Behavior Bricks

Programmers Quick Start Guide

Behavior Bricks allows to enrich the built-in Collection with new and customized actions and conditions throught a clean API. This tutorial introduces the Behavior Bricks *internal working*, and put that knowledge into practice throught the development of some *native behaviors* programmed directly using C#. The guide assumes you have some previous experience of programming, specifically using Unity3D, so it will not detail every single line of code.

This tutorial continues the small example created in the Quick Start Guide, where the player moves his avatar in the "environment" (a mere plane) using mouse clicks, and the enemy wanders around and pursues the player when he is near enough. We encourage you to follow that tutorial in the first place but, if you are impatient, its final version (and the starting point for this guide) is available in the Behavior Bricks package, under `Samples\QuickStartGuide\Done` folder. Obviously, you are supposed to have been loaded Behavior Bricks into a new Unity project. Refer to the download instructions in other case.

In this tutorial, we will create a new action that makes the enemy to shoot the player. We will learn about *latent actions* that require more than one game cycle, and how to make them CPU friendly to avoid wasting resources. After that, a new condition is also created, as an example of *perception*. The internal working of the priority selectors is introduced, so, again, the condition becomes CPU friendly.

If you want to experiment with the final version of the demo prior to following the tutorial, you can find it in the `Samples\ProgQuickStartGuide\Done` folder in the Behavior Bricks package.

Creating a native action: shooting

The last behavior asigned to the `Enemy` in the Quick Start Guide was composed of a priority selector that chooses between `MoveToGameObject` (to the `Player`) or just wander around. The desicion is taken depending on whether the condition `IsTargetClose` evaluates that the player is near enough. It is available in the Collection, in `Behavior/QuickStartGuide/DoneEnemyBehavior`.

If you play that scene, you may have noticed that if the `Enemy` reaches the player, he keeps joined to him and slowly orbitating. This is not, surely enough, a nice behavior for a frightening enemy. We would want our enemy to stop moving and shoot the player when the distance between them reaches a threshold. We will do that adding a new child behavior to the priority selector but we need the `Shoot` action in the first place.

Before getting our hands dirty, we should think about the action we want to program. In order to be reusable, it should be parameterized, in the same way as `MoveToGameObject` was parameterized with the target parameter. For the `Shoot` action, we need at least three parameters:

- The game object that represents the `bullet`. The action will clone that object and *shoot* it.
- The `shootPoint` where the `bullet` will be created. Usually, this position will be relative to the `Enemy`.

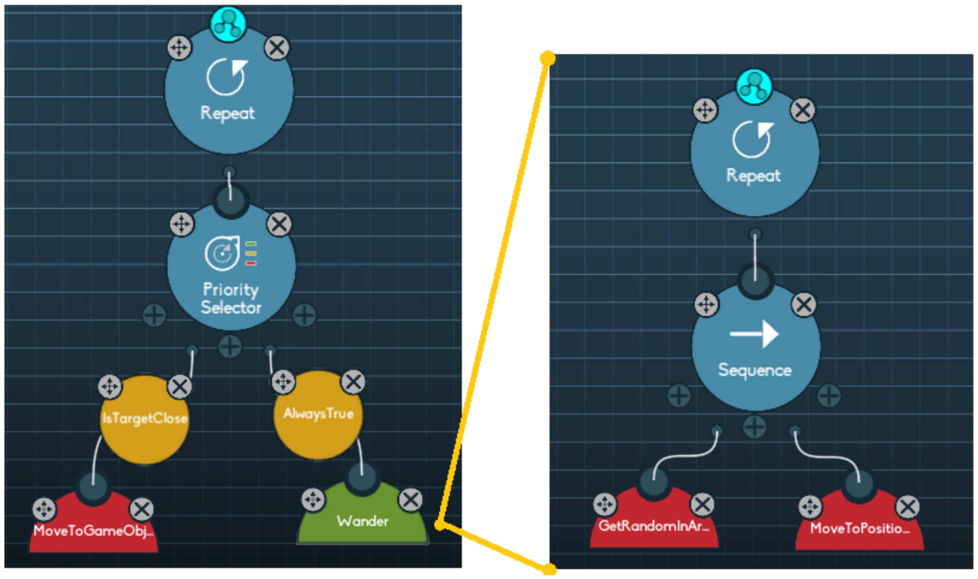


Figure 1: Initial enemy behavior from the quick start guide

- The initial bullet `velocity`. As a convention, the action will shoot the bullet through its *forward* axis.

A last debatable parameter would be the bullet lifetime. Our `Shoot` action could schedule the bullet destruction in the moment of its creation. Instead of that, we will leave that task to the bullet itself.

On the other hand, shooting is an *immediate action:* it finishes *in the same game loop* it starts. `GetRandomInArea`, used in the Quick Start Guide, is another example of immediate action. The opposite are *latent actions*, that last multiple game loops. `MoveToPosition` or `MoveToGameObject` are some of them. When creating behaviors, the existence of these two types is unimportant, but when programming a new action, an eye must be kept on that.

The easier way of programming a new action is to create a new subclass of `BasePrimitiveAction` of the `Pada1.BBCore.Framework` package. The key points are:

- The Behavior Bricks execution engine will periodically call its `OnUpdate()` method.
- The class *must have* an `Action` attribute that determines the name the action will have in the editor. This disconnects the class name and that one shown in the Collection, allowing even hierarchical names in a similar way as Unity allows for the shader assets.
- The class *could have* a `Help` attribute with an action description that will be shown in the editor.
- The class attributes (or properties) that should be available as action parameters in the editor *must have* an `InParam` attribute, specifying the visible name. This allows you both to use even private attributes as editor parameters, and to specify editor names with characters that would be invalid for a C# identifier (such as spaces). Optionally, when the attribute has a basic type, a default value can also be set.

