LimboAI

Release 1.0

Serhii Snitsaruk and the LimboAI contributors

Jun 19, 2026

GETTING STARTED

LimboAI is an open-source C++ module for **Godot Engine 4** providing a combination of **Behavior Trees** and **State Machines** for crafting your game's AI. It comes with a behavior tree editor, built-in documentation, visual debugger, and more! While it is implemented in C++, it fully supports GDScript for *creating your own tasks* and states. The full list of features is available on the LimboAI GitHub page.

Behavior Trees are powerful hierarchical structures used to model and control the behavior of agents in a game (e.g., characters, enemies, entities). They are designed to make it easier to create complex and highly modular behaviors for your games. To learn more about behavior trees, check out *Introduction to Behavior Trees*.

Hierarchical State Machines are finite state machines that allow any state to host their own sub-state machine. This allows you to tackle your AI's state and transition complexity by breaking down one big state machine into multiple smaller ones.

CHAPTER

ONE

GETTING LIMBOAI

LimboAI can be used as either a C++ module or as a GDExtension shared library. There are some differences between the two. In short, GDExtension version is more convenient to use but somewhat limited in features. The module version provides better editor experience and is slightly more performant, but it requires using custom engine builds including the export templates. Whichever you choose to use, your project will stay compatible with both and you can switch from one to the other any time.

Choose the version you'd like to use. If you're unsure, start with the GDExtension version. You can change your decision at any time - both versions are fully compatible.

1.1 Get GDExtension version

Precompiled builds are available on the official LimboAI GitHub page, and in the Asset Library.

GDExtension is the most convenient way of using the LimboAI plugin, but it comes with certain limitations:

- Documentation tooltips are not available.
- Handy *BBParam* property editor is not available in the extension due to dependencies on the engine classes that are not available in the Godot API.

  See also: What is GDExtension?

Installation instructions:

1. Make sure you're using the latest stable version of the Godot editor.
2. Create a new project for your experiments with LimboAI.
3. In Godot, click AssetLib tab at the top of the screen and search for LimboAI. Download it. LimboAI plugin will be downloaded with the demo project files. Don't mind the errors printed at this point, this is due to the extension library not being loaded just yet.
4. Reload your project with *Project -> Reload project*. There shouldn't be any errors printed now.
5. In the project files, locate a scene file called *showcase.tscn* and run it. It's the demo's entry point.

1.2 Get module version

Precompiled builds are available on the official LimboAI GitHub page.

Installation instructions:

1. In GitHub releases, download the latest pre-compiled release build for your platform.
2. Download the demo project archive from the same release.
3. Extract the pre-compiled editor and the demo project files.

4. Launch the pre-compiled editor binary, import and open the demo project.
5. Run the project.

Important: To export your game using the module version of LimboAI, make sure to use the pre-compiled export templates included in the same GitHub release build.

CHAPTER

TWO

USING LIMBOAI WITH C#

Here's how you can use the module version with C# to write your tasks and states.

1. Locate LimboAI NuGet package files.

Each provided build comes with a GodotSharp folder, which has packages under "GodotSharp/Tools/nupkgs/". Note down the directory path; you'll need it in the next step.

2. Add a local source for NuGet packages using the following command:

```
dotnet nuget add source path/to/limboai/nupkgs --name LimboNugetSource
```

3. Your C# project should be able to see LimboAI classes and compile.

Regarding GDExtension, I can only confirm success with the module version and C#. Unfortunately, I haven't had any luck with the GDExtension version yet. If you've had success with GDExtension, please let me know via Discord or email.

CHAPTER

THREE

CONTRIBUTING

We target the latest stable version of the Godot Engine for development until a third beta or a release candidate of an upcoming Godot release becomes available. If you want to contribute to the project, please ensure that you are using the corresponding Godot version.

We follow the Godot code style guidelines. Please use `clang-format` to maintain consistent styling. You can install `pre-commit` hooks for Git using `pre-commit install` to automate this process.

Please keep the minor versions backward-compatible when submitting pull requests.

We support building LimboAI as a module for the Godot Engine and as a GDExtension library. Make sure your contribution is compatible with both. Our CI workflow will verify that your changes can be compiled in both configurations.

3.1 Compiling as module of the Godot Engine

1. Clone the Godot Engine repository.
2. Switch to the latest stable tag.
3. Clone the LimboAI repository into the `modules/limboai` directory.

```
git clone https://github.com/godotengine/godot.git
git checkout 4.3-stable  # Replace "4.3-stable" with the latest stable tag
git clone https://github.com/limbonaut/limboai modules/limboai
```

Consult the Godot Engine documentation for detailed instructions on building the engine.

Unit tests can be compiled using the `tests=yes` build option. To execute them, run the compiled Godot binary with the `--test --tc="*[LimboAI]*"` command-line options.

3.1.1 Building C# Packages

If you want C# Support for LimboAI when building from source, follow Godot's instructions for building from source with C# Support: Compiling with .NET. LimboAI's classes will be included in the output from that build process.

3.2 Compiling as GDExtension library

You'll need the SCons build tool and a C++ compiler. See also Compiling. Run `scons target=editor` to build the plugin library for your current platform.

- SCons will automatically clone the `godot-cpp` repository if it doesn't already exist in the `limboai/godot-cpp` directory.
- By default, built targets are placed in the demo project: `demo/addons/limboai/bin/`.

Check `scons -h` for other options and targets.

3.3 Contributing to the documentation

Online documentation is created using Sphinx. Source files are located in the `doc/source` directory in RST format and can be built locally with `sphinx-build`. See the Sphinx documentation for more details.

Class documentation resides in XML files within the `doc_classes/` directory. If you create a new class or modify an existing one, you can run the compiled Godot binary with the `--doctool` option in the root of the Godot source code to generate the missing XML files or sections within those files in the class documentation. After editing the XML files, please run the compiled editor binary with the `--doctool` option again to update and tidy up the XML files.

Sphinx RST files for the class documentation are generated from XML files using the Godot script `make_rst.py` and stored in the `doc/source/classes` directory. This process is performed using our own script `scripts/update_rst.sh`. RST files in `doc/source/classes` should not be edited manually.

CHAPTER

FOUR

INTRODUCTION TO BEHAVIOR TREES

Note: Demo project includes a tutorial that provides an introduction to behavior trees through illustrative examples.

Behavior Trees (BT) are hierarchical structures used to model and control the behavior of agents in a game (e.g., characters, enemies, entities). They are designed to make it easier to create complex and highly modular behaviors for your games.

Behavior Trees are composed of tasks that represent specific actions or decision-making rules. Tasks can be broadly categorized into two main types: control tasks and leaf tasks. **Control tasks** determine the execution flow within the tree. They include *Sequence*, *Selector*, and *Invert*. **Leaf tasks** represent specific actions to perform, like moving or attacking, or conditions that need to be checked. The *BTTask* class provides the foundation for various building blocks of the Behavior Trees. Such tasks can *share data using the Blackboard*.

Note: To *create your own actions*, extend the *BTAction* class.

A Behavior Tree is usually processed each frame. It is traversed from top to bottom, with the control tasks determining the control flow. Each task has a *_tick* method, which performs the task's work and returns a status indicating its progress: `SUCCESS`, `FAILURE`, or `RUNNING`. `SUCCESS` and `FAILURE` indicate the outcome of finished work, while `RUNNING` status is returned when a task requires more than one tick to complete its job. These statuses determine how the tree progresses, with the `RUNNING` status usually meaning that the tree will continue execution during the next frame.

There are *four types of tasks*:

- **Actions** are leaf tasks that perform the actual work.
  - Examples: *PlayAnimation*, *Wait*.
- **Conditions** are leaf tasks that conduct various checks.
  - Examples: *CheckVar*, *InRange*.
- **Composites** can have one or more child tasks, and dictate the execution flow of their children.
  - Examples: *Sequence*, *Selector*, *Parallel*.
- **Decorators** can only have a single child and they change how their child task operates.
  - Examples: *AlwaysSucceed*, *Invert*, *TimeLimit*.

Sequence is one of the core composite tasks. It executes its child tasks sequentially, from first to last, until one of them returns `FAILURE`, or all of them result in `SUCCESS`. In other words, if any child task results in `FAILURE`, the *Sequence* execution will be aborted, and the *Sequence* itself will return `FAILURE`.

Selector is another essential composite task. It executes its child tasks sequentially, from first to last, until one of them returns `SUCCESS` or all of them result in `FAILURE`. In other words, when a child task results in `FAILURE`, it moves on to the next one until it finds the one that returns `SUCCESS`. Once a child task results in `SUCCESS`, the *Selector* stops and also returns `SUCCESS`. The purpose of the *Selector* is to find a child that succeeds.

Behavior Trees handle conditional logic using **condition tasks**. These tasks check for specific conditions and return either `SUCCESS` or `FAILURE` based on the state of the agent or its environment (e.g., "IsLowOnHealth", "IsTargetInSight"). Conditions can be used together with *Sequence* and *Selector* to craft your decision-making logic.

Note: To *create your own conditions*, extend the *BTCondition* class.

Check out the *BTTask* class documentation, which provides the foundation for various building blocks of Behavior Trees.

CHAPTER

FIVE

CREATING BEHAVIOR TREES

This chapter describes how to create and debug behavior trees.

5.1 Add a Behavior Tree to an agent

Follow these steps to add a behavior tree to a new or existing agent:

1. Make a scene file for your agent, or open an existing scene.
2. Add a *BTPlayer* node to your scene.
3. Select *BTPlayer*, and create a new behavior tree in the inspector.
4. Optionally, you can save the behavior tree to a file using the property's context menu.
5. Click the behavior tree property to open it in the LimboAI editor.

5.2 Debugging Behavior Trees

In Godot Engine, follow to "Bottom Panel > Debugger > LimboAI" tab. With the LimboAI debugger, you can inspect any currently active behavior tree within the running project. The debugger can be detached from the main editor window, which can be particularly useful if you have a HiDPI or a secondary display.

5.2.1 Inspecting task properties and blackboard

When you click on a task in the LimboAI debugger, its properties are shown in the remote Inspector panel, including the *Blackboard* reference. Click the *Blackboard* to open it and view all variables across the full scope chain, grouped by scope. You can also edit variable values live, which can be useful for testing and debugging your game's AI.

CHAPTER

SIX

CREATING CUSTOM TASKS

By default, user tasks should be placed in the `res://ai/tasks` directory. You can set an alternative location for user tasks in the `Project Settings` → `Limbo AI` (To see those options, `Advanced Settings` should be enabled in the Project Settings).

Each subdirectory within the user tasks directory is treated as a category. Therefore, if you create a subdirectory named "motion_and_physics," your custom tasks in that directory will automatically be categorized under "Motion And Physics."

When creating custom tasks, **extend one of the following** base classes: *BTAction*, *BTCondition*, *BTDecorator*, *BTComposite*. More on task types you can read in the *Introduction to Behavior Trees*.

> **Note:** To help you write new tasks, you can add a script template to your project using "Misc → Create script template" menu option.

Using the *Blackboard* is covered in *Accessing the Blackboard in a Task*.

6.1 Task anatomy

```
@tool
extends BTAction

# Task parameters.
@export var parameter1: float
@export var parameter2: Vector2

## Note: Each method declaration is optional.
## At minimum, you only need to define the "_tick" method.


# Called to generate a display name for the task (requires @tool).
func _generate_name() -> String:
    return "MyTask"


# Called to initialize the task.
func _setup() -> void:
    pass


# Called when the task is entered.
func _enter() -> void:
```

(continues on next page)

(continued from previous page)

```
    pass


# Called when the task is exited.
func _exit() -> void:
    pass


# Called each time this task is ticked (aka executed).
func _tick(delta: float) -> Status:
    return SUCCESS


# Strings returned from this method are displayed as warnings in the editor.
func _get_configuration_warnings() -> PackedStringArray:
    var warnings := PackedStringArray()
    return warnings
```

6.2 Example 1: A simple action

```
@tool
extends BTAction

## Shows or hides a node and returns SUCCESS.
## Returns FAILURE if the node is not found.

# Task parameters.
@export var node_path: NodePath
@export var visible := true


# Called to generate a display name for the task (requires @tool).
func _generate_name() -> String:
    return "SetVisible  %s  node_path: \"%s\"" % [visible, node_path]


# Called each time this task is ticked (aka executed).
func _tick(p_delta: float) -> Status:
    var n: CanvasItem = scene_root.get_node_or_null(node_path)
    if is_instance_valid(n):
        n.visible = visible
        return SUCCESS
    return FAILURE
```

6.3 Example 2: InRange condition

```
@tool
extends BTCondition
```

(continues on next page)

(continued from previous page)

```
## InRange condition checks if the agent is within a range of target,
## defined by distance_min and distance_max.
## Returns SUCCESS if the agent is within the defined range;
## otherwise, returns FAILURE.

@export var distance_min: float
@export var distance_max: float
@export var target_var: StringName = &"target"

var _min_distance_squared: float
var _max_distance_squared: float


# Called to generate a display name for the task.
func _generate_name() -> String:
    return "InRange (%d, %d) of %s" % [distance_min, distance_max,
        LimboUtility.decorate_var(target_var)]


# Called to initialize the task.
func _setup() -> void:
    _min_distance_squared = distance_min * distance_min
    _max_distance_squared = distance_max * distance_max


# Called when the task is executed.
func _tick(_delta: float) -> Status:
    var target: Node2D = blackboard.get_var(target_var, null)
    if not is_instance_valid(target):
        return FAILURE

    var dist_sq: float = agent.global_position.distance_squared_to(target.global_
↪position)
    if dist_sq >= _min_distance_squared and dist_sq <= _max_distance_squared:
        return SUCCESS
    else:
        return FAILURE
```

6.4 Creating tasks in C#

You can use the following script template for custom tasks:

```
using Godot;
using System;

[Tool]
public partial class _CLASS_ : _BASE_
{
    public override string _GenerateName()
    {
        return "_CLASS_";
```

(continues on next page)

(continued from previous page)

```
    }

    public override void _Setup()
    {
    }

    public override void _Enter()
    {
    }

    public override void _Exit()
    {
    }

    public override Status _Tick(double delta)
    {
        return Status.Success;
    }

    public override string[] _GetConfigurationWarnings()
    {
        return Array.Empty<string>();
    }
}
```

CHAPTER

SEVEN

SHARING DATA USING BLACKBOARD

To share data between different tasks and states, we employ a feature known as the *Blackboard*. The *Blackboard* serves as a central repository where tasks and states can store and retrieve named variables, allowing for seamless data interchange. Each instance of a behavior tree or a state machine gets its own dedicated *Blackboard*. It has the capability to store various data types, including objects and resources.

