Rollbacks for better resource management

Resource management is hard. But there is a more general problem, which includes it - management of side-effects.

I have already used this approach for 4 years. It reduces the dependencies, and makes the code sustainable. It also allows you to specify “undo” instructions next to the logic which has side effects, which makes code modular and more readable.

Problems

Let’s talk about services. It’s an asynchronous logic that lives for a long time and interacts with other services. There’re a number of problems:

- It is unclear what side effects service has, what it affects
- Will it dispose correctly if I change the logic? When it crashes? Did the author consider possible completion cases?

This leads to

- Unreleased resources: bundles, memleaks
- We rely on scene unloading to clean up for sure. It’s long, doesn’t give control, nor guarantees that it will execute in a correct manner
- We keep a lot of dependencies just for cleanup. In the form of class fields or passing them to dependent classes. It complicates code, refactoring and leads to bugs.

Problems with Dispose

The Dispose pattern is designed to solve this problem providing the unified way, but:

- you must explicitly store references (in the class fields) before calling Dispose. These’re dependencies that can be avoided.
- It’s not clear if they were initialized, and in what order they should be disposed (or rolled back)
- IDisposable does not oblige you to call it, you can forget easily. No wonder that Rider even has “code inspections” for this, but they only work for simple cases.
- In case of multiple IDisposables, it is not clear in what order to call them.
  Good practice is to dispose them in the opposite order of acquiring them. That is what the Rollback does internally, storing undo actions in a stack.
- There is quite a lot of extra code in Dispose methods to determine what state we are in.

(page break)

What is Rollback

It is an object that you rely on to restore the system to a certain point by performing “undo” actions. You add the “undo” action to the rollback, soon as you do something that has side effects.

Below is a simplified code to get the idea:

```
static class Rollback
{
   static IDisposable New (out IRollback rollback);
}


interface IRollback
{
  void Add (Action dispose);
  void Remove (Action dispose);
  bool IsDisposed {get;}
  IDisposable Nested (out IRollback nestedRollback);
}
```

Sometimes it’s easier to think of it as an inverse of Dispose pattern, or as an OnDispose event.

It's non-GC-friendly to keep things simple. But lowering allocations complicates the API and makes it more difficult to use.

(page break)

Dispose approach:

```
class Service : IDisposable
{
  AssetBundle bundle;
  ICloudService cloudService;
  Option<PopupWindow> popupInstance;

  Service (GAssetBundle bundle, ICloudService cloudService)
  {
     bundle.Load();
     this.bundle = bundle;

     cloudService.OnLoginResult += this.OnLoginResult;
     this.cloudService = cloudService;
  }

  async Task ShowPopupAsync (PopupWindow prefab)
  {
     this.popupInstance = Instantiate(prefab);
     // async logic
  }

  void Dispose ()
  {
     foreach(var popup in this.popupInstance)
     {
        popup.Destroy();
     }

     this.cloudService.OnLoginResult -= this.OnLoginResult;

     bundle.Unload();
  }
}
```

Rollbacks approach:

```
class Service
{
  IRollback rollback;

  Service (AssetBundle bundle, ICloudService cloudService, IRollback rollback)
  {
     bundle.Load();
     rollback.Add(() => bundle.Unload());

     cloudService.OnLoginResult += this.OnLoginResult;
     rollback.Add(() => cloudService.OnLoginResult -= this.OnLoginResult);

     this.rollback = rollback;
  }

  async Task ShowPopupAsync (PopupWindow prefab)
  {
     PopupWindow popup = Instantiate(prefab);
     this.rollback.Add(() => popup.Destroy());
     // async logic
  }
}
```

- We have reduced the dependencies: `bundle`, `cloudService`, `popopInstance`
- All Dispose code now became “dynamic”. Therefore if `ShowPopupAsync` was not called, it would not be destroyed, since we did not add it to Rollback. No need to check if it was created (like in Dispose approach)

Caller logic:

Dispose approach:

```
async Task ServicesFlowAsync ()
{
  Service service = new Service(bundle, cloudService);

  // async logic

  service.Dispose();
}
```

