Behavior Bricks

Quick Start Guide

What is Behavior Bricks?

Behavior Bricks is a plug-in for Unity3D that let you design the Artificial Intelligence (AI) for Non-player Characters (NPCs) in a graphical way. Using *behavior trees* as the graphical modelling language, Behavior Bricks provides rich and ready-to-use actions and conditions waiting for you to use them. But, if they are not enough for developing your next awesome game, you can always extend the built-in library using a powerful API. Behavior Bricks helps you in that task too, organizing your home-made behaviors in such a way that reusing them (in other games, or even in more complex behaviors) becomes a child's play.

This quick start guide shows you the basics of Behavior Bricks through the creation of your firsts behaviors; this should be enough for providing the first insights into the behavior editor. The guide assumes you have some previous experience using Unity3D, so it will not detail every single step done with it. You are also supposed to have been loaded Behavior Bricks into a new Unity project. Refer to the download instructions in other case.

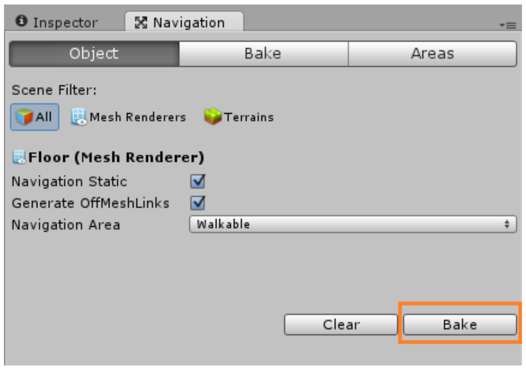
The final scene of the guide is available in the Behavior Bricks package, under `Samples\QuickStartGuide\Done` folder. The behaviors are available in the same folder in the Collection (more about it later).

Creating the scene

In this tutorial we create a small scene with two interactive game objects. The first one represents the player, and is moved around using mouse clicks. The second one is an "enemy" that wanders around, and pursues the player when sees him. For simplicity, all the models of the scene are basic primitives, so no assets are needed.

If you feel bored of setting up the initial scene, you can use the premade `InitialScene` in the `Samples\QuickStartGuide` folder of the Behavior Bricks package (remember to make a copy if you want to keep the original empty version). In other case, you should take the next steps:

- Create a plane (GameObject - Create other - Plane) and rename it to `Floor`. Set the position to (0, 0, 0), and the scale to (5, 1, 5) so it covers a bigger area. Check the `Static` checkbox near the Game Object name in the Inspector.
- Move the main camera to (0, 20, -30), and set the rotation to (45, 0, 0) in order to fit the plane into the view.
- Add a directional light. The default values will be ok.
- Create a sphere for the player. Rename it to `Player`, and place it in (0, 0.5, 0) so it will be over the floor.



Floor (Mesh Renderer)

Navigation Static ✓
Generate OffMeshLinks ✓
Navigation Area Walkable

Figure 1: Creating the navigation mesh

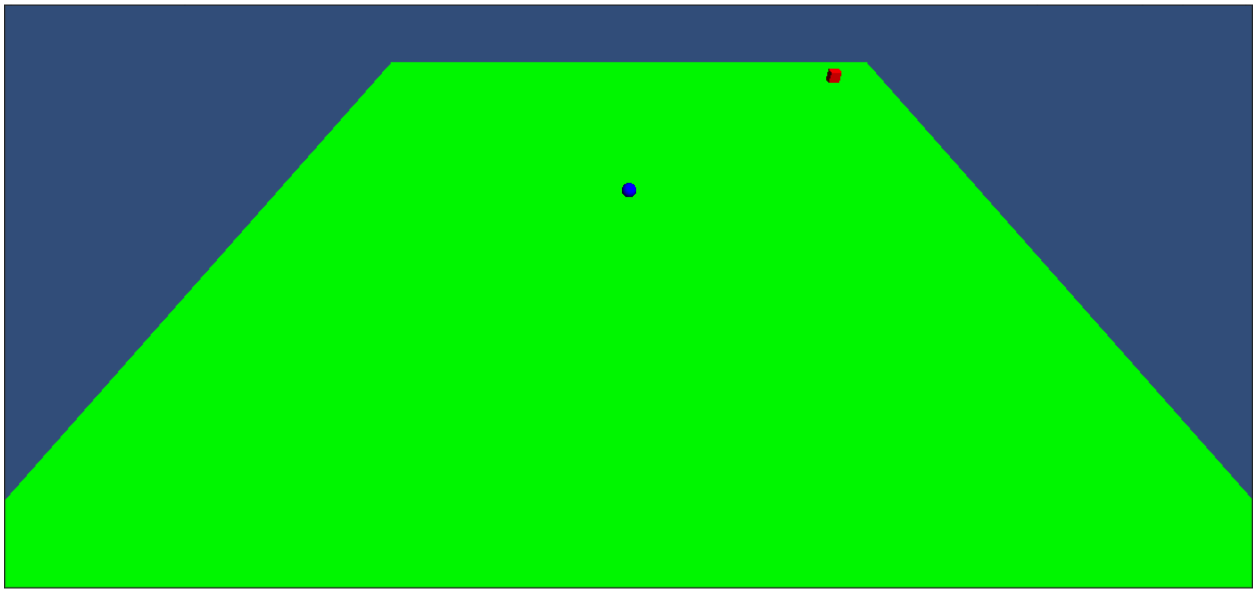


Figure 2: Initial scene

- Create a cube for the enemy. Rename it to `Enemy`, and place it in (20, 0.5, 20). It will be quite far away from the player, near the plane limits.
- Create three new materials, `Green`, `Blue` and `Red` with those flat colors and use them for the floor, player and enemy respectively.
- Create the *navitation mesh* that will be used by both the enemy and the player for pathfinding. Select the floor, go to Window - Navigation and press the *Bake* button. If you have not yet saved your scene, Unity will ask you to do so now.

The scene should be set up now.