Using the *Blackboard*, you can easily share data in your behavior trees, making the tasks in the behavior tree more flexible.

7.1 Accessing the Blackboard in a Task

Every *BTTask* has access to the *Blackboard*, providing a straightforward mechanism for data exchange. Here's an example of how you can interact with the *Blackboard* in GDScript:

```
@export var speed_var: StringName = &"speed"

func _tick(delta: float) -> Status:
    # Set the value of the "speed" variable:
    blackboard.set_var(speed_var, 200.0)

    # Get the value of the "speed" variable, with a default value of 100.0 if not found:
    var speed: float = blackboard.get_var(speed_var, 100.0)

    # Check if the "speed" variable exists:
    if blackboard.has_var(speed_var):
        # ...
```

If you are accessing a variable that holds an object instance, and it is expected that the instance may be null or freed, you can do it like this:

```
@export var object_var: StringName = &"object"

func _tick(delta: float) -> Status:
    # Get object instance stored in the "object" variable.
    # - Important: Avoid specifying a type for "obj" in GDScript
    #   to prevent errors when the instance is freed.
    var obj = blackboard.get_var(object_var)
    if is_instance_valid(obj):
        # ...
```

It is recommended to suffix variable name properties with `_var`, like in the example above, which enables the inspector to provide a more convenient property editor for the variable. This editor allows you to select or add the variable to the

blackboard plan, and provides a warning icon if the variable does not exist in the blackboard plan.

Note: The variable doesn't need to exist when you set it in code.

7.2 Editing the Blackboard Plan

The Blackboard Plan, associated with each *BehaviorTree* resource, dictates how the *Blackboard* initializes for each new instance of the *BehaviorTree*. BlackboardPlan resource stores default values, type information, and data bindings necessary for *BehaviorTree* initialization.

To add, modify, or remove variables from the Blackboard Plan, follow these steps:

1. Open the LimboAI editor and load the behavior tree you want to edit.
2. In the editor, click on the small button located inside the tab. This will open the *BlackboardPlan* in the Inspector.
3. In the Inspector, click the "Manage..." button to show the blackboard plan editor.
4. In the blackboard plan editor, you can add, remove, or reorder variables, and modify their data type and hint.
5. The hint provides additional information about the variable to the Inspector, such as minimum and maximum values for an integer variable. Learn more about property hints in the official Godot documentation.
6. You can specify the default values of the variables directly in the Inspector.

7.3 Overriding variables in BTPlayer

Each *BTPlayer* node also has a "Blackboard Plan" property, providing the ability to override values of the BehaviorTree's blackboard variables. These overrides are specific to the BTPlayer's scene and do not impact other scenes using the same *BehaviorTree*. To modify these values:

1. Select the BTPlayer node in the scene tree.
2. In the Inspector, locate the "Blackboard Plan" property.
3. Override the desired values to tailor the blackboard variables for the specific scene.

7.4 Task parameters

In some cases, it can be beneficial to allow behavior tree tasks to export parameters that can either be **bound to a blackboard variable or specified directly** by the user. For this purpose, LimboAI provides special parameter types that begin with "BB", such as *BBInt*, *BBBool*, *BBString*, *BBFloat*, *BBNode*, and more. For a complete list, please refer to the *BBParam* class reference.

Usage example:

```
extends BTAction

@export var speed: BBFloat

func _tick(delta: float) -> Status:
    var current_speed: float = speed.get_value(scene_root, blackboard, 0.0)
    ...
```

7.5 Connecting variables in BTs to HSMs: Variable mapping

Each *BTState* creates a new blackboard scope for its BehaviorTree instance. Because BehaviorTrees are reusable resources that can run in different contexts or on different agents, they do not automatically see variables defined in the HSM. At runtime, HSM variables are usually accessible inside the BT via the parent scope blackboard, but the recommended way to access them is through **mapping**.

> **Note**
>
> To learn more about scopes, see *Blackboard*.

When both the BehaviorTree and the HSM declare variables in their respective *BlackboardPlan* resources, the editor provides a **Mapping** section in the BlackboardPlan inspector. Mapping is the intended and recommended way to connect variables between related plans.

7.5.1 Key points about mapping

- Mapping connects two variables so they behave as a single logical variable at runtime.
- Linked variables are updated immediately in both scopes — they literally share the same state in memory (no polling or copying).
- Mapping is explicit: you decide which variables the BT exposes as inputs and outputs.
- Linkage is bidirectional — there is no distinction between input and output.
- Mapping does not create new variables automatically — the variables must already exist in both blackboard plans before they can be linked.

7.5.2 Inspector workflow

1. Define the required variables in each BlackboardPlan (HSM plan and BT plan, for example).
2. Select the *BTState* node in the scene or a *BTSubtree* inside a behavior tree.
3. In the Inspector, select the Blackboard Plan property.
4. In the BlackboardPlan resource inspector, locate the **Mapping** section.
5. Create mappings by pairing variables (`BT variable  parent variable`).
6. On startup the mappings are applied, and the variables are automatically linked for the running instance.

7.5.3 Programmatic alternative: linking variables in code

You can also link variables directly in code using *Blackboard.link_var*:

```
# Example: programmatically link a BT blackboard variable to an HSM blackboard variable
# Assume `hsm` is a LimboHSM instance and `bt_state` is a BTState instance
var hsm_bb := hsm.get_blackboard()
var bt_bb := bt_state.get_blackboard()

# NOTE: variable names do not need to match
bt_bb.link_var("target_pos", hsm_bb, "target_pos")
```

7.5.4 Best practices

- Declare all BT dependencies in the BT's BlackboardPlan instead of relying on parent scopes.
- Use **Mapping** to explicitly declare which variables should be shared with the HSM.
- Prefer the inspector-based Mapping workflow for clarity and editor support; use `link_var` only when runtime or programmatic setup is required.
- Avoid writing to `blackboard.get_parent()` from inside BT tasks unless you have a very specific reason and accept the coupling it introduces.

7.6 Advanced topic: Blackboard scopes

The *Blackboard* in LimboAI can act as a parent scope for another *Blackboard*. This means that if a specific variable is not found in the active scope, the system will look in the parent *Blackboard* to find it. This creates a "blackboard scope chain," where each *Blackboard* can have its own parent scope, and there is no limit to how many blackboards can be in this chain. It's important to note that the *Blackboard* doesn't modify values in the parent scopes.

Scopes are created automatically to prevent naming collisions between contextually separate environments:

- Within *BTNewScope*.
- Under *BTSubtree* decorators.
- With *LimboState* that have a non-empty blackboard plan defined.
- Under *LimboHSM* nodes: A new scope is created at the root level, and each *BTState* child also receives its own separate scope.

7.6.1 Sharing data between several agents

The blackboard scope mechanism can also be used for sharing data between several agents. In the following example, we have a group of agents, and we want to share a common target between them:

```
extends BTAction

@export var group_target_var: StringName = &"group_target"

func _tick(delta: float) -> Status:
    if not blackboard.has_var(group_target_var):
        var new_target: Node = acquire_target()
        # Set common target shared between agents in a group:
        blackboard.top().set_var(group_target_var, new_target)

    # Access common target shared between agents in a group:
    var target: Node = blackboard.get_var(group_target_var)
```

In this example, *blackboard.top()* accesses the root scope of the *Blackboard* chain. We assign that scope to each agent in a group through code:

```
class_name AgentGroup
extends Node2D
## AgentGroup node: Manages the shared Blackboard for agents in a group.
## Children of this node are assumed to be agents that belong to a common group.
## This implementation assumes that each agent has a "BTPlayer" node for AI.
```

(continues on next page)

(continued from previous page)

```
@export var blackboard_plan: BlackboardPlan

var shared_scope: Blackboard

func _ready() -> void:
    if blackboard_plan == null:
        shared_scope = Blackboard.new()
    else:
        shared_scope = blackboard_plan.create_blackboard()

    for child in get_children():
        var bt_player: BTPlayer = child.find_child("BTPlayer")
        if is_instance_valid(bt_player):
            bt_player.blackboard.set_parent(shared_scope)
```

In conclusion, the *Blackboard* scope chain not only prevents naming conflicts that can occur between state machines, behavior trees, and sub-trees, but it can also be used to share data between several agents.

7.7 Debugging blackboard variables

When debugging a running project, you can inspect *Blackboard* variables in the remote Inspector. Click on a task in the LimboAI debugger to view its properties, then click the *Blackboard* reference to see all variables across the full scope chain, grouped by scope. Variables can be edited live to aid in debugging. For state machines, select a *LimboState* node in the Remote Scene Tree to access its *Blackboard* in the same way.

CHAPTER

EIGHT

ACCESSING NODES IN THE SCENE TREE

There are several ways to access nodes in the agent's scene tree from a *BTTask*.

Note: The root node of the agent's scene tree can be accessed with the *scene_root* property.

8.1 With `BBNode` property

```
@export var cast_param: BBNode

func _tick(delta) -> Status:
    var node: ShapeCast3D = cast_param.get_value(scene_root, blackboard)
```

8.2 With `NodePath` property

```
@export var cast_path: NodePath

func _tick(delta) -> Status:
    var node: ShapeCast3D = scene_root.get_node(cast_path)
```

8.3 Using blackboard plan

You can *create a blackboard variable* in the editor with the type `NodePath` and point it to the proper node in the *BTPlayer* blackboard plan. By default, any `NodePath` variable will be replaced with the node instance when the blackboard is instantiated at runtime (see *BlackboardPlan.prefetch_nodepath_vars*).

```
extends BTCondition

@export var shape_var: StringName = &"shape_cast"

func _tick(delta) -> Status:
    var shape_cast: ShapeCast3D = blackboard.get_var(shape_var)
```

The property BTPlayer.prefetch_nodepath_vars should be set to `true`.

CHAPTER

NINE

CREATE STATE MACHINES

This guide will show how to set up and use a state machine using *LimboHSM*.

9.1 Initialization

To use the *LimboHSM* state machine, you first need to initialize it in your code. This is typically done in the `_ready` function of your script. Here's an example of how to do this:

```
@onready var hsm: LimboHSM = $LimboHSM
@onready var idle_state: LimboState = $LimboHSM/IdleState
@onready var move_state: LimboState = $LimboHSM/MoveState


func _ready() -> void:
    _init_state_machine()


func _init_state_machine() -> void:
    hsm.add_transition(idle_state, move_state, idle_state.EVENT_FINISHED)
    hsm.add_transition(move_state, idle_state, move_state.EVENT_FINISHED)

    hsm.initialize(self)
    hsm.set_active(true)
```

In this example, we first declare the state machine and the states we want to use. Then, in the `_init_state_machine` function, we add transitions between the states. Finally, we initialize the state machine and set it to active.

9.2 State example

You can define the behavior of a state in a script and attach it to the state node. Here's an example of a state that plays an animation on an `AnimationPlayer` and waits for it to finish:

```
extends LimboState
## PlayAnimation: Play an animation on AnimationPlayer, and wait for it to finish.

@export var animation_player: AnimationPlayer
@export var animation_name: StringName

func _enter() -> void:
    animation_player.play(animation_name)
```

(continues on next page)

(continued from previous page)

```
func _update(_delta: float) -> void:
    if not animation_player.is_playing() \
            or animation_player.assigned_animation != animation_name:
        dispatch(EVENT_FINISHED)
```

In this example, the `_enter` method is called when the state is entered, and it plays the specified animation. The `_update` method is called every frame, and it checks if the animation is finished. If it is, it dispatches the `EVENT_FINISHED` event, which can result in a transition to the next state.

