Proto* framework

The Proto* framework

Inspired in part by the articles of *Jean Simonet*:
http://gamasutra.com/blogs/JeanSimonet/20160128/264083/Logic_Over_Time.php and
http://gamasutra.com/blogs/JeanSimonet/20160310/267441/Beyond_the_State_Machine.php

I've been tinkering to implement *something* like this *(being able to write behavior-tree-like AI logic "more naturally", in a coroutine-like fashion)* for some time now, but never got far enough with it until recently, whence stumbling upon, and reading these articles gave me some justification for- and clarifications/extensions of my ideas, and helped me greatly in realizing my implementation, which is a bit different from the one provided along with the articles themselves.

> *(the difference in a nutshell, if you've read the articles: this is modeled on behavior trees instead of state machines, while also ditching the* `YieldInstuction`*s, and iterating over yielded* `Routine`*s themselves, resulting in a more* ***KISS/DRY****,* ***fluid*** *and* ***hackable*** *codebase.)*

Licensed as a **Unity Asset, with some additional extras**, aiming to be a *simple and to-the-point prototyping bootstrap solution* for your projects.

Table of Contents

The Proto* framework....1
- The idea: Using nested coroutines analogous to dynamic behavior trees (and state machines)....3
- How it works....4
- A very basic example....4
  - Thinker.cs....4
- Similarities (and dissimilarities) to behavior trees....5
  - Coroutine specifics....5
    - performing actions over multiple frames....5
    - “building” the behavior tree node-graph....5
    - stopping execution of a node....5
    - advanced topics: handling interruptions, and inter-node communications....5
  - Basic control flow....6
  - Executing a (nested) routine / coroutine....6
  - Parallel execution of routines....6
  - Conditional execution of routines....7
- A more in-depth, practical example....8
  - The Turret.cs script....8
- Under the hood, and extras: Pooling and Signals....13
  - The abstract Routine baseclass....13
  - The Subroutine subclass....13
  - The abstract Composite and Decorator subclasses....13
  - The Thinker class....13
  - The ManagedThinker and ThinkerManager classes....13
  - The StaticPool class....14
  - The PoolManager and Pooled classes....15
  - The Signal classes....16
  - Bonus: the static Intercept helper class....17
  - Profiling tips....18
    - Keeping your thinkers separated I....18
    - Keeping your thinkers separated II....18
    - Avoid thinkers that (mostly) idle....18
- Version history....19
- Copyright / Disclaimer....19

The idea: Using nested coroutines analogous to dynamic behavior trees (and state machines)

The basic idea is an implementation of a behavior-tree-like programming paradigm, or a *"**domain-specific-language**"* (DSL) if you will, using wrapped coroutine iterators and other execution flow control classes resulting in a runtime-dynamically-created "tree" of behavior nodes, specified more expressively trough *actual code* rather than *abstracted* node-based-trees/graphs.

Most of the `Selector`/`Sequence`/`etc` nested node logic in behavior-tree DSL-land can be replaced by writing actual code consisting of `if`/`switch`/`do`/`while`/`for` statements inside iterators yielding subroutine nodes.

The articles mentioned in the foreword also make strong points about the coroutine approach having the effect of being able to localize/encapsulate the current state of the AI, and point out, how instead of a pseudo-global "blackboard", having the state coupled locally and separately, in local variables of the coroutines / closures themselves can provide the same level functionality, but can be actually a safer and more reliable way to code, as – apart from making the code more followable – the inner state of a coroutine usually cannot be accidentally overridden from somewhere else deep in the tree by some forgotten child leaf/state resetting a "blackboard"-ed value.

This approach results in a much more readable (and maintainable) code-base, while allowing faster prototyping of AI behaviors and other stuff (like the code for a quest's interactions, NPC dialog trees, etc).