Wander: the first enemy behavior

It's time to create our first simple behavior for the `Enemy`. Go to the Behavior Bricks menu (Window - Behavior Bricks), and open the editor using one of the options (standalone or integrated window). That will open the *Behavior bricks editor*. Keep in mind that the window behaves like any other window in Unity3D: it

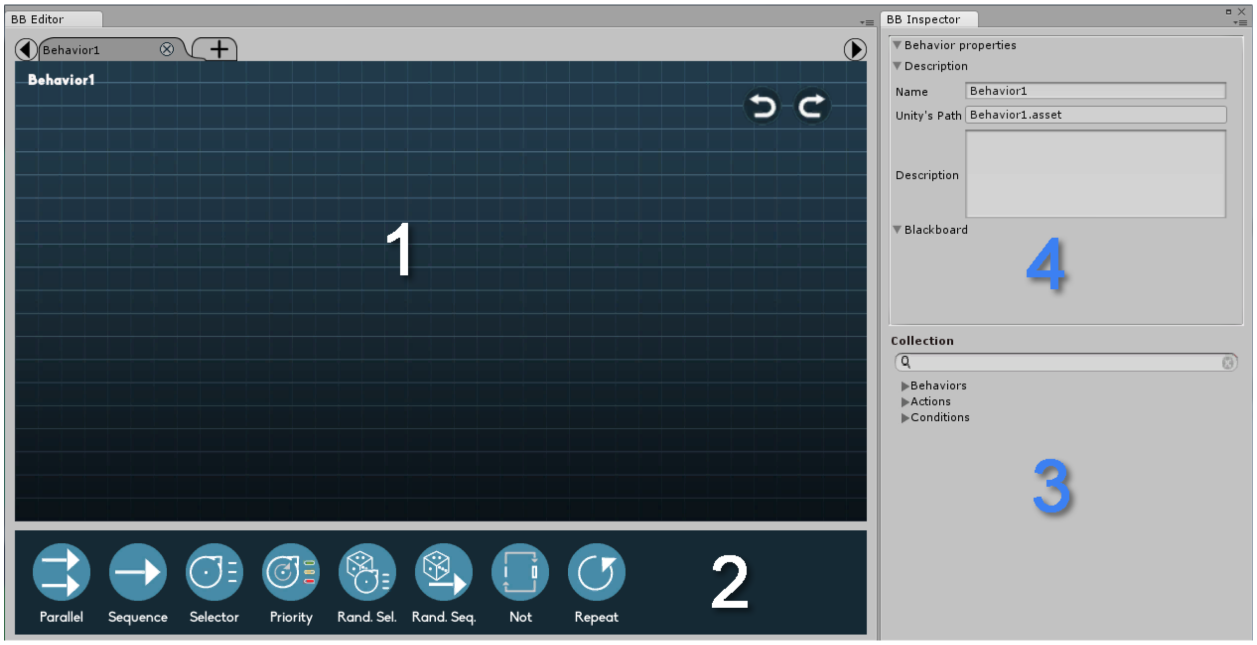


Figure 3: Behavior Bricks Editor

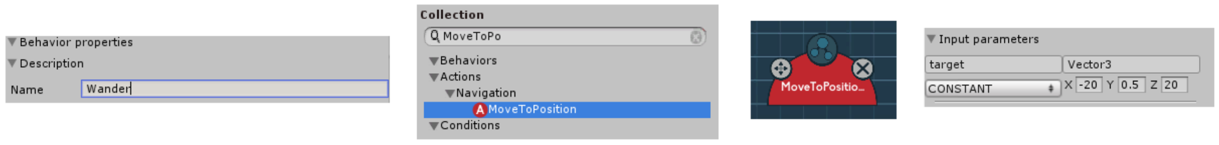
can be dragged to any location so you are able to customize your workspace as desired. Both options in the menu, standalone and integrated editor, are provided as a shorthand for two common configurations.

The editor window is divided in 4 parts:

1. *Behavior graphical editor*: this panel shows a *canvas* with the behavior being modeled (currently empty). Note the upper tabs: multiple behaviors can be opened at the same time. Using the `+` button a new behavior can be easily created.
2. *Available internal nodes*: they can be added to the current behavior by drag-and-drop. We will cover some of them later on.
3. *Properties panel*: it shows the properties of the selected element in the graphical editor.
4. *Behaviors Collection*: it enumerates, in a hierarchical view, all the project behaviors. Initially, only the built-in behaviors will be available, but it will also shown all those ones created by yourself, promoting reusability. In order to simplify the search, a text field is available for filtering the elements by name. Behaviors can be double-clicked to be opened, or dropped into the graphical editor to be used as sub-behaviors in the current one. Browse the hierarchy to have an idea about the built-in behaviors.

Our first step will be make the enemy to go to a fixed position, in (-20, 0.5, 20), moving parallel to the upper floor edge.

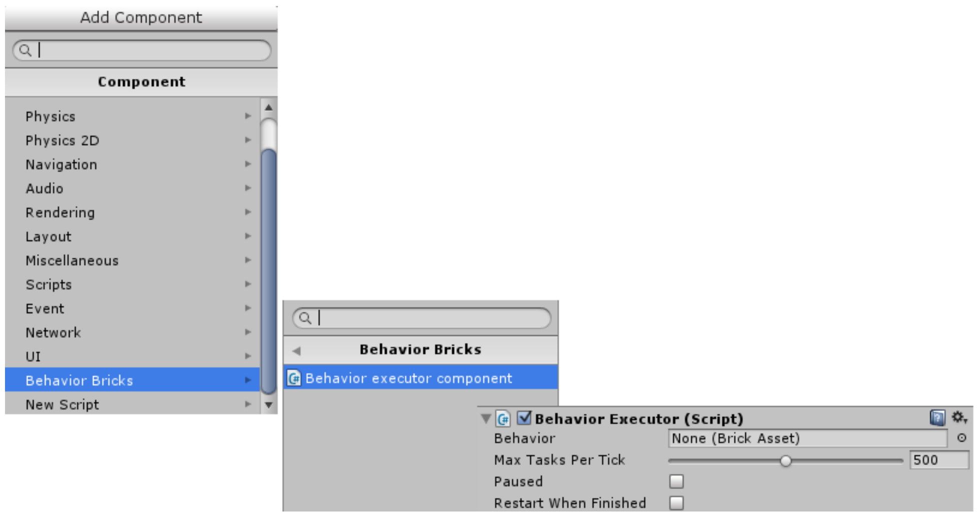
- Create a new Behavior using the + button and keep the default name `New BrickAsset`. Then rename it to `Wander` in the `BB Inspector`. Note the change in the Collection. Although it is not important for this tutorial, you can create a hierarchy using `/` in the name (for example `Basic/Wander`). Also you can notice that the name used in the Inspector and the asset name are independent.
- In the Collection, look for the action `MoveToPosition` (you can use the Search box), and drag it into the graphical editor.


