerializableScriptableObject, enabling easy integration into the Unity Editor as a scriptable object asset.

Use Case Example

BoolEventChannelSO class is particularly useful for managing events related to boolean states, such as toggling a UI interface, open and close panel for example. By using this class, you can create a centralized communication channel for boolean events across different parts of your Unity project.

Asset Creation

This class can be created as an asset in the Unity Editor using the CreateAssetMenu attribute. In the Unity Editor, right-click in the Project window. Navigate to "Create" > "Zanoshky" > "Events"

Example Usage

```
using UnityEngine;

public class ExampleListener : MonoBehaviour
{
    public BoolEventChannelSO boolEventChannel;

    private void OnEnable()
    {
        // Subscribe to the event
        boolEventChannel.OnEventRaised += HandleEvent;
    }

    private void OnDisable()
    {
        // Unsubscribe from the event to prevent memory leaks
        boolEventChannel.OnEventRaised -= HandleEvent;
    }

    private void HandleEvent(bool value)
    {
        // Do something with the boolean value received from the event
        Debug.Log("Event received with value: " + value);
    }
}
```

Tips

Subscribing and Unsubscribing: Properly subscribe to and unsubscribe from events in the OnEnable and OnDisable methods to avoid memory leaks. Null Check: Always check if the event is null before invoking it to prevent null reference exceptions. Asset Naming: Consider giving meaningful names to your BoolEventChannelSO assets to improve project organization and readability.

Conclusion

The Connectopia package provides a straightforward and reusable mechanism for handling all various events in Unity. Use it to enhance communication between components, making your code modular and maintainable.