Overview

Panda BT is a Behaviour Tree scripting framework for Unity.

The behaviour of a `GameObject` is defined by writing BT scripts, using a minimalist built-in language to describe the Behaviour Tree structure and its execution flow. A Behaviour Tree consists of a hierarchy of nodes where the deepest node are tasks, which are implemented in C#.

The Behaviour Tree is compactly display as a colour coded text directly within the Inspector. A Life View allows visualisation and debugging of the Behaviour Tree at run-time, providing relevant information at a glance.

Behaviour Tree in a nutshell

Behaviour Tree is an AI technique used in modern video games where the behaviour of an entity is described as a tree composed of an hierarchy of actions. As a whole, the tree is a single action recursively composed of sub-actions. For instances the action "Frying an egg" consists of the sub-actions: "Warming up a pan", "Breaking the egg", "Dump the egg content into the pan" and "Wait for 5 minutes". The sub-action "Warming up a pan" consists of "Take a pan", "Place it on the fire pit", "Lit the fire pit" and "Wait for 2 minutes" and so on... which is intuitive to define. Also, due to this hierarchical nature, Behaviour Tree is an ideal tool for building **scalable logic**.

A behaviour tree is composed of tasks. A task is a process that takes some among of time, or several ticks, to complete. The tick signal is propagated from the root recursively to the leaves. Once a task is completed, it returns a status which can be either *succeeded* or *failed*. Note that theses are **practical** statuses; they do not necessarily indicate whether an error as occurred or not. At any time a behaviour can be in the following state:

- Ready: The behaviour is not active.
- Running: The behaviour has been activated and it is processing.
- Succeeded: The behaviour has completed in success.
- Failed: The behaviour has completed in failure.

The status of a parent node depends on the statuses of its children. Depending on its type, a parent node has its own way to run its children and returns a status.

The overall logic of the tree is implicitly defined by its structure, which makes it very flexible to modify; you don't need to specify the transition from a node to another since it is defined by its position in the tree. Therefore you can easily move, delete or insert a node anywhere in the tree. Editing a behaviour tree is similar to writing a computer program: you don't specify the transitions from an instruction to another, since the transition is implied by the syntax of the programming language. This **programming analogy** is the based idea that inspired the development of Panda BT.

If you want to read more about Behaviour Tree, you can read theses online resources.

Installation

Download the Panda BT package from the Unity Asset Store then import it into your project. Once the package has been imported, a new directory named PandaBehaviour will be present at the root of the Assets folder. The PandaBehaviour folder contains the core of Panda BT and some examples to get started.

Panda BT (Script) Component

The Panda BT component is a container that holds BT scripts assigned as text files.

You can add this component to a `GameObject` from the Inspector by clicking on the "`Add Component`" button then by selecting `Scripts > Panda > Panda BT`.

in order to work, a Panda BT requires one or more BT scripts to be assigned to it. A BT script is a `TextAsset` describing trees which you can edit using your favourite text editor.

During the execution of the Behaviour Tree, a Live View of the execution is available in the Inspector, which provides a useful insight about the current Behaviour Tree state, as well as it's execution flow and other debugging information