9.3 Events and transitions

The *LimboHSM* comes with an **event system** that helps to **decouple transitions** from the state implementations. A transition is associated with a specific event, a starting state, and a destination state, and it is performed automatically when such an event is dispatched.

To register a transition and associate it with a specific event, you can use the *LimboHSM.add_transition* method:

```
hsm.add_transition(idle_state, move_state, &"movement_started")
```

In this example, we're registering a transition from the `idle_state` to the `move_state` when the `movement_started` event is dispatched.

A transition can be also associated with no particular starting state:

```
hsm.add_transition(hsm.ANYSTATE, move_state, &"movement_started")
```

Events are dispatched with the *LimboState.dispatch* method. It's important to note that this method can be called from anywhere in the state machine hierarchy and outside of it. Events are **propagated from the leaf to the root** state. This means that if an event is consumed by a state, it won't be propagated to its parent states.

States can also define **event handlers**, which are methods that react to specific events. These event handlers typically don't result in a state transition; they are simply a mechanism for states to react to particular events. You can use the *LimboState.add_event_handler* method to register event handlers in your states:

```
extends LimboState

func _setup() -> void:
    add_event_handler("movement_started", _on_movement_started)

func _on_movement_started(cargo = null) -> bool:
    # Handle the "movement_started" event here.
    # `cargo` can be passed with the event when calling `dispatch()`.
    # It's quite handy when you need to pass some data to the event handler.
    return true
```

If the event handler returns `true`, the event will be considered as consumed, and it won't propagate further or result in a state transition.

9.4 State anatomy

```
extends LimboState
```

(continues on next page)

(continued from previous page)

```
## Called once, when state is initialized.
func _setup() -> void:
    pass

## Called when state is entered.
func _enter() -> void:
    pass

## Called when state is exited.
func _exit() -> void:
    pass

## Called each frame when this state is active.
func _update(delta: float) -> void:
    pass
```

9.5 Using behavior trees with state machines

The *BTState* is a specialized state node in *LimboHSM* that can host a behavior tree. When a *BTState* is active, it executes the hosted behavior tree each frame, effectively using the behavior tree as its implementation.

This allows you to combine the power of behavior trees with the structure and control of state machines. Behavior trees are excellent for defining complex, hierarchical behaviors, while state machines are great for managing the flow and transitions between different behaviors.

Once the *LimboHSM* has been created, a *BTState* can be created for a given *LimboHSM*:

```
extends CharacterBody2D

var hsm: LimboHSM

@export var behavior_tree: BehaviorTree


func _ready():
    hsm = LimboHSM.new()
    add_child(hsm)

    var bt_state := BTState().new()
    bt_state.named("Supplied BT")

    # Set the behavior tree, or use a load("res://ai/trees/...") call
    bt_state.behavior_tree = behavior_tree
    bt_state.set_scene_root_hint(self)

    hsm.add_child(bt_state)
    hsm.initial_state = bt_state

    hsm.initialize(self)
    hsm.set_active(true)
```

9.6 Single-file state machine setup

In certain scenarios, such as prototyping or during game jams, it's practical to keep the state machine code in a single file. For such cases, *LimboHSM* **supports delegation** and provides **chained methods** for easier setup. Let's illustrate this with a practical code example:

```
extends CharacterBody2D

var hsm: LimboHSM

@onready var animation_player: AnimationPlayer = $AnimationPlayer


func _ready() -> void:
    _init_state_machine()


func _init_state_machine() -> void:
    hsm = LimboHSM.new()
    add_child(hsm)

    # Use chained methods and delegation to set up states:
    var idle_state := LimboState.new().named("Idle") \
        .call_on_enter(func(): animation_player.play("idle")) \
        .call_on_update(_idle_update)
    var move_state := LimboState.new().named("Move") \
        .call_on_enter(func(): animation_player.play("walk")) \
        .call_on_update(_move_update)

    hsm.add_child(idle_state)
    hsm.add_child(move_state)

    hsm.add_transition(idle_state, move_state, &"movement_started")
    hsm.add_transition(move_state, idle_state, &"movement_ended")

    hsm.initialize(self)
    hsm.set_active(true)


func _idle_update(delta: float) -> void:
    var dir: Vector2 = Input.get_vector(
        &"ui_left", &"ui_right", &"ui_up", &"ui_down")
    if not dir.is_zero_approx():
        hsm.dispatch(&"movement_started")


func _move_update(delta: float) -> void:
    var dir: Vector2 = Input.get_vector(
        &"ui_left", &"ui_right", &"ui_up", &"ui_down")
    var desired_velocity: Vector2 = dir * 200.0
    velocity = desired_velocity
    move_and_slide()
    if desired_velocity.is_zero_approx():
```

(continues on next page)

(continued from previous page)

```
        hsm.dispatch(&"movement_ended")
```

CHAPTER

TEN

IMPORTANT CLASSES

There are a lot of classes introduced in LimboAI. Here is a list of the most important ones you should know about:

10.1 BTTask

Inherits: *BT*

Inherited By: *BTAction*, *BTComment*, *BTComposite*, *BTCondition*, *BTDecorator*

Base class for all *BehaviorTree* tasks.

10.1.1 Description

Base class for all *BehaviorTree* tasks. A task is a basic building block in a *BehaviorTree* that represents a specific behavior or control flow. Tasks are used to create complex behaviors by combining and nesting them in a hierarchy.

A task can be one of the following types: action, condition, composite, or decorator. Each type of task has its own corresponding subclass: *BTAction*, *BTCondition*, *BTDecorator*, *BTComposite*.

Tasks perform their work and return their status using the *_tick* method. Status values are defined in *Status*. Tasks can be initialized using the *_setup* method. See also *_enter* & *_exit*.

Note: Do not extend **BTTask** directly for your own tasks. Instead, extend one of the subtypes mentioned above.

10.1.2 Properties

`Node`	*agent*	
Blackboard	*blackboard*	
`String`	*custom_name*	`""`
`float`	*elapsed_time*	
`Node`	*scene_root*	
Status	*status*	

10.1.3 Methods

void (No return value.)	*_enter*() virtual (This method should typically be overridden by the user to have any effect.)
void	*_exit*() virtual
`String`	*_generate_name*() virtual const (This method has no side effects. It doesn't modify any of the instance's member variables.)
`PackedStringAr`	*_get_configuration_warnings*() virtual const
void	*_setup*() virtual
Status	*_tick*(delta: `float`) virtual
void	*abort*()
void	*add_child*(task: *BTTask*)
void	*add_child_at_index*(task: *BTTask*, idx: `int`)
BTTask	*clone*() const
BehaviorTree	*editor_get_behavior_tree*()
Status	*execute*(delta: `float`)
BTTask	*get_child*(idx: `int`) const
`int`	*get_child_count*() const
`int`	*get_child_count_excluding_comments*() const
`int`	*get_index*() const
BTTask	*get_parent*() const
BTTask	*get_root*() const
`String`	*get_task_name*()
`bool`	*has_child*(task: *BTTask*) const
void	*initialize*(agent: `Node`, blackboard: *Blackboard*, scene_root: `Node`)
`bool`	*is_descendant_of*(task: *BTTask*) const
`bool`	*is_root*() const
BTTask	*next_sibling*() const
void	*print_tree*(initial_tabs: `int` = 0)
void	*remove_child*(task: *BTTask*)
void	*remove_child_at_index*(idx: `int`)

10.1.4 Property Descriptions

`Node` **agent**

- void **set_agent**(value: `Node`)
- `Node` **get_agent**()

The agent is the contextual object for the *BehaviorTree* instance. This is usually the parent of the *BTPlayer* node that utilizes the *BehaviorTree* resource.

Blackboard **blackboard**

- *Blackboard* **get_blackboard**()

Provides access to the *Blackboard*. Blackboard is used to share data among tasks of the associated *BehaviorTree*.

See *Blackboard* for additional info.

`String` **custom_name** = `""`

- VOID **set_custom_name**(value: `String`)
- `String` **get_custom_name**()

User-provided name for the task. If not empty, it is used by the editor to represent the task. See *get_task_name*.

`float` **elapsed_time**

- `float` **get_elapsed_time**()

Elapsed time since the task was "entered". See *_enter*.

Returns `0` when task is not `RUNNING`.

`Node` **scene_root**

- `Node` **get_scene_root**()

Root node of the scene the behavior tree is used in (e.g., the owner of the *BTPlayer* node). Can be uses to retrieve `NodePath` references.

Example:

```
extends BTAction

@export var node_path: NodePath

func _setup():
    var node: Node = scene_root.get_node(node_path)
```

Status **status**

- *Status* **get_status**()

Last execution *Status* returned by *_tick*.

10.1.5 Method Descriptions

VOID **_enter**() VIRTUAL

Called when task is "entered", i.e. when task is executed while not having a `RUNNING` *status*.

It is called before *_tick* in the execution order. This method is used when preparation is needed before main work begins, usually when it takes more than one tick to finish the task. See also *execute*.

VOID **_exit**() VIRTUAL

Called when task is "exited", i.e. after *_tick* returns `SUCCESS` or `FAILURE` status. See also *execute*.

`String` **_generate_name**() VIRTUAL CONST

Called to generate a display name for the task unless *custom_name* is set. See *get_task_name*.

`PackedStringArray` **_get_configuration_warnings**() VIRTUAL CONST

The string returned by this method is shown as a warning message in the behavior tree editor. Any task script that overrides this method must include `@tool` annotation at the top of the file.

VOID **_setup**() VIRTUAL

Called to initialize a task during initialization step. It is called only once before the task's first execution tick. This method allows you to set up any necessary state or configurations for the task before it begins executing.

Status **_tick**(delta: `float`) VIRTUAL

Called when task is "ticked", i.e. executed by *BTPlayer* or *BTState* during an update.

Returns execution status as defined in *Status*.

Note: Tasks perform their main function by implementing this method.

VOID **abort**()

Resets the task and its children recursively. If a task is in the `RUNNING` state, it is exited and its status is reset to `FRESH`.

VOID **add_child**(task: *BTTask*)

Adds a child task. The `task` is placed at the end of the children list.

VOID **add_child_at_index**(task: *BTTask*, idx: `int`)

Adds a child task. The `task` is placed at `idx` position in the children list.

BTTask **clone**() CONST

Duplicates the task and its children, copying the exported members. Sub-resources are shared for efficiency, except for *BBParam* subtypes, which are always copied. Used by the editor to instantiate *BehaviorTree* and copy-paste tasks.

BehaviorTree **editor_get_behavior_tree**()

Returns the behavior tree that owns this task. This is only available in the editor.

Status **execute**(delta: `float`)

Performs task's execution. The execution follows a specific sequence:

- If task's current *status* is not `RUNNING`, the *_enter* method is called first.
- Next, the *_tick* method is called next to perform the task's work.
- If the *_tick* method returns `SUCCESS` or `FAILURE` status, the *_exit* method will be called next as part of the execution cleanup.

BTTask **get_child**(idx: `int`) CONST

Returns a child task by specifying its index.

`int` **get_child_count**() CONST

Returns the number of child tasks.

`int` **get_child_count_excluding_comments**() CONST

Returns the number of child tasks not counting *BTComment* tasks.

`int` **get_index**() CONST

Returns the task's position in the behavior tree branch. Returns `-1` if the task doesn't belong to a task tree, i.e. doesn't have a parent.

BTTask **get_parent**() CONST

Returns the task's parent.

BTTask **get_root**() CONST

Returns the root task of the behavior tree.

`String` **get_task_name**()

The string returned by this method is used to represent the task in the editor.

Method *_generate_name* is called to generate a display name for the task unless *custom_name* is set.

`bool` **has_child**(task: *BTTask*) CONST

Returns `true` if `task` is a child of this task.

VOID **initialize**(agent: `Node`, blackboard: *Blackboard*, scene_root: `Node`)

Initilizes the task. Assigns *agent* and *blackboard*, and calls *_setup* for the task and its children.

The method is called recursively for each child task. `scene_root` should be the root node of the scene the behavior tree is used in (e.g., the owner of the node that contains the behavior tree).

`bool` **is_descendant_of**(task: *BTTask*) CONST

Returns `true` if this task is a descendant of `task`. In other words, this task must be a child of `task` or one of its children or grandchildren.

`bool` **is_root**() CONST

Returns `true` if this task is the root task of its behavior tree. A behavior tree can have only one root task.

BTTask **next_sibling**() CONST

Returns the next task after this task in the parent's children list.

Returns `null` if this task has no parent or it is the last child in the parent's children list.

VOID **print_tree**(initial_tabs: `int` = 0)

Prints the subtree that starts with this task to the console.

VOID **remove_child**(task: *BTTask*)

Removes `task` from children.

VOID **remove_child_at_index**(idx: `int`)

Removes a child task at a specified index from children.

10.2 BTPlayer

Inherits:

Player of *BehaviorTree* resources.

10.2.1 Description

BTPlayer node is used to instantiate and play *BehaviorTree* resources at runtime. During initialization, the behavior tree instance is given references to the agent, the *blackboard*, and the current scene root. The agent can be specified by the *agent_node* property (defaults to the BTPlayer's parent node).

