Reading time: 15 minutes - Last Updated: 2019.9.9

Active Logic Quick Start Guide

Inquiries and support: active.behaviours@gmail.com

NOTE: This guide may be out of date. Read the latest version online.

Be welcome! This document is a hands on introduction to Active Logic. Let's get started.

Using the API

Throughout this guide we assume that your source files include the following *using* statements:

```
using Active.Core;
using static Active.Core.status;
```

1. Status Expressions

Consider a basic soldier AI. A soldier may attack, defend or retreat; this can be modeled using a status expression:

```
status s = Attack() || Defend() || Retreat();
```

The above realizes a fallback *strategy* (commonly referred to as a *selector*), reading:

> *If possible, attack,*
> *otherwise, defend,*
> *otherwise, retreat.*

A status may be *complete*, *failing* or *running*. A value of *running* causes execution to yield until the next iteration.

Status expressions must be invoked frequently - usually, within update loops:

```
class Duelist : MonoBehaviour
{
    public status current = cont();

    void Update()
    {
        current = Attack() || Defend() || Retreat();
    }
}
```

While the above example is valid, Active Logic provides its own *MonoBehaviour* subclass, with several benefits (see §4).

➡ *When assigning a status directly, use done(), fail() or cont().*
➡ *Terms in a status expression are often referred to as* 'tasks' *or* 'subtasks'.
➡ *Because running is neither* true *nor* false*, status does not convert to bool. Instead, query the* running*,* failing *and* complete *properties.*

A key to understanding status expressions is that a later task does not get evaluated until prior tasks are traversed. For this reason, status functions are often implemented using the following approach:

```
status DoSomething(){
    if( GUARD ) return fail();
    REALLY_DO_SOMETHING;
}
```

If the guard condition restricts the generality of a status function, move it up the call stack:

```
// Using the conditional 'AND'...
status s = ( GUARD && DoSomething() ) || SomethingElse();
// The ternary conditional...
status s = GUARD ? DoSomething() : SomethingElse();
// ...or even like this:
status s = ( GUARD ? DoSomething() : fail() ) || SomethingElse();
```

Status expressions implement stateless control. In our model this approach is beneficial - depending on how Attack() is implemented, a wounded agent may receive healing and get back on the offense. Here is a possible implementation of the Attack() function:

```
status Attack(){
    if(health < 25) return fail();
    if(!threat)     return fail();
    return Engage(threat) && Strike(threat);
}
```

Notice the *Approach(threat) && Strike(threat)* idiom. In Active Logic, the conditional operator AND (&&) is used to implement action sequences.

- A sequence (&&) evaluates each subtask in sequential order. A prior subtask must complete before the next subtask may start.
- A selector (||) evaluates each subtask in preference order. A prior subtask must fail before alternatives are tried.

In BT, *selectors* and *sequences* are collectively known as composites.

In all, status expressions and status functions combine to form behavior trees. A key difference with other BT implementations is that the behavior trees are statically wired in program memory, instead of using explicit data structures (where useful, the latter is also supported).

2. Decorators

In the above example, we've hinted at a *Strike()* action. Generally, effective control requires a variety of small 'utilities' used to modulate behavior.

A staple of video game design, the humble cooldown is one such thing:

```
status Attack(){
    return Engage(threat) && Cooldown(1.0f)?[ Strike(threat) ];
}
```

Looking at the above snippet you might be asking 'where is the time stamp'?

Indeed, invoking *Cooldown(1.0f)* implies allocating storage for a *Cooldown* object; this is managed on your behalf by the underlying implementation.
For this reason, the above syntax only works if your behavior is derived from *UTask* or *Task*

The API offers a few built-in decorators; you may also create your own, custom decorators

➡ *Decorator syntax uses the null-conditional operator.*

3. Ordered composites

Status expressions are great but there is a utilitarian aspect to them which does not fit every use. Let's take an example:

```
status s = Reach(A) && Reach(B);
```

Applied repeatedly, assuming A and B are navigation waypoints, this would cause an actor to jitter around A: every time we move towards B, the first step gets undone. This condition (switching rapidly between subtasks, without achieving anything) is known as *dithering*.