Component properties

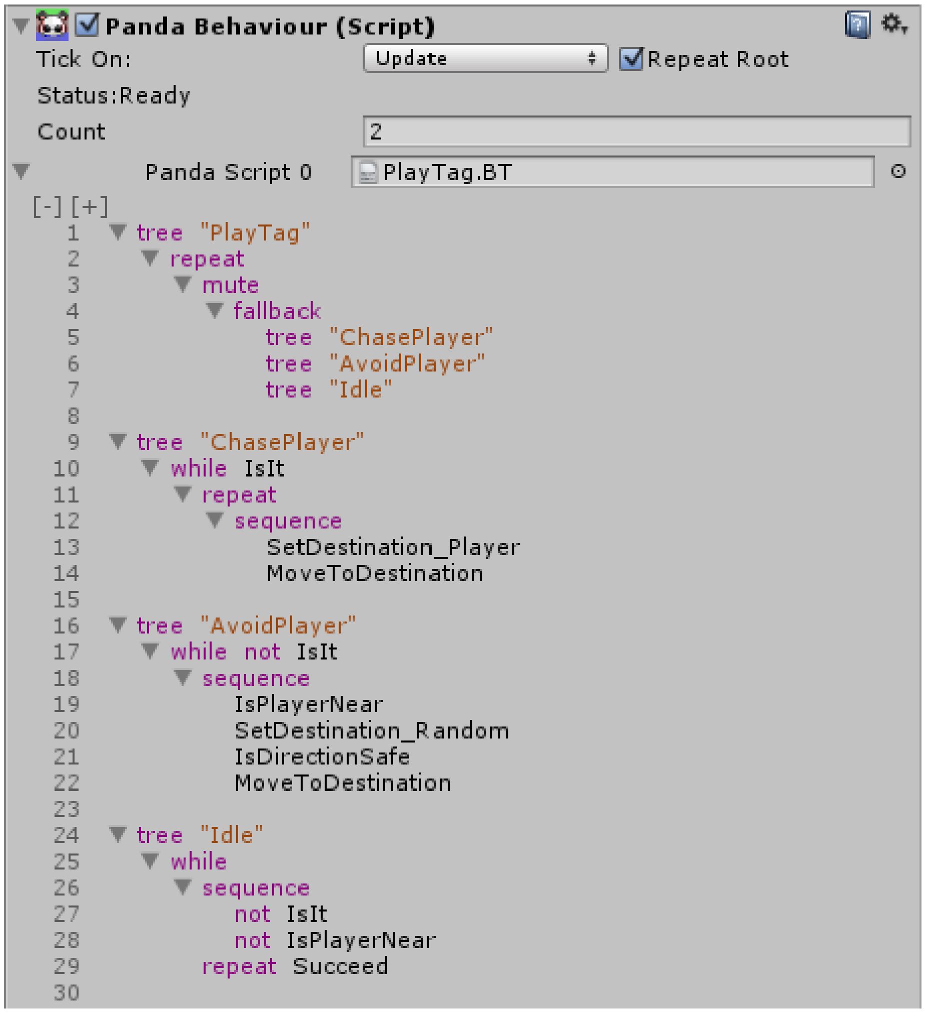
- Tick On

  Determines when the Behaviour Tree is ticked. The options are: `Update`, `LateUpdate`, `FixedUpdate` or `Manual`. When the `Manual` option is chosen, the Behaviour Tree will not be ticked unless the method `PandaBehaviour.Tick()` is called.

- Repeat Root

  When this option is enabled, the root node will be repeated after each completion. When "`Repeat Root`" is disabled and when the Behaviour Tree has completed, ticking the tree will have no effect unless `PandaBehaviour.Reset()` is called.

- Status



Panda Behaviour (Script)

Tick On: Update ✓ Repeat Root

Status:Ready

Count 2

Panda Script 0 PlayTag.BT

[-] [+]

```
tree "PlayTag"
    repeat
        mute
            fallback
                tree "ChasePlayer"
                tree "AvoidPlayer"
                tree "Idle"

tree "ChasePlayer"
    while IsIt
        repeat
            sequence
                SetDestination_Player
                MoveToDestination

tree "AvoidPlayer"
    while not IsIt
        sequence
            IsPlayerNear
            SetDestination_Random
            IsDirectionSafe
            MoveToDestination

tree "Idle"
    while
        sequence
            not IsIt
            not IsPlayerNear
        repeat Succeed

```

Figure 1: Visualizing behaviour tree script within Unity Inspector.

Displays the current status of the root node. `N/A` indicates that the Behaviour Tree is not effective.

- Count

  The number of `TextAsset` slots available for BT scripts.