Please note the name difference between the Behavior Bricks `OnUpdate()` method and the common `MonoBehavior::Update()`: this difference is deliberate. In fact, the class `BasePrimitiveAction` *does not* inherit from `MonoBehavior` at all. `OnUpdate()` must, also, return a `TaskStatus` value. This will inform the execution engine if the action has finished its execution or should still be invoked in the next game loop.

After all this background, our new action code should be self-explanatory.

- Create a new C# script and name it `ShootOnce.cs`.

- Open it into your preferred editor and substitute the code:

  ```
  using UnityEngine;

  using Pada1.BBCore;           // Code attributes
  using Pada1.BBCore.Tasks;     // TaskStatus
  using Pada1.BBCore.Framework; // BasePrimitiveAction

  [Action("MyActions/ShootOnce")]
  [Help("Clone a 'bullet' and shoots it throught the Forward axis with the " +
        "specified velocity.")]
  public class ShootOnce : BasePrimitiveAction
  {
      // Define the input parameter "shootPoint".
      [InParam("shootPoint")]
      public Transform shootPoint;

      // Define the input parameter "bullet" (the prefab to be cloned).
      [InParam("bullet")]
      public GameObject bullet;

      // Define the input parameter velocity, and provide a default
      // value of 30.0 when used as CONSTANT in the editor.
      [InParam("velocity", DefaultValue = 30f)]
      public float velocity;

      // Main class method, invoked by the execution engine.
      public override TaskStatus OnUpdate()
      {
          // Instantiate the bullet prefab.
          GameObject newBullet = GameObject.Instantiate(
                                      bullet, shootPoint.position,
                                      shootPoint.rotation * bullet.transform.rotation
                                  ) as GameObject;
          // Give it a velocity
          if (newBullet.GetComponent<Rigidbody>() == null)
              // Safeguard test, altough the rigid body should be provided by the
              // prefab to set its weight.
              newBullet.AddComponent<Rigidbody>();

          newBullet.GetComponent<Rigidbody>().velocity = velocity * shootPoint.forward;
          // The action is completed. We must inform the execution engine.
          return TaskStatus.COMPLETED;

      } // OnUpdate

  } // class ShootOnce
  ```

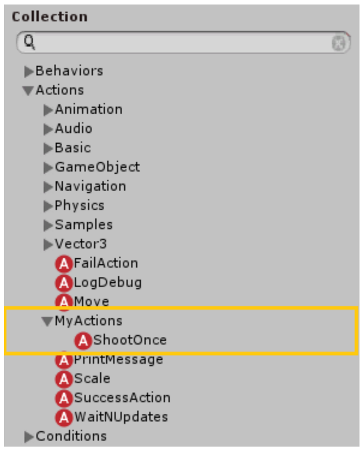
Some things to note:

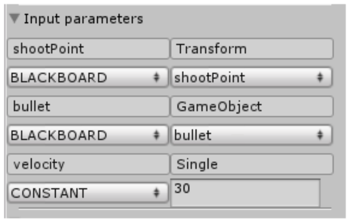
- The class is named `ShootOnce` and it will be refered as such in the Collection, but it will be in the `MyActions` folder as indicated by the `Action` attribute.
- The class inherits from `BasePrimitiveAction` instead of `MonoBehavior`.

- The class fields (`shootPoint`, `bullet` and `velocity`) have all the `InParam` attribute to inform that they should be accesible from the Behavior Bricks editor. The names shown by the editor will be the same, altough we could have decided to use others (for example `shoot point` with a space).
- The `velocity InParam` attribute provides a default value. When the parameter is not set using the editor, a `CONSTANT` with that value will be used.
- The `OnUpdate` method instantiates (clones) the provided bullet prefab, makes some calculation to decide the position and orientation and returns `COMPLETED` so the execution engine knows that it has just finished.

Once the action is available, it's time to use it in the `Enemy` game object behavior. As said before, our starting point is the final version created in the Quick Start Guide, and it is available in the Collection, under the `Behavior/Samples/QuickStartGuide` folder with the name `DoneEnemyBehavior`. You can clone it (selecting the root node and clicking on the `Export` button in the inspector), duplicate it by selecting it and pressing CTRL+D, create it again from scratch or just change it. In any case, open the behavior in the editor.

- Look for the `ShootOnce` action in the Collection, and drop it into the canvas. Note the action parameters in the inspector and, specifically, the default value for the `velocity` parameter.





Input parameters

shootPoint	Transform
BLACKBOARD	shootPoint
bullet	GameObject
BLACKBOARD	bullet
velocity	Single
CONSTANT	30

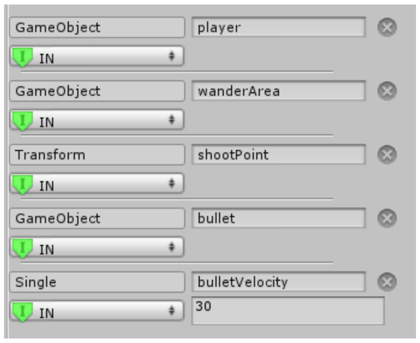
Figure 2: Our action in the Collection, and its parameters in the inspector

- We want the shooting parameters to be available in the inspector of those game objects that use the behavior. Select *blackboard* for all the three input parameters, and create a new field for each one. Use the default names except for `velocity`, where something like `bulletVelocity` is preferred to avoid confusion.

- The `velocity` default value has vanished. Keep in mind that it is only used when the parameter is set *using a constant* in the editor. When using a blackboard parameter, the default value specified in the code attribute is discarded. Fortunately, we can still provide a default value in the behavior. Click on any empty space in the canvas to see the behavior properties in the inspector. You will see the blackboard parameters just created. As `bulletVelocity` has a primitive type, you can provide the 30 again.