How it works

- you write `IEnumerable<Routine>` return-typed **coroutines**
- which `yield return` special `Routine` instructions (or `null`s)
  between blocks of regular (coroutine) code
- you "**invoke**" child nodes using various `(Routine).Run (...);` -like calls
- you only need to tick the `Update()` function on the root node (`Routine`) yourself
- a dynamic "behavior tree", with a current "active branch" will be automatically maintained *trough yielding, in the background*
  (with additional pooling and reusing of nodes for efficiency)

A very basic example

...illustrating the points above (we'll cover the more in-depth stuff later):

Thinker.cs

```
Using Proto;
Using Proto.AI;
Using UnityEngine;

// a simple monobehaviour baseclass, running a dynamic behavior tree
public class Thinker : MonoBehaviour
{
    // the root node of our dynamic tree
    Routine routine;

    protected virtual void OnEnable ()
    {
        // initialize the root node: Run the Think() coroutine
        routine = Subroutine.Run (Think ());
    }

    protected virtual void OnDisable ()
    {
        // dispose the behavior, when not needed anymore
        if (routine != null)
        {
            routine.Dispose ();
            routine = null;
        }
    }

    protected virtual void FixedUpdate ()
    {
        // tick the root node
        if (routine != null && routine.running)
            routine.Update ();
    }

    protected virtual IEnumerable<Routine> Think ()
    {
        // do nothing for now, we'll override later
        // just skip a frame, then terminate
        yield return null;
    }
}

// -- EOF:Thinker.cs --
```

Similarities (and dissimilarities) to behavior trees

Coroutine specifics

performing actions over multiple frames

behavior trees nodes usually return with a `Running` state when their inner logic wants to signal "*done for the current frame*", while coroutines can simply do

```
yield return null;
```

to suspend execution of the logic until the coroutine is ticked the next time.

"building" the behavior tree node-graph

```
yield return Subroutine.Run (IEnumarable<Routine> routine)
yield return Parallel.Run (params object[] routines)
yield return While.Run (Func<bool> condition, Routine routine)
yield return Until.Run (Routine condition, Routine routine)
```

yielding any other value (any subclass of `Routine` in an `IEnumerable<Routine>` coroutine) will result in execution of that (child) routine in place of the current routine, returning execution to the current (parent) routine when the child routine is finished running.

or, in *behavior-tree-speak*: every `yield return`-ed `Routine` means a (sequential) child node in the current iterator "`Sequence`" in the behavior tree being built at run-time.

stopping execution of a node

```
yield break;
```

is supported, however you can easily work around and can usually avoid resorting to it, resulting in more readable and usually much neater code.

> to dispel any confusions: a `yield break` immediately stops the **whole coroutine**, and **not just the current** `switch`/`do`/`while`/`for` **block** inside the coroutine!
>
> this is because the coroutine is actually compiled down to a special iterator by the compiler internally, and a `yield break` is the way to stop the *whole* iterator when iterating, so consequently you just can't skip to the end of the "current block" with it.

thus, a `yield break` **always stops the coroutine *iterator as a whole*,**
and returns execution to it's caller (parent) routine immediately.

advanced topics: handling interruptions, and inter-node communications

the in-depth example (*found later in this document*) shows examples of handling **outside interruptions** of the coroutine from within a coroutine itself using special `try`/`finally` blocks (leaving out the `catch` clauses) to restore side-effects on a mid-run termination, and how this can be useful in some situations. (see: `Turret.cs`: `TrackTarget()`)

the example also demonstrates how you can use closures / lambda functions to communicate information between nodes. (see: `Turret.cs`: `Thinker()` + `FindTargetInRadius()`)

Basic control flow

the major difference between behavior trees and this implementation is that:

there are no *direct* analogues for...

```
Sequence
Selector
Loop
Inverter
Succeeder
```

… *(etc)*

type behavior nodes, as you can easily replace the logic they impose by writing coroutines containing actual `if`/`switch`/`do`/`while`/`for` statement logic.

for some other behavior-tree functionalities however, you can `yield return` special routines using various `(Routine).Run(...)` -like calls.

Executing a (nested) routine / coroutine

```
yield return Subroutine.Run (IEnumarable<Routine> routine)
```

execute the coroutine in place, returning to the caller when it finishes.

note: the coroutine can execute and finish in the same frame,
returning immediately to the caller.