- BT Script #

  A TextAsset slot that can receive a BT script. You can compile BT scripts at runtime using `PandaBehaviour.Compile(string source)` or `PandaBehaviour.Compile(string[] sources)`.

Nodes folding

You can use the [–] and [+] buttons to respectively collapse and unfold the Behaviour Tree hierarchy.

Opening C# implementations of tasks

When you double click on a task name in the Inspector, the file containing the task implementation will be opened in your external text editor at the task definition.

Script Viewer

The Panda BT component integrate a Script Viewer that works in two modes: Static View and Live View. The Static View mode is for inspecting the code whereas the Live View mode is for visualizing the Behaviour Tree state at run-time.

Static View

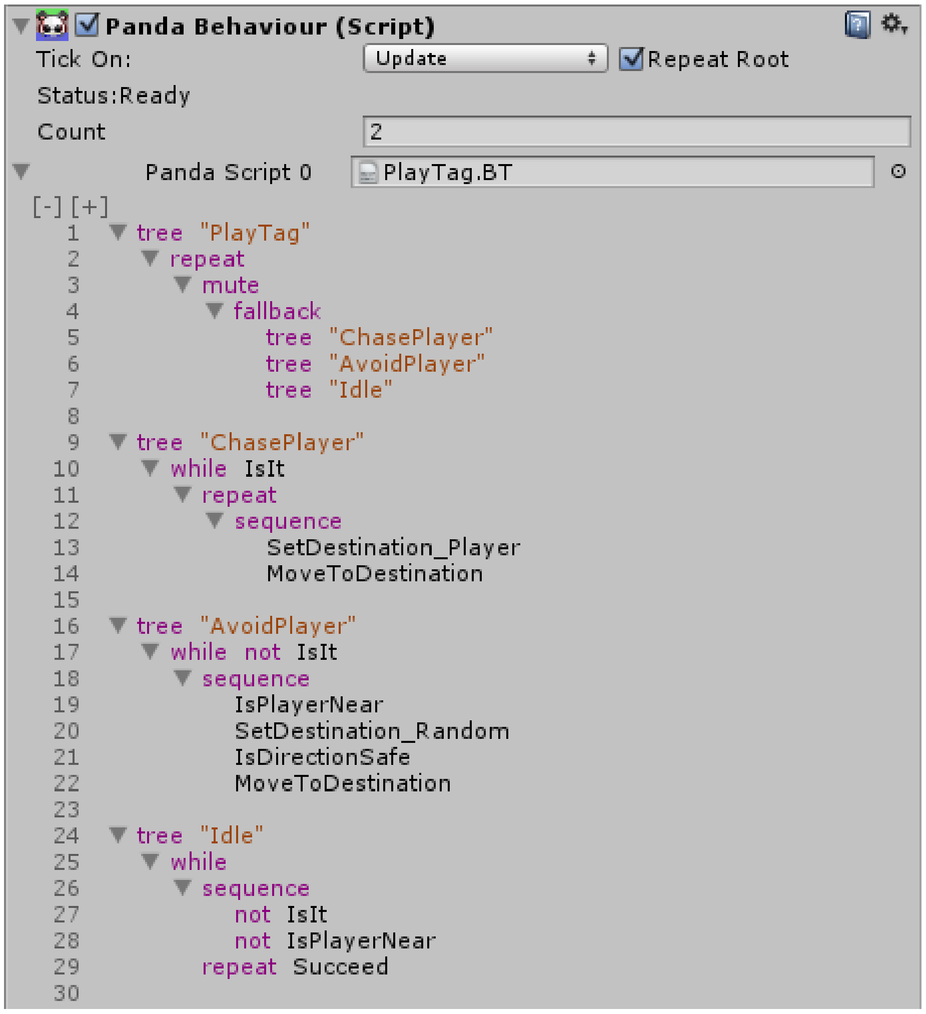
When the editor is not in play mode, the Behaviour Tree is not active and the the Script Viewer is in Static View mode. In this mode the color indicates the type of element:

- Purple: structural nodes
- Orange: node parameters
- Black (or White for the Unity Dark Skin) : tasks
- Grey: comments

Live View

In play mode, the Behaviour Tree is executing (if there is no error). The script viewer is in Live View mode. In this mode, the color indicates current status of the nodes:

- Green: The node has succeeded.