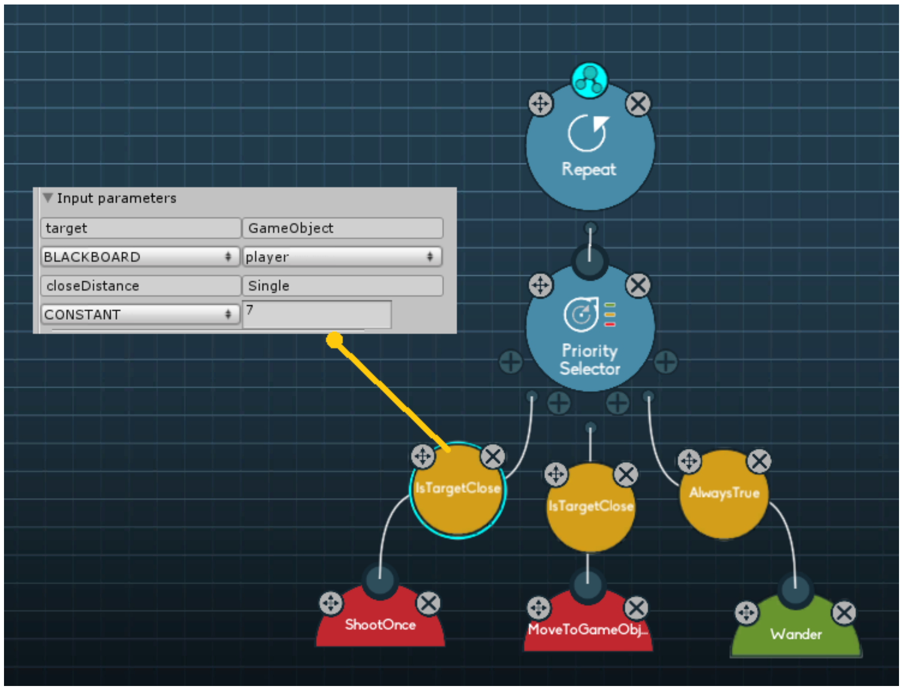
- Connect the priority selector and the `ShootOnce` action using the first handler so shooting will be the first child to be considered (higher priority).

- Drag and drop the `Perception/IsTargetClose` condition into the question mark of the new edge. Choose the `Player` as `target`, and 7 as `closeDistance`. As you might remember, the condition for the `MoveToGameObject` action (pursue the player) is exactly the same but the `closeDistance`, set as 15.



GameObject	player
IN	
GameObject	wanderArea
IN	
Transform	shootPoint
IN	
GameObject	bullet
IN	
Single	bulletVelocity
IN	30

Figure 3: `ShootOnce` blackboard parameters



Input parameters

target	GameObject
BLACKBOARD	player
closeDistance	Single
CONSTANT	7

Figure 4: Complete behavior using the `ShootOnce` action

- Close the editor. We will now create the bullet prefab that will be cloned each time the enemy shoots:
  - Create a new capsule, and call it `Bullet`.
  - Change the scale to (0.3, 0.3, 0.3) and the rotation to (90, 0, 0) so it will lay horizontally.
  - Add it a new C# script and change the `Start()` method so the bullet autodestroys itself after two seconds:

    ```
    void Start() {
        Destroy(gameObject, 2);
    }
    ```

  - Drag and drop the `Bullet` into the Project panel so a new prefab is created based on it, and remove the game object.

- Now we must create the shooting point. We will create a child node in the `Enemy` game object, that will be placed in the middle of his front face. When the enemy moves, so do the shooting point.
  - Create a new empty game object and call it `shootPoint`.
  - Drag it so it will be a child node of the `Enemy`.
  - Set the position to (0, 0, 0.5) so it will be in the middle of the front face.

- Select the `Enemy` again and configure the behavior properties:
  - Drag and drop the bullet *prefab* into the `bullet` parameter.
  - Drag and drop the `shootPoint` child object into the `shootPoint` parameter.

Finally, play the scene. The enemy should pursue the player, as usual, but when he is near enough, it will stop moving and start shooting at him.

Accesing the Game Object from the action: the GOAction class

In its current state, using our `EnemyBehavior` simultaneously in many different enemies can be quite exasperating. Specifically, for each enemy, we should manually set the shooting point parameter to the pertinent child game object. Apart from being boring and error prone, it is arguable if a shoot action for a game object should be able to shoot from a non-child point to the game object itself.

A better approach could be the action to *look for* the child game object. But this would require our `ShootOnce` class to have access to the container game object. Unfortunately, as said before, `BasePrimitiveAction` *is not* a `MonoBehavior` so the game object is unreachable.

If an action needs to have access to the container game object, we can inherit from `BBUnity.Actions.GOAction`. Note the different namespace: `Pada1.BBCore` classes are completely independent from the Unity framework, but `BBUnity` ones are not. Although, due to the execution engine needs, `GOAction` does not inherit from `MonoBehavior` either, it *does provide* a `gameObject` field that will be the container game object at runtime.

- Edit the `ShootOnce` class and change the base class to `BBUnity.Actions.GOAction`. You can import the namespace and remove the `using` clause for `Pada1.BBCore.Framework` that it will not be used anymore.

- Add the code for automatically setting the `shootPoint` parameter in the `OnUpdate` method:

  ```
  public override TaskStatus OnUpdate()
  {
      if (shootPoint == null)
      {
          shootPoint = gameObject.transform.Find("shootPoint");
  ```

```
            if (shootPoint == null)
            {
                Debug.LogWarning("shoot point not specified. ShootOnce will not work " +
                                 "for " + gameObject.name);
                return TaskStatus.FAILED;
            }
        }
        // Instantiate the bullet prefab.
        [ ... ]
    } // OnUpdate
```

- Select the `Enemy` in the scene hierarchy and remove the value for the `shootPoint` behavior parameter.
- Play the scene. Note that the action finds correctly the shoot point.