Where an actor are intended to follow a sequence of steps *by design*, use an ordered composite:

```
status s = Seq()[           // Or 'Sel' for ordered selectors
    and ? Reach(A) :        //
    // . . .                // Can use any number of subtasks
    and ? Reach(B) : end ]; // Use 'end' or 'loop'
```

➡ *The above syntax is available in a* UTask *context. Once again, this is because ordered composites, similar to decorators, are stateful (an int tracks the currently executing subtask).*

With an ordered composite, the following is true…

For an ordered sequence:

- Subtasks are attempted in order. Once a subtask has completed, it will not be attempted again and we move on to the next subtask.
- If any subtask in a sequence fails, the sequence is aborted (*status.failing*)
- Once the last subtask in a sequence has succeeded, the sequence is complete (*status.complete*)

For an ordered selector:

- Subtasks are attempted in order. Once a subtask has failed, it will not be attempted again and we move on to the next subtask.
- After completing any subtask the selector is complete;
- If the last subtask in a selector fails, the selector has failed.

Once an ordered composite has succeeded or failed, further iterations have no effect (*end* keyword) or the composite does loop over (*loop* keyword).
At any time, invoke *UTask.Reset()* to return a composite to its initial state.

➡ *Ordered sequences and selectors may use or embed within, status expressions.*

4. Tasks and frame agents

We have already mentioned *UTask*. In Active Logic, *UTask* is a useful base class derived from *MonoBehaviour.* Specifically, *UTask* provides the following benefits:

1. *UTask* updates via the *Step()* method. Unlike *Update()* this ensures that subtasks are not running simultaneously when they should not.

To define a task you only need to inherit from Task/UTask and override the *Step()* method:

```
public class MyTask : UTask
{
    override protected status Step()
    {
        /* do something… */
        return anyStatus;
    }
}
```

2. *UTask* objects may partake status expressions. Say for example we want a farmer AI, but with defensive abilities:

```
public class DefensiveFarmer : UTask
{
    Farming farm;
    Fighting fight;

    override protected status Step() => fight || farm;
}
```

In the above we simply combine both roles using a status expression.

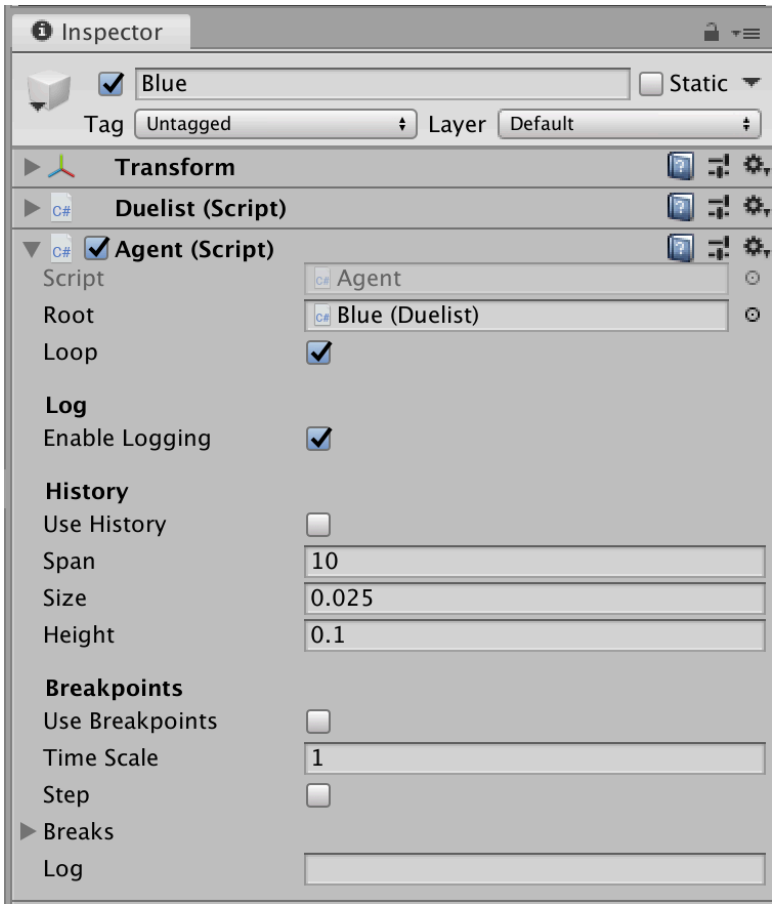
3. UTask allows a concise syntax for decorators and ordered composites

Strictly speaking, you can use decorators and ordered composites without *UTask*. However the syntax is usually less concise and less readable.

Let's have a look at a more complete example involving *UTask*.

```
using Active.Core;
using static Active.Core.status;

public class Duelist : UTask{

    float health = 100;

    override protected status Step()
        => Attack() || Defend() || Retreat();


    status Attack() => threat && health > 25
        ? Engage(threat) && Cooldown(1.0f)?[ Strike(threat) ]
        : fail();

    status Defend()  => undef();

    status Retreat() => undef();

}
```

➡ undef() *denotes an unimplemented status function. While debugging, undef randomizes its return value, which is useful for testing incomplete models;* undef() *is only allowed in the Unity Editor.*



Since *UTask* inherits from *MonoBehaviour,* The *Duelist* component may be added to a Unity game object. A task, however, does not run on its own.