Behavior properties
Description
Name Wander

Collection
MoveToPo
Behaviors
Actions
Navigation
MoveToPosition
Conditions

Input parameters
target Vector3
CONSTANT X -20 Y 0.5 Z 20

Figure 4: Steps for creating the first behavior


Add Component
Component
Physics
Physics 2D
Navigation
Audio
Rendering
Layout
Miscellaneous
Scripts
Event
Network
UI
Behavior Bricks
New Script

Behavior Bricks
Behavior executor component

Behavior Executor (Script)
Behavior None (Brick Asset)
Max Tasks Per Tick 500
Paused
Restart When Finished

Figure 5: Steps for adding the executor component

- Observe the properties of the new node in the Inspector. Specifically, you will find the *input parameters*, that constitute the information the action needs to know exactly how to proceed. `MoveToPosition` needs the target position, a `Vector3`. Set the values to (-20, 0.5, 20).
- All changes are automatically saved in your project so, once done, you can close the Behavior Bricks editor.

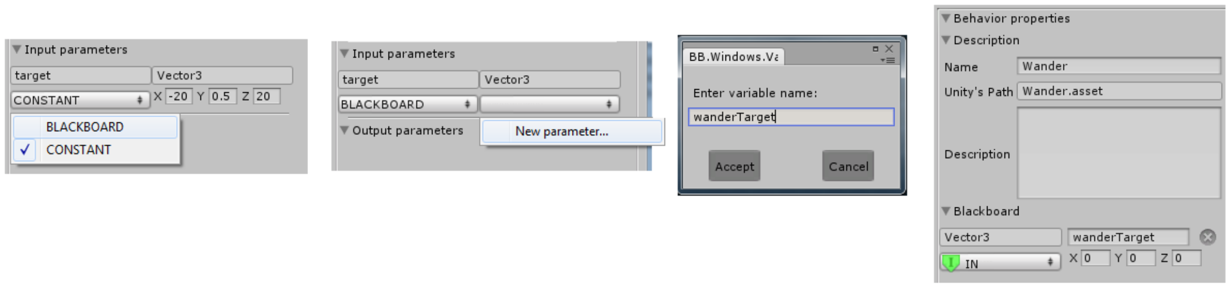
At runtime, behaviors are executed by the *behavior executor*, a component that must be added to any game object that would be controlled by Behavior Bricks. Select the `Enemy` and add the component Behavior Bricks - Behavior Executor.

The new component shown in the Inspector provides a place to drop the behavior to execute. From the asset folders, look for the 'New BrickAsset' behavior, rename it to 'Wander', and drop it in the Behavior variable. Launch your project and you will see the `Enemy` moving to his left. Be aware of the Unity warning "*The Enemy game object does not have a Nav Mesh Agent component to navigate. One with default values has been added*". This is due to a missing component in the `Enemy`. Specifically, `MoveToPosition` action uses the scene nav mesh, which requires a nav mesh agent component. Unity automatically adds it on runtime when missing, and warns about that. You can avoid the warning adding that component beforehand yourself.

Generalizing behaviors: the blackboard

Our behavior moves the `Enemy` to (-20, 0.5, 20), a position hard-wired directly in the behavior. This is usually a bad idea. Keep in mind that we could have *more than one game object* sharing the same behavior and we usually will want to finetune some parameters of the behavior in order to provide diversity.

We can, then, do this better generalizing the behavior using the *blackboard*. The behavior blackboard acts as the *behavior memory*, structured by means of attribute-value pairs. Actions like `MoveToPosition` can read



target: Vector3 — CONSTANT — X -20 Y 0.5 Z 20

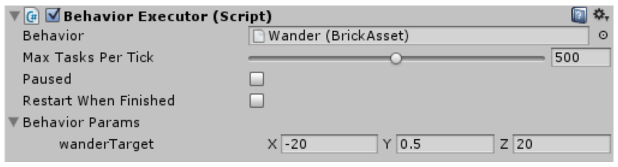
target: Vector3 — BLACKBOARD

Enter variable name: wanderTarget

Name: Wander
Unity's Path: Wander.asset
Description:

Vector3: wanderTarget — IN — X 0 Y 0 Z 0

Figure 6: Steps for creating an entry in the blackboard



Behavior: Wander (BrickAsset)
Max Tasks Per Tick: 500
Paused: ☐
Restart When Finished: ☐
wanderTarget: X -20 Y 0.5 Z 20

Figure 7: `wanderTarget` parameter in the behavior executor component

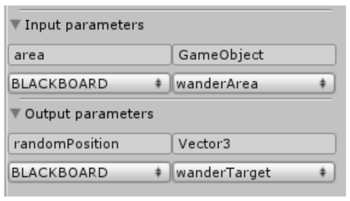
their parameters from there instead of using hard-wired constant values and, even better, the blackboard can be configured externally from the game object Inspector.