Note the `TaskStatus.FAILED` value returned if the shoot point is not found. That notifies the execution engine that the action cannot be executed, and the behavior should adapt itself (depending on the internal nodes of the tree). It is arguable whether ending with `FAILED` here is a good idea, or it would be better to just return `COMPLETED` after notifying the problem in the console; after all, this error should not happen, and we want a way to detect it during development. In any case, if your personal preferences are such that you think is better to consider the action has failed, that is the way for doing it.

Before ending this section, just a warning. You could be tempted to always use `GOAction` instead of `BasePrimitiveAction` because it is a richer class. But, `GOAction` instances have a runtime penality during their initialization so `BasePrimitiveAction` should be preferred when possible.

Latent actions: shooting multiple times

Although our action just shoots a bullet, if you have played the scene you will have noticed that the enemy shoots a high-frequency stream of bullets.

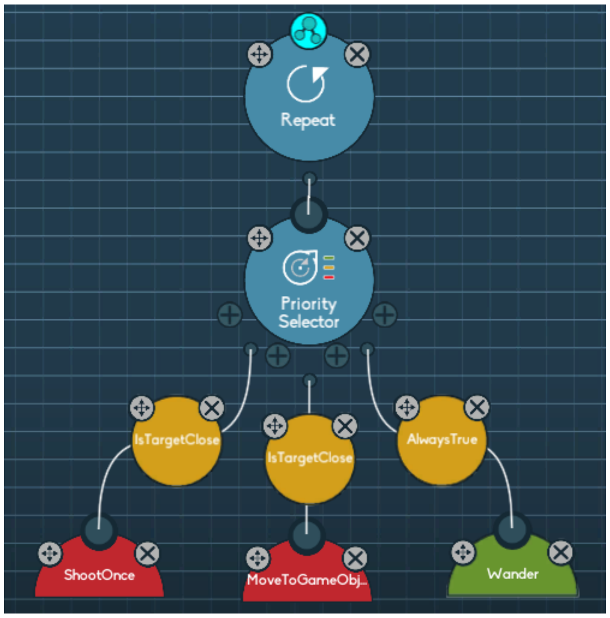


Figure 5: Final behavior using the `ShootOnce` action

Our `ShootOnce` action ends immediately after instantiating the bullet. That causes *the end of the parent* priority selector. Fortunately, the root node is a `Repeat` that relaunches the behavior again in the next game loop. Usually, the condition that determines whether the player is near enough to be shooted will still be true and the enemy shoots again, repeating the complete cycle. The bullet stream becomes maximum, because the enemy is shooting as fast as possible.

We could solve this issue improving the behavior with a sequence and a delay. But for this tutorial, we will create a second action, `Shoot`, that shoots continuosly, adding a configurable delay between shoots. The action will *never* ends.

- Create a new C# script and name it `Shoot.cs`.
- Remove the template code added by Unity, and create a new class that inherits from `ShootOnce`. Add the `Action` and `Help` attributes conveniently. Note that all action parameters from the base class will be available also in the editor for the new action.
- Add a new integer parameter `delay`, that will provide the game cycles count that the action will wait between two consecutive shoots. Set a default value of 30 in the `InParam` attribute.
- Add a private field `elapsed` that will store the game cycles elapsed since the last shoot. This field *should not* have an `InParam` attribute, because it is a private field the action needs for internal use.
- The `OnUpdate()` method should start checking the delay. Usually it will increase the `elapsed` field and exit. Eventually, it will need to instantiate a new bullet, invoking the `ShootOnce::OnUpdate()` parent method. The important point here is the method *must* return, in both cases, `TaskStatus.RUNNING` so the execution engine will know that the action has not finished and must be invoked in the next game cycle. Our `Shoot` becomes a *latent action*.

```
using Pada1.BBCore;           // Code attributes
using Pada1.BBCore.Tasks;     // TaskStatus

[Action("MyActions/Shoot")]
[Help("Periodically clones a 'bullet' and shoots it throught the Forward axis " +
      "with the specified velocity. This action never ends.")]
public class Shoot : ShootOnce
{
    // Define the input parameter delay, with the waited game loops between shoots.
    [InParam("delay", DefaultValue=30)]
    public int delay;

    // Game loops since the last shoot.
    private int elapsed = 0;

    // Main class method, invoked by the execution engine.
    public override TaskStatus OnUpdate()
    {
        if (delay > 0)
        {
            ++elapsed;
            elapsed %= delay;
            if (elapsed != 0)
                return TaskStatus.RUNNING;
        }

        // Do the real shoot.
        base.OnUpdate();
        return TaskStatus.RUNNING;
    } // OnUpdate

} // class Shoot
```

- Open the editor and change the `EnemyBehavior` so it uses the new `Shoot` action.
- Remember to configure the blackboard parameters like we did before in `ShootOnce`.
- Play the scene. The bullets are shooted more slowly.

Returning `RUNNING` could look familiar if you know the concept of *coroutine*. A coroutine is similar to a function that has the ability to pause itself and return the execution back to Unity. When restarted, *Unity coroutines* will continue its execution where they left off; note that the `RUNNING` state in Behavior Bricks works differently because the `OnUpdate()` method is restarted from scratch. If actions were programmed using coroutines, our `Shoot` action would have been:

```
IEnumerator coroutine() {
    while(true) {
        if (delay > 0) {
            while(elapsed != delay) {
                ++elapsed;
                yield return null; // Execution back to Unity
            }
            elapsed = 0;
        }
        base.OnUpdate();
        yield return null;
    } // infinite loop
}
```

Actions life cycle

The keen reader may have realized that the existence of a `shootPoint` will be checked each time a bullet is instantiated. When developing our `ShootOnce` action, it seemed quite reasonable to add the check in the `OnUpdate()` method because it was an immediate action. But, from `Shoot`, that `OnUpdate()` method will be called multiple times, and the check will be repeated. We can improve this situation knowing the *action life cycle*.

