# SwissArmyLib Documentation

***Release 1.0***

**Casper Christiansen**

**Sep 21, 2018**

# Contents

# CHAPTER 1

# Getting started

## 1.1 Adding it to your project

### 1.1.1 Method 1: Binaries (Recommended)

Download the latest binaries from here and extract them somewhere in your Unity project's *Assets* folder.

### 1.1.2 Method 2: Copy source

Download the repository and extract it somewhere.

Then from the extracted folder copy the **Archon.SwissArmyLib** folder into your project **Assets** folder somewhere.

Do the same thing with the **Archon.SwissArmyLib.Editor** folder but place that one in another folder called "Editor".

#### Start using it

That's it! You can freely start using the library. Check out the sidebar for some help with using the different utilities.

## 1.2 Building the source

Just download the project, open the project in Visual Studio (or use MSBuild via CLI) and build it. There's nothing more to it.

# CHAPTER 2

# BetterCoroutines

Coroutines is an awesome hack, but unfortunately they're not that cheap. BetterCoroutines is a very similar but much more performant alternative to Unity's coroutines. It runs faster and reduces the amount of garbage generated. Furthermore you have more control over which update loop they run in and their lifetime is not dependent on gameobjects.

That said, they still aren't super cheap and should be used sparingly (if at all) if you're shooting for high performance. Their performance is similar to that of More Effective Coroutines.

## 2.1 Usage

### 2.1.1 Defining your coroutine

Defining your coroutine method is exactly the same as with Unity's coroutines:

```
IEnumerator MyRoutine()
{
    Debug.Log("Let's wait a frame!");
    yield return null;
    Debug.Log("That was awesome!");
}
```

### 2.1.2 Starting a coroutine

Instead of using *MonoBehaviour.StartCoroutine()* you should use *BetterCoroutines.Start()*.

```
BetterCoroutines.Start(MyRoutine());
```

This'll run *MyRoutine* in the regular update loop.

If you instead want it to run in LateUpdate or FixedUpdate you can do the following:

```
BetterCoroutines.Start(MyRoutine(), UpdateLoop.LateUpdate);

// or

BetterCoroutines.Start(MyRoutine(), UpdateLoop.FixedUpdate);
```

If you want your coroutines to stop automatically when a gameobject or component is destroyed/disabled, you can do so with an overload:

```
BetterCoroutines.Start(MyRoutine(), myGameObject);

// or

BetterCoroutines.Start(MyRoutine(), myComponent);
```

### 2.1.3 Stopping a coroutine

The start method actually returns the id of the started coroutine, which you can use to stop it prematurely if needed.

```
int myCoroutineId = BetterCoroutines.Start(MyRoutine());

// later

bool wasStopped = BetterCoroutines.Stop(myCoroutineId);
```

*wasStopped* will now be true if the coroutine was found and stopped, otherwise it will be false.

### 2.1.4 Stopping all coroutines

If you suddenly have the urge to stop all coroutines you can do so easily:

```
// stops all coroutines regardless of which update loop they're part of
BetterCoroutines.StopAll();

// stops all coroutines in a specific update loop
BetterCoroutines.StopAll(UpdateLoop.FixedUpdate);
```

### 2.1.5 Checking whether a specific coroutine is running

You can check if a coroutine with a specific ID is currently running.

```
bool isRunning = BetterCoroutines.IsRunning(myCoroutineId);
```

### 2.1.6 Pausing a coroutine

BetterCoroutines supports pausing a coroutine as well.

```
BetterCoroutines.Pause(myCoroutineId);
BetterCoroutines.Unpause(myCoroutineId);

// or
```

(continues on next page)

(continued from previous page)

```
BetterCoroutines.SetPaused(myCoroutineId, true);
BetterCoroutines.SetPaused(myCoroutineId, false);
```

You can can check whether a coroutine is currently paused with *IsPaused(id)*.

```
bool isPaused = BetterCoroutines.IsPaused(myCoroutineId);
```

### 2.1.7 Yield Operations

#### Wait a frame

Just as with Unity's coroutines you can wait a single frame by yielding *null*.

If this is not verbose enough for you, you can instead use the constant BetterCoroutines.WaitForOneFrame.

```
IEnumerator WaitAFrameRoutine()
{
    Debug.Log("First frame");
    yield return null;
    Debug.Log("Second frame");
    yield return BetterCoroutines.WaitForOneFrame;
    Debug.Log("Three frame");
}
```

#### Subcoroutines

You can yield a coroutine to suspend the current one until the yielded coroutine is done.

```
IEnumerator MyRoutine()
{
    yield return MySubroutine();

    // or

    yield return BetterCoroutines.Start(MySubroutine());
}

IEnumerator MySubroutine()
{
    yield return null;
}
```

#### WaitForSeconds

You can suspend a coroutine for a certain time either in scaled or unscaled time.

```
IEnumerator WaitForSecondsRoutine()
{
    // waits 1 second in scaled time
    yield return BetterCoroutines.WaitForSeconds(1);
```

(continues on next page)

(continued from previous page)

```
    // waits 1 second in unscaled time
    yield return BetterCoroutines.WaitForSeconds(1, false);
}
```

### WaitForSecondsRealtime

You can also suspend a coroutine for a specific time in realtime.

```
IEnumerator WaitForSecondsRealtimeRoutine()
{
    yield return BetterCoroutines.WaitForSecondsRealtime(1);
}
```

### WaitWhile

You can also suspend a coroutine until a delegate returns false. The delegate is invoked each frame.

```
IEnumerator WaitWhileRoutine()
{
    // waits until space is released (assuming it's pressed right now)
    yield return BetterCoroutines.WaitWhile(() => Input.GetKey(KeyCode.Space));
}
```

### WaitForEndOfFrame

BetterCoroutines also has support for Unity's special WaitForEndOfFrame instruction.

It'll suspend the coroutine until right before the current frame is displayed.

While you could create a new instance every time, there's no reason to punish the GC. Just use BetterCoroutines.WaitForEndOfFrame instead.

```
IEnumerator EndOfFrameRoutine()
{
    yield return BetterCoroutines.WaitForEndOfFrame;

    // take screenshot or whatever
}
```

### WaitUntil

Similar to *WaitWhile* but instead it waits until the delegate returns true. The delegate is invoked each frame.

```
IEnumerator WaitUntilRoutine()
{
    // waits until Space is pressed.
    yield return BetterCoroutines.WaitUntil(() => Input.GetKeyDown(KeyCode.Space));
}
```

### WaitForWWW

Yielding a WWW object will suspend the coroutine until the download is done.

```
IEnumerator WaitForWWWRoutine()
{
    WWWW www = new WWW("https://gitlab.com/archoninteractive/SwissArmyLib/raw/master/
↪logo.png");

    yield return www;
    // or
    yield return BetterCoroutines.WaitForWWW(www);

    // www.texture is now downloaded
}
```

### WaitForAsyncOperation

Yielding an AsyncOperation will suspend the coroutine until that AsyncOperation is done.

```
IEnumerator WaitForAsyncOperationRoutine()
{
    AsyncOperation operation = SceneManager.LoadSceneAsync(0);

    yield return operation;
    // or
    yield return BetterCoroutines.WaitForAsyncOperation(operation);

    // scene is now loaded
}
```

**Todo:** Write some examples for BetterCoroutines

CHAPTER 3

# Events

## 3.1 Event

A simple event handler that supports both interface and delegate listeners. Listeners can have a specific priority that affects whether they're called before or after other listeners. If multiple listeners have the same priority, they will be called in the order they were added.

All listeners will be notified even if one of the invocations throws an exception.

If you need to pass along some data to the listeners, you can use the alternative **Event<T>** version.

Events are differentiated by an integer. You are expected to define them as constants and not just keeping track of the ids in your head.

If you need global events, take a took at *GlobalEvents*.

### 3.1.1 Usage

#### Creating

Define your event's id somewhere, eg. in a static EventIds class.

```
public static class EventIds
{
    public const int MyEvent = 0;
}
```

Now create the event handler.

```
public class SomeClassWithAnEvent
{
    public readonly Event MyEventWithoutArgs = new Event(EventIds.MyEvent);

    // or if you need args:
```

(continues on next page)

(continued from previous page)

```
    public readonly Event<int> MyEventWithArgs = new Event<int>(EventIds.MyEvent);
}
```

That's it, the event handlers are ready to be used.