- Open the `Wander` behavior in the editor, and select the `MoveToPosition` action.
- In the BB Inspector, go to the *input parameters* and note the combobox where `CONSTANT` is selected by default. Change it to `BLACKBOARD`. The old value (-20, 0.5, 20) will be substituted by a new combo where the blackboard entry must be selected.
- Choose "New parameter" in that combo and name it `wanderTarget` in the dialog box.
- Click over any empty area in the canvas graphical editor in order to see the behavior properties in the Inspector. Note the new *behavior input parameter* with that name. You can change the value for the parameter there.

As seen, the value for the new blackboard field can be globally changed in the behavior properties. But, more interesting, it can also be changed in the behavior executor component properties:

- Close the editor.
- Select the `Enemy` game object if needed.
- Check the behavior executor component. Note the new *behavior param*, `wanderTarget`.
- Change the value to (-20, 0.5, 20) in order to mimic the initial configuration.

Although it is not needed for this tutorial, it is possible to assign the same `Wander` behavior to as many game object as desired, using different `wanderTarget` values for all of them, creating diversity. The value provided for the `wanderTarget` parameter *in the behavior* becomes a *default value* used for those game objects that have not had it changed in their executor component. This acts in a similar way to the values in the *prefabs* and their instances.



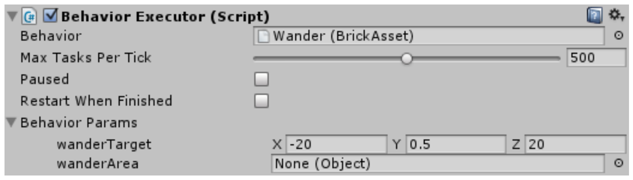
Input parameters

area	GameObject
BLACKBOARD	wanderArea

Output parameters

randomPosition	Vector3
BLACKBOARD	wanderTarget

Figure 8: `GetRandomInArea` properties



Behavior Executor (Script)

Behavior	Wander (BrickAsset)
Max Tasks Per Tick	500
Paused	☐
Restart When Finished	☐

Behavior Params

wanderTarget	X -20 Y 0.5 Z 20
wanderArea	None (Object)

Figure 9: Two parameters in the behavior

Composing behaviors: internal nodes

A wander behavior that just goes to a fixed position is hardly motivating. We will improve it selecting a random position.

- Open again the editor and edit the `Wander` behavior.
- Look for the `GetRandomInArea` action, and drop it into the canvas. When executed, this action chooses a position from an area (specified by means of a Game Object) and write it back into the blackboard.
- Check the *action parameters* in the Inspector. Note the `area` *input* parameter, and the `randomPosition` *output* parameter.
- Create a new blackboard field for the first one and call it `wanderArea`.
- Select the previous `wanderTarget` as the blackboard param where the action will write the selected position.

  The key point here is that the actions `GetRandomInArea` and `MoveToPosition` are now *linked* by the `wanderTarget` blackboard parameter.
- Close the editor and select, if needed, the `Enemy` game object.
- Note the behavior executor component. Now it has *two* parameters, the old `wanderTarget` and the new one `wanderArea`.
- `wanderTarget` is actually an *internal* value that should not manipulated from the outside. It is something like *temporal information* that the behavior needs, but that is unimportant when using the behavior in a game object.
- Open again `Wander` behavior in the editor.
- Click over an empty area of the editor canvas in order to see the behavior properties in the Inspector.
- Note *both* parameters, `wanderTarget` and `wanderArea` labeled as "`IN`".
- Change the type of the `wanderTarget` parameter to "`LOCAL`".
- Note that the `wanderTarget` parameter has vanished from the behavior executor component in the `Enemy`.

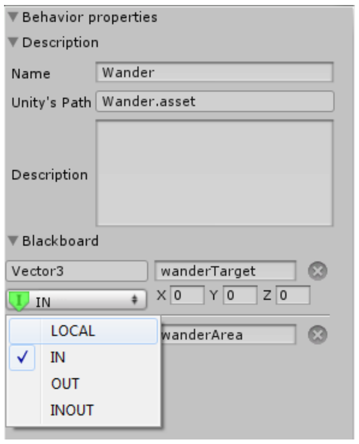

▼ Behavior properties
▼ Description
Name: Wander
Unity's Path: Wander.asset
Description:
▼ Blackboard
Vector3 | wanderTarget
IN | X 0 | Y 0 | Z 0
wanderArea

Figure 10: Changing the type of a blackboard parameter

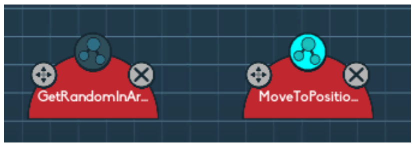


Figure 11: Two actions, one of them the tree root

In this moment we have two actions in our behavior but what of them will be executed earlier? Obviously, we want the `GetRandomInArea` to be executed before `MoveToPosition`, but if you observe both actions, you will note the different color in the upper-middle circle.

The action *with the highlighted circle* (`MoveToPosition`) will be executed in the first place. In fact, `GetRandomInArea` will *not be* executed at all. In order to create a behavior with *two* actions, we need to *composed them*: our behavior becomes *a tree*.

- Drag-and-drop the `Sequence` node from the bottom area of the editor into the behavior canvas. The concrete position is unimportant: you can move it afterwards. Arrange the nodes in a way similar to the figure. A `Sequence` is an internal node that executes *its children* in order.
- Drag and drop the bottom *handle* in the `Sequence` into the top circle in the `GetRandomInArea`. This will create an *edge* relating both *nodes*. Observe that two new handles appear in both sides of the edge end in the `Sequence` node. Those can be used to create new edges keeping an eye on the *order*. An edge can be easily removed dragging any of its ends and dropping it into any empty space in the canvas.
- Create a new edge from the `Sequence` into `MoveToPosition`. Observe that the hightlighted circle moves to the `Sequence` node. Our behavior *is now a valid tree*. The executor will start with the sequence, that will, in turn, execute the `GetRandomInArea` action and then the `MoveToPosition` one, following a left-to-right order.
- The sequence execute its children *once*. We want the enemy to continually wander around, so when it reaches the random position, another one should be selected and the enemy should go there.
- Insert a `Repeat` node into the canvas. This internal node executes its child behavior again and again.
- Create an edge from the `Repeat` node to the `Sequence`. Observe how the hightlighted circle readjusts itself.
- Close the editor. Remember that behaviors are automatically saved.