Parallel execution of routines

```
yield return Parallel.Run (params object[])
```

executes multiple subroutines at once, in parallel
returning to the caller once (all / any one) of the subroutines have terminated.

this **variable argument list** function can accept and understand multiple types of parameters, in any order, including:

`Parallel.ExitPolicy`	for overriding the default `RequireAll` exit-policy to a `RequireOne` exit-policy (see note below)
`Routine`	run the routine in parallel with the others
`Routine[]`	run the routines in parallel with the others
`List<Routine>`	run the routines in parallel with the others
`IEnumerable<Routine>`	`Subroutine.Run()` the coroutine in parallel

note: the **(default)** `RequireAll` exit-policy **waits until all** subroutines have finished running, while the `RequireOne` exit-policy will terminate *(interrupt)* the remaining running subroutines **when any one** of the subroutines finishes running.

also note: specifying **multiple exit-policies** in the argument list will make **only the last one** specified take effect.

Conditional execution of routines

```
yield return While.Run (Func<bool> condition, Routine routine)
yield return Until.Run (Routine condition, Routine routine)
```

`While`: executes the subroutine **while** the boolean condition **stays** `true`

`Until`: executes the subroutine in parallel **until** the condition subroutine **finishes running.**

note: the `routine` *may* be terminated *(interrupted),* when the `condition` becomes `false` / the `condition` routine finishes running.

the `routine` may also finish long before the `condition` triggers, and will **not** be looped in such case. (**tip:** wrap the inner logic of the `routine` in a `while(true)` loop to do that.)

note: you can also pass an `IEnumerable<Routine>` coroutine in place of the `Routine` parameters, these will be fed trough a `Subroutine.Run()` call automatically.

additionally, for *syntactic sugar,* `Until` and `While` can be used interchangeably: as both have the same overloads (referencing each other internally), and both were designed to do a similar thing: one is for checking boolean conditions, the other for checking a subroutine's running status.

this allows for more code legibility in situations where we would want to write something like *"until(foundtarget, idle)"* or *"while(targetvisible, shoot)"* with, for example, both using coroutines as conditions.

the rule of thumb is:
if the first parameter is a...

`Func<bool>` (a lambda/delegate/callback)	...it runs a `While` routine instance
`Routine` or `IEnumerable<Routine>`	...it runs an `Until` routine instance

the full (and interchangeable) overload list for both classes:

```
yield return (While/Until).Run (Func<bool>, Routine)
yield return (While/Until).Run (Func<bool>, Ienumerable<Routine>)
yield return (Until/While).Run (Routine, Routine)
yield return (Until/While).Run (Routine, Ienumerable<Routine>)
yield return (Until/While).Run (IEnumerable<Routine>, Routine)
yield return (Until/While).Run (IEnumerable<Routine>, IEnumerable<Routine>)
```

A more in-depth, practical example

The `Turret.cs` script

Let's take a look at the code driving the turret in the provided example scene, with some **additional code comments** explaining the nuts and bolts.

The class was modeled on the example class used in the article at: http://gamasutra.com/blogs/JeanSimonet/20160310/267441/Beyond_the_State_Machine.php, to highlight the differences of the implementations, and to provide a practical, guided reference on the usage patterns and benefits of the approach, based around the same ideas as discussed in the article.

You can see this script in action by loading the example scene, clicking on the play button, then clicking on the arena ground to make the target character move around the turret.

`Turret.cs`:

```
// 1/5
Using Proto;
Using Proto.AI;
Using System;
Using System.Collections.Generic;
Using UnityEngine;

// we subclass the Thinker class seen before
public class Turret : Thinker
{
    // some inspector fields
    public Bullet bulletPrefab;         // the bullet spawned
    public GameObject trackingLight;    // enabled when target is in radius
    public Transform barrel;            // spawn point/direction of bullet

    public float targetRadius = 10f;    // detection radius

    public float trackSpeed = 2f;       // turret turn/track speed
    public float shotDelay = 1f;        // delay between shots

    //-----------------------------------------------------.

    protected override void OnEnable ()
    {
        // disable the tracking light by default
        trackingLight.SetActive (false);

        // don't forget to call base.OnEnable ();
        base.OnEnable ();
    }

    //-----------------------------------------------------
    // a (bad) example of a coroutine with a self-contained state
    // wait n seconds

    protected IEnumerable<Routine> Wait (float seconds)
    {
        // the inner “state”, as a local variable
        float start = Time.time;

        while (Time.time - start < seconds)
            yield return null;
    }

// continued on the next page...
```