### Invoking

Invoking an event is as simple as calling:

```
MyEventWithoutArgs.Invoke();

// or with args:
MyEventWithArgs.Invoke(10);
```

### Subscribing

#### Interface listener

First of all you have to implement the **IEventListener** (no args) or **IEventListener<T>** (with args) interface, which has an *OnEvent()* method that will be called, as the name suggests, when an event is invoked.

```
public class SomeListenerWithoutArgs : IEventListener
{
    public void OnEvent(int eventId)
    {

    }
}

public class SomeListenerWithArgs : IEventListener<int>
{
    public void OnEvent(int eventId, int args)
    {

    }
}
```

Then add yourself as a listener for the event:

```
MyEvent.AddListener(myListener);
```

And that's it! Your *OnEvent()* method will be called when the event is invoked.

#### Delegate listener

Using a delegate listener is simpler and works for static classes as well, but also less performant since they often allocate short-living memory.

First define the methods you want called:

```
void OnMyEventWithoutArgs()
{

}

void OnMyEventWithArgs(int args)
{

}
```

Then add them as listeners to the events:

```
MyEventWithoutArgs.AddListener(OnMyEventWithoutArgs);
MyEventWithArgs.AddListener(OnMyEventWithArgs);
```

### Prioritization

You can optionally pass in a priority if you want your listener to be notified before or after other listeners.

```
MyEvent.AddListener(myListener);
MyEvent.AddListener(myOtherListener, -1000);
```

*myOtherListener* will now get called before *myListener*, despite being added later.

### Unsubscribing

Removing yourself as a listener is even easier, simply call:

```
MyEvent.RemoveListener(myListener);
```

### Clearing

In case you ever need to completely clear all listeners for an event:

```
MyEvent.Clear();
```

**Todo:** Examples for:

- Events without args
- Events with args
- Events with prioritized listeners

## 3.2 GlobalEvents

A manager of events that do not belong to any specific object but instead can be listened to by anyone and invoked by anyone.

Useful for GameLoaded, MatchEnded and similar events.

This uses *Event* instances behind the scene.

If you need to pass along some data to the listeners, you can use the alternative **GlobalEvents<T>** version.

Events are differentiated by an integer. You are expected to create constants to define your events.

If you just need regular local events, take a took at *Event*.

## 3.2.1 Usage

### Invoking

Invoking an event is as simple as calling:

```
GlobalEvents.Invoke(eventId);
```

### Subscribing

#### Interface listener

First of all you have to implement the **IEventListener** interface, which has an *OnEvent(int eventId)* method that will be called, as the name suggests, when an event is invoked.

Then add yourself as a listener for an event:

```
GlobalEvents.AddListener(eventId, listener);
```

You can optionally pass in a priority if you want your listener to be notified before or after other listeners.

And that's it! Your *OnEvent(id)* method will be called when the event is invoked.

#### Delegate listener

Using a delegate listener is simpler and works for static classes as well, but also less performant since they often allocate short-living memory.

```
void MyMethod()
{
}

GlobalEvents.AddListener(eventId, MyMethod);
```

### Unsubscribing

Removing yourself as a listener is even easier, simply call:

```
GlobalEvents.RemoveListener(eventId, listener);
```

Or if you want your listener to be unsubscribed from all events you can call:

```
GlobalEvents.RemoveListener(listener);
```

### Clearing

In case you ever need to completely clear all listeners for an event:

```
GlobalEvents.Clear(eventId);
```

Or maybe you want a completely clean slate:

```
GlobalEvents.Clear();
```

## 3.2.2 Examples

### Global events without args

In this example we want the UI to be notified when the player dies or is revived, so we can show or hide the death screen accordingly.

```
public static class EventIds
{
    public const int PlayerDied = 0,
                     PlayerRevived = 1;
}

public class Player : MonoBehaviour
{
    public void Kill()
    {
        // play death animation

        GlobalEvents.Invoke(EventIds.PlayerDied);
    }

    public void Revive()
    {
        // praise the lord, we were revived!

        GlobalEvents.Invoke(EventIds.PlayerRevived);
    }
}

public class UIManager : MonoBehaviour, IEventListener
{
    private void OnEnable()
    {
        GlobalEvents.AddListener(EventIds.PlayerDied, this);
        GlobalEvents.AddListener(EventIds.PlayerRevived, this);
    }

    private void OnDisable()
    {
        GlobalEvents.RemoveListener(EventIds.PlayerDied, this);
        GlobalEvents.RemoveListener(EventIds.PlayerRevived, this);
    }

    public void OnEvent(int eventId)
    {
```

(continues on next page)

(continued from previous page)

```
        switch (eventId)
        {
            case PlayerRevived:
                HideDeathScreen();
                break;
            case PlayerDied:
                ShowDeathScreen();
                break;
        }
    }
}
```

### Global event with args

This time we have a multiplayer game and need to keep track of player stats: Kills and deaths. Instead of having the player class find and call the scoreboard itself, we make the scoreboard rely on a PlayerKilled event instead. When a player is killed it invokes the event and sends along information about the event (who the victim is, and who the killer is). This way other systems that might need to know when players are killed, can be notified too without even having to change the Player class again.

```
public class PlayerKilledArgs : EventArgs
{
    public Player Victim;
    public Player Killer;
}

public static class EventIds
{
    public const int PlayerKilled = 0;
}

public class Player : MonoBehaviour
{
    // we store it in a field, so we can reuse the same instance and avoid generating␣
↪garbage
    private PlayerKilledArgs _playerKilledArgs = new PlayerKilledArgs();

    public void Kill(Player killer)
    {
        // play death animation

        // update event args
        _playerKilledArgs.Victim = this;
        _playerKilledArgs.Killer = killer;

        GlobalEvents<EventArgs>.Invoke(EventIds.PlayerKilled, _playerKilledArgs);
    }
}

public class Scoreboard : MonoBehaviour, IEventListener<EventArgs>
{
    private void OnEnable()
    {
        GlobalEvents<EventArgs>.AddListener(EventIds.PlayerKilled, this);
    }
```

(continues on next page)

(continued from previous page)

```
    private void OnDisable()
    {
        GlobalEvents<EventArgs>.RemoveListener(EventIds.PlayerKilled, this);
    }

    public void OnEvent(int eventId, EventArgs args)
    {
        if (eventId == EventIds.PlayerKilled)
        {
            var playerKilledArgs = (PlayerKilledArgs) args;

            AddKill(PlayerKilledArgs.Killer);
            AddDeath(PlayerKilledArgs.Victim);
        }
    }
}
```

**Todo:** Examples for:

- Global events with prioritized listeners

## 3.3 ManagedUpdate

A relay for Unity update events. Also useful for non-**MonoBehaviours** that needs to be part of the update loop as well.

Check out *ManagedUpdateBehaviour* for a simple **MonoBehaviour** subclass that makes use of this instead of regular events.

Events that you can listen to:

- ManagedUpdate.OnUpdate
- ManagedUpdate.OnLateUpdate
- ManagedUpdate.OnFixedUpdate

You can also create custom update loops that for example runs once every second.

### 3.3.1 Why?

Here's why you might want to use this. In short; avoid overhead of Native C++ –> Managed C# calls.

### 3.3.2 Usage

*Subscribing* to one of the relayed events:

- ManagedUpdate.OnUpdate
- ManagedUpdate.OnLateUpdate
- ManagedUpdate.OnFixedUpdate

You can also use the method *AddListener(eventId, listener)* in ManagedUpdate but it's only required if you use custom update loops.

### Custom Update Loops

In addition to the regular Unity update loops, ManagedUpdate also supports custom ones that you can control when runs. You can create such a loop by creating an instance of a class that implements the **ICustomUpdateLoop** interface (though it's simpler if you just subclass **CustomUpdateLoopBase**). There's two simple implementations already available if you need them: **TimeIntervalUpdateLoop** and **FrameIntervalUpdateLoop**.

```
var updateLoop = new TimeIntervalUpdateLoop(eventId: 1000, interval: 1);
```

Then add the instance to ManagedUpdate by calling *AddCustomUpdateLoop()*.

```
ManagedUpdate.AddCustomUpdateLoop(updateLoop);
```

Subscribing to a custom update loop requires using the *AddListener(eventId, listener)* in ManagedUpdate (or you could use the updateloop instance directly).

```
Action listener = () => Debug.Log("Booyah!");
ManagedUpdate.AddListener(eventId: 1000, listener);
```

If you want to know the time difference since these updates were last invoked, you can call **ManagedUpdate.DeltaTime** or **ManagedUpdate.UnscaledDeltaTime**.

```
Action listener = () =>
{
    Debug.LogFormat("It has been {0} seconds since the last update!", ManagedUpdate.
↪DeltaTime);
};
ManagedUpdate.AddListener(eventId: 1000, listener);
```

## 3.3.3 Examples

### MonoBehaviour

Check out *ManagedUpdateBehaviour* if you need something like this.

```
public class SimpleManagedUpdateBehaviour : MonoBehaviour, IEventListener
{
    protected virtual void OnEnable()
    {
        ManagedUpdate.OnUpdate.AddListener(this);
    }

    protected virtual void OnDisable()
    {
        ManagedUpdate.OnUpdate.RemoveListener(this);
    }

    protected virtual void OnEvent(int eventId)
    {
        if (eventId == ManagedUpdate.EventIds.Update)
        {
```

(continues on next page)

(continued from previous page)

```
            // regular ol' update logic
        }
    }
}
```

### Non-MonoBehaviour

There's not much to it, you just need to subscribe *somewhere*. In this example we just control whether the class receives updates through its *Enabled* property.

```
public class ManagedUpdateListener : IEventListener
{
    private bool _enabled = false;
    public bool Enabled
    {
        get { return _enabled; }
        set
        {
            if (_enabled == value)
                return;

            _enabled = value;

            if (_enabled)
                ManagedUpdate.OnUpdate.AddListener(this);
            else
                ManagedUpdate.OnUpdate.RemoveListener(this);
        }
    }

    void OnEvent(int eventId)
    {
        if (eventId == ManagedUpdate.EventIds.Update)
        {
            // regular ol' update logic
        }
    }
}
```

## 3.4 ManagedUpdateBehaviour

An abstract subclass of **MonoBehaviour** that uses *ManagedUpdate* for update events.