Panda Behaviour (Script)

Tick On: Update ✓ Repeat Root

Status:Ready

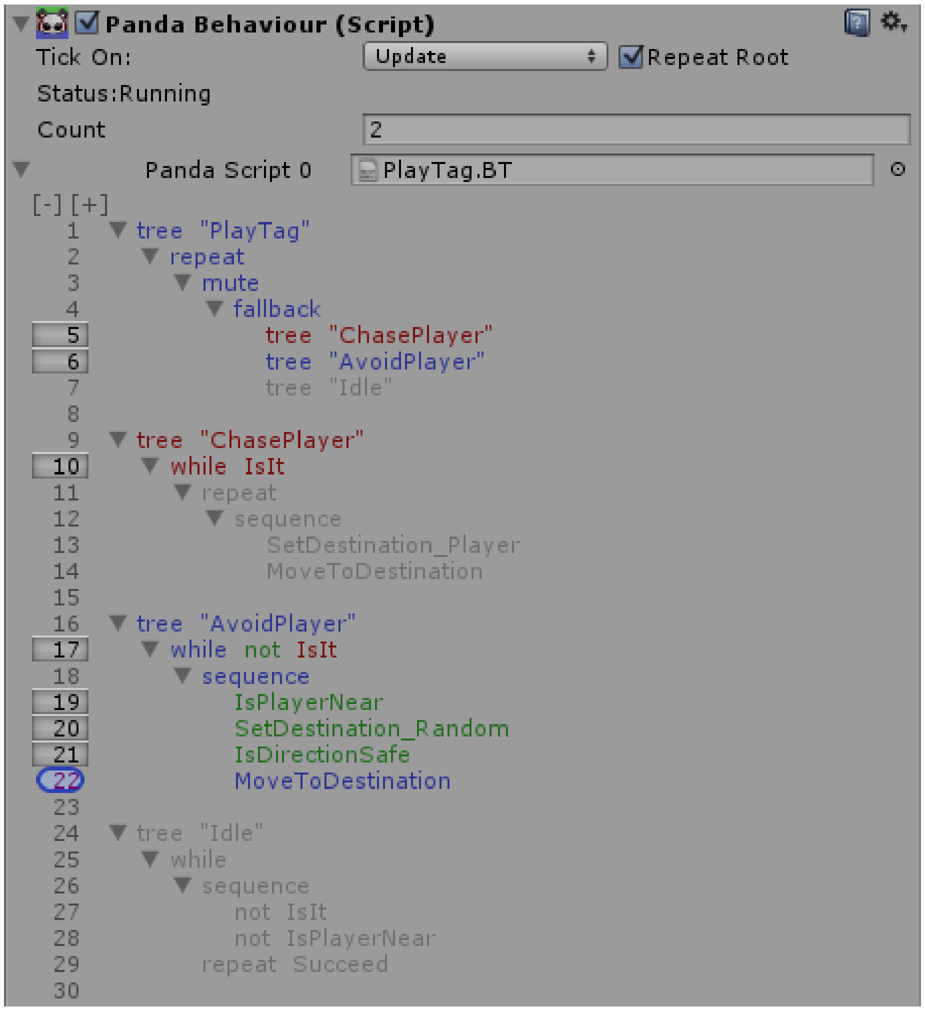
Count 2

Panda Script 0 PlayTag.BT

[-] [+]

```
tree "PlayTag"
    repeat
        mute
            fallback
                tree "ChasePlayer"
                tree "AvoidPlayer"
                tree "Idle"

tree "ChasePlayer"
    while IsIt
        repeat
            sequence
                SetDestination_Player
                MoveToDestination

tree "AvoidPlayer"
    while not IsIt
        sequence
            IsPlayerNear
            SetDestination_Random
            IsDirectionSafe
            MoveToDestination

tree "Idle"
    while
        sequence
            not IsIt
            not IsPlayerNear
        repeat Succeed

```

Figure 2: Visualizing behaviour tree script within Unity Inspector.