Turret.cs:

```
// 2/5

    //--------------------------------------------------------
    // another simple example:
    // turn toward initial facing, aka: the idle state

    IEnumerable<Routine> Idle (Vector3 startAngle)
    {
        while (true)
        {
            transform.forward = Vector3.RotateTowards
            (
                transform.forward,
                startAngle,
                Time.deltaTime * trackSpeed,
                0f
            );

            yield return null;
        }
    }

    //--------------------------------------------------------
    // firing projectiles at intervals

    // demonstrates yielding another Routine with "yield return"
    // (and passing control to it until it finishes)

    IEnumerable<Routine> FireProjectiles ()
    {
        while (true)
        {
            // spawn a projectile
            // we use the PoolManager class to spawn a Bullet prefab
            Bullet bullet = PoolManager.Spawn<Bullet>
            (
                bulletPrefab,
                barrel.position,
                barrel.rotation
            );

            bullet.gameObject.SetActive (true);

            // yielding another coroutine
            // Wait() shotDelay seconds before next shot
            yield return Subroutine.Run (Wait (shotDelay));
        }
    }

// ...continued on the next page...
```

Turret.cs:

```
// 3/5

    //-------------------------------------------------------
    // override the Think coroutine used by the base Thinker class
    // this will be ticked on FixedUpdate() instead

    // demonstrates advanced control flow routines
    // using While/Until and Parallel.Run() calls
    // to tie together the other coroutines in this example

    protected override IEnumerable<Routine> Think ()
    {
        // save turret start angle
        Vector3 startAngle = transform.forward;

        while (true)
        {
            // this local var is part of the routine's self-contained state
            // and cannot be accidentally overridden from somewhere else
            Transform target = null;

            // look for a target
            yield return Until.Run
            (
                // this coroutine looks for a target,
                // and puts it in target before completing
                // using a lambda callback
                FindTargetInRadius
                (
                    targetRadius,
                    (found) => (target = found)
                ),
                // this happens while we look for a target
                // (or 'until' we've found one)
                Idle (startAngle)
            );

            if (target != null)
            {
                // we have a target
                // track and shoot simultaneously
                yield return While.Run
                (
                    // the (lambda) condition
                    () =>
                        (
                            Vector3.Distance
                            (target.position, transform.position)
                                < targetRadius
                        ),
                    // this happens while the condition stays true
                    Parallel.Run // simultaneously...
                    (
                        // ...track the target
                        TrackTarget (target),
                        // ...and fire projectiles in timed intervals
                        FireProjectiles ()
                    )
                );

                // target left the area, reset and look for another
                target = null;
            }
        }
    }

// ...continued on the next page...
```

Turret.cs:

```
// 4/5

    //-------------------------------------------------------
    // tracking the target
    // advanced example

    // demonstrates handling of outside interruptions
    // of the coroutine

    // when using While / until / Paralell.ExitPolicy.RequireOne
    // (or yield break-ing from the inside on some occasions)
    // running routines may be interrupted mid-run

    // these interruptions can be handled with try/finally blocks
    // ie: try/catch blocks that don't need to catch anything,
    //     but still have a finally clause

    // rule of thumb: all finally blocks, whose try blocks were entered
    //  - this means nested try/finally blocks too! -
    // execute upon terminating (disposing) of a routine (iterator)

    IEnumerable<Routine> TrackTarget (Transform target)
    {
        try
        {
            // enable tracking light
            trackingLight.SetActive (true);

            // turn towards target
            while (true)
            {
                Vector3 direction =
                    (target.position - transform.position).normalized;

                transform.forward = Vector3.RotateTowards
                (
                    transform.forward,
                    direction,
                    Time.deltaTime *  trackSpeed,
                    0f
                );

                yield return null;
            }
        }
        finally
        {
            // disable tracking light when tacking is interrupted
            trackingLight.SetActive (false);
        }
    }

// ...continued on the next page...
```