Apart from the `OnUpdate()` method, `BasePrimitiveAction` subclasses can override some other methods that will be invoked from the execution engine in the right moments:

- `OnStart()`: used when the action is first started. Subclasses can do whatever initialization they need here.
- `OnEnd()`: called when the action ends with `COMPLETED`.
- `OnFailedEnd()`: called when the action ends with `FAILED`.
- `OnAbort()`: called when the execution engine aborts the execution of the action. This usually occurs if a higher priority action must be started because some condition has changed (in a priority selector). Actions have here the oportunity to graciously stop its execution and release any resources.

As far as our `ShootOnce` action, we would use the `OnStart()` method for seeking the shooting point instead of doing it each `OnUpdate()` invokation:

```
// Be careful! This is the ShootOnce action, not Shoot!

// Initialization method. If not established, we look for the shooting point.
public override void OnStart()
{
    if (shootPoint == null)
    {
        shootPoint = gameObject.transform.Find("shootPoint");
        if (shootPoint == null)
        {
            Debug.LogWarning("shoot point not specified. ShootOnce will not work " +
                             "for " + gameObject.name);
        }
    }
    base.OnStart();
}

// Main class method, invoked by the execution engine.
public override TaskStatus OnUpdate()
{
    if (shootPoint == null)
    {
        return TaskStatus.FAILED;
    }
    [ ... ]
} // OnUpdate
```

Note that `OnStart()` cannot inform the execution engine about the failed initialization, so unfortunately, we must postpone it to the `OnUpdate()`. In honor of the truth, we could solve this issue using some other overridable methods of the `Action` class, but they are out of the scope of this tutorial.

Suspended actions: avoiding polling

In its current state, our `Shoot` action is not user friendly because the `delay` parameter is measured in game loops, instead of a more intuitive unit as seconds. Even worst, it is not CPU friendly either, because it decides whether a new bullet must be instantiate *each game cycle*. If the `delay` parameter is high, the action will be using more and more CPU for doing nothing.

In order to avoid this *polling*, the Behavior Bricks execution model allows an action to *suspend itself*. This means that it has not finished yet, but the execution engine should not invoke its `OnUpdate()` method in the next game cycles. The action assumes the responsability of asking the execution engine to be resumed again in some instant in the future, using the `resume()` method.

The hard point here is, in fact, to resume the action. Some kind of event-driven infrastructure must be available in the game so the action receives a message informing it that it should be restarted. We will see one example of this idea later on. Now, instead of improving our `Shoot` action, we will illustrate the suspension feature with a simpler action that will be useful for our final behavior in the next sections. Specifically, we will implement a low-cost infinite action.

An action that never ends but just keep the behavior doing nothing could be implemented with an `OnUpdate()` method returning `RUNNING`. But that would be a waste of resources because the behavior will received its CPU slice each game cycle. A better aproach is returning `SUSPEND`. The behavior will continue “live” with no end, but it will not consume any CPU at all. Note that this is also better than a behavior with a `Repeat` node with the built-in `SuccessAction` as only child:

```
using UnityEngine;

using Pada1.BBCore;           // Code attributes
using Pada1.BBCore.Tasks;     // TaskStatus
using Pada1.BBCore.Framework; // BasePrimitiveAction

[Action("MyActions/SleepForever")]
[Help("Low-cost infinite action that never ends. It does not consume CPU at all.")]
public class SleepForever : BasePrimitiveAction
{

    // Main class method, invoked by the execution engine.
    public override TaskStatus OnUpdate()
    {
        return TaskStatus.SUSPENDED;

    } // OnUpdate

} // class SleepForever
```

Conditions: perceiving the environment

In the previous sections we were focused on *actions* that constitute the behavior *actuators*. Now we will center in their *sensors*, called *conditions*, that perceive the environment state in order to adapt the behavior. In this section we will add a script to the scene light so it simulates the day-night cycle. The enemy will perceive the ligth, and stop moving (he goes “to sleep”) when the darkness exceeds a threshold.

- Select the `Directional light` game object and add it a new C# script called `DayNightCycle`:

  ```
  using UnityEngine;

  public class DayNightCycle : MonoBehaviour
  {
      // Complete day-night cycle duration (in seconds).
      public float dayDuration = 10.0f;

      // Read-only property that informs if it is currently night time.
      public bool isNight { get; private set; }

      // Private field with the day color. It is set to the initial light color.
      private Color dayColor;

      // Private field with the hard-coded night color.
      private Color nightColor = Color.white * 0.1f;

      void Start()
      {
          dayColor =  GetComponent<Light>().color;
      }

      void Update()
      {
          float lightIntensity = 0.5f +
  ```

```
                            Mathf.Sin(Time.time * 2.0f * Mathf.PI / dayDuration) / 2.0f;
            isNight = (lightIntensity < 0.3);
             GetComponent<Light>().color = Color.Lerp(nightColor, dayColor, lightIntensity);
        }
    } // class DayNightCycle
```

The `lightIntensity` calculation looks quite tricky. It uses the math `sin` function that cycles between -1.0 and 1.0 depending on the parameter. We must scale the parameter so that cycle corresponds with our day duration. We must also adjust the resulting value to convert the [-1.0, 1.0] interval of the `sin` function to [0.0, 1.0] as the subsequent `Lerp` (linear interpolation) requires.

- Select the `Directional light` again, and label it with a new tag called `MainLight`. The condition will look for the light by that tag. Please note that a real game will had a better thought out implementation, using some kind of game state manager, for example. But, for shortness sake, we will get along with this one.