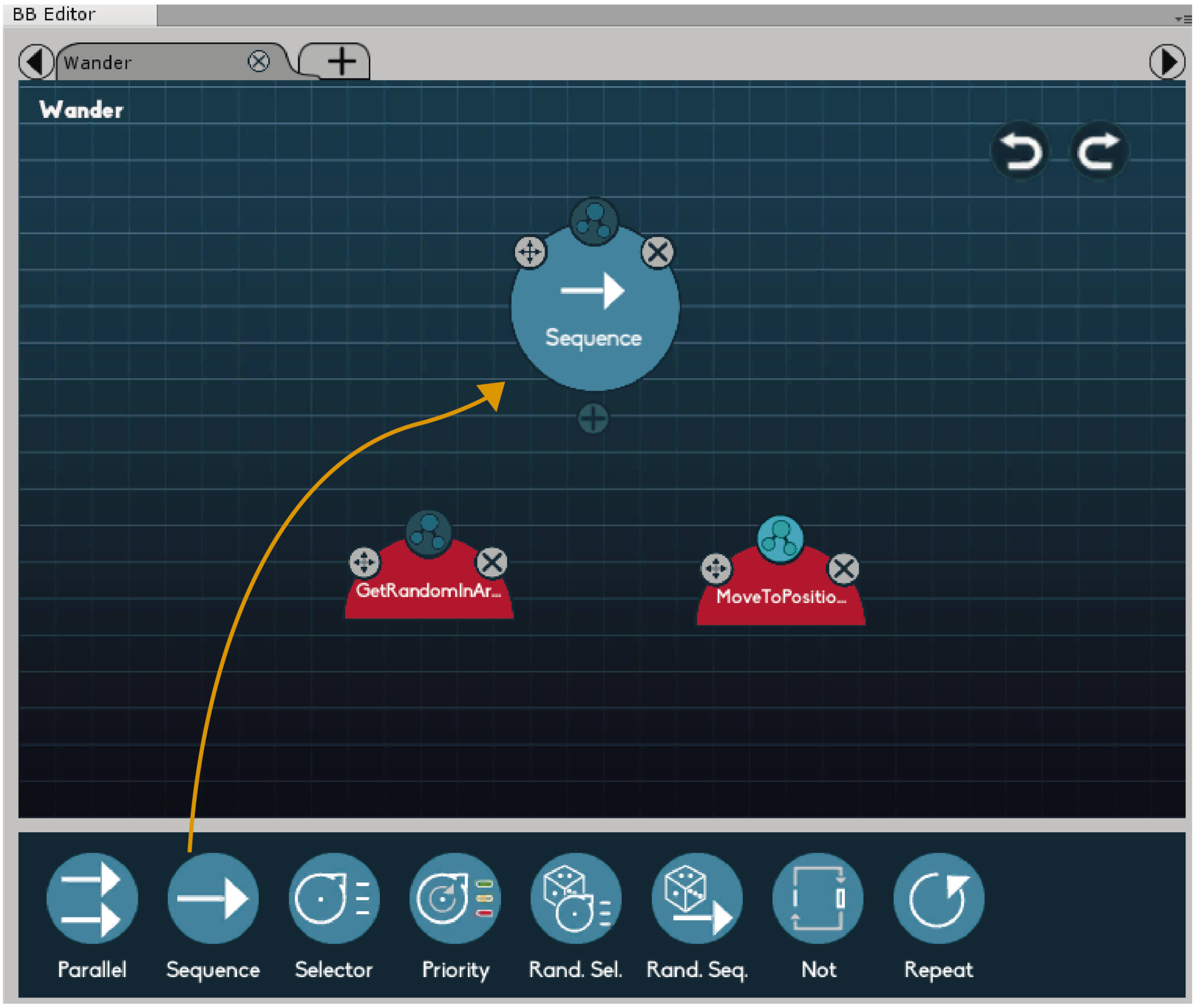


Figure 12: Adding a `Sequence`

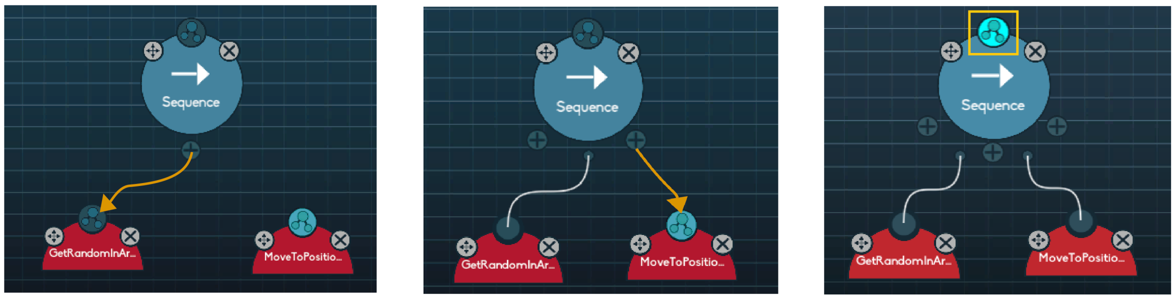


Figure 13: Steps for creating the tree

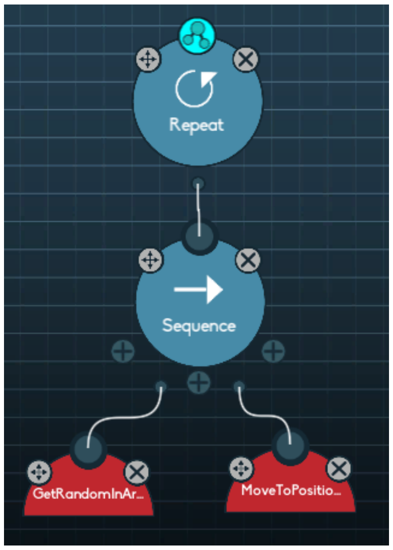


Figure 14: Complete wander behavior

- Select, if needed, the `Enemy` game object and observe the behavior executor parameters. Remember that the `wanderArea` is still unestablished so the `GetRandomInArea` does not know the area where the position should be chosen from. Specifically, `GetRandomInArea` requires a game object *with a box or sphere collider*.
- Select the `Floor` game object, and add it a Box collider. Set (10, 0, 10) as its size.
- Select the `Enemy` game object, and drag and drop the `Floor` into the `wanderArea` parameter.
- Play the scene. Observe the wander behavior of the enemy. It will continually move around to random positions.

The final version of this `Wander` behavior is available in `Behaviors/Samples/QuickStartGuide` Collection folder.

Conditions and perception: nodes that fail

Although Behavior Bricks is mainly designed for creating the NPC and tactic or strategic AI, it can also be used for creating any other game logic, as menu control, HUD movement or even dynamic music manipulation. In this section we will create a behavior *for the player*, that will let us introduce new actions and our first condition.

We want the player to move around using the mouse. This requires three steps:

- Detect the mouse click. This provides us with a coordinate in *screen space* (2D).
- Convert the click position into a 3D coordinate using a *raycast* to detect the object under the mouse.
- Ask the player avatar to move to that position.

When the user clicks the screen, his avatar will move to that position, so we need to create a behavior that *feels* the environment and acts according to it. This *perception* is managed through *conditions*.

Conditions are nodes that, when reached, end immediately and inform their parent nodes whether they were evaluated as `true` or `false`. In behavior jargon, when a node returns `false` it is said to *fail*, and to *succeed* if returns `true`. The distinctive feature of the internal nodes (`Sequence`'s and `Repeat's` are the only ones we have used so far) is how they react to the values returned by their children conditions (if they *fail* or *succeed*).

In this section we will use the built-in `CheckMouseButton` condition that test if the user has just pressed one of the mouse buttons and ends with success in that case.

- Open the editor window and click the `+` sign near the tabs, in the top of the behavior canvas.
- Rename the new created behavior to `ClickAndGo`.
- Add a `CheckMouseButton` node into it (remember the use of the search box). Note that this node is under the `Condition` branch and it has a different color.