Turret.cs:

```
    // 5/5

        //--------------------------------------------------------
        // finding a target in range
        // advanced example

        // demonstrates returning a value to the caller via a callback

        // for the receiving end, see the overridden Thinker() function
        // near the top after "look for a target" in the While.Run() call

        IEnumerable<Routine> FindTargetInRadius
            (
                float radius,
                Action<Transform> foundTarget
            )
        {
            Collider target = null;

            while (target == null)
            {
                List<Collider> colliders = new List<Collider>
                (
                    Physics.OverlapSphere (transform.position, radius)
                );

                target = colliders.Find
                (
                    (collider) =>
                        collider.GetComponentInParent<Target> () != null
                );

                yield return null;
            }

            // pass target to caller
            foundTarget (target.GetComponentInParent<Target> ().transform);
        }
    }

    // ...last page
    // -- EOF:Turret.cs --
```

Under the hood, and extras: Pooling and Signals

The abstract `Routine` baseclass

which is an `abstract` coroutine representation, implemented as a combination of a `BehaviorTreeNode` and a `YieldInstruction`.

The `Subroutine` subclass

which is a special coroutine implementation that ticks an `IEnumerable<Routine>` iterator, and handles executing the `yield return`-ed `Routine`-s from it, acting effectively as the "current executing branch" forming nodes of the dynamic "behavior tree".

The abstract `Composite` and `Decorator` subclasses

are the behavior-tree node analogous implementations of:

- an `abstract Composite` routine, the base class of the `Parallel` routine, and;
- an `abstract Decorator` routine, the base class of the `While`/`Until` routines.

The `Thinker` class

is available as the concrete implementation of a `Thinker` discussed earlier,
used by the provided example scene.

The `ManagedThinker` and `ThinkerManager` classes

are an alternate drop-in-replacement implementation for the `Thinker`, providing the same functionality, but utilizing unly one Unity callback, the `FixedUpdate` in the `ThinkerManger` to tick all `ManagedThinker`-s.

to use, just subclass from `ManagedThinker` instead of `Thinker`, and have a manager `GameObejct` with a `ThinkerManager` component attached (and active) in the scene.

you'll most likely won't need it, unless you're optimizing for that extra few percents of cpu, but provided as an example for e.g. a quest tracking system, with a centralized manager of routines that could govern what happens during a quest, etc.

The `StaticPool` class

is used for pooling `Routine` instances by `Type`
(reducing the memory allocation footprint of the framework greatly).

this class also enables a simple pooling mechanism for any `Type` easily.

(the framework also contains a similar solution for pooling `GameObject` prefabs, which will be described in the next section.)

usage:

```
// instantiate a new instance,
// Request() returns null if no pooled instance is available,
// so we also call 'new' here, if that's the case
T instance = (StaticPool.Request<T> () ?? new T ());

// reinitialize the instance
// (here) by using a late constructor provided by the class
instance.Construct (params);

// - see the Run()/Construct() functions in the routines for examples
// - and enforcing pooling with private/protected constructors


// ... use the object in the code ...


// later: return an instance to the pool
StaticPool.Return (instance); // we don't need the <Type> here

// protip:
// this can be automatically called from an implementation of
(IDisposable)instance.Dispose ();
// provided by the class
// see Routine.Dispose (); for IDisposable example
```

The `PoolManager` and `Pooled` classes

implements a simple pooling mechanism for `GameObject` prefabs, where the pools are accessed by using a prefab `GameObject` (sometimes of an attached `Component`) as the key (instead of a `Type` as in a `StaticPool`).