### 3.4.1 Usage

Just inherit your **MonoBehaviours** from **ManagedUpdateBehaviour** instead and implement one (or more) of the interfaces: **IUpdateable**, **ILateUpdateable** and **IFixedUpdateable**, **ICustomUpdateable**.

Instead of Unity's magic *Update()*, *LateUpdate()* and *FixedUpdate()* methods you should use the corresponding methods from the interfaces.

If you override *Start()*, *OnEnable()* or *OnDisable()* make sure you remember to call the base method, to save yourself a debugging headache.

#### Execution Order

While **ManagedUpdateBehaviour** doesn't respect Unity's ScriptExecutionOrder, it does however have its own counterpart. Do however note that this is completely separate from ScriptExecutionOrder and it only affects the order that **ManagedUpdateBehaviours** will have their methods called.

To use it just override the *ExecutionOrder* property like so:

```
// Will now be called before other behaviours with a higher ExecutionOrder
protected override int ExecutionOrder { get { return -1000; } }
```

### 3.4.2 Examples

#### Simple2DMovement

```
public class Simple2DMovement : ManagedUpdateBehaviour, IUpdateable
{
    public float Speed = 5;

    public void OnUpdate()
    {
        var dir = new Vector3(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical"),␣
↪0);
        transform.position += dir * Speed * Time.deltaTime;
    }
}
```

#### Custom Update Loop

This example shows you how you can easily use a custom update loop with ManagedUpdateBehaviour. The example assumes that the custom update loops are already created and added to ManagedUpdate.

```
public static class EventIds
{
    public const int CustomUpdate1 = 1000,
                     CustomUpdate2 = 1001;
}

public class Simple2DMovement : ManagedUpdateBehaviour, ICustomUpdateable
{
    public void GetCustomUpdateIds()
    {
        // we return an array of the custom update loops we want to receive events␣
↪for.
        return new[]{ EventIds.CustomUpdate1, EventIds.CustomUpdate2 };
    }

    public void OnCustomUpdate(int eventId)
    {
        switch (eventId)
        {
            case EventIds.CustomUpdate1:
                Debug.Log("CustomUpdate1 ran!");
                Debug.Log("Seconds since it last ran: " + ManagedUpdate.DeltaTime);
```

(continues on next page)

(continued from previous page)

```
                return;

            case EventIds.CustomUpdate2:
                Debug.Log("CustomUpdate2 ran!");
                Debug.Log("Seconds since it last ran: " + ManagedUpdate.DeltaTime);
                return;
        }
    }
}
```

## 3.5 TellMeWhen

**TellMeWhen** is a simple utility for whenever you need something to be called after a certain time. Instead of relying on Unity's (relatively expensive) *Invoke()*, a coroutine with a *WaitForSeconds()* or simply checking each update, you can just ask **TellMeWhen** to notify when the time has passed. Optionally you can pass some arguments along with the notification or make it happen repeatedly every Nth second.

### 3.5.1 Usage

#### Scheduling

First step is implementing the **ITimerCallback** interface in whatever class you want to be able to receive the timer callbacks.

Then all you have to do is call one of the static **TellMeWhen** scheduling methods:

```
TellMeWhen.Exact(time, listener, id, args);
TellMeWhen.Seconds(seconds, listener, id, args);
TellMeWhen.Minutes(minutes, listener, id, args);

TellMeWhen.ExactUnscaled(time, listener, id, args);
TellMeWhen.SecondsUnscaled(seconds, listener, id, args);
TellMeWhen.MinutesUnscaled(minutes, listener, id, args);
```

The 'normally' named methods use Unity's ol' **Time.time**, which is scaled according to **Time.timeScale**. If you use that for pausing the game, but don't want your timers to be delayed accordingly, you might want to use the *Unscaled* methods. These will as the name says use **Time.unscaledTime** instead, which completely ignore the time scale.

#### Multiple timers

If you have multiple different timers for the same listener you might want to differentiate by an id. The scheduling methods take an optional id that will be passed to the listener when the timer is triggered.

#### Arguments

You can optionally pass an object along to the listener when scheduling.

#### Canceling

You can cancel a specific timer by calling *Cancel(listener, eventId)* or just clear everything for a listener by calling *Cancel(listener)*.

### 3.5.2 Examples

#### Single timer, no args

```
public class Bomb : MonoBehaviour, ITimerCallback
{
        public float CountdownTime = 5;

        private void Awake()
        {
                TellMeWhen.Seconds(CountdownTime, this);
        }

        public void OnTimesUp(int id, object args)
        {
                // boom!
        }
}
```

#### Single repeating timer

```
public class DamageOverTime : MonoBehaviour, ITimerCallback
{
        public float DamagePerSecond = 1;

        private void OnEnable()
        {
                TellMeWhen.Seconds(1, this, repeating:true);
        }

        private void OnDisable()
        {
                // clears all timers for this listener (but of course we only have␣
↪one anyway)
                TellMeWhen.Cancel(this);
        }

        // called every second
        public void OnTimesUp(int id, object args)
        {
                ImaginaryHealth.Damage(DamagePerSecond);
        }
}
```

#### Single timer with args

```
public class GameObjectPool : ITimerCallback
{
        public void Despawn(GameObject gameObject)
        {
                // disable gameObject and back to pool
        }

        public void DelayedDespawn(GameObject gameObject, float delay)
        {
                TellMeWhen.Seconds(delay, this, args:gameObject);
        }

        public void OnTimesUp(int id, object args)
        {
                var target = args as GameObject;
                Despawn(target);
        }

        // ... yadda yadda rest of pool
}
```

### Multiple timers

```
public class Bomb : MonoBehaviour, ITimerCallback
{
        public float CountdownTime = 5;

        private static class Timers
        {
                public const int Tick = 0,
                                 Explode = 1;
        }

        private void Awake()
        {
                TellMeWhen.Seconds(1, this, Timers.Tick, repeating: true);
                TellMeWhen.Seconds(CountdownTime, this, Timers.Explode);
        }

        public void OnTimesUp(int id, object args)
        {
                switch (id)
                {
                        case Timers.Tick:
                                // play ticking animation
                                break;
                        case Timers.Explode:
                                // since Timers.Tick is repeating, we have to cancel␣
↪it
                                // so it doesn't continue after the big boom boom
                                TellMeWhen.Cancel(Timers.Tick, this);
                                // boom!
                                break;
                }
        }
}
```

| Class | Purpose |
|---|---|
| *Event* | Simple event handler that supports both interface and delegate listeners. |
| *GlobalEvents* | Manager of events that do not belong to any specific object and can be listened to by anyone and invoked by anyone. |
| *TellMeWhen* | Simple utility for whenever you need something to be called after a certain time. |
| *ManagedUpdate* | Relay for Unity update events. |
| *ManagedUpdateBehaviour* | Abstract subclass of **MonoBehaviour** that uses *ManagedUpdate* for update events. |

CHAPTER 4

# Object Pooling

## 4.1 PoolHelper

A global pool manager for GameObjects.

If you need it for regular objects take a look at the generic version *PoolHelper<T>*.