- Check its parameters. Confirm that the `button` is set to `Left`, so the condition will check if the user has pressed the left mouse button.
- Add a `FromMouseToWorld` action. This action convert the current mouse position from screen coordinates to world coordinates. It has four parameters:
  - `camera`: camera used for the translation. Add a new blackboard field and keep the default name, `camera`.
  - `mask`: layer mask used for filtering the raycasting. Add a new blackboard field and keep the default name.
  - `selectedGameObject`: an *output* parameter where the action will write back the game object under the mouse, if any. We are not interested on it in this behavior, so we will keep it unassigned.
  - `selectedPosition`: *output* parameter with the 3D position of the mouse relative to the chosen camera. Add a new blackboard field and keep the default name.
- Add a `MoveToPosition`. Change its `target` parameter to the `selectedPosition` field in the blackboard.
- Click into any empty space in the behavior canvas to see the behavior properties in the Inspector. Change the `selectedPosition` field type from `OUT` to `LOCAL` because it is not important for the game object.
- Add a `Sequence` and make those three nodes its children.
- Add a `Repeat` node so the sequence loops again and again and the user can move around to different positions.
- Close the editor and add a Behavior executor component in the `Player` game object. Attach it the `ClickAndGo` behavior.
- Configure the `camera` parameter with the `Main Camera`, and set the `mask` to `Everything`, so the screen-to-world transformation is done using the camera view, and the raycasting considers all objects in the scene.
- To avoid the warning on runtime, add the nav mesh agent component to the `Player`.
- Play the scene and click around and note that now the player moves to the selected positions.

It is worth mentioning that while doing the behavior, we have completely ignored the fact that the `CheckMouseButton` condition returns a value. When a child condition succeeds (returns `true`), `Sequence`'s will continue executing their next child, if any, with no delay. But if a condition fails (returns `false`), they will end immediately and fail themselves. As you might expect, `Sequence`'s succeed when they have executed all their children and none has failed (returned `false`).

For our example, it is also important how `Repeat` nodes react to conditions.

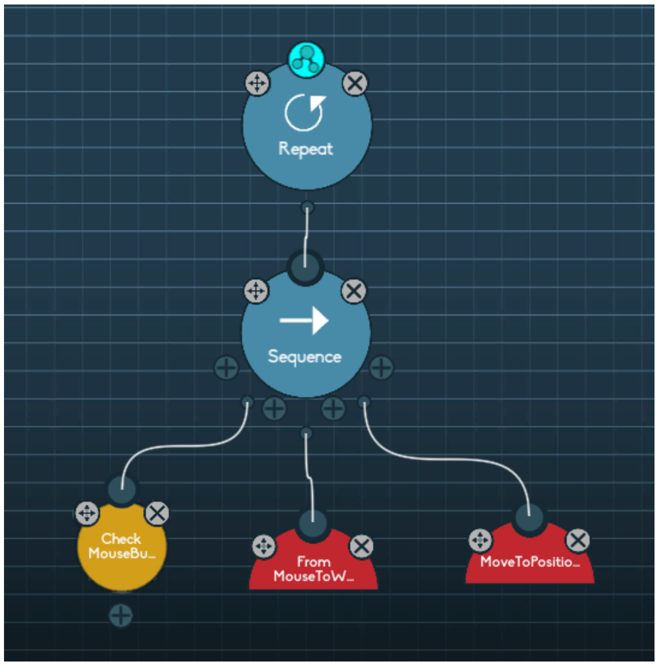


Figure 15: `ClickAndGo` behavior

When playing the scene, is likely that the user does not press the left mouse button in the very first game loop. So the `CheckMouseButton` will end with failure. That will cause the `Sequence` also fails, and so the `Repeat`. The `Player` behavior will then end inmediately (with failure) and the behavior executor will not have anything to do afterwards.