For an introduction to behavior trees, see *BehaviorTree*.

10.2.2 Properties

`bool`	*active*	`true`
`NodePath`	*agent_node*	`NodePath("..")`
BehaviorTree	*behavior_tree*	
Blackboard	*blackboard*	
BlackboardPlan	*blackboard_plan*	
`bool`	*monitor_performance*	`false`
UpdateMode	*update_mode*	`1`

10.2.3 Methods

BTInstance	*get_bt_instance*()
VOID	*restart*()
VOID	*set_bt_instance*(bt_instance: *BTInstance*)
VOID	*set_scene_root_hint*(scene_root: `Node`)
VOID	*update*(delta: `float`)

10.2.4 Signals

behavior_tree_finished(status: `int`)

Deprecated: Use *updated* signal instead.

Emitted when the behavior tree has finished executing and returned `SUCCESS` or `FAILURE`.

Argument `status` holds the status returned by the behavior tree. See *Status*.

---

updated(status: `int`)

Emitted when BTPlayer has finished the behavior tree update.

Argument `status` holds the status returned by the behavior tree. See *Status*.

10.2.5 Enumerations

enum **UpdateMode**:

UpdateMode **IDLE** = `0`

Execute behavior tree during the idle process.

UpdateMode **PHYSICS** = `1`

Execute behavior tree during the physics process.

UpdateMode **MANUAL** = `2`

Behavior tree is executed manually by calling *update*.

10.2.6 Property Descriptions

`bool` **active** = `true`

- VOID **set_active**(value: `bool`)
- `bool` **get_active**()

If `true`, the behavior tree will be executed during update.

`NodePath` **agent_node** = `NodePath("..")`

- VOID **set_agent_node**(value: `NodePath`)

- `NodePath` **get_agent_node**()

Path to the node that will be used as the agent. Setting it after instantiation will have no effect.

BehaviorTree **behavior_tree**

- VOID **set_behavior_tree**(value: *BehaviorTree*)
- *BehaviorTree* **get_behavior_tree**()

BehaviorTree resource to instantiate and execute at runtime.

Blackboard **blackboard**

- VOID **set_blackboard**(value: *Blackboard*)
- *Blackboard* **get_blackboard**()

Holds data shared by the behavior tree tasks. See *Blackboard*.

BlackboardPlan **blackboard_plan**

- VOID **set_blackboard_plan**(value: *BlackboardPlan*)
- *BlackboardPlan* **get_blackboard_plan**()

Stores and manages variables that will be used in constructing new *Blackboard* instances.

`bool` **monitor_performance** = `false`

- VOID **set_monitor_performance**(value: `bool`)
- `bool` **get_monitor_performance**()

If `true`, adds a performance monitor to "Debugger->Monitors" for each instance of this **BTPlayer** node.

UpdateMode **update_mode** = 1

- VOID **set_update_mode**(value: *UpdateMode*)
- *UpdateMode* **get_update_mode**()

Determines when the behavior tree is executed. See *UpdateMode*.

10.2.7 Method Descriptions

BTInstance **get_bt_instance**()

Returns the behavior tree instance.

VOID **restart**()

Resets the behavior tree's execution. Each running task will be aborted and the next tree execution will start anew. This method does not reset *Blackboard*.

VOID **set_bt_instance**(bt_instance: *BTInstance*)

Sets the *BTInstance* to play. This method is useful when you want to switch to a different behavior tree instance at runtime. See also *BehaviorTree.instantiate*.

VOID **set_scene_root_hint**(scene_root: `Node`)

Sets the `Node` that will be used as the scene root for the newly instantiated behavior tree. Should be called before the **BTPlayer** is added to the scene tree (before `NOTIFICATION_READY`). This is typically useful when creating **BTPlayer** nodes dynamically from code.

VOID **update**(delta: `float`)

Executes the root task of the behavior tree instance if *active* is `true`. Call this method when *update_mode* is set to *MANUAL*. When *update_mode* is not *MANUAL*, the *update* will be called automatically. See *UpdateMode*.

10.3 Blackboard

Inherits:

A key/value storage for sharing among *LimboHSM* states and *BehaviorTree* tasks.

10.3.1 Description

Blackboard is where data is stored and shared between states in the *LimboHSM* system and tasks in a *BehaviorTree*. Each state and task in the *BehaviorTree* can access this Blackboard, allowing them to read and write data. This makes it easy to share information between different actions and behaviors.

Blackboard can also act as a parent scope for another Blackboard. If a specific variable is not found in the active scope, it looks in the parent Blackboard to find it. A parent Blackboard can itself have its own parent scope, forming what we call a "blackboard scope chain." Importantly, there is no limit to how many Blackboards can be in this chain, and the Blackboard doesn't modify values in the parent scopes.

New scopes can be created using the *BTNewScope* and *BTSubtree* decorators. Additionally, a new scope is automatically created for any *LimboState* that has defined non-empty Blackboard data or for any root-level *LimboHSM* node.

10.3.2 Methods

void	*bind_var_to_property*(var_name: StringName, object: Object, property: StringName, create: bool = false)
void	*clear*()
void	*erase_var*(var_name: StringName)
Blackboard	*get_parent*() const
Variant	*get_var*(var_name: StringName, default: Variant = null, complain: bool = true) const
Dictionary	*get_vars_as_dict*() const
bool	*has_var*(var_name: StringName) const
void	*link_var*(var_name: StringName, target_blackboard: *Blackboard*, target_var: StringName, create: bool = false)
Array[StringName]	*list_vars*() const
void	*populate_from_dict*(dictionary: Dictionary)
void	*print_state*() const
void	*set_parent*(blackboard: *Blackboard*)
void	*set_var*(var_name: StringName, value: Variant)
Blackboard	*top*() const
void	*unbind_var*(var_name: StringName)

10.3.3 Method Descriptions

void **bind_var_to_property**(var_name: StringName, object: Object, property: StringName, create: bool = false)

Establish a binding between a variable and the object's property specified by `property` and `object`. Changes to the variable update the property, and vice versa. If `create` is `true`, the variable will be created if it doesn't exist.

---

void **clear**()

Removes all variables from the Blackboard. Parent scopes are not affected.

---

void **erase_var**(var_name: StringName)

Removes a variable by its name.

Blackboard **get_parent**() const

Returns a Blackboard that serves as the parent scope for this instance.

Variant **get_var**(var_name: StringName, default: Variant = null, complain: bool = true) const

Returns variable value or `default` if variable doesn't exist. If `complain` is `true`, an error will be printed if variable doesn't exist. If the variable doesn't exist in the current **Blackboard** scope, it will look in the parent scope **Blackboard** to find it.

Dictionary **get_vars_as_dict**() const

Returns all variables in the Blackboard as a dictionary. Keys are the variable names, values are the variable values. Parent scopes are not included.

`bool` **has_var**(var_name: `StringName`) CONST

Returns `true` if the Blackboard contains the `var_name` variable, including the parent scopes.

VOID **link_var**(var_name: `StringName`, target_blackboard: *Blackboard*, target_var: `StringName`, create: `bool` = false)

Links a variable to another Blackboard variable. If a variable is linked to another variable, their state will always be identical, and any change to one will be reflected in the other. If `create` is `true`, the variable will be created if it doesn't exist.

You can use this method to link a variable in the current scope to a variable in another scope, or in another Blackboard instance. A variable can only be linked to one other variable. Calling this method again will overwrite the previous link. However, it is possible to link to the same variable from multiple different variables.

Array[`StringName`] **list_vars**() CONST

Returns all variable names in the Blackboard. Parent scopes are not included.

VOID **populate_from_dict**(dictionary: `Dictionary`)

Fills the Blackboard with multiple variables from a dictionary. The dictionary keys must be variable names and the dictionary values must be variable values. Keys must be StringName or String.

VOID **print_state**() CONST

Prints the values of all variables in each scope.

VOID **set_parent**(blackboard: *Blackboard*)

Assigns the parent scope. If a value isn't in the current Blackboard scope, it will look in the parent scope Blackboard to find it.

VOID **set_var**(var_name: `StringName`, value: `Variant`)

Assigns a value to a variable in the current Blackboard scope. If the variable doesn't exist, it will be created. If the variable already exists in the parent scope, the parent scope value will NOT be changed.

Blackboard **top**() CONST

Returns the topmost **Blackboard** in the scope chain.

VOID **unbind_var**(var_name: `StringName`)

Remove binding from a variable.

10.4 BBParam

Inherits:

Inherited By: *BBAabb*, *BBArray*, *BBBasis*, *BBBool*, *BBByteArray*, *BBColor*, *BBColorArray*, *BBDictionary*, *BBFloat*, *BBFloat32Array*, *BBFloat64Array*, *BBInt*, *BBInt32Array*, *BBInt64Array*, *BBNode*, *BBPlane*, *BBProjection*, *BBQuaternion*, *BBRect2*, *BBRect2i*, *BBString*, *BBStringArray*, *BBStringName*, *BBTransform2D*, *BBTransform3D*, *BBVariant*, *BBVector2*, *BBVector2Array*, *BBVector2i*, *BBVector3*, *BBVector3Array*, *BBVector3i*, *BBVector4*, *BBVector4i*

A base class for LimboAI typed parameters.

10.4.1 Description

A base class for LimboAI typed parameters, with the ability to reference a *Blackboard* variable or hold a raw value of a specific Variant.Type.

Note: Don't instantiate. Use specific subtypes instead.

10.4.2 Properties

`Variant`	*saved_value*	`null`
ValueSource	*value_source*	`0`
`StringName`	*variable*	

10.4.3 Methods

Variant.Type	*get_type*() CONST
`Variant`	*get_value*(scene_root: `Node`, blackboard: *Blackboard*, default: `Variant` = null)

10.4.4 Enumerations

enum **ValueSource**:

ValueSource **SAVED_VALUE** = `0`

The value is stored directly within the BBParam resource.

ValueSource **BLACKBOARD_VAR** = 1

The value is referenced by a variable name and retrieved from the *Blackboard*. The variable name is stored within the BBParam resource.

10.4.5 Property Descriptions

`Variant` **saved_value** = `null`

- VOID **set_saved_value**(value: `Variant`)
- `Variant` **get_saved_value**()

Stores the parameter value when *value_source* is set to *SAVED_VALUE*. The data type of the value is determined by *get_type*.

ValueSource **value_source** = `0`

- VOID **set_value_source**(value: *ValueSource*)
- *ValueSource* **get_value_source**()

Specifies the source of the value for BBParam. See *ValueSource*.

`StringName` **variable**

- VOID **set_variable**(value: `StringName`)
- `StringName` **get_variable**()

Stores the name of a *Blackboard* variable when *value_source* is set to *BLACKBOARD_VAR*.

10.4.6 Method Descriptions

Variant.Type **get_type**() CONST

Returns the expected data type of the parameter.

`Variant` **get_value**(scene_root: `Node`, blackboard: *Blackboard*, default: `Variant` = null)

Returns the value of the parameter.

10.5 LimboState

Inherits:

Inherited By: *BTState*, *LimboHSM*

A state node for Hierarchical State Machines (HSM).

10.5.1 Description

A LimboAI state node for Hierarchical State Machines (HSM).

You can create your state behavior by extending this class. To implement your state logic, you can override *_enter*, *_exit*, *_setup*, and *_update*. Alternatively, you can delegate state implementation to external methods using the `call_on_*` methods.

For additional details on state machines, refer to *LimboHSM*.

10.5.2 Properties

`StringName`	*EVENT_FINISHED*
`Node`	*agent*
Blackboard	*blackboard*
BlackboardPlan	*blackboard_plan*

10.5.3 Methods

void	*_enter*() virtual
void	*_exit*() virtual
void	*_setup*() virtual
void	*_update*(delta: `float`) virtual
void	*add_event_handler*(event: `StringName`, handler: `Callable`)
LimboState	*call_on_enter*(callable: `Callable`)
LimboState	*call_on_exit*(callable: `Callable`)
LimboState	*call_on_update*(callable: `Callable`)
void	*clear_guard*()
`bool`	*dispatch*(event: `StringName`, cargo: `Variant` = null)
LimboState	*get_root*() const
`bool`	*is_active*() const
LimboState	*named*(name: `String`)
void	*set_guard*(guard_callable: `Callable`)

10.5.4 Signals

entered()

Emitted when the state is entered.

---

exited()

Emitted when the state is exited.

---

setup()

Emitted when the state is initialized.

---

updated(delta: `float`)

Emitted when the state is updated.

10.5.5 Property Descriptions

`StringName` **EVENT_FINISHED**

- `StringName` **event_finished**()

A commonly used event that indicates that the state has finished its work.

`Node` **agent**

- void **set_agent**(value: `Node`)
- `Node` **get_agent**()

An agent associated with the state, assigned during initialization.

Blackboard **blackboard**

- *Blackboard* **get_blackboard**()

A key/value data store shared by states within the state machine to which this state belongs.

BlackboardPlan **blackboard_plan**

- VOID **set_blackboard_plan**(value: *BlackboardPlan*)
- *BlackboardPlan* **get_blackboard_plan**()

Stores and manages variables that will be used in constructing new *Blackboard* instances.