Panda Behaviour (Script)

Tick On: Update   Repeat Root

Status:Running

Count 2

Panda Script 0 PlayTag.BT

[-] [+]

```
tree "PlayTag"
    repeat
        mute
            fallback
                tree "ChasePlayer"
                tree "AvoidPlayer"
                tree "Idle"

tree "ChasePlayer"
    while IsIt
        repeat
            sequence
                SetDestination_Player
                MoveToDestination

tree "AvoidPlayer"
    while not IsIt
        sequence
            IsPlayerNear
            SetDestination_Random
            IsDirectionSafe
            MoveToDestination

tree "Idle"
    while
        sequence
            not IsIt
            not IsPlayerNear
        repeat Succeed
```

Figure 3: Visualizing behaviour tree script within Unity Inspector at runtime.

- Red: The node has failed.
- Blue: The node is running.
- Light grey: The node is ready (it has not been ticked since the last Behaviour Tree reset).
- Dark grey: Comments or debug info.

When the line number is highlighted, it means that a leaf node on that line has been ticked on the current frame.

In-Inspector Editor (Pro Only)

- You can drag & drop the nodes around to organize the tree hierarchy (similarly to the Unity scene Hierarchy tab). The node parameters are also editable similarly to any inspector input field. Copy/Paste/Cut/Delete/Duplicate work as expected and Undo/Redo is supported (the corresponding keys and shortcuts are functional).
- When you modify a script through this editor, the modified version concerns only the current GameObject instance unless you "apply" the modifications (which will save the modifications to the original script). You can also "revert" the modifications. This mechanism is similar to how prefabs work. For instance, you can use this feature to have local variations without duplicating scripts.
- Task creation: You can select where to insert a new task with a right click (dragging with RMB will display a visual cue about where the task will be inserted. Then, at the selected location, drop-down menu will be available listing all the available tasks, grouped by MonoBehaviours, implemented on the current GameObject (see attached screenshot).

Debugging

You can set break points and display custom text describing the task progression in the inspector by using the Debug Info.

Break points (Pro only)

You can set and unset a break point by clicking on a line number (only for lines containing nodes). The line number will turn yellow indicating that the break point is set.

When the execution flow reaches a line with a break point set, the Editor will pause and that line number will turn purple. You can resume the game or skip to the next frame by clicking the usual pause or forward buttons in the Unity tool bar.

The break point will be activated only when a node is starting, that is on the first tick of the node. Visually in the Inspector, this event happens when the node turn its color from grey to another color.

Debug Info

Optionally, you can display custom text that would give insight into how a task is currently performing.

In Life View, the Debug Info is visible in the Inspector next to a node when it has been ticked. You can set the displayed text from the C# task method by assigning a string to the following task property:

```
ThisTask.debugInfo
```

Implementing tasks in C

Each task in a BT script requires a C# implementation (a method, a field or a property) defined in any MonoBehaviour script attached to the GameObject. A single MonoBehaviour can contains several task definitions.

Example:

```
using UnityEngine;
using Panda;

public class Example : MonoBehaviour
{
    [Task]
    void SetColor(float r, float g, float b)
    {
        var rdr = this.GetComponent<Renderer>();
        rdr.material.color = new Color(r, g, b);

        ThisTask.Succeed();
    }
}
```

Here, the task SetColor will set the color of the GameObject to the given rgb value then it will succeeds immediately.