Fortunately, we can configure the `Repeat` node *policy*.

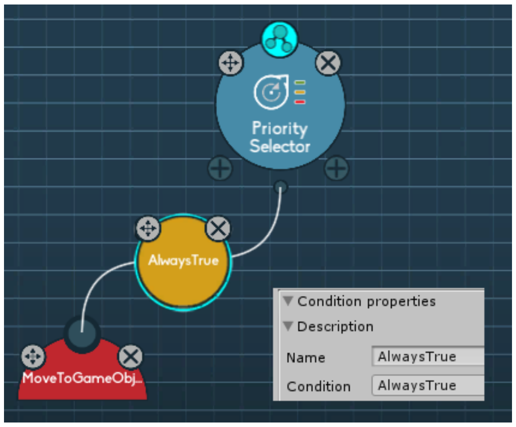
- Open the `ClickAndGo` behavior in the editor.
- Select the `Repeat` node.
- Observe the Inspector and note the policy combo box. By default, it is set to "Continue when child fails". This will force the `Repeat` node to retry its child even when it has ended with failure.

Just in case you are wondering, the `Sequence` node *has not* a configurable policy similar to that one in `Repeat`. As said before, the way sequences react to children failures is in fact the decisive feature that become them in sequences in the first place. If you need an internal node that *tries* the next child when the previous one fails, then you would want to use a `Selector`. Selectors end as soon as one child ends with success, and execute the next child when the preceding one has failed.

Currently, both the enemy and the player moves at the same speed.

- Select the `Player` game object.
- In the nav mesh agent component, set the speed to 7.0 so it will move faster. This will allow to run away from the enemy in the next sections.

You can find the final behavior under `Behaviors/Samples/QuickStartGuide` folder in the Collection with the name `DoneClickAndGo`.



▼ Condition properties
▼ Description
Name AlwaysTrue
Condition AlwaysTrue

Figure 16: Default priority condition

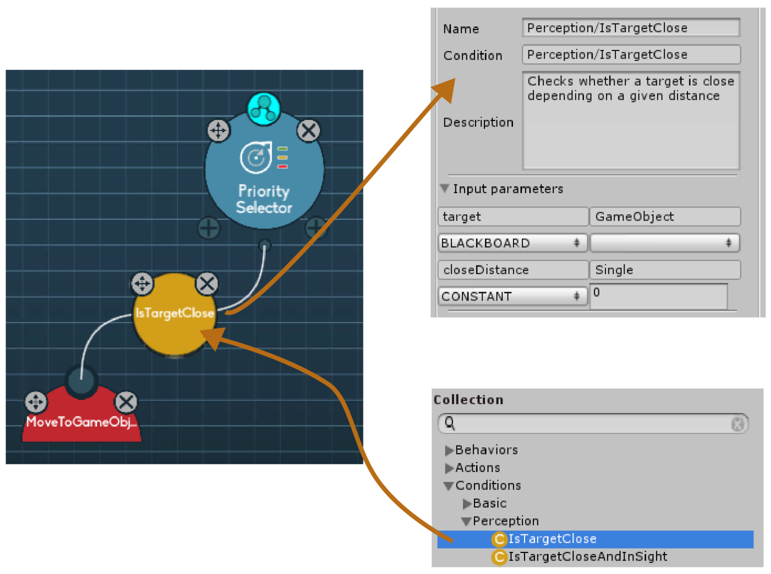
Changing your mind with priority selectors: wander and pursuing

Now that the player can move around, the enemy seems quite dump because he moves around completely ignoring the player. In this section we will create a new enemy behavior that will pursue the player when he is "seen", and will come back to a home (fixed) position in other case.

Note that, in fact, this behavior *includes other two*. The prefered behavior is pursuing the player. But this is only possible if the player is near enough to be seen. In other case, the enemy will execute the fallback behavior consisting on moving to the home position. As soon as the player falls in the enemy field of influence, that movement should be aborted and the preferred behavior started.

This composition of behaviors with *different priorities* is achieved through the `Priority` internal node. It is a *selectors* in the sense that as soon as one of its children ends with success, it also ends. But each child has a *condition* that must be true in order to launch it. And, more important, when a child is being executed, *all* the conditions of preceding sibling nodes are evaluated and, if any of them becomes true, the less priority behavior is inmediately aborted and that one is started.

- Create a new behavior and name it `EnemyBehavior`.
- Drag and drop a `Priority` node into the canvas.
- Add an action `MoveToGameObject`. It is similar to the known `MoveToPosition` but the parameter will be a game object instead of a concrete 3D position.
  - Add a new blackboard field for the `target` parameter and name it `player`.
- Create an edge from the `Priority` to the `MoveToGameObject`. Note the yellow node in the middle of the edge.
- Select the yellow condition node. The Inspector shows the condition that must be true in order to trigger the child behavior. By default, it will be `AlwaysTrue`, useful for the fallback behavior.
- Look for the `IsTargetClose` condition in the Collection, and drag it inside the node directly from the collection. Don't drag it to the canvas and then move it.
- Note the change in the Inspector.
- `IsTargetClose` has two parameters:
  - Set the `target` parameter to the `player` blackboard field.
  - Set 15 as the close distance.