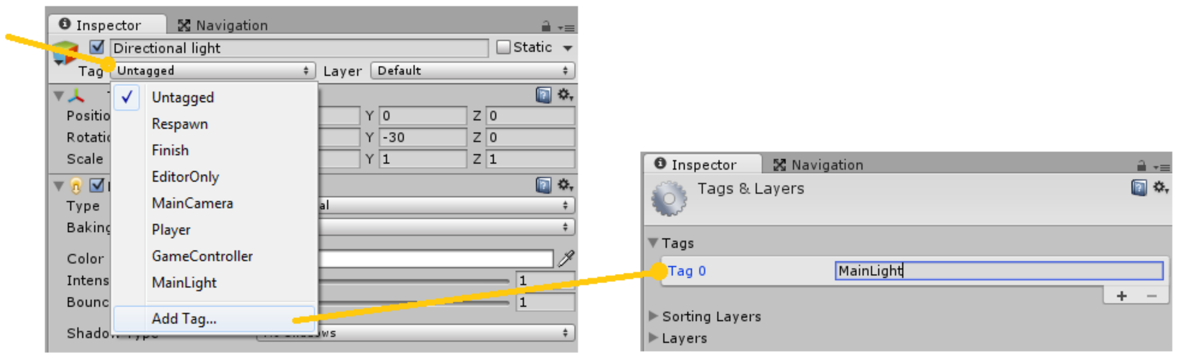


Figure 6: Adding the `MainLight` tag

In Behavior Bricks, conditions are implemented creating a new subclass of `ConditionBase` of the `Pada1.BBCore.Framework` package. It is essentially equal to the known `BasePrimitiveAction` with the exception of some key points:

- The class *must have* a `Condition` attribute that determines the condition name in the Collection. It plays the same role than the `Action` attribute in actions.
- The Behavior Bricks execution engine will invoke to its `bool Check()` method. The condition must return the result of its perception (`true` or `false`). As `OnUpdate()` for actions, this is the main method of the class.

As actions, conditions could have a `Help` attribute, and parameters for the editor (tagged with `InParam` attributes). The executor engine will use the `Check()` returned value to know if the condition ended with success or failure. That result will determine how the behavior will carry on depending on its structure (internal nodes).

Finally, be aware that, as with actions, `ConditionBase` does not inherit from `MonoBehavior` so the container game object is not available. If you need it, you can inherit from `GOCondition` (from `BBUnity.Conditions` package), althouth it has a penality at runtime and, when possible, should be avoided.

Let us now put into practice all our new knowledge.

- Create a new C# script and name it `IsNight.cs`.

- Open it into your preferred editor and substitute the code:

```
using UnityEngine;

using Pada1.BBCore;           // Code attributes
using Pada1.BBCore.Framework; // ConditionBase

[Condition("MyConditions/IsNight")]
[Help("Checks whether it is night. It searches for the first light labeld with " +
      "the 'MainLight' tag, and looks for its DayNightCycle script, returning the" +
      "informed state. If no light is found, false is returned.")]
public class IsNightCondition : ConditionBase
{
    public override bool Check()
    {
        GameObject light = GameObject.FindGameObjectWithTag("MainLight");
        if (light != null)
        {
            DayNightCycle dnc = light.GetComponent<DayNightCycle>();
            if (dnc != null)
                return dnc.isNight;
        }

        return false;
    }
} // class IsNightCondition
```

- Open the `EnemyBehavior` in the editor.
- Add a new action `SleepForever` and join it with the priority selector as its first child.
- Set the condition for this new branch to `IsNightCondition`. As soon as night arrives, the condition will be true and the `SleepForever`, with higher priority, will be chosen.
- Play the scene. Note that the enemy stops wherever was doing when the light is nearly off.

The keen reader may have realized that all the child actions (even the sub-behavior `Wander`) never end, and the enemy just chooses between them depending on the result of the conditions. This means that the root `Repeat` node has become unnecessary and can be removed.

Latent conditions: improving the efficiency

We know, for previous sections, that actions can last more than one game cycle. When possible, it is preferable to suspend the actions so they do not waste CPU cycles. Our `SleepForever` is a nice example: while sleeping, the `Enemy` does not consume resources.

On the other hand, conditions are atomic. They must provide an answer in the `Check()` method as soon as possible, and they cannot ask for extra time.

But when a condition is used in a priority selector or a guard decorator, something unpleasant occurs: the execution engine could need to continually call to its `Check()` method to monitor its state. That will happen with higher priority conditions that are currently false, or with the first condition that is currently true. In our example, imagine the light is on (it is daytime), and the enemy is near enough to the player, so it is shooting at him. The state is summarized in the figure, where the `Shoot` action is hightlighted as the current action.