### 4.1.1 Usage

#### Spawning

```
SomeMonobehaviour somePrefabInstance = PoolHelper.Spawn<SomeMonobehaviour>(myPrefab);
```

#### Despawning

```
PoolHelper.Despawn(somePrefabInstance);
```

You can also despawn objects after a delay:

```
// despawns after 1 sec in scaled time
PoolHelper.Despawn(somePrefabInstance, 1f);

// unscaled time
PoolHelper.Despawn(somePrefabInstance, 1f, true);
```

### 4.1.2 Examples

#### Simple bullet pool

```
public class Bullet : MonoBehaviour, IPoolable
{
    public Vector3 Position;
    public Vector3 Velocity;
    public event Action<Bullet, Collider> Impact;

    void Update()
    {
        Position += Velocity * Time.deltaTime;
    }

    void OnTriggerEnter(Collider otherCollider)
    {
        Impact(this, otherCollider);
    }

    void IPoolable.OnSpawned()
    {

    }

    void IPoolable.OnDespawned()
    {
        // reset state
        Position = Vector3.zero;
        Velocity = Vector3.zero;
        Impact = null;
    }
}

public class Gun : MonoBehaviour
{
    public float BulletSpeed = 5;
    public float BulletLifeTime = 10;
    public Bullet BulletPrefab;

    void Fire()
    {
        var bullet = PoolHelper.Spawn(BulletPrefab);
        bullet.Position = transform.position;
        bullet.Velocity = transform.forward * BulletSpeed;
        bullet.Impact += OnImpact;

        // despawn the bullet after a delay if it hasn't already been despawned
        PoolHelper.Despawn(bullet, BulletLifeTime);
    }

    void OnImpact(Bullet bullet, Collider otherCollider)
    {
        // inflict damage or something

        PoolHelper.Despawn(bullet);
    }
```

(continues on next page)

(continued from previous page)

```
}
```

## 4.2 PoolHelper<T>

A global pool manager for regular objects with empty constructors.

If you need it for GameObjects take a look at the non-generic *PoolHelper*.

### 4.2.1 Usage

#### Spawning

```
SomeClass someObject = PoolHelper<SomeClass>.Spawn();
```

#### Despawning

```
PoolHelper<SomeClass>.Despawn(someObject);
```

You can also despawn objects after a delay:

```
// despawns after 1 sec in scaled time
PoolHelper<SomeClass>.Despawn(someObject, 1f);

// unscaled time
PoolHelper<SomeClass>.Despawn(someObject, 1f, true);
```

### 4.2.2 Examples

#### Simple bullet pool

```
public class Bullet
{
    public Vector3 Position;
    public Vector3 Velocity;
    public event Action<Bullet, object> Impact;

    void Update()
    {
        Position += Velocity * Time.deltaTime;

        if (weHitSomething)
            Impact(this, whatWeHit);
    }
}

public class Gun
{
    Vector3 muzzlePosition;
```

(continues on next page)

(continued from previous page)

```
    Vector3 direction;
    float bulletSpeed = 5;
    float bulletLifeTime = 10;

    void Fire()
    {
        var bullet = PoolHelper<Bullet>.Spawn();
        bullet.Position = muzzlePosition;
        bullet.Velocity = direction * bulletSpeed;
        bullet.Impact += OnImpact;

        // despawn the bullet after a delay if it hasn't already been despawned
        PoolHelper<Bullet>.Despawn(bullet, bulletLifeTime);
    }

    void OnImpact(Bullet bullet, object otherObject)
    {
        bullet.Impact -= OnImpact;
        PoolHelper<Bullet>.Despawn(bullet);
    }
}
```

## 4.3 Pool<T>

A simple object pool for any type of objects that supports custom factory methods, event callbacks and timed despawns.

If the objects implement the *IPoolable* interface they will be notified when they're spawned and despawned.

### 4.3.1 Usage

#### Creating the pool

```
Pool<SomeClass> myPool = new Pool<SomeClass>(() => new SomeClass());
```

Since we use a delegate for creating the instances, you are not limited to just using the empty constructor:

```
Pool<SomeClass> myPool = new Pool<SomeClass>(() =>
{
    var instance = new SomeClass();
    instance.SomeFloat = 3f;
    instance.SomeMethod();
    return instance;
});
```

#### Spawning

```
SomeClass someObject = myPool.Spawn();
```

### Despawning

```
myPool.Despawn(someObject);
```

You can also despawn objects after a delay:

```
// despawns after 1 sec in scaled time
myPool.Despawn(someObject, 1f);

// unscaled time
myPool.Despawn(someObject, 1f, true);
```

## 4.3.2 Examples

### Simple bullet pool

```
public class Bullet
{
    public Vector3 Position;
    public Vector3 Velocity;
    public event Action<Bullet, object> Impact;

    void Update()
    {
        Position += Velocity * Time.deltaTime;

        if (weHitSomething)
            Impact(this, whatWeHit);
    }
}

public class Gun
{
    Vector3 muzzlePosition;
    Vector3 direction;
    float bulletSpeed = 5;
    float bulletLifeTime = 10;

    Pool<Bullet> _bulletPool = new Pool<Bullet>(() => new Bullet());

    void Fire()
    {
        var bullet = _bulletPool.Spawn();
        bullet.Position = muzzlePosition;
        bullet.Velocity = direction * bulletSpeed;
        bullet.Impact += OnImpact;

        // despawn the bullet after a delay if it hasn't already been despawned
        _bulletPool.Despawn(bullet, bulletLifeTime);
    }

    void OnImpact(Bullet bullet, object otherObject)
    {
        bullet.Impact -= OnImpact;
        _bulletPool.Despawn(bullet);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

## 4.4 GameObjectPool

A subclass of *Pool<T>* that supports GameObjects.

If the objects implement the *IPoolable* interface they will be notified when they're spawned and despawned.

If you need multiple *IPoolable* components in an instance to be notified take a look at the *PoolableGroup* component.

### 4.4.1 Multiscene support

When a pool is created you can set multiscene support to either off or on.

If it's off the pool will reside in the current active scene. If the scene is unloaded the pool itself along with any despawned instances will be destroyed along with the scene.

If it's on, it will instead be marked as *DontDestroyOnLoad* and despawned instances will survive any scene unloading.

### 4.4.2 Usage

#### Creating the pool

```
var myPool = new GameObjectPool<SomeMonobehaviour>(somePrefab, multiScene:true);
```

We can also control the instantiating ourselves by using the overload with a factory method:

```
var myPool = new GameObjectPool<SomeMonobehaviour>("MyPoolName", multiScene:true, ()␣
↪=>
{
    var instance = Object.Instantiate(somePrefab);
    instance.SomeFloat = 3f;
    instance.SomeMethod();
    return instance;
});
```

#### Spawning

```
SomeMonobehaviour somePrefabInstance = myPool.Spawn();

// or

var position = new Vector3(0, 0, 5);
var rotation = Quaternion.Euler(0, 0, 90);
var parent = someTransform;
SomeMonobehaviour somePrefabInstance = myPool.Spawn(position, rotation, parent);
```

### Despawning

```
myPool.Despawn(somePrefabInstance);
```

You can also despawn objects after a delay:

```
// despawns after 1 sec in scaled time
myPool.Despawn(somePrefabInstance, 1f);

// unscaled time
myPool.Despawn(somePrefabInstance, 1f, true);
```

### Destroying the pool

You can destroy the pool and any despawned items it contains by calling *Dispose()*.

```
myPool.Dispose();
myPool = null;
```

## 4.4.3 Examples

### Simple bullet pool

```
public class Bullet : MonoBehaviour
{
    public Vector3 Position;
    public Vector3 Velocity;
    public event Action<Bullet, Collider> Impact;

    void Update()
    {
        Position += Velocity * Time.deltaTime;
    }

    void OnTriggerEnter(Collider otherCollider)
    {
        Impact(this, otherCollider);
    }
}

public class Gun : MonoBehaviour
{
    public float BulletSpeed = 5;
    public float BulletLifeTime = 10;
    public Bullet BulletPrefab;

    private GameObjectPool<Bullet> _bulletPool;

    void Awake()
    {
        _bulletPool = new GameObjectPool<Bullet>(BulletPrefab, multiScene:false);
    }

```

(continues on next page)

(continued from previous page)

```
    void OnDestroy()
    {
        _bulletPool.Dispose();
        _bulletPool = null;
    }

    void Fire()
    {
        var bullet = _bulletPool.Spawn();
        bullet.Position = transform.position;
        bullet.Velocity = transform.forward * BulletSpeed;
        bullet.Impact += OnImpact;

        // despawn the bullet after a delay if it hasn't already been despawned
        _bulletPool.Despawn(bullet, BulletLifeTime);
    }

    void OnImpact(Bullet bullet, Collider otherCollider)
    {
        bullet.Impact -= OnImpact;
        _bulletPool.Despawn(bullet);
    }
}
```

## 4.5 IPoolable

An interface that poolable classes can implement to get notified when they're spawned or despawned.

Need multiple MonoBehaviours to get notified when their prefab instance is spawned or despawned then look at *PoolableGroup*.

### 4.5.1 Usage

Just implement the interface in the pooled class.