Name: Perception/IsTargetClose
Condition: Perception/IsTargetClose
Description: Checks whether a target is close depending on a given distance

Input parameters

target	GameObject
BLACKBOARD	
closeDistance	Single
CONSTANT	0

Collection
- Behaviors
- Actions
- Conditions
  - Basic
  - Perception
    - IsTargetClose
    - IsTargetCloseAndInSight

Figure 17: Setting a new condition in a priority child

- Add a new `MoveToPosition` node and make it the second child of the priority, becoming the fallback behavior. Set the `target` parameter to the constant value (-20, 0.5, 0).
- Close the editor and set the new behavior to the enemy, substituting the previous one, `Wander`. Note that the old one will not be removed from your behavior Collection.
- Set the `player` parameter to the `Player` game object.
- Play the scene. The enemy will move to his home position (in the center of the left floor edge). In the middle of his path, the player will enter in his influence radius and the more priority action `MoveToGameObject` will be triggered.
- Run away from the enemy using the mouse.
- Note that, once far enough, the enemy stops pursuing the player and go to his home position.

Once there, the enemy will not pursue the player any longer. A `Priority selector` ends when their children end, so the behavior finishes when the enemy reaches its home position. Make the `Priority` child of a new `Repeat` node, and play the scene again. Now the enemy will pursue the player even when he is in his home position.

Composing bricks: reusing behaviors

Behavior bricks promotes behavior reusability not only because behaviors can be used simultaneosly in many game objects, but because behaviors can be used as *black boxes* in other ones. In that way, the *Collection* becomes the conerstone of the reusability in Behavior Bricks.

- Open the `EnemyBehavior` in the editor.

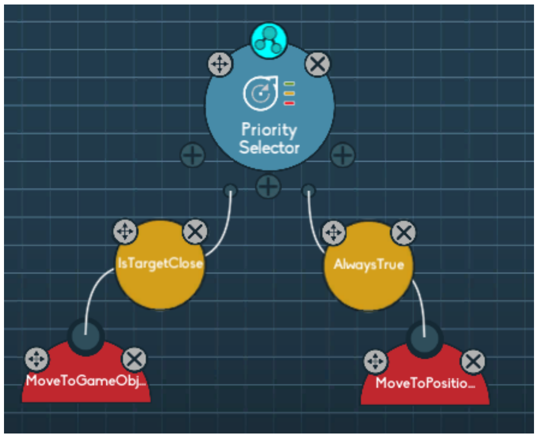


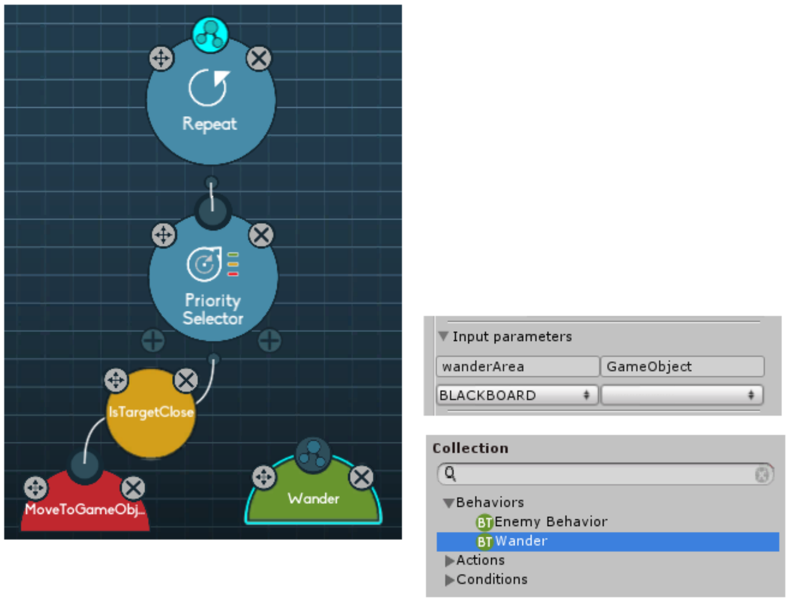
Figure 18: Priority selector with two children

- Remove the `MoveToPosition` action clicking on the `x` sign in the top-right corner of the node.
- Look for the `Wander` behavior in the Collection, and *drop it* into the canvas. Note that this is the behavior we made in a previous section. Keep in mind that if you double clicked the behavior in the Collection, you will open it in a new tab. Dropping it into the `EnemyBehavior` canvas *incorporates it* as any other action. The `EnemyBehavior` will take advantage of any subsequent improvement done to the original `Wander` behavior.
- Note the `wanderArea` in the behavior parameters. This was the only `IN` blackboard parameter we created. When we used the behavior in the executor component, the parameter had to be set in the game object Inspector. When used in a bigger behavior, it must be configured directly in the editor of the using behavior.
- In this case, we want to "publish" this parameter towards the `EnemyBehavior` users. Assign it a new blackboard field and keep the default name, `wanderArea`.
- Click in any empty area on the editor canvas to see the behavior properties in the Inspector. Note both `IN` parameters, `player` and `wanderArea`.
- Make the `Wander` node the fallback child of the `Priority`.
- Remember to assign the Floor GameObject to the wanderArea parameter in the Unity Inspector.
- Play the scene. Now the `Enemy` wander around to random places. When he perceives the player, he abandons his wander behavior and starts pursuing. If the player runs away from him, then he starts wandering again.

As always, you can find the final behavior in the Collection, under `Behaviors/Samples/QuickStartGuide` folder.

What's next?

Congratulation! You've just finished the first Behavior Bricks tutorial. You have learned the basics of the editor, and the main concepts: actions, conditions, blackboard, internal nodes, and reusability. If you want to test your newly acquired knowledge, you can, for example, improve the player behavior. Currently, once a position has been selected, it cannot be changed until the player avatar has reached it. It would be great if the player could select a different position using the mouse to overwrite the previous one. Using a priority selector it is easier that it sounds! And it you need some inspiration, you can always have a look at the behavior in the Collection. Finally, remember if you have had any problem following this tutorial, you have available the final scene in the `Samples\QuickStartGuide\Done` folder of the Behavior Bricks package.



Input parameters

wanderArea	GameObject
BLACKBOARD	

Collection

- Behaviors
  - Enemy Behavior
  - Wander
- Actions
- Conditions

Figure 19: Adding a behavior as any other action