To make this runnable, add a frame agent (*Add Component* > *Active Logic* > *Agent*). Then, in the frame agent inspector, assign *Duelist* as root (drag the *Duelist* component onto the frame agent's 'root' field).

You may download and try the complete Duelist demo online.

5. Logging and the Log-Tree

Active Logic offers a log-tree feature; this is helpful, in that the behavior of complex agents is rarely transparent, and tracing update loops running at 10~120Hz is not convenient.

In Unity, access the Log-Tree window from the Window menu; then, enable logging from the Frame Agent inspector

Currently, logging requires annotating the source. The following example demonstrates this.

➡ *If you want interactive feedback, but do not enjoy annotating sources, we offer a freely available logging automation, Prolog.*
Compared to the API's logging feature, Prolog does not provide hierarchic views. It does, however, reflect every function call without the need for manual input.

```
using Active.Core;

public class Duelist : UTask{

    float health = 100;

    override protected status Step() => Eval(
        Attack() || Defend() || Retreat()
    );

    status Attack() => Eval(                                          // 1
        !threat     ? fail("No enemy around") :                       // 2
        health < 25 ? fail(log && $"Health too low :{health}")  :  // 3
        Engage(threat) && Cooldown(1.0f)?[ Strike(threat) ]
    );

    status Defend()  => undef();

    status Retreat() => undef();

    status Engage(Transform threat){
        var dist = Vector3.Distance(transform.position,
                                    threat.position);
        return Eval(
            dist < 1f ? done(log && "In range") : Approach()
        )[log && $"At {dist:0.#}meters from target"];
    }

    status Approach() => undef;

}
```

The above example illustrates everything you need to know to get started with logging:

1: Whenever using *return* or =>, use *Eval:*

```
status Attack() => Eval( STATUS_EXPRESSION );
```

or

```
status Attack() { return Eval( STATUS_EXPRESSION ); }
```

The exception to this is when you return a status constant:

```
status JustDoIt() => done();  // Eval not required here
```

***2:* With *done()*, *fail()* and *cont()*, a custom log message may be passed as argument**

Passing a literal string to *fail:*

```
fail("No enemy around")
```

When passing a formatted string, use string interpolation and the *log && message syntax:*

```
fail(log && $"Health too low :{health}")  :
```

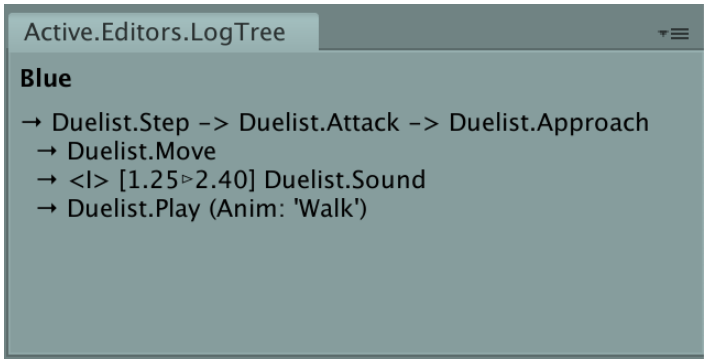
The above syntax ensures best performance. If *log &&* is omitted, string formatting will be performed every time, even in release builds. In contrast, the correct syntax will only format strings when logging is enabled for a particular object.

To avoid errors resulting in logging overheads, you may to define the AL_STRICT tag in your response file (see ***cs.rsp.template***). When this flag is enabled, omitting “log &&” causes a compile error.

3: Also attach log messages using the indexer notation

```
status Engage
=> Eval( STATUS_EXP )[log && $"At {dist:0.#}meters from target"];
```

Consistently annotating your sources produces well formed, hierarchic views of your AIs and controllers.