```
public class MyPooledClass : IPoolable
{
    public void OnSpawned()
    {
        Debug.Log("I'm alive!");
    }

    public void OnDespawned()
    {
        Debug.Log("Back into the pool I go :(");
    }
}

// somewhere else
var myInstance = PoolHelper<MyPooledClass>.Spawn();
Debug.Log("I just spawned myInstance");
```

(continues on next page)

(continued from previous page)

```
PoolHelper<MyPooledClass>.Despawn(myInstance);
Debug.Log("I just despawned myInstance");
```

Will output:

```
I'm alive!
I just spawned myInstance
Back into the pool I go :(
I just despawned myInstance
```

## 4.6 PoolableGroup

A helpful component in case you want multiple *IPoolable* components to receive callbacks.

It manages a list of *IPoolable* components in it's children and serializes the references to avoid having to do so at runtime for each instantiated object.

### 4.6.1 Usage

Just add it to the root the prefab and use it as the prefab reference for the pool.

```
var instance = PoolHelper.Spawn<PoolableGroup>(myPrefab);

PoolHelper.Despawn<PoolableGroup>(instance);
```

A simple object pooling solution for both regular objects as well as GameObjects.

| Class | Purpose |
|---|---|
| *PoolHelper* | A convenient global pool manager for GameObjects. |
| *PoolHelper<T>* | A convenient global pool manager for regular objects with empty constructors. |
| *Pool<T>* | A simple object pool for any type of objects that supports custom factory methods, event callbacks and timed despawns. |
| *GameObject-Pool<T>* | A subclass of Pool<T> that supports GameObjects. |
| *IPoolable* | An interface that poolable classes can implement to get notified when they're spawned or despawned. |
| *PoolableGroup* | A helpful component in case you want multiple IPoolable components to receive callbacks. |

# CHAPTER 5

# Automata

## 5.1 FiniteStateMachine<T>

A simple Finite State Machine with states as objects inspired by Prime31's excellent StateKit.

If your state classes have an empty constructor, the state machine can create the states automatically when needed (using *ChangeStateAuto<T>()*). If not you should create the state instance yourself and register the state in the machine.

Whether or not a null state is valid is up to your design.

**Todo:** Write how to use FiniteStateMachine<T>

### 5.1.1 Examples

#### Simple machine without automatic state creation

Let's create some oversimplified AI for an enemy that should chase the player down and attack him once he's within range.

```
public class IdleState : FsmState<Enemy>
{
    public override void Begin()
    {
        base.Begin();

        // start idle animation
    }

    public override void Reason()
    {
```

(continues on next page)

(continued from previous page)

```
        base.Reason();

        if (Context.Target != null)
        {
            // we've gained a target, let's chase him down!
            Machine.ChangeState<ChaseState>();
        }
    }
}

public class ChaseState : FsmState<Enemy>
{
    public override void Begin()
    {
        base.Begin();

        // start running animation
    }

    public override void Reason()
    {
        base.Reason();

        if (Context.Target == null)
        {
            // we lost our target, return to idle state
            Machine.ChangeState<IdleState>();
        }
        else if (Context.GetDistanceToTarget() < Context.AttackRange)
        {
            // we're within range to attack!
            Machine.ChangeState<AttackState>();
        }
    }

    public override void Act(float deltaTime)
    {
        base.Act(deltaTime);

        // move towards the target player
        Context.Move(dirToTarget);
    }
}

public class AttackState : FsmState<Enemy>
{
    public override void Begin()
    {
        base.Begin();

        // start attack animation
    }

    public override void Reason()
    {
        base.Reason();
```

(continues on next page)

(continued from previous page)

```
        if (Context.Target == null)
        {
            // target is lost, return to normal
            Machine.ChangeState<IdleState>();
        }
        else if (Context.GetDistanceToTarget() > Context.AttackRange)
        {
            // the player is getting away! return to chasing him
            Machine.ChangeState<ChaseState>();
        }
    }

    public override void Act(float deltaTime)
    {
        base.Act(deltaTime);

        // damage target
    }
}

public class Enemy : MonoBehaviour
{
    public Player Target { get; set; }

    FiniteStateMachine<Enemy> _stateMachine;

    void Awake()
    {
        _stateMachine = new FiniteStateMachine<Enemy>(this, new IdleState());
        _stateMachine.RegisterState(new ChaseState());
        _stateMachine.RegisterState(new AttackState());
    }

    void Update()
    {
        // look for the player and set the Target property

        // update the active state
        _stateMachine.Update(Time.deltaTime);
    }

    float GetDistanceToTarget()
    {
        // return distance
    }
}
```

### Simple machine with automatic state creation

Same code as before, but with two small differences:

- We don't register the states after creating the machine
- We use *ChangeStateAuto<T>()*

```
public class IdleState : FsmState<Enemy>
{
    public override void Begin()
    {
        base.Begin();

        // start idle animation
    }

    public override void Reason()
    {
        base.Reason();

        if (Context.Target != null)
        {
            // we've gained a target, let's chase him down!
            Machine.ChangeStateAuto<ChaseState>();
        }
    }
}

public class ChaseState : FsmState<Enemy>
{
    public override void Begin()
    {
        base.Begin();

        // start running animation
    }

    public override void Reason()
    {
        base.Reason();

        if (Context.Target == null)
        {
            // we lost our target, return to idle state
            Machine.ChangeStateAuto<IdleState>();
        }
        else if (Context.GetDistanceToTarget() < Context.AttackRange)
        {
            // we're within range to attack!
            Machine.ChangeStateAuto<AttackState>();
        }
    }

    public override void Act(float deltaTime)
    {
        base.Act(deltaTime);

        // move towards the target player
        Context.Move(dirToTarget);
    }
}

public class AttackState : FsmState<Enemy>
{
```

(continues on next page)

(continued from previous page)

```
    public override void Begin()
    {
        base.Begin();

        // start attack animation
    }

    public override void Reason()
    {
        base.Reason();

        if (Context.Target == null)
        {
            // target is lost, return to normal
            Machine.ChangeStateAuto<IdleState>();
        }
        else if (Context.GetDistanceToTarget() > Context.AttackRange)
        {
            // the player is getting away! return to chasing him
            Machine.ChangeStateAuto<ChaseState>();
        }
    }

    public override void Act(float deltaTime)
    {
        base.Act(deltaTime);

        // damage target
    }
}

public class Enemy : MonoBehaviour
{
    public Player Target { get; set; }

    FiniteStateMachine<Enemy> _stateMachine;

    void Awake()
    {
        _stateMachine = new FiniteStateMachine<Enemy>(this, new IdleState());
    }

    void Update()
    {
        // look for the player and set the Target property

        // update the active state
        _stateMachine.Update(Time.deltaTime);
    }

    float GetDistanceToTarget()
    {
        // return distance
    }
}
```

## 5.2 PushdownAutomaton<T>

A simple Pushdown Automaton with states as objects.

If your state classes have an empty constructor, the state machine can register the states automatically when needed (using *PushStateAuto<TState>()* and *ChangeStateAuto<TState>()*). If not you should register the states yourself using *RegisterStateType<TState>()* or *RegisterStateType()* and use the regular.

The machine will automatically pool the states so you don't have to worry about it.

Whether or popping the last state is valid is up to your design.

**Todo:** Write about how to use PushdownAutomaton.

### 5.2.1 Examples

#### Game State Management

```
public class MainScreen : PdaState<ScreenManager>
{
    public override Begin()
    {
        // show main menu
    }

    public override Reason()
    {
        if (playButton.WasPressed)
        {
            Machine.PushStateAuto<GameScreen>();
            Machine.PushStateAuto<LoadingScreen>();
        }
        else if (howToButton.WasPressed)
            Machine.PushStateAuto<HowToPlayPopup>();
        else if (optionsButton.WasPressed)
            Machine.PushStateAuto<OptionsScreen>();
    }

    public override End()
    {
        // hide main menu
    }
}

public class HowToPlayPopup : PdaState<ScreenManager>
{
    public override Begin()
    {
        // show popup containing tutorial
    }

    public override Reason()
    {
        if (closeButton.WasPressed)
```

(continues on next page)

(continued from previous page)