10.5.6 Method Descriptions

VOID **_enter**() VIRTUAL

Called when the state is entered.

VOID **_exit**() VIRTUAL

Called when the state is exited. This happens on a transition to another state, and when the state machine is removed from the scene tree (e.g., when the node is freed with Node.queue_free or the scene changes). Due to implementation details, *_exit* will not be called on Object.free!

VOID **_setup**() VIRTUAL

Called once during initialization. Use this method to initialize your state.

VOID **_update**(delta: `float`) VIRTUAL

Called during the update. Implement your state's behavior with this method.

VOID **add_event_handler**(event: `StringName`, handler: `Callable`)

Registers a `handler` to be called when `event` is dispatched. The handler function should have the following signature:

```
func my_event_handler(cargo=null) -> bool:
```

If the handler returns `true`, the event will be consumed. Cargo is an optional parameter that can be passed to the handler. See also *dispatch*.

LimboState **call_on_enter**(callable: `Callable`)

A chained method that connects the *entered* signal to a `callable`.

LimboState **call_on_exit**(callable: `Callable`)

A chained method that connects the *exited* signal to a `callable`.

LimboState **call_on_update**(callable: `Callable`)

A chained method that connects the *updated* signal to a `callable`.

VOID **clear_guard**()

Clears the guard function, removing the `Callable` previously set by *set_guard*.

`bool` **dispatch**(event: `StringName`, cargo: `Variant` = null)

Recursively dispatches a state machine event named `event` with an optional argument `cargo`. Returns `true` if the event was consumed.

Events propagate from the leaf state to the root state, and propagation stops as soon as any state consumes the event. States will consume the event if they have a related transition or event handler. For more information on event handlers, see *add_event_handler*.

LimboState **get_root**() CONST

Returns the root **LimboState**.

`bool` **is_active**() CONST

Returns `true` if it is currently active, meaning it is the active substate of the parent *LimboHSM*.

LimboState **named**(name: `String`)

A chained method for setting the name of this state.

VOID **set_guard**(guard_callable: `Callable`)

Sets the guard function, which is a function called each time a transition to this state is considered. If the function returns `false`, the transition will be disallowed.

10.6 LimboHSM

Inherits: *LimboState*

Event-based Hierarchical State Machine (HSM).

10.6.1 Description

Event-based Hierarchical State Machine (HSM) that manages *LimboState* instances and facilitates transitions between them. LimboHSM is a *LimboState* in itself and can also serve as a child of another LimboHSM node.

10.6.2 Properties

LimboState	*ANYSTATE*	
LimboState	*initial_state*	
UpdateMode	*update_mode*	1

10.6.3 Methods

VOID	*add_transition*(from_state: *LimboState*, to_state: *LimboState*, event: `StringName`, guard: `Callable` = Callable())
VOID	*change_active_state*(state: *LimboState*)
LimboState	*get_active_state*() CONST
LimboState	*get_leaf_state*() CONST
LimboState	*get_previous_active_state*() CONST
`bool`	*has_transition*(from_state: *LimboState*, event: `StringName`) CONST
VOID	*initialize*(agent: `Node`, parent_scope: *Blackboard* = null)
VOID	*remove_transition*(from_state: *LimboState*, event: `StringName`)
VOID	*set_active*(active: `bool`)
VOID	*update*(delta: `float`)

10.6.4 Signals

active_state_changed(current: *LimboState*, previous: *LimboState*)

Emitted when the currently active substate is switched to a different substate.

10.6.5 Enumerations

enum **UpdateMode**:

UpdateMode **IDLE** = `0`

Update the state machine during the idle process.

UpdateMode **PHYSICS** = `1`

Update the state machine during the physics process.

UpdateMode **MANUAL** = `2`

Manually update the state machine by calling *update* from a script.

10.6.6 Property Descriptions

LimboState **ANYSTATE**

- *LimboState* **anystate**()

Useful for defining a transition from any state.

LimboState **initial_state**

- VOID **set_initial_state**(value: *LimboState*)
- *LimboState* **get_initial_state**()

The substate that becomes active when the state machine is activated using the *set_active* method. If not explicitly set, the first child of the LimboHSM will be considered the initial state.

UpdateMode **update_mode** = 1

- VOID **set_update_mode**(value: *UpdateMode*)
- *UpdateMode* **get_update_mode**()

Specifies when the state machine should be updated. See *UpdateMode*.

10.6.7 Method Descriptions

VOID **add_transition**(from_state: *LimboState*, to_state: *LimboState*, event: `StringName`, guard: `Callable` = Callable())

Establishes a transition from one state to another when `event` is dispatched. Both `from_state` and `to_state` must be immediate children of this **LimboHSM**.

Optionally, a `guard` function can be specified, which must return a boolean value. If the guard function returns `false`, the transition will not occur. The guard function is called immediately before the transition is considered. For a state-wide guard function, check out *LimboState.set_guard*.

```
func my_guard() -> bool:
    return is_some_condition_met()
```

VOID **change_active_state**(state: *LimboState*)

Changes the currently active substate to `state`. If `state` is already active, it will be exited and reentered.

`state` must be a child of this **LimboHSM**.

LimboState **get_active_state**() CONST

Returns the currently active substate.

LimboState **get_leaf_state**() CONST

Returns the currently active leaf state within the state machine.

LimboState **get_previous_active_state**() CONST

Returns the previously active substate.

`bool` **has_transition**(from_state: *LimboState*, event: `StringName`) CONST

Returns `true` if there is a transition from `from_state` for a given `event`.

VOID **initialize**(agent: `Node`, parent_scope: *Blackboard* = null)

Initiates the state and calls *LimboState._setup* for both itself and all substates.

VOID **remove_transition**(from_state: *LimboState*, event: `StringName`)

Removes a transition from a state associated with specific `event`.

VOID **set_active**(active: `bool`)

When set to `true`, switches the state to *initial_state* and activates state processing according to *update_mode*.

VOID **update**(delta: `float`)

Calls *LimboState._update* on itself and the active substate, with the call cascading down to the leaf state. This method is automatically triggered if *update_mode* is not set to *MANUAL*.

10.7 BTState

Inherits: *LimboState*

A state node for *LimboHSM* that hosts a *BehaviorTree*.

10.7.1 Description

BTState is a *LimboState* node that manages a *BehaviorTree* to provide behavior logic for the state. It instantiates and runs the *BehaviorTree* resource, dispatching a state machine event upon `SUCCESS` or `FAILURE`. Event names are customizable through *success_event* and *failure_event*. For further details on state machine events, see *LimboState.dispatch*.

10.7.2 Properties

BehaviorTree	*behavior_tree*	
`StringName`	*failure_event*	`&"failure"`
`bool`	*monitor_performance*	`false`
`StringName`	*success_event*	`&"success"`

10.7.3 Methods

BTInstance	*get_bt_instance*() CONST
VOID	*set_scene_root_hint*(scene_root: `Node`)

10.7.4 Property Descriptions

BehaviorTree **behavior_tree**

- void **set_behavior_tree**(value: *BehaviorTree*)
- *BehaviorTree* **get_behavior_tree**()

A *BehaviorTree* resource that defines state behavior.

`StringName` **failure_event** = `&"failure"`

- void **set_failure_event**(value: `StringName`)
- `StringName` **get_failure_event**()

HSM event that will be dispatched when the behavior tree results in `FAILURE`. See *LimboState.dispatch*.

`bool` **monitor_performance** = `false`

- void **set_monitor_performance**(value: `bool`)
- `bool` **get_monitor_performance**()

If `true`, adds a performance monitor to "Debugger->Monitors" for each instance of this **BTState** node.

`StringName` **success_event** = `&"success"`

- void **set_success_event**(value: `StringName`)
- `StringName` **get_success_event**()

HSM event that will be dispatched when the behavior tree results in `SUCCESS`. See *LimboState.dispatch*.

10.7.5 Method Descriptions

BTInstance **get_bt_instance**() const

Returns the behavior tree instance.

---

void **set_scene_root_hint**(scene_root: `Node`)

Sets the `Node` that will be used as the scene root for the newly instantiated behavior tree. Should be called before the state machine is initialized. This is typically useful when creating **BTState** nodes dynamically from code.

Full class reference is available in the side bar.

CHAPTER

ELEVEN

BBAABB

Inherits: *BBParam*

AABB-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWELVE

BBARRAY

Inherits: *BBParam*

Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTEEN

BBBASIS

Inherits: *BBParam*

Basis-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FOURTEEN

BBBOOL

Inherits: *BBParam*

Bool-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FIFTEEN

BBBYTEARRAY

Inherits: *BBParam*

ByteArray-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

SIXTEEN

BBCOLOR

Inherits: *BBParam*

Color-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

SEVENTEEN

BBCOLORARRAY

Inherits: *BBParam*

ColorArray-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

EIGHTEEN

BBDICTIONARY

Inherits: *BBParam*

Dictionary-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

NINETEEN

BBFLOAT

Inherits: *BBParam*

Float-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTY

BBFLOAT32ARRAY

Inherits: *BBParam*

PackedFloat32Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYONE

BBFLOAT64ARRAY

Inherits: *BBParam*

PackedFloat64Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYTWO

BBINT

Inherits: *BBParam*

Integer-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYTHREE

BBINT32ARRAY

Inherits: *BBParam*

PackedInt32Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYFOUR

BBINT64ARRAY

Inherits: *BBParam*

PackedInt64Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYFIVE

BBNODE

Inherits: *BBParam*

Node-type parameter for *BehaviorTree* tasks. See *BBParam*.

25.1 Description

Node-type parameter intended for use with *BehaviorTree* tasks. See *BBParam*.

If the source is a blackboard variable, it allows any type extended from `Object`.

CHAPTER

TWENTYSIX

BBPLANE

Inherits: *BBParam*

Plane-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYSEVEN

BBPROJECTION

Inherits: *BBParam*

Projection-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYEIGHT

BBQUATERNION

Inherits: *BBParam*

Quaternion-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

TWENTYNINE

BBRECT2

Inherits: *BBParam*

Rect2-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTY

BBRECT2I

Inherits: *BBParam*

Rect2i-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYONE

BBSTRING

Inherits: *BBParam*

String-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYTWO

BBSTRINGARRAY

Inherits: *BBParam*

StringArray-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYTHREE

BBSTRINGNAME

Inherits: *BBParam*

StringName-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYFOUR

BBTRANSFORM2D

Inherits: *BBParam*

Transform2D-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYFIVE

BBTRANSFORM3D

Inherits: *BBParam*

Transform3D-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYSIX

BBVARIANT

Inherits: *BBParam*

Variant-type parameter for *BehaviorTree* tasks. See *BBParam*.

36.1 Description

Variant-type parameter intended for use with *BehaviorTree* tasks. See *BBParam*.

The data type is specified by *type*.

36.2 Properties

Variant.Type	*type*	`0`

36.3 Property Descriptions

Variant.Type **type** = `0`

- VOID **set_type**(value: Variant.Type)
- Variant.Type **get_type**()

Specified Variant.Type for the parameter value.

CHAPTER

THIRTYSEVEN

BBVECTOR2

Inherits: *BBParam*

Vector2-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYEIGHT

BBVECTOR2ARRAY

Inherits: *BBParam*

Vector2Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

THIRTYNINE

BBVECTOR2I

Inherits: *BBParam*

Vector2i-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FORTY

BBVECTOR3

Inherits: *BBParam*

Vector3-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FORTYONE

BBVECTOR3ARRAY

Inherits: *BBParam*

Vector3Array-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FORTYTWO

BBVECTOR3I

Inherits: *BBParam*

Vector3i-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FORTYTHREE

BBVECTOR4

Inherits: *BBParam*

Vector4-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FORTYFOUR

BBVECTOR4I

Inherits: *BBParam*

Vector4i-type parameter for *BehaviorTree* tasks. See *BBParam*.

CHAPTER

FORTYFIVE

BEHAVIORTREE

Inherits:

Contains Behavior Tree data.

45.1 Description

Behavior Trees are hierarchical structures used to model and control the behavior of agents in a game (e.g., characters, enemies, entities). They are designed to make it easier to create complex and highly modular behaviors for your games.

Behavior Trees are composed of tasks that represent specific actions or decision-making rules. Tasks can be broadly categorized into two main types: control tasks and leaf tasks. Control tasks determine the execution flow within the tree. They include *BTSequence*, *BTSelector*, and *BTInvert*. Leaf tasks represent specific actions to perform, like moving or attacking, or conditions that need to be checked. The *BTTask* class provides the foundation for various building blocks of the Behavior Trees. BT tasks can share data with the help of *Blackboard*. See *BTTask.blackboard* and *Blackboard*.

Note: To create your own actions, extend the *BTAction* class.

The BehaviorTree is executed from the root task and follows the rules specified by the control tasks, all the way down to the leaf tasks, which represent the actual actions that the agent should perform or conditions that should be checked. Each task returns a status when it is executed. It can be `SUCCESS`, `RUNNING`, or `FAILURE`. These statuses determine how the tree progresses. They are defined in *Status*.