The [Task] attribute

A task is implemented as a void method with the exact same name as in the BT script (task names are case sensitive) and the same number of parameters and parameter types. This method must be preceded with the [Task] attribute in order to be recognized by the Behaviour Tree engine.

When preceded with the [Task] attribute, a method returning a boolean, a boolean field or a boolean property can as well implement a task. In these cases, the task will complete immediately (the task will last only one tick) and the completion status will depends on the returned value of the boolean: true for succeeded and false for failed.

ThisTask

`ThisTask` gives access to the run-time instance of a task, providing information and interactions with the task. `ThisTask` is valid only within the scope of task methods.

Methods

A task will be in the running state until one of the following completion methods is called:

- `void Succeed()`

  Indicates that the task has succeeded.

- `void Fail()`

  Indicates that the task has failed.

- `void Complete(bool hasSucceeded)`

  Indicates that the task has succeeded or failed by respectively setting the value of *hasSucceeded* to true or false.

Properties

- `bool isStarting`

  (read only) True on the first tick of a task. Usually used to perform initialization.

- `string debugInfo`

  (read/write) Text displayed at run-time next to the node in the Inspector, after the node has been ticked.

- `object item`

  Arbitrary object attached to the task. Usually used to store data required for the progression of the task.

- `Status status`

  (read only) Returns the current status.

- `static bool isInspected`

  (read only) Whether the task is currently displayed in the Inspector. (Used to avoid GC allocation occurring when formatting string)

Writing BT scripts

The Panda BT component requires at least one BT script assigned as a TextAsset. You can create a new Panda Script asset by right clicking on a folder in the Project tab then by selecting Create > Panda BT Tree Script. A new TextAsset containing the default BT script will be created:

```
tree "Root"
    Succeed
```

Any Behaviour Tree must have a root tree, defined by the **tree** node having the string parameter "Root" as name. Here the child of the root tree is the task *Succeed*, which always succeeds immediately.

This Behaviour Tree does not do that much; it simply returns "Succeeded". So you want to edit this file and write your own Behaviour Tree so that it makes something interesting.

Writing a BT scripts consists of specifying one or more hierarchies of nodes. A node can be a structural node, which are explained below, or a task node, which is a low level task referencing a C# implementation.

Let's write a first BT script that cycle the color of the `GameObject` using the task `SetColor` defined above.

```
tree "Root"
    sequence
        SetColor(1.0, 0.0, 0.0)
        Wait 1.0

        SetColor(0.0, 1.0, 0.0)
        Wait 1.0

        SetColor(0.0, 0.0, 1.0)
        Wait 1.0
```

The sequence node is used to run the tasks one after another. The `Wait` task, is a build-in task that succeeds after a given number of seconds.

The color of the `GameObject` will cycle red-green-blue and each color will be displayed for a duration of one second.

Behaviour Tree, a hierarchy of nodes

A Behaviour Tree consists of an hierarchy of nodes. The hierarchy is implied by the text indentation and the position of the nodes on the same lines.

Indentation parenting

The hierarchy is defined by the text indentation (spaces or tabulations); the children of a node is indented to the right. Depending of its type, a node can have from none to several children.

Example:

```
tree "Root"
    parallel
        sequence
            task_A
            task_B
            task_C
        while
            not
               condition
            task_D
```

The **`tree`** node has the **`parallel`** node as child. The **`parallel`** node has two children: the **`sequence`** node and the **`while`** node. The **`sequence`** node has three children: `task_A`, `task_B` and `task_C` . The **`while`** node has two children: the **`not`** node and `task_D`. The **`not`** node has the `condition` node as child.

Single line parenting

In order to save vertical space or to improve readability, it's possible to write parent/child relationship on the same line. When several nodes are written on the same line, a node is parented to the first structural node from its left.