`GameObject` instances are **released** from the pool in a state of **being disabled**, ready to be re-initialized by the caller, and then **expected to be manually enabled** with a `gameObject.SetActive (true);` call.

the instances will be **automatically returned** to the pool **when disabling them** (by

calling `gameObject.SetActive (false);`), handled via the `Pooled` component, which gets auto-added to each pooled `GameObject` instance spawned by the `PoolManager`.

usage:

```
// spawn a GameObject from a prefab pool, using a GameObject as a key
// using PoolManager.Spawn() the same way (same overloads)
// as we would call GameObject.Instantiate ()
// (but: Spawn() always returns a disabled GameObject!)
GameObject instance = PoolManager.Spawn (prefabGO);

// or, using a Bullet MonoBehaviour as the key
Bullet bullet = PoolManager.Spawn<Bullet> (bulletPrefab);
// internally, the pool used will be the bulletPrefab.gameObject pool

// reset stuff/reinitialize pooled instance
bullet.transform.position = barrel.position;
bullet.transform.rotation = barrel.rotation;

// activate gameObject in the scene manually
// (remember: PoolManager.Spawn() returns a disabled GameObject!)
bullet.gameObject.SetActive (true);


// ... do stuff in the scene ...


// return to the pool (handled by the Pooled behavior)
bullet.gameObject.SetActive (false);

// for more examples, see the
// Bullet/Trail/Hit classes provided for the example scene
```

note: about the `Pooled.cs` file:

this contains a `#define NICE_POOLS`, which, when enabled, will make pooled objects reside in an organized hierarchy under an empty `GameObject`, called *"PoolManager"* in the scene, (unless, of course, you reparent their transforms on your re-initialization of instances) allowing pools to behave nicer in the editor hierarchy while testing.

commenting out this line will **dump all** the disabled/enabled pooled objects in the **root of the active scene**, without any hierarchy at all (but still tracked / pooled the same way by the `PoolManager` as you would expect).

The `Signal` classes

are meant to be used as a compile-time, type-safe event-system, implementing the *observable design pattern* with `System.Action` delegate type callbacks.

based on the Signals implementation found in *StrangeIoC*, written by *Will Corvin*: strangeioc.wordpress.com/2013/09/18/signals-vs-events-a-strange-smackdown-part-2-of-2/

side note: sadly, the C# spec forbids me to make these more DRY, as we cannot use any delegate types as a generic parameter to a would-be `BaseSignal<CallbackType>` class.

so in contrast to *Strange*, i opted not to have a base class (as it felt a little forced without inversion of control/injection), and instead opted to provide five very similar, but thus simpler classes, each with different callback signatures matching each of the five `System.Action` delegate types:

`class Signal`	for `Action()` callbacks
`class Signal<T>`	for `Action(T)` callbacks
`class Signal<T,U>`	for `Action(T1,T2)` callbacks
`class Signal<T,U,V>`	for `Action(T1,T2,T3)` callbacks
`class Signal<T,U,V,W>`	for `Action(T1,T2,T3,T4)` callbacks

tip: if you need more than four differently typed callback parameters for a signal, you're probably better off enclosing them in a single, typed value-object (i.e. in a custom `class`/`struct`, as member fields), and passing that as a single parameter instead.

usage:

```
// defining Signals as a members/static variables
public Signal<float> tookDamage = new Signal<float> ();
public static Signal OnSomethingHappened = new Signal ();

// you can also subclass, and make your own signal types
public class DamagedBySignal : Signal<float,Actor> { }

// attach/remove listeners
// the compiler will warn about mismatched callback signatures
void OnEnable ()
{
    tookDamage.AddListener (TakeDamage);
    // you can attach listeners that remove themselves once signaled
    OnSomeThingHappened.AddOnce (DoSomething);
    // ! you can not add the exact same listener twice (the same way)
    OnSomeThingHappened.AddOnce (DoSomething); // won't fire twice
}

void OnDisable ()
{
    tookDamage.RemoveListener (TakeDamage);
    OnSomeThingHappened.RemoveListener (DoSomething);
}

// use simple parameter type-safe listeners for the signals
void TakeDamage (float damage) { }
void DoSomething () { }

// (...and later from somewhere else in the code:)
// dispatch a signal to all listeners
instance.tookDamage.Dispatch (120f);
TheClass.OnSomethingHappened.Dispatch ();
```