Behavior Trees handle conditional logic using condition tasks. These tasks check for specific conditions and return either `SUCCESS` or `FAILURE` based on the state of the agent or its environment (e.g., "IsLowOnHealth", "IsTargetInSight"). Conditions can be used together with *BTSequence* and *BTSelector* to craft your decision-making logic.

Note: To create your own conditions, extend the *BTCondition* class.

Check out the *BTTask* class, which provides the foundation for various building blocks of Behavior Trees.

45.2 Properties

BlackboardPlan	*blackboard_plan*	
`String`	*description*	`""`

45.3 Methods

BehaviorTree	*clone*() const
void	*copy_other*(other: *BehaviorTree*)
BTTask	*get_root_task*() const
BTInstance	*instantiate*(agent: Node, blackboard: *Blackboard*, instance_owner: Node, custom_scene_root: Node = null) const
void	*set_root_task*(task: *BTTask*)

45.4 Signals

plan_changed()

Emitted when the *BlackboardPlan* changes.

45.5 Property Descriptions

BlackboardPlan **blackboard_plan**

- void **set_blackboard_plan**(value: *BlackboardPlan*)
- *BlackboardPlan* **get_blackboard_plan**()

Stores and manages variables that will be used in constructing new *Blackboard* instances.

String **description** = ""

- void **set_description**(value: String)
- String **get_description**()

User-provided description of the **BehaviorTree**.

45.6 Method Descriptions

BehaviorTree **clone**() const

Makes a copy of the BehaviorTree resource.

void **copy_other**(other: *BehaviorTree*)

Become a copy of another behavior tree.

BTTask **get_root_task**() const

Returns the root task of the BehaviorTree resource.

BTInstance **instantiate**(agent: `Node`, blackboard: *Blackboard*, instance_owner: `Node`, custom_scene_root: `Node = null`) CONST

Instantiates the behavior tree and returns *BTInstance*. `instance_owner` should be the scene node that will own the behavior tree instance. This is typically a *BTPlayer*, *BTState*, or a custom player node that controls the behavior tree execution. Make sure to pass a *Blackboard* with values populated from *blackboard_plan*. See also *BlackboardPlan.populate_blackboard* & *BlackboardPlan.create_blackboard*.

If `custom_scene_root` is not `null`, it will be used as the scene root for the newly instantiated behavior tree; otherwise, the scene root will be set to `instance_owner.owner`. Scene root is essential for *BBNode* instances to work properly.

VOID **set_root_task**(task: *BTTask*)

Assigns a new root task to the **BehaviorTree** resource.

CHAPTER

FORTYSIX

BEHAVIORTREEDATA

Inherits:

Represents current state of a *BehaviorTree* instance.

46.1 Description

This class is used by the LimboAI debugger for the serialization and deserialization of *BehaviorTree* instance data.

Additionally, it can be used with *BehaviorTreeView* to visualize the current state of a *BehaviorTree* instance. It is meant to be utilized in custom in-game tools.

46.2 Methods

BehaviorTreeData	*create_from_bt_instance*(bt_instance: *BTInstance*) STATIC (This method doesn't need an instance to be called, so it can be called directly using the class name.)

46.3 Method Descriptions

BehaviorTreeData **create_from_bt_instance**(bt_instance: *BTInstance*) STATIC

Returns current state of the `bt_instance` encoded as a **BehaviorTreeData**, suitable for use with *BehaviorTreeView*.

Behavior tree instance can be acquired with *BTPlayer.get_bt_instance*.

CHAPTER

FORTYSEVEN

BEHAVIORTREEVIEW

Inherits:

Visualizes the current state of a *BehaviorTree* instance.

47.1 Description

Visualizes the current state of a *BehaviorTree* instance. See also *BehaviorTreeData*.

47.2 Properties

int	*update_interval_msec*	0

47.3 Methods

void	*clear*()
void	*update_tree*(behavior_tree_data: *BehaviorTreeData*)

47.4 Signals

task_selected(type_name: `String`, script_path: `String`)

Emitted when a task item is selected in **BehaviorTreeView**.

47.5 Property Descriptions

`int` **update_interval_msec** = `0`

- void **set_update_interval_msec**(value: `int`)
- `int` **get_update_interval_msec**()

Minimum delay between two updates (in milliseconds). Set to higher values for a lower CPU load.

47.6 Method Descriptions

void **clear**()

Clears the tree view.

---

void **update_tree**(behavior_tree_data: *BehaviorTreeData*)

Updates the representation of a *BehaviorTree* instance. See also *BehaviorTreeData*.

CHAPTER

FORTYEIGHT

BLACKBOARDPLAN

Inherits:

Stores and manages variables that will be used in constructing new *Blackboard* instances.

48.1 Properties

`bool`	*prefetch_nodepath_vars*	`true`

48.2 Methods

Blackboard	*create_blackboard*(prefetch_root: `Node`, parent_scope: *Blackboard* = null, prefetch_root_for_base_plan: `Node` = null)
Blackboard-Plan	*get_base_plan*() CONST
`Callable`	*get_parent_scope_plan_provider*() CONST
`bool`	*is_derived*() CONST
VOID	*populate_blackboard*(blackboard: *Blackboard*, overwrite: `bool`, prefetch_root: `Node`, prefetch_root_for_base_plan: `Node` = null)
VOID	*set_base_plan*(blackboard_plan: *BlackboardPlan*)
VOID	*set_parent_scope_plan_provider*(callable: `Callable`)
VOID	*sync_with_base_plan*()

48.3 Property Descriptions

`bool` **prefetch_nodepath_vars** = `true`

- VOID **set_prefetch_nodepath_vars**(value: `bool`)
- `bool` **is_prefetching_nodepath_vars**()

Enables or disables `NodePath` variable prefetching. If `true`, `NodePath` values will be replaced with node instances when the *Blackboard* is created.

48.4 Method Descriptions

Blackboard **create_blackboard**(prefetch_root: `Node`, parent_scope: *Blackboard* = null, prefetch_root_for_base_plan: `Node` = null)

Constructs a new instance of a *Blackboard* using this plan. If `NodePath` prefetching is enabled, `prefetch_root` will be used to retrieve node instances for `NodePath` variables and substitute their values.

BlackboardPlan **get_base_plan**() CONST

Returns the base plan. See *is_derived*.

`Callable` **get_parent_scope_plan_provider**() CONST

Returns the parent scope plan provider - a callable that returns a **BlackboardPlan**.

`bool` **is_derived**() CONST

Returns `true` if this plan is derived from another, i.e., the base plan is not `null`. A derived plan can only contain variables that are present in the base plan, and only variable values can be different.

VOID **populate_blackboard**(blackboard: *Blackboard*, overwrite: `bool`, prefetch_root: `Node`, prefetch_root_for_base_plan: `Node` = null)

Populates `blackboard` with the variables from this plan. If `overwrite` is `true`, existing variables with the same names will be overwritten. If `NodePath` prefetching is enabled, `prefetch_root` will be used to retrieve node instances for `NodePath` variables and substitute their values.

VOID **set_base_plan**(blackboard_plan: *BlackboardPlan*)

Sets the base plan. If assigned, this plan will be derived from the base plan.

Use with caution, as it will remove variables not present in the base plan. Only use this for custom tooling.

VOID **set_parent_scope_plan_provider**(callable: `Callable`)

Sets the parent scope plan provider - a callable that returns a **BlackboardPlan**. Used to provide hints in the inspector. When set, mapping feature becomes available.

VOID **sync_with_base_plan**()

Synchronizes this plan with the base plan: removes variables not present in the base plan, and updates type information. Only use this for custom tooling.

CHAPTER

FORTYNINE

BT

Inherits:

Inherited By: *BTTask*

Base of *BTTask*.

49.1 Enumerations

enum **Status**:

Status **FRESH** = `0`

Task wasn't executed yet or it was aborted and reset.

Status **RUNNING** = `1`

Task is being performed and hasn't finished yet.

Status **FAILURE** = `2`

Task has finished with failure.

Status **SUCCESS** = `3`

Task has finished with success.

CHAPTER

FIFTY

BTACTION

Inherits: *BTTask* < *BT*

Inherited By: *BTAwaitAnimation*, *BTCallMethod*, *BTConsolePrint*, *BTEvaluateExpression*, *BTFail*, *BTPauseAnimation*, *BTPlayAnimation*, *BTRandomWait*, *BTSetAgentProperty*, *BTSetVar*, *BTStopAnimation*, *BTWait*, *BTWaitTicks*

Base class for all *BehaviorTree* actions.

50.1 Description

Base class for all actions within a *BehaviorTree*. You can create your own actions by extending the **BTAction** class.

Represents a specific behavior or action in a *BehaviorTree* that an agent should execute. Actions are the lowest level of the *BehaviorTree* hierarchy and are responsible for performing the actual work required to achieve a goal. Actions do not have child tasks.

A single action can perform a task within one or multiple ticks. If it takes more than one tick to complete the task, the action should return `RUNNING` status. When the task is finished, the action returns either `SUCCESS` or `FAILURE` to indicate the outcome.

BTALWAYSFAIL

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that converts `SUCCESS` into `FAILURE`.

CHAPTER

FIFTYTWO

BTALWAYSSUCCEED

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that converts `FAILURE` into `SUCCESS`.

CHAPTER

FIFTYTHREE

BTAWAITANIMATION

Inherits: *BTAction* < *BTTask* < *BT*

BT action that waits for an animation to finish playing.

53.1 Description

BTAwaitAnimation action waits for an animation on the specified `AnimationPlayer` node to finish playing and returns `SUCCESS`.

Returns `SUCCESS` if the specified animation has finished playing or if the specified animation is not currently playing.

Returns `FAILURE` if the specified animation doesn't exist or if the action fails to get the `AnimationPlayer` node.

53.2 Properties

`StringName`	*animation_name*	`&""`
BBNode	*animation_player*	
`float`	*max_time*	`1.0`

53.3 Property Descriptions

`StringName` **animation_name** = `&""`

- void **set_animation_name**(value: `StringName`)
- `StringName` **get_animation_name**()

Animation's key within the `AnimationPlayer` node.

BBNode **animation_player**

- void **set_animation_player**(value: *BBNode*)
- *BBNode* **get_animation_player**()

Parameter that specifies the `AnimationPlayer` node.

`float` **max_time** = `1.0`

- VOID **set_max_time**(value: `float`)
- `float` **get_max_time**()

The maximum duration to wait for the animation to complete (in seconds). If the animation doesn't finish within this time, BTAwaitAnimation will stop waiting and return `SUCCESS`.

CHAPTER

FIFTYFOUR

BTCALLMETHOD

Inherits: *BTAction* < *BTTask* < *BT*

BT action that calls a method on a specified `Node` or `Object`.

54.1 Description

BTCallMethod action calls a *method* on the specified `Node` or `Object` instance and returns `SUCCESS`.

Returns `FAILURE` if the action encounters an issue during the method execution.

54.2 Properties

Array[*BBVariant*]	*args*	`[]`
`bool`	*args_include_delta*	`false`
`StringName`	*method*	`&""`
BBNode	*node*	
`StringName`	*result_var*	`&""`

54.3 Property Descriptions

Array[*BBVariant*] **args** = `[]`

- void **set_args**(value: Array[*BBVariant*])
- Array[*BBVariant*] **get_args**()

The arguments to be passed when calling the method.

`bool` **args_include_delta** = `false`

- void **set_include_delta**(value: `bool`)
- `bool` **is_delta_included**()

Include delta as a first parameter and shift the position of the rest of the arguments if any.

`StringName` **method** = `&""`

- void **set_method**(value: `StringName`)
- `StringName` **get_method**()

The name of the method to be called.

BBNode **node**

- void **set_node_param**(value: *BBNode*)
- *BBNode* **get_node_param**()

Specifies the `Node` or `Object` instance containing the method to be called.

`StringName` **result_var** = &""

- void **set_result_var**(value: `StringName`)
- `StringName` **get_result_var**()

if non-empty, assign the result of the method call to the blackboard variable specified by this property.

CHAPTER

FIFTYFIVE

BTCHECKAGENTPROPERTY

Inherits: *BTCondition* < *BTTask* < *BT*

BT condition that checks agent's property value.

55.1 Description

BTCheckAgentProperty examines the agent's property value and compares it to *value*.

Returns `SUCCESS` or `FAILURE` based on the *check_type*.

55.2 Properties

CheckType	*check_type*	`0`
`StringName`	*property*	`&""`
BBVariant	*value*	

55.3 Property Descriptions

CheckType **check_type** = `0`

- void **set_check_type**(value: *CheckType*)
- *CheckType* **get_check_type**()

The type of check to be performed.

`StringName` **property** = `&""`

- void **set_property**(value: `StringName`)
- `StringName` **get_property**()

Parameter that specifies the agent's property to be compared.

BBVariant **value**

- void **set_value**(value: *BBVariant*)
- *BBVariant* **get_value**()

Parameter that specifies the value against which an agent's property will be compared.

CHAPTER

FIFTYSIX

BTCHECKTRIGGER

Inherits: *BTCondition* < *BTTask* < *BT*

BT condition that checks a trigger (a boolean variable).

