API Extensibility Manual

ProjectDoors API Extensibility Manual

1. Introduction

The **ProjectDoors** API provides a modular and extensible framework for managing doors, interactions, and character movement within Unity projects. This manual offers an overview of key components, including event channels, controllers, and editor utilities, and guides developers on extending the API.

2. Event Channels

Event channels facilitate communication between objects without direct dependencies.

- **MovementControlEventChannel.cs**
  Raises `OnStopMovement` and `OnResumeMovement` events for movement controlMovementControlEventCha....
- **DoorEventChannel.cs**
  Handles `OnOpenRequest` and `OnCloseRequest` events with position and force parametersDoorEventChannel.
- **InteractionEventChannel.cs**
  Sends `OnInteractRequest` events with object positions, useful for triggering door interactionsInteractionEventChannel.
- **ProximityEventChannel.cs**
  Manages `OnEnterProximity` and `OnExitProximity` events when the player enters or exits a specified rangeProximityEventChannel.

3. Core Components

- **Door.cs**
  Controls door behaviors including sliding, rotation, dual-door setups, and audio integration. Supports pivot adjustments for custom rotation pointsDoor.
- **DoorLever.cs**
  Allows lever-controlled doors with customizable rotation, sound effects, and proximity-based interactionsDoorLever.
- **DoorTrigger.cs**
  Automates door opening/closing when objects enter/exit trigger zones, with configurable delaysDoorTrigger.
- **ClickAgentController.cs**
  Manages movement for GameObjects using `NavMeshAgent` or `CharacterController`, providing `StopAgentMovement` and `ResumeAgentMovement` methods ClickAgentController.
- **ProximityStopper.cs**
  Stops player movement within proximity of objects and resumes after a delay, integrating with `ClickAgentController`ProximityStopper.
- **RandomWanderAI.cs**
  Implements AI for random wandering within defined areas using NavMesh agentsRandomWanderAI.
- **LayeredNavMeshObstacle.cs**
  Manages dynamic NavMesh obstacles activated based on object proximity and layer prioritiesLayeredNavMeshObstacle.

4. Editor Tools

- **DoorEditor.cs**
  Extends the Unity Editor to provide visual tools for configuring doors, pivots, dual-door setups, and lever controlsDoorEditor.
- **PlayerActionsEditor.cs**
  Enables in-editor customization of player input bindings from Unity's New Input SystemPlayerActionsEditor.
- **DoorCreationWizard.cs**
  Provides a step-by-step wizard for setting up new doors and player charactersDoorCreationWizard.
- **DoorHierarchyIcons.cs**
  Displays custom icons in the Unity Hierarchy for doors with active pivots or knobs in edit modeDoorHierarchyIcons.

5. Extending the API

- **Adding Custom Event Channels**:
  Inherit from `ScriptableObject` and define new events. Register events in existing components using `Resources.Load<T>()`.

Creating a New Event Channel

```
using UnityEngine;
using System;

[CreateAssetMenu(menuName = "ProjectDoors/EventChannels/CustomEventChannel")]
public class CustomEventChannel : ScriptableObject
{
    public event Action<string> OnCustomEvent;

    public void RaiseCustomEvent(string message)
    {
        OnCustomEvent?.Invoke(message);
    }
}
```

- **Creating New Door Types**:
  Extend the `Door` class, overriding methods like `Open`, `Close`, `ForceOpen`, and `ForceClose`.

Extending the Door Script with New Behavior

```
public class CustomDoor : Door
{
    public override void Open(Vector3 userPosition)
    {
        base.Open(userPosition);
        Debug.Log("Custom Door Opened with additional behavior.");
    }

    public override void Close()
    {
        base.Close();
        Debug.Log("Custom Door Closed with additional behavior.");
    }
}
```

- **Adding Player Interactions**:
  Modify `PlayerActions.cs` to detect new interactable objects by extending the `LayerMask` and handling new tags.

Modify `PlayerActionsEditor.cs` to add new input actions dynamically:

```
if (GUILayout.Button("Add Custom Input"))
{
    InputAction newAction = new InputAction("CustomAction", InputActionType.Button, "<Keyboard>/c");
    playerActions.GetInteractActionReference().action.AddBinding(newAction.bindings[0]);
    EditorUtility.SetDirty(target);
}
```

- **Editor Customizations**:
  Add new tabs to `DoorEditor.cs` by extending `DrawTabsWithWrap` and creating new `Draw*Settings` methods.

Extend `DoorEditor.cs` to add a new tab for Custom Settings:

```
case 10: // CUSTOM
    DrawCustomSettings(door);
    break;

private void DrawCustomSettings(Door door)
{
    DrawTitle("-- CUSTOM SETTINGS --");
    DrawBackgroundBox(() =>
    {
        door.customSetting = EditorGUILayout.FloatField("Custom Value", door.customSetting);
    });
}
```

6. Best Practices

- **Event-Driven Design**: Use event channels for communication to maintain decoupling.
- **Inspector Validation**: Implement `OnValidate` methods to ensure data integrity.
- **Editor Scripting**: Use `EditorUtility.SetDirty` and `Undo.RecordObject` for saving changes in custom editors.
- **Code Reuse**: Utilize utility methods like `EnsureComponent<T>` for runtime component addition.