```
            Machine.PopState();
    }

    public override End()
    {
        // hide popup containing tutorial
    }
}

public class OptionsScreen : PdaState<ScreenManager>
{
    public override Begin()
    {
        // show options menu
    }

    public override Reason()
    {
        if (closeButton.WasPressed)
            Machine.PopState();
    }

    public override End()
    {
        // hide options menu
    }
}

public class LoadingScreen : PdaState<ScreenManager>
{
    public override Begin()
    {
        // show loading overlay
    }

    public override Reason()
    {
        if (doneLoading)
            Machine.PopState();
    }

    public override End()
    {
        // hide loading overlay
    }
}

public class GameScreen : PdaState<ScreenManager>
{
    public override Begin()
    {
        // show gameplay screen
    }

    public override Reason()
    {
        if (player.IsDead)
```

(continues on next page)

(continued from previous page)

```
            Machine.ChangeStateAuto<GameOverScreen>();
    }

    public override End()
    {
        // hide gameplay screen
    }
}

public class ScreenManager
{
    private PushdownAutomaton<Menu> _machine;

    public ScreenManager()
    {
        _machine = new PushdownAutomaton<Menu>(this);
        _machine.ChangeStateAuto<MainScreen>();
        _machine.PushStateAuto<LoadingScreen>();
    }

    public void Update(float deltaTime)
    {
        _machine.Update(deltaTime);
    }
}
```

| Class | Purpose |
|---|---|
| *FiniteStateMachine<T>* | A ‘states as objects’ state machine. |
| *PushdownAutomaton<T>* | Same as FiniteStateMachine<T> but with a stack of states. |

# CHAPTER 6

# Spatial Partitioning

## 6.1 Quadtree<T>

A GC-friendly Quadtree implementation.

Use the static *Create()* and *Destroy()* methods for creating and destroying trees. If you forget to *Destroy()* a tree when you're done with it, it will instead be collected by the GC as normal, but the nodes will not be recycled.

If you're doing 3D, take a look at *Octree&lt;T&gt;*.

### 6.1.1 Usage

To be done.

### 6.1.2 Examples

To be done.

## 6.2 Octree<T>

A GC-friendly Octree implementation.

Use the static *Create()* and *Destroy()* methods for creating and destroying trees. If you forget to *Destroy()* a tree when you're done with it, it will instead be collected by the GC as normal, but the nodes will not be recycled.

If you're doing 2D, take a look at *Quadtree&lt;T&gt;*.

### 6.2.1 Usage

To be done.

### 6.2.2 Examples

To be done.

## 6.3 Bin2D<T>

A simple GC-friendly two-dimensional Bin (aka Spatial Grid) implementation.

When you're done with the Bin, you should call *Dispose()* so its resources can be freed in their object pool. If you forget this, no harm will be done but memory will be GC'ed.

If you're doing 3D, take a look at *Bin3D<T>*.

### 6.3.1 Usage

#### Creation

```
int gridWidth = 25;
int gridHeight = 20;
float cellWidth = 1;
float cellHeight = 1;

Bin2D<object> bin = new Bin2D<object>(gridWidth, gridHeight, cellWidth, cellHeight);
```

When you no longer need the Bin2D, you should call *Dispose()* on it:

```
bin.Dispose();
```

#### Retrieving potential matches

You can retrieve a set of objects that potentially intersects a rect.

```
HashSet<object> results = new HashSet<object>();

Rect queryRect = Rect.MinMaxRect(4, 4, 9, 9);
bin.Retrieve(queryRect, results);
```

You should recycle the results HashSet for future calls to avoid GC allocations. Remember to clear it before you reuse it.

#### Insertion

```
object item = new object();
Rect itemBounds = Rect.MinMaxRect(4, 4, 5, 5);

bin.Insert(item, itemBounds);
```

#### Removing

To remove an item you need to know the bounds that was used to add it.

```
bin.Remove(item, itemBoundsWhenAdded);
```

In case you don't know the previous bounds you can use *Remove(item)* which goes through every cell and removes the item. It's much cheaper if you cache the bounds instead, but it's there if you need it.

```
bin.Remove(item);
```

#### Updating

Updating an item's bounds in the bin is easy. Just like when removing an item, you have to know the bounds that was used when adding it.

```
bin.Update(item, oldItemBounds, newItemBounds);
```

#### Clearing

You can clear all items from the bin with *Clear()*.

```
bin.Clear();
```

## 6.4 Bin3D<T>

A simple GC-friendly three-dimensional Bin (aka Spatial Grid) implementation.

When you're done with the Bin, you should call *Dispose()* so its resources can be freed in their object pool. If you forget this, no harm will be done but memory will be GC'ed.

If you're doing 3D, take a look at *Bin2D<T>*.

### 6.4.1 Usage

#### Creation

```
int gridWidth = 25;
int gridHeight = 20;
int gridDepth = 10;
float cellWidth = 1;
float cellHeight = 1;
float cellDepth = 1;

Bin3D<object> bin = new Bin3D<object>(gridWidth, gridHeight, gridDepth, cellWidth,␣
↪cellHeight, cellDepth);
```

When you no longer need the Bin3D, you should call *Dispose()* on it:

```
bin.Dispose();
```

### Retrieving potential matches

You can retrieve a set of objects that potentially intersects an axis-aligned bounding box.

```
HashSet<object> results = new HashSet<object>();

var center = new Vector3(5, 5, 3);
var size = new Vector3(2, 2, 1);
Bounds queryBounds = new Bounds(center, size);

bin.Retrieve(queryBounds, results);
```

You should recycle the results HashSet for future calls to avoid GC allocations. Remember to clear it before you reuse it.

### Insertion

```
object item = new object();

var center = new Vector3(4, 4, 4);
var size = new Vector3(1, 2, 1);
Bounds itemBounds = new Bounds(center, size);

bin.Insert(item, itemBounds);
```

### Removing

To remove an item you need to know the bounds that was used to add it.

```
bin.Remove(item, itemBoundsWhenAdded);
```

In case you don't know the previous bounds you can use *Remove(item)* which goes through every cell and removes the item. It's much cheaper if you cache the bounds instead, but it's there if you need it.

```
bin.Remove(item);
```

### Updating

Updating an item's bounds in the bin is easy. Just like when removing an item, you have to know the bounds that was used when adding it.

```
bin.Update(item, oldItemBounds, newItemBounds);
```

### Clearing

You can clear all items from the bin with *Clear()*.

```
bin.Clear();
```

There's a bunch of useful Spatial Partitioning system implementations available in SwissArmyLib.

| Class | Purpose |
|---|---|
| *Quadtree<T>* | A GC-friendly Quadtree implementation. |
| *Octree<T>* | A GC-friendly Octree implementation. |
| *Bin2D<T>* | A simple GC-friendly two-dimensional Bin (aka Spatial Grid) implementation. |
| *Bin3D<T>* | A simple GC-friendly three-dimensional Bin (aka Spatial Grid) implementation. |

CHAPTER 7

# Resource System

## 7.1 ResourcePool

**Todo:** Write about ResourcePool

## 7.2 ResourceRegen

**Todo:** Write about ResourceRegen

## 7.3 Shield

**Todo:** Write about Shield

| Component | Purpose |
|---|---|
| *ResourcePool* | A generic resource pool that can be used for health, mana, shield, energy etc. |
| *ResourceRegen* | Regenerates the resource in a resource pool over time either gradually or in intervals. |
| *Shield* | A resource pool subclass that protects another resource pool by absorbing some or all of the damage. |

# CHAPTER 8

# Collections

## 8.1 Grid2D<T>

A fixed (but resizable) 2D grid of items.

### 8.1.1 Usage

#### Creation

```
// creates a 100x100 grid
Grid2D<float> myGrid = new Grid2D<float>(100, 100);
```

You can also define what value cells should have by default:

```
Grid2D<float> myGrid = new Grid2D<float>(100, 100, 1f);
```

#### Changing a cell

```
myGrid[3, 10] = 128f;

// or

myGrid.Set(3, 10, 128f);
```

#### Fill

You can fill everything within a rectangle with *Fill()*.

```
int minX = 30, minY = 40;
int maxX = 50, maxY = 60;
myGrid.Fill(128f, minX, minY, maxX, maxY);
```

#### Clear

You can clear the whole grid with a specific value by using *Clear()*.

```
// sets every cell to the default value
myGrid.Clear();

// sets every cell to 128f
myGrid.Clear(128f);
```

#### Resize

If the size of the grid suddenly doesn't cut it you can always resize it.

The cell contents are kept the same and any new cells are initialized with the *DefaultValue*.

Growing the grid will allocate new arrays, shrinking will not.

```
myGrid.Resize(300, 100);
```

**Todo:** Write some examples for Grid2D

## 8.2 Grid3D<T>

A fixed (but resizable) 3D grid of items.

### 8.2.1 Usage

#### Creation

```
// creates a 100x100x100 grid
Grid3D<float> myGrid = new Grid3D<float>(100, 100, 100);
```

You can also define what value cells should have by default:

```
Grid3D<float> myGrid = new Grid3D<float>(100, 100, 100, 1f);
```

#### Changing a cell

```
myGrid[3, 10, 30] = 128f;

// or
```

(continues on next page)

(continued from previous page)

```
myGrid.Set(3, 10, 30, 128f);
```

#### Fill

You can fill everything within a cube with *Fill()*.

