Unity Stories is a state container for games built in Unity utilizing Scriptable Objects.

Influences

Unity Stories is inspired and influenced by Redux and Flux.

Forerunner implementations in Unity / C# that also have influenced Unity Stories are: - redux-unity-3d - Unidux

Motivation

The general approach to building scripts in Unity often generates a code base that is monolithic. This results in that your code is cumbersome to test, non-modular and hard to debug and understand.

The aim of Unity Stories is to seperate concerns between your game state and the implementation of your game logic making your scripts modular. The will make protyping your game faster and makes it easier to make changes to your code base even though your project has grown large.

Installation

Import unitypackage from latest releases or download and import into your project from the Unity Asset Store.

Usage

In order to utilize this library you should understand how flux and redux works. See links above.

Story and Story Actions

Create a Stories object (Assets/Create/Unity Stories/Stories) and an Entry Story (Press "Create Entry Story" button on Stories asset or Assets/Create/Unity Stories/Entry Story). If created from the window menu, drag and drop the Entry Story to the Stories object.

Create your Stories (state containers) by inheriting from the abstract Story class and connect them to the Entry Story. Here is an example of a simple story with two int variables, one that is persistied between plays and one that is initalized each time we start the game:

```
[CreateAssetMenu(menuName = "Unity Stories/Example1/Stories/Count Story")]
public class CountStory : Story
{
    // Variables that you want to keep track of in your story.
    public int count = 0;
    public int countNotPresisted = 0;

    // Init your variables here that you don't want to be persisted between plays.
    public override void InitStory()
    {
        countNotPresisted = 0;
    }

    // Actions / factories
    public class IncrementCount : GenericAction<CountStory>
    {
        public override void Action(CountStory story)
        {
            story.count++;
            story.countNotPresisted++;
        }
    }
    public static GenericFactory<IncrementCount, CountStory> IncrementCountFactory
= new GenericFactory<IncrementCount, CountStory>();

    public class DecrementCount : GenericAction<CountStory>
    {
        public override void Action(CountStory story)
        {
            story.count--;
            story.countNotPresisted--;
        }
    }
    public static GenericFactory<DecrementCount, CountStory> DecrementCountFactory
= new GenericFactory<DecrementCount, CountStory>();
}
```

Below is a breakdown of what is going on in our `CountStory`: - First we define the variables that we want to keep track of in this Story. This is what the Story is all about. You can store any data or object that you want to keep track of and change when StoryActions are dispatched. - The InitStory() method is used if you want to initalize variables each play. - Lastly we define our StoryActions and corrsponding factories (these can be defined in a seperate file if that is preferable). StoryActions can be dispatched from your code in order to change the state in our Story. The StoryAction can contain data and is responsible to define how it changes our Story's data (defined in `void Action(...)`). In the above example we define 2 StoryActions (and corrsponding factories) that increments and decrements our variables stored in the Story. The example is using `GenericAction` / `GenericFactory` in order to reduce boilerplate code, but if needed / wanted it is possible to extend `StoryAction` directly.

One major difference from Redux is that we don't define a reducer. Instead we let StoryActions define how we change our Story / state. Another major difference is that a StoryAction actual

mutates our Story / state. This is because Unity Stories tries to minimize the amount of garbage being generated.

Even though it is possible, a Story's state should never be altered directly, always dispatch a StoryAction.

Dispatch Story Actions

When the Story is defined you can now use is it in your code. Here is an example of how you would dispatch a StoryAction (using our defined factories) from a button click:

```
public class Button : MonoBehaviour
{
    public StoriesHelper storiesHelper;

    public void OnClick_Inc()
    {
        storiesHelper.Dispatch(CountStory.IncrementCountFactory.Get());
    }

    public void OnClick_Dec()
    {
        storiesHelper.Dispatch(CountStory.DecrementCountFactory.Get());
    }
}
```

Don't forget to assign the Stories object to the StoriesHelper from the inspector

Use The Data From The Story

You can now use the values in this Story by connecting to your Stories from another script. Here is an example of displaying the values in an UI text element:

```
public class CountText_Example1 : MonoBehaviour
{
    public Text countText;
    public Text countNotPersistedText;
    public StoriesHelper storiesHelper;

    void Start()
    {
        storiesHelper.Setup(gameObject, MapStoriesToProps);
    }

    void SetCountText(int count)
    {
        countText.text = "Count is: " + count;
    }

    void SetCountTextNotPersisted(int count)
    {
```

```
        countNotPersistedText.text = "Not persisted count is: " + count;
    }

    public void MapStoriesToProps(Story story)
    {
        SetCountText(story.Get<CountStory>().count);
        SetCountTextNotPersisted(story.Get<CountStory>().countNotPresisted);
    }
}
```

Don't forget to assign the Stories object to the StoriesHelper from the inspector

Connectors

In order to seperate the story code from where the story data is consumed you can define a connector class. In the example above you would then remove the Stories specific code in `CountText_Example1` (`stories.Connect` and `MapStoriesToProps`) and make the setters (`SetCountText` and `SetCountTextNotPersisted`) public. You would then create a connector class looking like this:

```
using UnityEngine;
using UnityStories;

public class ConnectCountToText_Example1 : MonoBehaviour
{
    public StoriesHelper storiesHelper;
    public CountText_Example1 countText_Example1;

    void Start()
    {
        storiesHelper.Setup(gameObject, MapStoriesToProps);
    }

    void MapStoriesToProps(Story story)
    {
        countText_Example1.SetCountText(story.Get<CountStory>().count);

countText_Example1.SetCountTextNotPersisted(story.Get<CountStory>().countNotPresist
ed);
    }
}
```

Stories Helper

The stories helper class helps your setup your connection to your stories and provides a public `Dispatch` function. Use this instead of writing boilerplate code for `Connect` / `Disconnect` and `Listen` / `RemoveListener` in your MonoBehaviour scripts. Declare the `StoriesHelper` in your MonoBehaviour, drag and drop your stories object from the inspector and call the Setup function from your Start / Awake function.

```
using UnityEngine;
using UnityStories;

public class MyMonoBehaviour : MonoBehaviour
{
    public StoriesHelper storiesHelper;

    void Start()
    {
        storiesHelper.Setup(gameObject, MapStoriesToProps);
    }

    void MapStoriesToProps(Story story)
    {
        // Map your stories
    }
}
```

More Examples

See more examples of how to use Unity Stories in the Examples folder.

Middleware

Unity Stories is allowing users to use and define enhancers (like Redux allows users to enhance their store). Unity Stories ships with one enhancer creator, ApplyMiddleware, that is making it possible to apply middleware to the dispatch method. Logger is a middleware defined in Unity Stories that shows the API and a simple example of how a middleware can be defined.

For performance reasons it is preferable to always define StoreAction factories. However, if you want to define your own middleware there are things to take into consideration when you for example wants to perform an async task in an middleware based on the StoreAction. Internally, Unity Stories is keeping track of StoreActions (when using the StoryActionFactoryHelper). In order to keep the StoreAction alive for an async task the `KeepActionAlive()` must be used. It is also important that `ReleaseActionForReuse()` is called when the StoreAction can be released for reuse.

Performance

First of all, it is strongly recommended to create StoryAction factories in order to minimize garbage. Furthermore, in order to avoid unnecessary garbage collection reference types (for example strings) in StoryActions should be avoided if used often (for example in the Update loop) when possible.

Forum

Unity Forum Thread