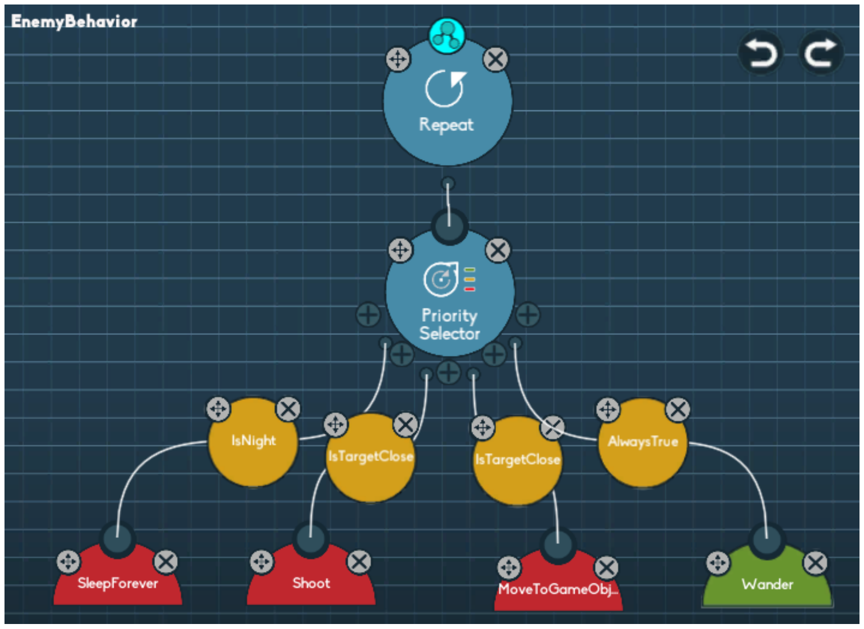


Figure 7: Final enemy behavior

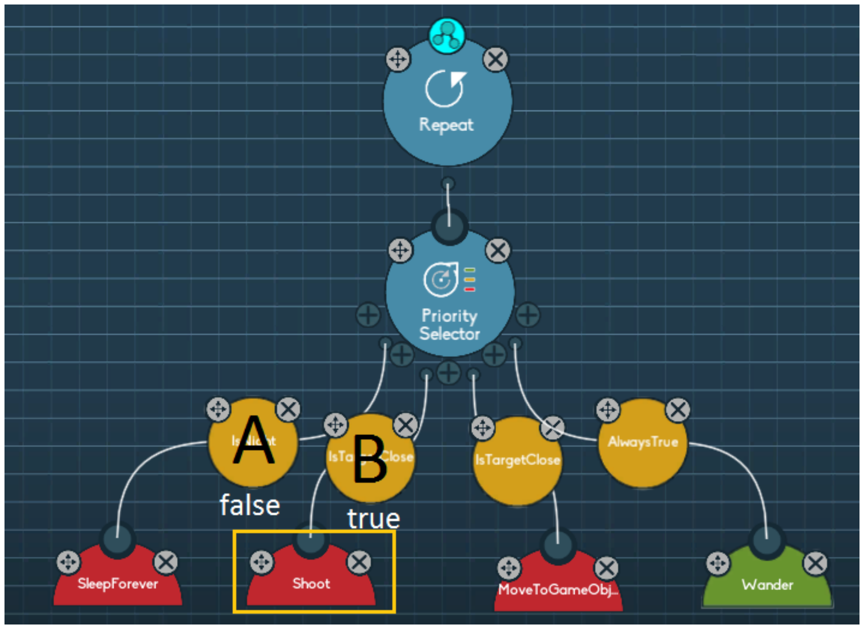


Figure 8: Execution state example

In this situation, the execution state must guarantee *each cycle* that the night has not yet fallen (A condition is still false) *and* the player is still near enough to be shooted (B condition is still true). As far as we currently know, that requires the execution engine to invoke Check() in both conditions. In general, if a priority selector is executing its n-th child, then n conditions must be checked. When night has fallen, our SleepForever does not require CPU at all; but the IsNightCondition is checked every frame in order to know if the sleeping SleepForever should be still used.

Fortunately, the execution engine provides a way to improve the performance of conditions *when they are used in priority selectors* (or in guards) to avoid so much polling. If a condition will be remain with the same result continuously until some event occurs, the condition can inform the execution engine not to bother asking it. Implementation is a bit more complex, but far more CPU friendly.

We will put into practice this feature with our day-night example. But before going deeper into the details, we need the light to notify that the state has changed (the "sun" has just rised or set) so our condition can avoid polling.

We will do that using *events*, the C# native mechanism for a class to provide notifications to clients when something interesting happens. Specifically, our DayNightCycle script will have an OnChanged event that will be trigger when the "sun" rises or sets.

- Add a new event to the DayNightCycle.cs script:

  ```
  // Event raised when sun rises or sets.
  public event System.EventHandler OnChanged;
  ```

- Change the Upate method so the event is triggered accordingly. For simplicity, we are not using the EventArgs parameter. A more elaborated implementation could inform whether night has just fallen, or a new day has come:

  ```
  void Update()
  {
      float lightIntensity = 0.5f +
                             Mathf.Sin(Time.time * 2.0f * Mathf.PI / dayDuration) / 2.0f;
      if (isNight != (lightIntensity < 0.3))
      {
          isNight = !isNight;
          if (OnChanged != null)
              OnChanged(this, System.EventArgs.Empty);
      }
      GetComponent<Light>().color = Color.Lerp(nightColor, dayColor, lightIntensity);
  }
  ```

  Now, any object can receive notifications from the light using:

  ```
  theLight.OnChanged += <myMethod>;
  ```

We are now prepared to come back to our condition. Remember that for avoding polling in priority selectors, the execution engine would like to know when a false condition has become true or viceversa. There are two methods in BaseCondition that help for that:

- TaskStatus MonitorCompleteWhenTrue(): invoked by the execution engine when the condition was previously false. It must return COMPLETED as soon as it has become true.
- TaskStatus MonitorFailWhenFalse(): invoked by the execution engine when the condition was previously true. It must return FAILED as soon as it has become false.

By default, both methods return RUNNING when the result has not changed, denoting that nothing has changed. For clarifying, the default implementation in BaseCondition of both methods is, in essence:

```
public virtual Tasks.TaskStatus MonitorFailWhenFalse()
{
    if (Check())
        return Tasks.TaskStatus.RUNNING;
    else
        return Tasks.TaskStatus.FAILED;
}

public virtual Tasks.TaskStatus MonitorCompleteWhenTrue()
{
    if (!Check())
        return Tasks.TaskStatus.RUNNING;
    else
        return Tasks.TaskStatus.COMPLETED;
}
```

The important point here is that a subclass can override this code and return SUSPEND instead of RUNNING. The execution engine will assume the value of the condition will remain true (for MonitorFailWhenFalse) or false (for MonitorCompleteWhenTrue) in the near future, and it will stop checking the value each game cycle, saving CPU time. As for actions, conditions are responsible of knowing when the condition has changed, and notifying this fact to the execution engine to know about that.