The same example as above can be written, with the same hierarchy, as follow:

```
tree "Root"
    parallel
        sequence task_A task_B task_C
        while not condition
            task_D
```

Structural nodes

There are two types of nodes: the task nodes and the structural nodes. The task nodes, the leafs of the tree, are the low level actions implemented in C#, whereas the structural nodes are the Panda BT language elements, which are highlighted as purple keywords in the Inspector when the Unity Editor is not in play mode.

The structural nodes defines the execution flow of the Behaviour Tree.

The Panda BT language is essentially composed of 10 structural nodes:

- The **`tree`** node is an identifier node which gives a name to a node hierarchy and can be referenced from anywhere-else in the BT scripts, similarly to how functions and methods are used in other programming language.
- The **`sequence`** node and **`fallback`** node implement sequence and selector respectively.
- The **`parallel`** node and **`race`** node are the asynchronous version of the sequence and fallback.
- The **`random`** node randomly selects and runs one of its children.
- The **`repeat`** node implements iterations.
- The **`while`** node implements conditional run.
- The **`not`** node and **`mute`** node modify the returned status of their child.

Each structural node has its own way *to run* (i.e. to tick until completion) its children and it has its own requirements for returning one of the statuses (running, succeeded or failed).

All details about each node type, how they run their children and when they returned each status, are describe below.

If you want to see how the nodes behave in a real an application, you can check this demo which provides example for each structural node.

Tree

The **`tree`** node has a *name* specified as a string parameter. A tree definition has one child and can be referenced from another trees by its name. Any Behaviour Tree requires exactly one root tree named “Root”, which is the first node being ticked, therefore the entrance point of the execution flow. When a tree node is ticked, it equally ticks its child and it returns the same status has its child.

```
tree "Name"
    child
```

Returns	When
Running	The child is running.
Succeeded	The child has succeeded.
Failed	The child has failed.

Sequence

The **`sequence`** node runs its children one by one, from top to bottom, as long as they succeed. It succeeds when all children succeed and it fails on the first child that fails.

```
sequence
    child_0
    child_1
    ...
    child_k
```

Returns	When
Running	A child is running.
Succeeded	All the children have succeeded.
Failed	One child has failed.

Fallback

The **`fallback`** node (or selector) runs its child one by one, from top to bottom, as long as they fail. It succeeds on the first child that succeeds and it fails when all children fail.

```
fallback
    child_0
    child_1
    ...
    child_k
```

Returns	When
Running	A child is running.
Succeeded	One child has succeeded.
Failed	All the children have failed.

Parallel

The **`parallel`** node runs all its children at the same time. It succeeds when all the children succeed and it fails on the first child that fails.

```
parallel
    child_0
    child_1
    ...
    child_k
```

Returns	When
Running	At least one child is running and none has failed.
Succeeded	All child has succeeded.
Failed	One child has failed.

Race

The **`race`** node runs all its children at the same time. It succeeds on the first child that succeeds and it fails when all the children fail.

```
race
    child_0
    child_1
    ...
    child_k
```

Returns	When
Running	At least one child is running and none has succeeded.
Succeeded	One child has succeeded.
Failed	All the children have failed.

Random

The **random** node randomly selects and runs one of its children. It returns the same status as the selected child; it succeeds when the selected child succeeds and fails when the selected child fails.

Optional weights *w_i* can be specified for each *child_i* in order to define a probability distribution for the selection. Higher weight value means higher chance of being selected.

```
random(w_0, w_1, ..., w_k)
    child_0
    child_1
    ...
    child_k
```

Returns	When
Running	The selected child is running.
Succeeded	The selected child has succeeded.
Failed	The selected child has failed.

Repeat

The **repeat** node runs its child n times. If n is unspecified, the repetition is infinite. It succeeds after n successful iterations of the *child* and it fails when the *child* fails.

```
repeat n
    child
```

Returns	When
Running	The *child* has succeeded less than n times.

Returns	When
Succeeded	The *child* has succeeded n times. (If n is unspecified, the **repeat** node never succeeds.)
Failed	The *child* has failed.