```
int minX = 30, minY = 40, minZ = 23;
int maxX = 50, maxY = 60, maxZ = 80;
myGrid.Fill(128f, minX, minY, minZ, maxX, maxY, maxZ);
```

#### Clear

You can clear the whole grid with a specific value by using *Clear()*.

```
// sets every cell to the default value
myGrid.Clear();

// sets every cell to 128f
myGrid.Clear(128f);
```

#### Resize

If the size of the grid suddenly doesn't cut it you can always resize it.

The cell contents are kept the same and any new cells are initialized with the *DefaultValue*.

Growing the grid will allocate new arrays, shrinking will not.

```
myGrid.Resize(300, 100, 150);
```

**Todo:** Write some examples for Grid2D

## 8.3 PooledLinkedList<T>

PooledLinkedList<T> is a wrapper for LinkedList<T> that recycles its LinkedListNode<T> instances to reduce GC allocations.

You can either have each PooledLinkedList<T> have their own pool, or make them use a shared one.

### 8.3.1 Usage

The API is identical to LinkedList<T> except for two extra constructors, so check out its MSDN documentation.

#### Shared pool

If you do not pass in a node pool in the constructor, it will create its own pool. If you have a bunch of lists with the same type of item, you might want to consider making them share a single node pool.

```
Pool<LinkedListNode<int>> nodePool = new Pool<LinkedListNode<int>>(() => new␣
↪LinkedListNode<int>());

PooledLinkedList<int> list1 = new PooledLinkedList<int>(nodePool);
PooledLinkedList<int> list2 = new PooledLinkedList<int>(nodePool);
```

## 8.4 DelayedList<T>

A list wrapper that delays adding or removing items from the list until its method *ProcessPending()* is called. Useful to avoid problems when you want to be able to change a list in the midst of iterating through it.

### 8.4.1 Usage

#### Creation

```
DelayedList<object> myList = new DelayedList<object>();
```

You can also pass in a **IList<T>** instance if you need it to be something other than a regular **List<T>**.

```
IList<object> myCustomList = new SomeOtherListType<object>();
DelayedList<object> myList = new DelayedList<object>(myCustomList);
```

#### Processing changes

Once you want pending changes to be applied to the list (eg. before iterating through the items), you should call *ProcessPending()*.

Pending changes will be executed in the order they were originally called.

```
DelayedList<object> myList = new DelayedList<object>();

myList.Add(someObject);
Debug.Log(myList.Count);

myList.ProcessPending();
Debug.Log(myList.Count);
```

Will output:

```
0
1
```

#### Adding

```
myList.Add(someObject);
```

After adding an item they still won't be in the list until you call *ProcessPending()*.

#### Removing

```
myList.Remove(someObject);
```

Or by index:

```
myList.RemoveAt(3);
```

After removing an item they will still be in the list until you call *ProcessPending()*.

#### Clearing

You can also queue up clearing the list, removing every previous pending change and all items currently contained in the list the next time *ProcessPending()* is called.

If you add any other changes after calling *Clear()* they will get added as expected once *ProcessPending()* is called.

```
myList.Clear();
```

If you don't want to have to call ProcessPending and just have all items and pending changes cleared instantly, you can instead call the aptly named *ClearInstantly()*.

Do not however that you should naturally not call this while iterating through the items.

```
myList.ClearInstantly();
```

If you only want to clear the pending changes and keep the contents the same, you can call *ClearPending()* at any time.

```
myList.ClearPending();
```

### 8.4.2 Examples

**Todo:** Write some DelayedList examples.

## 8.5 PrioritizedList<T>

A list where the items are kept sorted by their priorities.

**Todo:** Write about how to use PrioritizedList

**Todo:** Write some examples for PrioritizedList

## 8.6 DictionaryWithDefault<TKey,TValue>

Just a regular Dictionary<TKey,TValue> but instead of throwing an error when a key doesn't exist it just returns a specified default value.

### 8.6.1 Usage

**Todo:** Write how to use DictionaryWithDefault

### 8.6.2 Examples

**Todo:** Write some examples for DictionaryWithDefault

There's a bunch of collection types available in **SwissArmyLib**, mostly just used internally but perhaps someone might find them useful.

| Class | Purpose |
|---|---|
| *Grid2D<T>* | A fixed (but resizable) 2D grid of items. |
| *Grid3D<T>* | Same as Grid2D<T> but with an extra axis. |
| *PooledLinkedList<T>* | A wrapper for LinkedList<T> that recycles its LinkedListNode<T> instances to reduce GC allocations. |
| *De-layedList<T>* | A list wrapper that delays adding or removing item from the list until its method *Pro-cessPending()* is called. Useful to avoid problems when you want to be able to add or remove from a list in the midst of iterating through it. |
| *Priori-tizedList<T>* | A list where the items are kept sorted by their priorities. |
| *Dictionary-WithDe-fault<TKey,TValue>* | Just a regular Dictionary<TKey,TValue> but instead of throwing an error when a key doesn't exist it just returns a specified default value. |

# CHAPTER 9

# Utils

## 9.1 BetterTime

BetterTime is just a wrapper for Unity's Time class. The thing with their class is that every call has a marshal overhead since it calls their native C++ code. Since most of the properties don't change as often as they're called, it is worthwhile to cache them. This is what BetterTime does in the background.

It is a small performance benefit, but it is totally free as it's just about writing **BetterTime** instead of **Time** and everything remains the same.

### 9.1.1 Usage

Every UnityEngine.Time property is wrapped and the API stays the same. Just call BetterTime instead of Time.

Eg. instead of:

```
transform.position += velocity * Time.deltaTime;
```

You should write:

```
transform.position += velocity * BetterTime.DeltaTime;
```

## 9.2 ServiceLocator

A (somewhat) simple implementation of the service locator pattern. The **ServiceLocator** knows about **MonoBehaviours** and how to work with them. Creating scene-specific resolvers that only live as long as their respective scene is also supported.

### 9.2.1 Singleton

Whenever a singleton type is resolved the same instance is always returned.

#### Lazyloading

You can also choose to defer the creation of a singleton instance until the instance is first looked up. You can do this by simply using the overloads with 'Lazy' in their name.

### 9.2.2 Transient

If you register a type as transient, a new instance of that type will be created each time it is resolved.

If you for some reason need to work with transient **MonoBehaviours** (Please tell me why, I'm honestly very curious), you should remember to destroy them yourself again when no longer used.

### 9.2.3 Scene-specific resolvers

Both singleton and transient resolvers can be set as scene-specific, so they only live as long as the scene they're in. This works for both **MonoBehaviours** and regular objects.

#### Active scene as defined by Unity

Please note that when you load a new scene, the **MonoBehaviours** in that scene will have their *Awake()* method called before their scene becomes the active one. This means you can't rely on *SceneManager.GetActiveScene()* to return the scene they're in, so you might want to use *GameObject.scene* to specify which scene to register the resolver for.

### 9.2.4 Usage

**Todo:** To be done.

### 9.2.5 Examples

#### Singleton

```
interface IStore
{
    bool Purchase(string sku);
}

class AppleIAP : IStore
{
    public bool Purchase(string sku)
    {
        // buy the item using the App Store API
    }
}
```

(continues on next page)

(continued from previous page)

```
class GoogleIAP : IStore
{
    public bool Purchase(string sku)
    {
        // buy the item using Google's In App Billing API
    }
}

class NullStore : IStore
{
    public bool Purchase(string sku)
    {
        // do nothing or maybe log the action for debugging purposes
    }
}

// At game startup:
class Game
{
    void Initialize()
    {
#if UNITY_ANDROID
        ServiceLocator.RegisterSingleton<IStore, GoogleIAP>();
#elif UNITY_IOS
        ServiceLocator.RegisterSingleton<IStore, AppleIAP>();
#else
        ServiceLocator.RegisterSingleton<IStore, NullStore>();
#endif
    }
}