Please note that these two methods, MonitorFailWhenFalse and MonitorCompleteWhenTrue are *only used* for priority selectors and guard decorators. When developing a condition, you should *always* implement the Check() method, that will be used when the condition is used "stand-alone". In fact, the implementation pattern for both Monitor*When* methods consists on invoking the Check() method, and if the condition has not changed (the default implementation would return RUNNING) do whatever it's needed to receive some kind of callback, and return SUSPEND.

With all this in mind, a better implementation of our IsNight condition, using the event we have just added, becomes:

```
using UnityEngine;

using Pada1.BBCore;           // Code attributes
using Pada1.BBCore.Tasks;     // TaskStatus
using Pada1.BBCore.Framework; // ConditionBase

[Condition("MyConditions/IsNight")]
[Help("Checks whether it is night. It searches for the first light labeld with " +
      "the 'MainLight' tag, and looks for its DayNightCycle script, returning the" +
      "informed state. If no light is found, false is returned.")]
public class IsNightCondition : ConditionBase
{

    public override bool Check()
    {
        if (searchLight())
            return light.isNight;
        else
            return false;
```

```
}

// Method invoked by the execution engine when the condition is used in a priority
// selector and its last value was false. It must return COMPLETED when the value
// becomes true. In other case, it can return RUNNING if the method should be
// invoked again in the next game cycle, or SUSPEND if we will be notified of the
// change through any other mechanism.
public override TaskStatus MonitorCompleteWhenTrue()
{
    if (Check())
        return TaskStatus.COMPLETED;
    else
    {
        // Light does not exist, or is "off". We must register ourselves in the
        // light event so we will be notified when the sun rises. In the mean time,
        // we do not need to be called anymore.
        if (light != null)
        {
            light.OnChanged += OnSunrise;
        }
        return TaskStatus.SUSPENDED;
        // We will never awake if light does not exist.
    }
} // MonitorCompleteWhenTrue

// Similar to MonitorCompleteWhenTrue, but used when the last condition value was
// true and the execution engine is checking that it has not become false.
public override TaskStatus MonitorFailWhenFalse()
{
    if (!Check())
        // Light does not exist, or is "off".
        return TaskStatus.FAILED;
    else
    {
        // Light exists, and is "on". We suspend ourselves
        // until sunrise.
        light.OnChanged += OnSunset;
        return TaskStatus.SUSPENDED;
    }
} // MonitorFailWhenFalse

// Method attached to the light event that will be called when the light is "on"
// again. We remove ourselves from the event, and notify the execution engine
// that the new condition value is false (it is not night anymore).
public void OnSunset(object sender, System.EventArgs night)
{
    light.OnChanged -= OnSunset;
    EndMonitorWithFailure();
}

// Similar to OnSunset, but used when we are monitoring the sunrise.
public void OnSunrise(object sender, System.EventArgs e)
{
    light.OnChanged -= OnSunrise;
```

```
        EndMonitorWithSuccess();
    }

    // Search the global light, and stores in the light field. It returns true if
    // the light was found.
    private bool searchLight()
    {
        if (light != null)
            return true;

        GameObject lightGO = GameObject.FindGameObjectWithTag("MainLight");
        if (lightGO == null)
            return false;
        light = lightGO.GetComponent<DayNightCycle>();
        return light != null;
    }

    private DayNightCycle light;

} // class IsNightCondition
```

`MonitorCompleteWhenTrue()` and `MonitorFailWhenFalse()` are symmetric, so we only detail here the first one. Imagine the execution engine starts the priority selector. It calls `Check()` and notices that the result is false (the sun is in the sky). It analyzes the subsequent children and decides which sub-behavior execute (that is not important for our current analysis). In the next cycles, it will not call `Check()` directly, but `MonitorCompleteWhenTrue()`. Our overriden implementation checks the light just once and, if it is still "on", it registers itself in the light event. The light is supposed to trigger the event when its state changes (becomes "off"), so we return `SUSPEND` to the execution engine, freeing valuable resources.

When the sun sets, the light triggers the event, and our `OnSunset` method is called. We unregister from the event and, more important, notify the execution engine that our monitorization must end because we are now `true`. That is the purpose of the `EnMonitorWithFailure()` invokation.

In the next cycle, the execution engine will use `MonitorFailWhenFalse()` to detect when the condition becomes false again. The way it works is similar to `MonitorCompleteWhenTrue()` and should be clear now.

We still have an outstanding issue. Imagine our `IsNight` condition is used in the *second child* of a priority selector. Additionally, suppose the higher priority condition results to be false, and `IsNight` is true, so the execution engine has invoke our `MonitorFailWhenFalse` and we are suspended now, attached to the light event.

Eventually, the first condition could be true. In that moment, it is not important anymore for the execution engine whether `IsNight` changes its value. In fact, it makes not sense if the condition informs the execution about such a change using any of the `EnMonitorWith*()` methods. That means that the `IsNight` condition is neither interested about a light change, but it is still registered in the event.

To solve this issue, the execution engine will call a new condition method, `OnAbort()`, equivalent to that namesake method in `Action`. Regarding `IsNight` condition, all this means that we should implement that method and unregister from any event we could have registered previously:

```
public override void OnAbort()
{
    if (searchLight())
    {
        light.OnChanged -= OnSunrise;
        light.OnChanged -= OnSunset;
    }
    base.OnAbort();
}
```

What's next?

Congratulation! After completing this guide you should have adquired a deep knowledge of the Behavior Brick internals, and should be able to create your own actions and conditions. They constitute the key point in any behavior, so this hability will allow you to create richer NPCs and game mechanics adapted to your own projects. Remember if you have had any problem following this tutorial, you have available the final scene in the `Samples\ProgrammersQuickStartGuide\Done` folder of the Behavior Bricks package.

Now, enjoy your new adquired skills. But don't forget the motto: "two weeks programming can save you from one hour in the library". Before creating your own action or condition, browse the Collection! Behavior Bricks provides a rich built-in set of action and conditions, ready-to-use. Use it at will!