56.1 Description

BTCheckTrigger verifies whether the *variable* is set to `true`. If it is, the task switches it to `false` and returns `SUCCESS`. Otherwise, it returns `FAILURE`.

BTCheckTrigger can function as a "gate" within a *BTSequence*: when the trigger variable is set to `true`, it permits the execution of subsequent tasks and then changes the variable to `false`.

56.2 Properties

`StringName`	*variable*	`&""`

56.3 Property Descriptions

`StringName` **variable** = `&""`

- VOID **set_variable**(value: `StringName`)
- `StringName` **get_variable**()

A boolean variable on the blackboard used as a trigger. See also *BTTask.blackboard*.

If variable's value is `true`, **BTCheckTrigger** will switch it to `false` and return `SUCCESS`.

If variable's value is `false`, **BTCheckTrigger** will return `FAILURE`.

CHAPTER

FIFTYSEVEN

BTCHECKVAR

Inherits: *BTCondition* < *BTTask* < *BT*

BT condition that checks a variable on the blackboard.

57.1 Description

BTCheckVar evaluates the *variable* against *value* and returns `SUCCESS` or `FAILURE` based on the *check_type*.

57.2 Properties

CheckType	*check_type*	`0`
BBVariant	*value*	
`StringName`	*variable*	`&""`

57.3 Property Descriptions

CheckType **check_type** = `0`

- void **set_check_type**(value: *CheckType*)
- *CheckType* **get_check_type**()

The type of check to be performed.

BBVariant **value**

- void **set_value**(value: *BBVariant*)
- *BBVariant* **get_value**()

A parameter that specifies the value against which the *variable* will be compared.

`StringName` **variable** = `&""`

- void **set_variable**(value: `StringName`)
- `StringName` **get_variable**()

The name of the variable to check its value.

CHAPTER

FIFTYEIGHT

BTCOMMENT

Inherits: *BTTask* < *BT*

BTComment adds annotations or explanatory notes to a *BehaviorTree*.

58.1 Description

BTComment is used to add annotations or explanatory notes to a *BehaviorTree*.

Comments are not executed as part of the tree and are removed from runtime instances of the *BehaviorTree*.

CHAPTER

FIFTYNINE

BTCOMPOSITE

Inherits: *BTTask* < *BT*

Inherited By: *BTDynamicSelector*, *BTDynamicSequence*, *BTParallel*, *BTProbabilitySelector*, *BTRandomSelector*, *BTRandomSequence*, *BTSelector*, *BTSequence*

Base class for BT composite tasks.

59.1 Description

Base class for all *BehaviorTree* composite tasks. You can create your own composite tasks by extending the BTComposite class.

Composite is a control task within a *BehaviorTree* that contains one or more child tasks. It defines the structure and flow of the *BehaviorTree* by specifying how child tasks are executed. Composites can be used to group related tasks into a single unit, making it easier to manage and maintain the *BehaviorTree*. Examples of *BehaviorTree* composites include *BTSelector*, *BTSequence*, and *BTParallel*.

CHAPTER

SIXTY

BTCONDITION

Inherits: *BTTask* < *BT*

Inherited By: *BTCheckAgentProperty*, *BTCheckTrigger*, *BTCheckVar*

Base class for BT conditions.

60.1 Description

Base class for all *BehaviorTree* conditions. You can create your own conditions by extending the **BTCondition** class.

Condition is a task within a *BehaviorTree* that checks for a specific condition before executing subsequent tasks. It is often used inside composite tasks to control the execution flow. Conditions are used to verify the state of the environment, check for the presence of an enemy, or evaluate the health status of the agent. The use of condition tasks in a *BehaviorTree* can improve system efficiency and prevent unnecessary actions. However, they may not be suitable for complex decision-making processes, and too many condition tasks can make the *BehaviorTree* difficult to read.

Conditions typically don't take multiple ticks to finish and return either `SUCCESS` or `FAILURE` immediately.

CHAPTER

SIXTYONE

BTCONSOLEPRINT

Inherits: *BTAction* < *BTTask* < *BT*

BT action task that outputs text to the console.

61.1 Description

BTConsolePrint action outputs text to the console and returns `SUCCESS`. It can include placeholders for format arguments similar to GDScript's % operator.

61.2 Properties

`PackedStringArray`	*bb_format_parameters*	`PackedStringArray()`
`String`	*text*	`""`

61.3 Property Descriptions

`PackedStringArray` **bb_format_parameters** = `PackedStringArray()`

- VOID **set_bb_format_parameters**(value: `PackedStringArray`)
- `PackedStringArray` **get_bb_format_parameters**()

The values of format parameters are used as format arguments for the text that will be printed.

Note: The returned array is *copied* and any changes to it will not update the original property value. See `PackedStringArray` for more details.

`String` **text** = `""`

- VOID **set_text**(value: `String`)
- `String` **get_text**()

The text to be printed, which can include multiple placeholders to be substituted with format arguments. These placeholders follow the same format as GDScript's % operator and typically start with '%' followed by a format specifier. For instance: %s for string, %d for integer, %f for real.

CHAPTER

SIXTYTWO

BTCOOLDOWN

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that executes its child task only if *duration* time has passed since the previous execution.

62.1 Description

BTCooldown runs its child task only if *duration* time has passed since the last successful child task execution. It will only consider successful executions unless *trigger_on_failure* is set to `true`.

Returns `RUNNING`, if the child task results in `RUNNING`.

Returns `SUCCESS`, if the child task results in `SUCCESS`, and triggers the cooldown timer.

Returns `FAILURE`, if the child task results in `FAILURE` or if *duration* time didn't pass since the previous execution.

62.2 Properties

`StringName`	*cooldown_state_var*	`&""`
`float`	*duration*	`10.0`
`bool`	*process_pause*	`false`
`bool`	*start_cooled*	`false`
`bool`	*trigger_on_failure*	`false`

62.3 Property Descriptions

`StringName` **cooldown_state_var** = `&""`

- VOID **set_cooldown_state_var**(value: `StringName`)
- `StringName` **get_cooldown_state_var**()

A boolean variable used to store the cooldown state in the *Blackboard*. If left empty, the variable will be automatically generated and assigned.

If the variable's value is set to `true`, it indicates that the cooldown is activated. This feature is useful for checking the cooldown state from other parts of the tree or sharing it among different sections of the *BehaviorTree*.

`float` **duration** = `10.0`

- VOID **set_duration**(value: `float`)
- `float` **get_duration**()

Time to wait before permitting another child's execution.

`bool` **process_pause** = `false`

- VOID **set_process_pause**(value: `bool`)
- `bool` **get_process_pause**()

If `true`, process cooldown when the `SceneTree` is paused.

`bool` **start_cooled** = `false`

- VOID **set_start_cooled**(value: `bool`)
- `bool` **get_start_cooled**()

If `true`, initiate a cooldown as if the child had been executed before the first BT tick.

`bool` **trigger_on_failure** = `false`

- VOID **set_trigger_on_failure**(value: `bool`)
- `bool` **get_trigger_on_failure**()

If `true`, the cooldown will be activated if the child task also returns `FAILURE`. Otherwise, the cooldown will only be triggered when the child task returns `SUCCESS`.

CHAPTER

SIXTYTHREE

BTDECORATOR

Inherits: *BTTask* < *BT*

Inherited By: *BTAlwaysFail*, *BTAlwaysSucceed*, *BTCooldown*, *BTDelay*, *BTForEach*, *BTInvert*, *BTNewScope*, *BTProbability*, *BTRepeat*, *BTRepeatUntilFailure*, *BTRepeatUntilSuccess*, *BTRunLimit*, *BTTimeLimit*

Base class for BT decorators.

63.1 Description

Base class for all *BehaviorTree* decorators. You can create your own decorators by extending the BTDecorator class.

A decorator is a task within a *BehaviorTree* that alters the behavior of its child task. Decorators can have only one child task.

Decorators can be used to add conditions, limits, or other constraints to the execution of a task. Examples of *BehaviorTree* decorators include *BTInvert*, *BTRepeat*, and *BTCooldown*. The use of *BehaviorTree* decorators can simplify the design and implementation of complex behaviors by adding additional logic to existing tasks.

CHAPTER

SIXTYFOUR

BTDELAY

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that introduces a delay before executing its child task.

64.1 Description

BTDelay introduces a delay of *seconds* before executing its child task. Returns `RUNNING` during the delay period.

64.2 Properties

`float`	*seconds*	`1.0`

64.3 Property Descriptions

`float` **seconds** = `1.0`

- VOID **set_seconds**(value: `float`)
- `float` **get_seconds**()

Delay duration in seconds.

CHAPTER

SIXTYFIVE

BTDYNAMICSELECTOR

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that executes tasks from scratch every tick until first `SUCCESS`.

65.1 Description

BTDynamicSelector executes its child tasks sequentially, from first to last, until any child returns `SUCCESS`. Unlike *BTSelector*, it will execute tasks from the beginning every tick, reevaluating their statuses. It is quite useful when you want to retry higher-priority behaviors in every tick.

Returns `RUNNING` if a child task results in `RUNNING`. BTDynamicSelector will remember the last `RUNNING` child, but, unlike *BTSequence*, on the next execution tick, it will reexecute preceding tasks and reevaluate their return statuses. If any of the preceding tasks doesn't result in `FAILURE`, it will abort the remembered `RUNNING` task.

Returns `FAILURE` if all child tasks result in `FAILURE`.

Returns `SUCCESS` if a child task results in `SUCCESS`.

CHAPTER

SIXTYSIX

BTDYNAMICSEQUENCE

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that executes tasks from scratch every tick as long as they return `SUCCESS`.

66.1 Description

BTDynamicSequence executes its child tasks sequentially, from first to last, for as long as they return `SUCCESS`. Unlike *BTSequence*, it will execute tasks from the beginning every tick, reevaluating their statuses. It is quite useful when you want to recheck conditions preceding a long-running action during each tick and abort the `RUNNING` action when any condition results in `FAILURE`.

Returns `RUNNING` if a child task results in `RUNNING`. BTDynamicSequence will remember the last `RUNNING` child, but, unlike *BTSequence*, on the next execution tick, it will reexecute preceding tasks and reevaluate their return statuses. If any of the preceding tasks doesn't result in `SUCCESS`, it will abort the remembered `RUNNING` task.

Returns `FAILURE` if a child task results in `FAILURE`.

Returns `SUCCESS` if all child tasks result in `SUCCESS`.

CHAPTER

SIXTYSEVEN

BTEVALUATEEXPRESSION

Inherits: *BTAction* < *BTTask* < *BT*

BT action that evaluates an `Expression` against a specified `Node` or `Object`.

67.1 Description

BTEvaluateExpression action evaluates an *expression_string* on the specified `Node` or `Object` instance and returns `SUCCESS` when the `Expression` executes successfully.

Returns `FAILURE` if the action encounters an issue during the `Expression` parsing or execution.

67.2 Properties

`String`	*expression_string*	`""`
`bool`	*input_include_delta*	`false`
`PackedStringArray`	*input_names*	`PackedStringArray()`
Array[*BBVariant*]	*input_values*	`[]`
BBNode	*node*	
`StringName`	*result_var*	`&""`

67.3 Methods

Error	*parse*()

67.4 Property Descriptions

`String` **expression_string** = `""`

- VOID **set_expression_string**(value: `String`)
- `String` **get_expression_string**()

The expression string to be parsed and executed.

Warning: Call *parse* after updating *expression_string* to update the internal `Expression` as it won't be updated automatically.

`bool` **input_include_delta** = `false`

- VOID **set_input_include_delta**(value: `bool`)
- `bool` **is_input_delta_included**()

If enabled, the input variable `delta` will be added to *input_names* and *input_values*.

Warning: Call *parse* after toggling *input_include_delta* to update the internal `Expression` as it won't be updated automatically.

`PackedStringArray` **input_names** = `PackedStringArray()`

- VOID **set_input_names**(value: `PackedStringArray`)
- `PackedStringArray` **get_input_names**()

List of variable names within *expression_string* for which the user will provide values for through *input_values*.

Warning: Call *parse* after updating *input_names* to update the internal `Expression` as it won't be updated automatically.

Note: The returned array is *copied* and any changes to it will not update the original property value. See `PackedStringArray` for more details.

Array[*BBVariant*] **input_values** = `[]`

- VOID **set_input_values**(value: Array[*BBVariant*])
- Array[*BBVariant*] **get_input_values**()

List of values for variables specified in *input_names*. The values are mapped to the variables by their array index.

BBNode **node**

- VOID **set_node_param**(value: *BBNode*)
- *BBNode* **get_node_param**()

Specifies the `Node` or `Object` instance containing the method to be called.

`StringName` **result_var** = `&""`

- VOID **set_result_var**(value: `StringName`)
- `StringName` **get_result_var**()

if non-empty, assign the result of the method call to the blackboard variable specified by this property.

67.5 Method Descriptions

Error **parse**()

Calls Expression.parse considering *input_include_delta* and *input_names* and returns its error code.

CHAPTER

SIXTYEIGHT

BTFAIL

Inherits: *BTAction* < *BTTask* < *BT*

BT action that always returns `FAILURE`.