While

The **`while`** node has two children: the first one is the *condition* and the second one is the *action*. It repeats the *condition* and runs the *action* unless the *condition* fails. The *action* is started after the first success of the *condition*. It succeeds if the *action* succeeds and it fails when any child fails.

```
while
    condition
    action
```

Returns	When
Running	Any child is running.
Succeeded	The *action* has succeeded.
Failed	The *condition* or the *action* has failed.

Not

The **`not`** node inverts the completion status of its child. It succeeds when the *child* fails and fails when the *child* succeeds.

```
not
    child
```

Returns	When
Running	The child is running.
Succeeded	The child has failed.
Failed	The child has succeeded.

Mute

The **`mute`** node always succeeds whenever its *child* succeeds or fails.

```
mute
    child
```

Returns	When
Running	The *child* is running.
Succeeded	The *child* has failed or has succeeded.

Returns	When
Failed	(The **mute** node never fails.)

Built-in Tasks

The `PandaBehaviour` component implements several sets of built-in tasks (with source available). Theses tasks are ready to be used from any BT scripts attached to a `PandaBehaviour` component.

Status

Task	Description
void Succeed ()	Succeed immediately.
void Fail ()	Fail immediately.
void Running ()	Run indefinitely.

Time

Task	Description
void Wait (float duration)	Run for *duration* seconds then succeed.
void Wait (int ticks)	Run for *ticks* ticks then succeed.
void RealtimeWait (float duration)	Run for *duration* unscaled seconds then succeed.

Input

Task	Description
void IsKeyDown (string keycode)	Succeed if the key *keycode* is down on the current tick, otherwise fail.
void IsKeyUp (string keycode)	Succeed if the key *keycode* is up on the current tick, otherwise fail.
void IsKeyPressed (string keycode)	Succeed if the key *keycode* is pressed on the current tick, otherwise fail.
void IsMouseButtonPressed (int button)	Succeed if the mouse button *button* is pressed on the current tick, otherwise fail.
void IsMouseButtonUp (int button)	Succeed if the mouse button *button* is up on the current tick, otherwise fail.
void IsMouseButtonDown (int button)	Succeed if the mouse button *button* is down on the current tick, otherwise fail.

Task	Description
void IsButtonUp (string buttonName)	Succeed if the button *buttonName* is up on the current tick, otherwise fail.
void IsButtonDown (string buttonName)	Succeed if the button *buttonName* is down on the current tick, otherwise fail.
void IsButtonPressed (string buttonName)	Succeed if the button *buttonName* is pressed on the current tick, otherwise fail.
void WaitKeyDown (string keycode)	Run indefinitely and succeed when the key *keycode* is down.
void WaitAnyKeyDown ()	Run indefinitely and succeed when any key is down.
void WaitKeyUp (string keycode)	Run indefinitely and succeed when the key *keycode* is up.
void WaitMouseButtonUp (int button)	Run indefinitely and succeed the mouse button *button* is up.
void WaitMouseButtonDown (int button)	Run indefinitely and succeed the mouse button *button* is down.
void WaitButtonUp (string buttonName)	Run indefinitely and succeed the button *buttonName* is up.
void WaitButtonDown (string buttonName)	Run indefinitely and succeed the button *buttonName* is down.
void HoldKey (string keycode, float duration)	Run indefinitely and complete when the key *keycode* is up. Succeed if the key had been held for *duration* seconds, otherwise fail.
void IsNextKeyDown (string keycode)	Run indefinitely and complete when any key is down. Succeed if the key is *keycode* , otherwise fail.

Debug

Task	Description
void DebugLog (string message)	Log *message* to the console and succeed immediately.
void DebugLogError (string message)	Log the error *message* to the console and succeed immediately.
void DebugLogWarning (string message)	Log the warning *message* to the console and succeed immediately.
void DebugBreak ()	Break (pose the editor) and succeed immediately.