Bonus: the static `Intercept` helper class

provides a handy vector math function for anticipating the future position a moving target with a projectile fired from a moving gun:

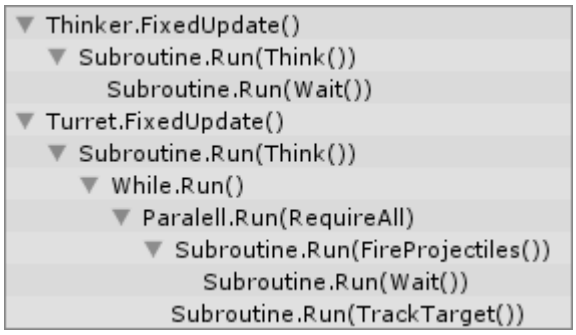
```
static Vector3 Intercept.Point
(
    Vector3 shooterPosition,
    Vector3 shooterVelocity,
    float shotSpeed,
    Vector3 targetPosition,
    Vector3 targetVelocity
);
```

the parameters `shooterPosition` and `shooterVelocity` are the position and velocity of the **shooter,** `shotSpeed` is the speed of the **projectile,** and `targetPosition` and `targetVelocity` are the position and velocity of the **target.**

the resulting `Vector3` is the position where the path of the target and a projectile fired at `shotSpeed` from the shooter (while also inheriting the speed/inertia from the shooter) *can intersect*, i.e. the point the shooter should be aiming for a hit. if the equations can not predict such an intercept point ahead, the `targetPosition` parameter will be returned unchanged.

tip: for example, by substituting `shotSpeed` with the maximum possible speed of the shooter itself, we can also use this function calculate a point ahead of a moving target, anticipating it's movement while chasing it, etc.

Profiling tips



The `Routine` classes are profiler-friendly, showing up as additional rows in the hierarchical data of the **CPU Usage** area of the profiler under the row of the function that ticks them *(as presented on the right).*

What you can see here are the so-called ***current executing branches*** of the ***behavior trees*** we are building at run-time.

In the example on the right, the `Thinker` is waiting on something, and the `Turret` has found it's `target` and is now simultaneously tracking it and shooting at it while it's in range.

Keeping your thinkers separated I

if you decide to use a similar architecture in your project for your actors as given in the example (ie: a base `Thinker` class with multiple subclasses as actors), you may find it useful to override the Unity callback that ticks the root routine (in our case `FixedUpdate`):

```
#if UNITY_EDITOR
protected override void FixedUpdate ()
{
    base.FixedUpdate ();
}
#endif
```

this makes the profiler show the subroutines of the actual subclass under it's own row instead under `Thinker.FixedUpdate` *(see above)*

tip: if you tick your routines elsewhere (from something like a `ManagedThinker`) you can wrap the `Routine.Update` call with a `Profiler.BeginSample` and `Profiler.EndSample` call to make it show up on a separate row under the caller *(see the Unity Manual for additional reference on the `Profiler` class).*

Keeping your thinkers separated II

instead of one behemoth *does-all-the-thinking* `MonoBehaviour`, have multiple simpler concurrent thinkers, for example, by having an `EnemyVehicle` handling the navigation and a `Turret` handling the shooting for a `Tank` actor, we can for example also enable/disable these separately, when needed.

Avoid thinkers that (mostly) idle

thinkers that mostly just wait for something to happen, like a `Destructable` spawning an explosion and some debris when the health drops below zero, ***could*** *hog up cpu/memory resources,* depending on the complexity of the coroutine (and it's parents) which in this case means the local and outside lambdas, etc (which are then mirrored internally in the compiler-generated code), *especially when lots of instances exist in the scene at the same time*, and so it's wiser to instead to **implement such behaviors as on-event triggered, and ticked separately, only when really needed.**

Version history

1.0	Apr-2016	Initial release on the Asset Store.

Copyright / Disclaimer