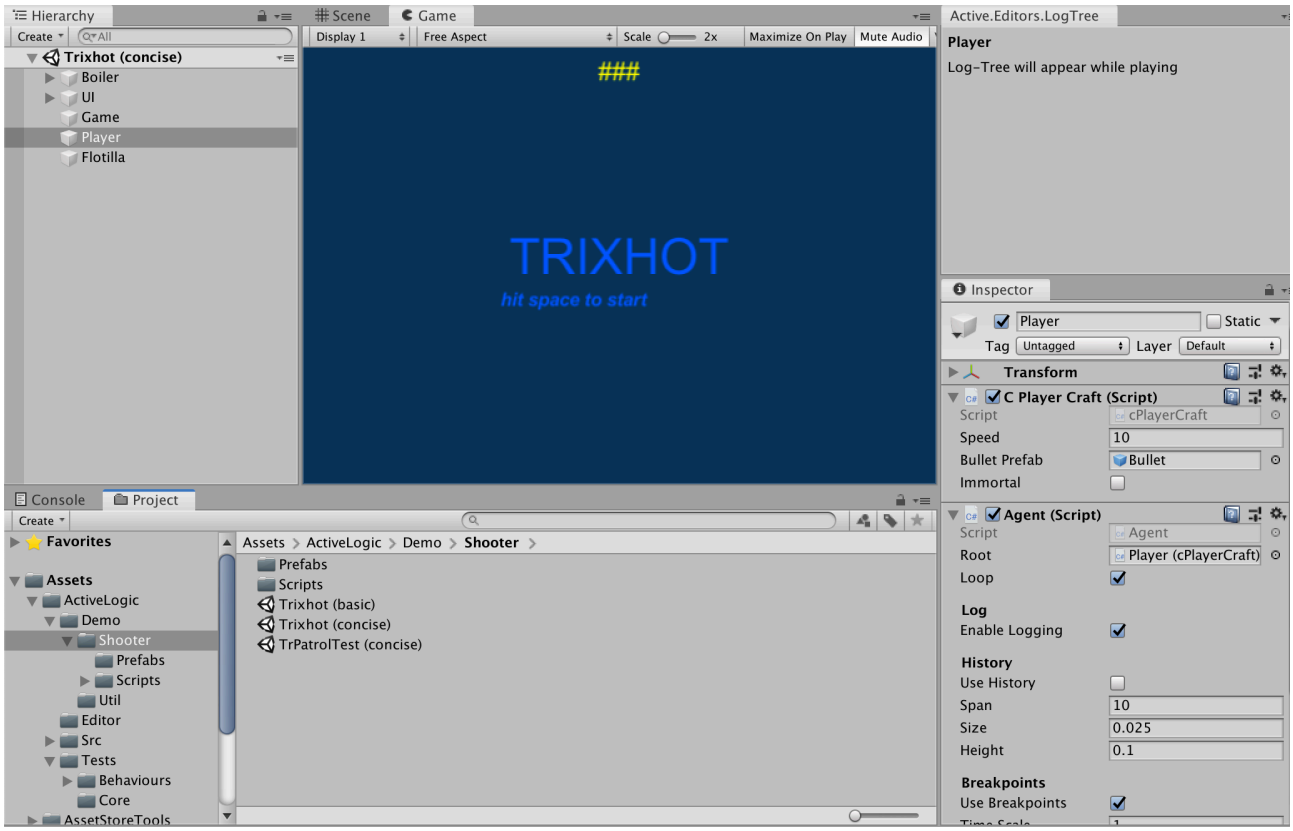


6. Built-in demo

The API ships with a built-in demo, Trixhot. The demo is a minimal but comprehensive shooting game, illustrating the use of Active Logic to implement the following areas:

- Game controller
- User input
- NPC generation
- NPC AI

The demo is under *ActiveLogic* > *Demo* > *Shooter*



Two implementations are provided:

- The *basic* demo is written in a style that is less concise, but possibly more readable. Expression bodied members are avoided throughout. When possible, logic is broken into several lines of code. It is recommended to look at this first, especially for beginners.
- The *concise* demo is written in a compact, modern style. Expression bodied members are used whenever possible.

In addition, the concise demo is designed to (optionally) build using special compilation flags.

Thus you may enable *AL_BEST_PERF* and *AL_THREAD_SAFE* via a response file. In order to do so, rename **csc.rsp.template** to **csc.rsp** and uncomment matching flags.

AL_THREAD_SAFE disables syntax which is not thread safe. In general this is not important because Unity's game loop is single threaded.

AL_BEST_PERF disables syntax which does not yield optimal performance. This is provided as an exploratory tool to help you understand how alternative syntax affects performance. To optimize your application, use the profiler.

Turning *AL_BEST_PERF* on will disable certain features of the API, resulting in decreased readability and maintainability. In contrast, switching only performance intensive segments of your application gives you the best of both worlds (while AL_BEST_PERF is disabled, all syntactic variants are available).

Going Further

In addition to this guide, reference documentation is available online:

https://github.com/active-logic/activelogic-cs/blob/master/Doc/Reference/Overview.md