Rollbacks approach:

```
async Task ServicesFlowAsync ()
{
 IDisposable disposeHandle = Rollback.New(out IRollback servicesRollback);
 Service service = new Service(bundle, cloudService, servicesRollback);

 // async logic

 disposeHandle.Dispose();
}
```

- We don't need to bookkeep service anymore, just store the `disposeHandle`
- We could pass `servicesRollback` as a parameter to every feature that needs to be disposed within the same group
- `serviceDispose.Dispose()` executes the rollback

(page break)

Benefits

- Fewer dependencies, the code has not lost stability, you do not need to bookkeep dependencies just to dispose them.
- There is no need to think over different scenarios. Rollback will not be able to run instructions that you did not add to it.
    - The stability is higher, because even in the case of a non-standard completion (`Exception`), you clean up only those actions that you managed to add.
- After using Rollbacks for 4 years, Dispose feels like a brittle hardcode and a step back.

More benefits

if implemented in the project as a conventions:

- **Restore point**
  The line where Rollback is declared is the “restore point” to which the system will return to on `rollback.Dispose()`
  See the Advanced example below
- **Richer API**
    - Could indicate the lifetime of an object.
    - Could enforce users to specify the “lifetime”, ie `bundle.Load(rollback)`
- **Code readability**
  As integration progresses through all codebase, **control flow** and features “hierarchy” becomes clear, starting from bootstrap code. Which simplifies adaptation for newcomers, or even colleagues who didn’t work on certain parts of the code.
    - The unified way to find out the side effects of a service is by reading the code
    - Object lifetime is easy to track
        - Rollback group could be tracked following arguments up to definition
        - Moment of dispose could be tracked following
    - Goes great with `async/yield` syntax combined with `using` (using implicitly calls `finally {rollbackHandle.Dispose()}`
- **Cascade disposing, nested rollbacks**
  You could add nested rollbacks which could be disposed separately or by disposing parent rollback
  See the Advanced example below
- **Debug tools**
    - could inspect current undisposed rollbacks (ie, collecting the code lines via CallerInfo attributes)
    - assert all rollbacks are disposed on crash or game restart
- Could be used as a **collector for unmanaged resources**
- Restarting the game feature is out of the box
  Disposing the root rollbackHandle will return the game to its initial state, so you could start over clean (without unloading the scenes, disposing pools, etc)

- Restarting the game takes much less time on the device.
- Benefit from [disabling Domain Reload feature of Unity Editor 2019.4+](#) Switching to / from Playmode happens on the orders of magnitude faster than with reload. You could try it on the Advanced example below, and it's also tested on bigger projects.

Advanced example

Using a simple example, it is difficult to show all the benefits that this approach gives in real projects, when the code base grows.

I have not yet had the free time to prepare a separate example. Below is a prototype of the game, which had to be done alone in 2 days without using ready-made code (like a game jam). The emphasis was on the arcade physics of the car (off-road drifting). There is room for improvement in architecture due to the extreme time restrictions.
https://gitlab.com/korchoon/racing-game-test/

Highlights:

1. `Rollback` is called differently - `Scope`, but the idea is the same.
   Implementation can be found here:
   https://gitlab.com/korchoon/racing-game-test/-/blob/master/Assets/Scripts/Utils/Scope.cs

2. `Starter.cs` is a bootstrap, which declares the root `Scope` (Rollback). This is the "restore point", to which the game returns at the moment of a full restart.

3. The main interesting part there is an example of `SubScope` (NestedRollbacks), below is their hierarchy:
   `scope` (root one, with `_disposeScope` as a dispose handle)
   - `levelScope`
     - `gameScope`
     - `pauseScope`

4. Each of them can be completed separately or from the parent Scope (Rollback)

5. Considering that there was almost no time, even here Rollbacks helped to keep the flow in order and there were no problems with cleaning up.

The link shows the options for using the API:
https://gitlab.com/search?utf8=%E2%9C%93&search=scope&group_id=&project_id=20015440&scope=&search_code=true&snippets=false&repository_ref=master&nav_source=navbar