// An "Ad Removal" purchase button:
class BuyButton
{
    public void OnPressed()
    {
        var success = ServiceLocator.Resolve<IStore>().Purchase("AdRemoval");
        if (success)
        {
            PlaySound("ka-ching.ogg");
            AdManager.Enabled = false;
        }
    }
}
```

### Transient

**Todo:** To be done.

### Scene-specific singletons

**Todo:** To be done.

## 9.3 Attributes

### 9.3.1 ReadOnlyAttribute

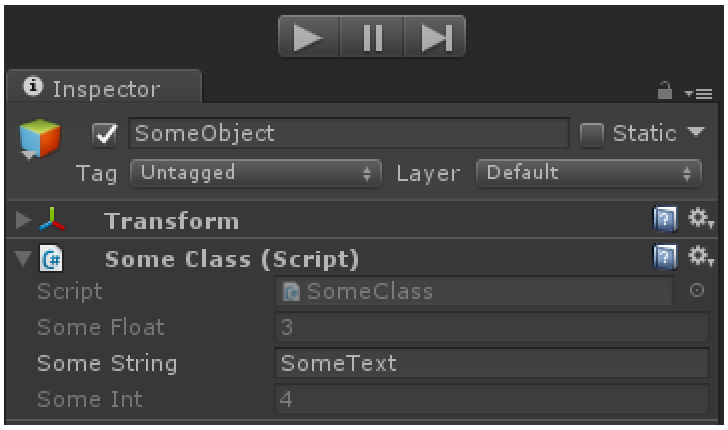
Makes fields marked with it uninteractable via the inspector. You can make the field only be readonly during play mode by setting *OnlyWhilePlaying* to true.

#### Usage

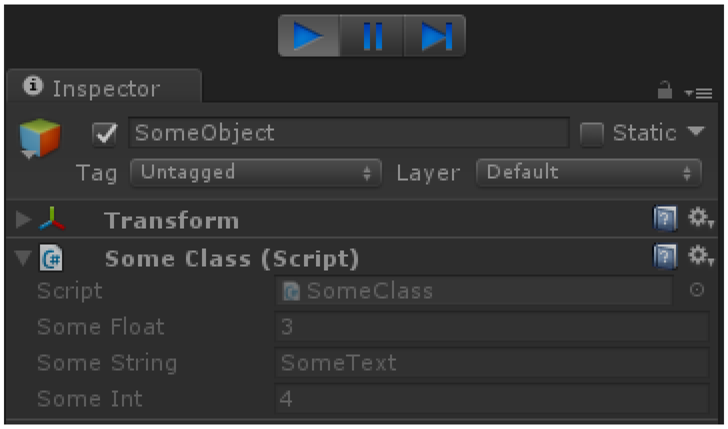
Simply sprinkle the **[ReadOnly]** attribute on a serialized field.

```
public class SomeClass : MonoBehaviour
{
    [ReadOnly]
    public float SomeFloat = 3;

    [ReadOnly(OnlyWhilePlaying=true)]
    public string SomeString = "SomeText";

    [SerializeField, ReadOnly]
    private int _someInt = 4;
}
```

Results in:



Script SomeClass
Some Float 3
Some String SomeText
Some Int 4



SomeObject
Tag Untagged Layer Default
Script SomeClass
Some Float 3
Some String SomeText
Some Int 4

### 9.3.2 ExecutionOrderAttribute

Sets a default script execution order for a MonoBehaviour.

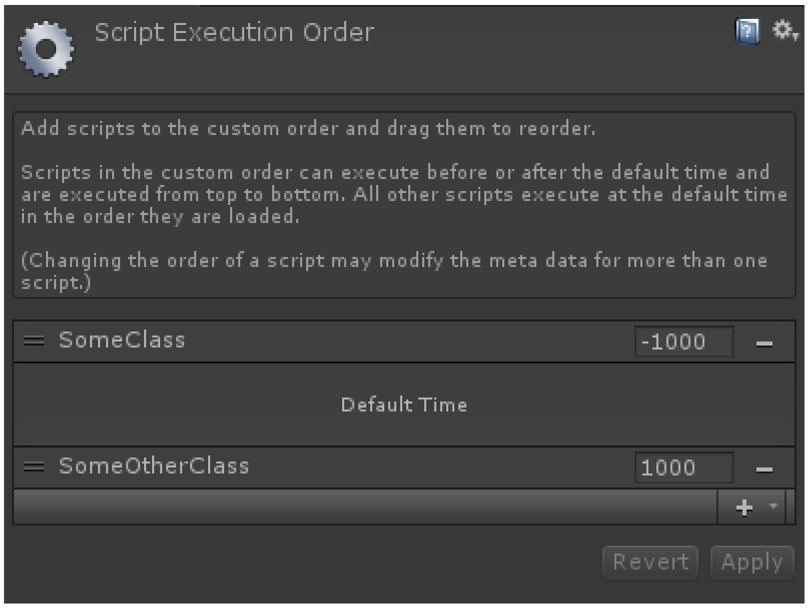
You can also force the order, so that it can't be changed manually.

#### Usage

Just slam the attribute on a MonoBehaviour and specify which order value it should have. If you need it to be forced, then just set *Forced* to true.

```
[ExecutionOrder(Order = -1000)]
public class SomeClass : MonoBehaviour
{

}

[ExecutionOrder(Order = 1000, Forced = true)]
public class SomeOtherClass : MonoBehaviour
{

}
```

Results in:



Add scripts to the custom order and drag them to reorder.

Scripts in the custom order can execute before or after the default time and are executed from top to bottom. All other scripts execute at the default time in the order they are loaded.

(Changing the order of a script may modify the meta data for more than one script.)

| | | |
|---|---|---|
| SomeClass | -1000 | – |
| Default Time | | |
| SomeOtherClass | 1000 | – |

| Attribute | Purpose |
|---|---|
| *ReadOnlyAttribute* | Makes fields marked with it uninteractable via the inspector. |
| *ExecutionOrderAttribute* | Makes fields marked with it uninteractable via the inspector. |

| Topic | Summary |
|---|---|
| *BetterTime* | A wrapper for Unity's Time class that caches its values to improve performance. |
| *ServiceLocator* | A (somewhat) simple implementation of the service locator pattern. |
| *Attributes* | Some useful attributes when working with Unity. |

Repository • API Reference

# CHAPTER 10

# About

**SwissArmyLib** is an attempt to create a collection of useful utilities with a focus on being performant. It is primarily intended for Unity projects, but feel free to rip parts out and use them for whatever you want.

A very important part in the design decisions of this library was to keep the garbage generation low. This means you will probably frown a little upon the use of interfaces for callbacks, instead of just using delicious delegates. It also means using the trusty old *for* loops for iterating through collections where possible.

There's a lot of libraries with some of the same features, but they're often walled off behind a restrictive or ambiguous license. This project is under the very permissive MIT license and we honestly do not care what you use it for.

# CHAPTER 11

# Features

- *Events*
  - Supports both interface and delegate listeners
  - Can be prioritized to control call order
  - Check out *GlobalEvents* if you need.. well.. global events.
- *Timers*
  - Supports both scaled and unscaled time
  - Optional arbitrary args to pass in
  - Also uses interfaces for callbacks to avoid garbage
- *Coroutines*
  - More performant alternative to Unity's coroutines with a very similar API.
- *Automata*
  - *Finite State Machine*
  - *Pushdown Automaton*
- *Pooling*
  - Support for both arbitrary classes and GameObjects
  - *IPoolable* interface for callbacks
    - *PoolableGroup* component in case multiple IPoolable components needs to be notified
  - Timed despawns
- *Service Locator*
  - An implementation of the Service Locator pattern
  - Aware of MonoBehaviours and how to work with them
  - Supports scene-specific resolvers

- Supports both singletons and short-lived objects
  - Singletons can be lazy loaded
- *Managed Update Loop*
  - An update loop maintained in managed space to avoid the overhead of Native C++ –> Managed C#
  - Useful for non-MonoBehaviours that needs to be part of the update loop
  - Optional *ManagedUpdateBehaviour* class for easy usage
- *Spatial Partitioning*
  - GC-friendly implementations of common space-partitioning systems
- *Resource Pool*
  - Generic and flexible resource pool (health, mana, energy etc.)
- Gravity
  - Flexible gravitational system
  - Useful for planet gravity, black holes, magnets and all that sort of stuff.
- Misc
  - *BetterTime*
    - A wrapper for Unity's static Time class that caches the values per frame to avoid the marshal overhead.
    - About 4x faster than using the Time class directly, but we're talking miniscule differences here.
  - Shake
    - Useful for creating proper screen shake
  - *Some collection types*
  - *Some useful attributes*
    - *ExecutionOrder*
      - Sets a default (or forces) an execution order for a MonoBehaviour
    - *ReadOnly*
      - Makes fields uninteractable in the inspector
  - A few other tiny utilities

## 11.1 Download

Binaries for the bleeding edge can be found here. Alternatively you can either *build it yourself* (very easily) or simply *copy the source code into your Unity project* and call it a day.

## 11.2 License

MIT - Do whatever you want. :)

## 11.3 Contributing

Pull requests are very welcome!

I might deny new features if they’re too niche though, but it’s still very much appreciated!