CHAPTER

SIXTYNINE

BTFOREACH

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that executes its child task for each element of an `Array`.

69.1 Description

BTForEach executes its child task for each element of an `Array`. During each iteration, the next element is stored in the specified *Blackboard* variable.

Returns `RUNNING` if the child task results in `RUNNING` or if the child task results in `SUCCESS` on a non-last iteration.

Returns `FAILURE` if the child task results in `FAILURE`.

Returns `SUCCESS` if the child task results in `SUCCESS` on the last iteration.

69.2 Properties

`StringName`	*array_var*	`&""`
`StringName`	*save_var*	`&""`

69.3 Property Descriptions

`StringName` **array_var** = `&""`

- VOID **set_array_var**(value: `StringName`)
- `StringName` **get_array_var**()

A variable within the *Blackboard* that holds an `Array`, which is used for the iteration process.

`StringName` **save_var** = `&""`

- VOID **set_save_var**(value: `StringName`)
- `StringName` **get_save_var**()

A *Blackboard* variable used to store an element of the array referenced by *array_var*.

CHAPTER

SEVENTY

BTINSTANCE

Inherits:

Represents a runtime instance of a *BehaviorTree* resource.

70.1 Description

Can be created using the *BehaviorTree.instantiate* method.

70.2 Properties

`bool`	*monitor_performance*	`false`

70.3 Methods

`Node`	*get_agent*() CONST
Blackboard	*get_blackboard*() CONST
Status	*get_last_status*() CONST
`Node`	*get_owner_node*() CONST
BTTask	*get_root_task*() CONST
`String`	*get_source_bt_path*() CONST
`bool`	*is_instance_valid*() CONST
VOID	*register_with_debugger*()
VOID	*unregister_with_debugger*()
Status	*update*(delta: `float`)

70.4 Signals

freed()

Emitted when the behavior tree instance is freed. Used by debugger to unregister.

updated(status: `int`)

Emitted when the behavior tree instance has finished updating.

70.5 Property Descriptions

`bool` **monitor_performance** = `false`

- VOID **set_monitor_performance**(value: `bool`)
- `bool` **get_monitor_performance**()

If `true`, adds a performance monitor for this instance to "Debugger->Monitors" in the editor.

70.6 Method Descriptions

`Node` **get_agent**() CONST

Returns the agent of the behavior tree instance.

Blackboard **get_blackboard**() CONST

Returns the blackboard of the behavior tree instance.

Status **get_last_status**() CONST

Returns the execution status of the last update.

`Node` **get_owner_node**() CONST

Returns the scene `Node` that owns this behavior tree instance.

BTTask **get_root_task**() CONST

Returns the root task of the behavior tree instance.

`String` **get_source_bt_path**() CONST

Returns the file path to the behavior tree resource that was used to create this instance.

`bool` **is_instance_valid**() CONST

Returns `true` if the behavior tree instance is properly initialized and can be used.

VOID **register_with_debugger**()

Registers the behavior tree instance with the debugger.

VOID **unregister_with_debugger**()

Unregisters the behavior tree instance from the debugger. This is typically not necessary, as the debugger will automatically unregister the instance when it is freed.

Status **update**(delta: `float`)

Ticks the behavior tree instance and returns its status.

CHAPTER

SEVENTYONE

BTINVERT

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that transforms `FAILURE` into `SUCCESS` and `SUCCESS` into `FAILURE`.

71.1 Description

BTInvert transforms `FAILURE` into `SUCCESS` and `SUCCESS` into `FAILURE`.

CHAPTER

SEVENTYTWO

BTNEWSCOPE

Inherits: *BTDecorator* < *BTTask* < *BT*

Inherited By: *BTSubtree*

BT decorator that creates a new *Blackboard* scope.

72.1 Description

BTNewScope creates a new *Blackboard* scope during initialization and populates it with data. See *Blackboard*.

Returns the status of the child task execution.

72.2 Properties

BlackboardPlan	*blackboard_plan*

72.3 Property Descriptions

BlackboardPlan **blackboard_plan**

- void **set_blackboard_plan**(value: *BlackboardPlan*)
- *BlackboardPlan* **get_blackboard_plan**()

Stores and manages variables that will be used in constructing new *Blackboard* instances.

CHAPTER

SEVENTYTHREE

BTPARALLEL

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that executes all of its child tasks simultaneously.

73.1 Description

BTParallel executes all of its child tasks simultaneously. Note that BTParallel doesn't involve multithreading. It processes each task sequentially, from first to last, in the same tick before returning a result. If one of the abort criterea is met, any tasks currently `RUNNING` will be terminated, and the result will be either `FAILURE` or `SUCCESS`. The *num_failures_required* determines when BTParallel fails and *num_successes_required* when it succeeds. When both are fullfilled, it gives priority to *num_failures_required*.

If set to *repeat*, all child tasks will be re-executed each tick, regardless of whether they previously resulted in `SUCCESS` or `FAILURE`.

Returns `FAILURE` when the required number of child tasks result in `FAILURE`. When *repeat* is set to `false`, if none of the criteria were met and all child tasks resulted in either `SUCCESS` or `FAILURE`, BTParallel will return `FAILURE`.

Returns `SUCCESS` when the required number of child tasks result in `SUCCESS`.

Returns `RUNNING` if none of the criterea were fulfilled, and either *repeat* is set to `true` or a child task resulted in `RUNNING`.

73.2 Properties

`int`	*num_failures_required*	`1`
`int`	*num_successes_required*	`1`
`bool`	*repeat*	`false`

73.3 Property Descriptions

`int` **num_failures_required** = 1

- VOID **set_num_failures_required**(value: `int`)
- `int` **get_num_failures_required**()

If the specified number of child tasks return `FAILURE`, BTParallel will also return `FAILURE`.

`int` **num_successes_required** = 1

- VOID **set_num_successes_required**(value: `int`)
- `int` **get_num_successes_required**()

If the specified number of child tasks return `SUCCESS`, BTParallel will also return `SUCCESS`.

`bool` **repeat** = `false`

- VOID **set_repeat**(value: `bool`)
- `bool` **get_repeat**()

When `true`, the child tasks will be executed again, regardless of whether they previously resulted in a `SUCCESS` or `FAILURE`.

When `false`, if none of the criteria were met, and all child tasks resulted in a `SUCCESS` or `FAILURE`, BTParallel will return `FAILURE`.

CHAPTER

SEVENTYFOUR

BTPAUSEANIMATION

Inherits: *BTAction* < *BTTask* < *BT*

BT action that pauses the playback of an animation on the specified `AnimationPlayer` node.

74.1 Description

BTPauseAnimation action pauses the playback of an animation on the specified `AnimationPlayer` node and returns `SUCCESS`.

Returns `FAILURE` if the action fails to get the `AnimationPlayer` node.

74.2 Properties

BBNode	*animation_player*

74.3 Property Descriptions

BBNode **animation_player**

- VOID **set_animation_player**(value: *BBNode*)
- *BBNode* **get_animation_player**()

Parameter that specifies the `AnimationPlayer` node.

CHAPTER

SEVENTYFIVE

BTPLAYANIMATION

Inherits: *BTAction* < *BTTask* < *BT*

BT action that plays an animation on the specified `AnimationPlayer` node.

75.1 Description

BTPlayAnimation action plays an animation on the specified `AnimationPlayer` node. If the *await_completion* is greater than `0`, the action will wait for the animation to complete, with the maximum waiting time equal to *await_completion*.

Returns `SUCCESS` if the animation finishes playing or if the elapsed time exceeds *await_completion*. When *await_completion* is set to `0`, BTPlayAnimation doesn't wait for the animation to finish and immediately returns `SUCCESS`.

Returns `RUNNING` if the animation is still playing and the elapsed time is less than *await_completion*.

Returns `FAILURE` if the action fails to play the requested animation.

75.2 Properties

`StringName`	*animation_name*	`&""`
BBNode	*animation_player*	
`float`	*await_completion*	`0.0`
`float`	*blend*	`-1.0`
`bool`	*from_end*	`false`
`float`	*speed*	`1.0`

75.3 Property Descriptions

`StringName` **animation_name** = `&""`

- VOID **set_animation_name**(value: `StringName`)
- `StringName` **get_animation_name**()

Animation's key within the `AnimationPlayer` node. If empty, BTPlayAnimation will resume the last played animation if the `AnimationPlayer` was previously paused.

BBNode **animation_player**

- VOID **set_animation_player**(value: *BBNode*)
- *BBNode* **get_animation_player**()

Parameter that specifies the `AnimationPlayer` node.

`float` **await_completion** = `0.0`

- VOID **set_await_completion**(value: `float`)
- `float` **get_await_completion**()

The maximum duration to wait for the animation to complete (in seconds). If the animation doesn't finish within this time, BTPlayAnimation will stop waiting and return `SUCCESS`.

`float` **blend** = `-1.0`

- VOID **set_blend**(value: `float`)
- `float` **get_blend**()

Custom blend time (in seconds). See AnimationPlayer.play.

`bool` **from_end** = `false`

- VOID **set_from_end**(value: `bool`)
- `bool` **get_from_end**()

Play animation from the end. Used in combination with negative *speed* to play animation in reverse. See AnimationPlayer.play.

`float` **speed** = `1.0`

- VOID **set_speed**(value: `float`)
- `float` **get_speed**()

Custom playback speed scaling ratio. See AnimationPlayer.play.

CHAPTER

SEVENTYSIX

BTPROBABILITY

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that executes its child task with a given probability.

76.1 Description

BTProbability executes its child task with a given probability defined by *run_chance*.

Returns the result of the child task if it was executed; otherwise, it returns `FAILURE`.

76.2 Properties

`float`	*run_chance*	`0.5`

76.3 Property Descriptions

`float` **run_chance** = `0.5`

- VOID **set_run_chance**(value: `float`)
- `float` **get_run_chance**()

Probability that defines how likely the child task will be executed.

CHAPTER

SEVENTYSEVEN

BTPROBABILITYSELECTOR

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that chooses a child task to execute based on attached probabilities.

77.1 Description

BTProbabilitySelector chooses a child task to execute based on attached probabilities. It is typically used for decision-making purposes. Probability distribution is calculated based on weights assigned to each child task.

Returns `SUCCESS` when a child task results in `SUCCESS`.

Returns `RUNNING` when a child task results in `RUNNING`.

The behavior of BTProbabilitySelector when a child task results in `FAILURE` depends on the *abort_on_failure* value:

- If *abort_on_failure* is `false`, when a child task results in `FAILURE`, BTProbabilitySelector will normalize the probability distribution over the remaining children and choose a new child task to be executed. If all child tasks fail, the composite will return `FAILURE`.
- If *abort_on_failure* is `true`, when a child task results in `FAILURE`, BTProbabilitySelector will not choose another child task to be executed and will immediately return `FAILURE`.

77.2 Properties

`bool`	*abort_on_failure*	`false`

77.3 Methods

`float`	*get_probability*(child_idx: `int`) CONST
`float`	*get_total_weight*() CONST
`float`	*get_weight*(child_idx: `int`) CONST
`bool`	*has_probability*(child_idx: `int`) CONST
VOID	*set_probability*(child_idx: `int`, probability: `float`)
VOID	*set_weight*(child_idx: `int`, weight: `float`)

77.4 Property Descriptions

`bool` **abort_on_failure** = `false`

- VOID **set_abort_on_failure**(value: `bool`)
- `bool` **get_abort_on_failure**()

If `true`, BTProbabilitySelector will not choose another child to execute and will return `FAILURE` when a child task results in `FAILURE`.

77.5 Method Descriptions

`float` **get_probability**(child_idx: `int`) CONST

Returns the child task's selection probability.

`float` **get_total_weight**() CONST

Returns the total weight of all child tasks.

`float` **get_weight**(child_idx: `int`) CONST

Returns the child task's weight within the weighted probability selection algorithm.

`bool` **has_probability**(child_idx: `int`) CONST

Returns whether the child task at index `child_idx` participates within the weighted probability selection algorithm and has a probability assigned to it. Returns `false` for *BTComment* tasks.

VOID **set_probability**(child_idx: `int`, probability: `float`)

Sets the child task's weight calculated based on the desired probability.

VOID **set_weight**(child_idx: `int`, weight: `float`)

Sets the child task's weight for the weighted probability selection algorithm.

CHAPTER

SEVENTYEIGHT

BTRANDOMSELECTOR

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that executes tasks in random order until first `SUCCESS`.

78.1 Description

BTRandomSelector executes its child tasks in a random order until any child returns `SUCCESS`. If a child task results in `FAILURE`, BTRandomSelector will immediately execute another child task until one of them returns `SUCCESS` or all of them result in `FAILURE`.

Returns `RUNNING` if a child task results in `RUNNING`. BTRandomSelector will remember the execution order and the last child task that returned `RUNNING`, ensuring it resumes from that point in the next tick.

Returns `FAILURE` if all child tasks result in `FAILURE`.

Returns `SUCCESS` if a child task results in `SUCCESS`.

CHAPTER

SEVENTYNINE

BTRANDOMSEQUENCE

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that executes tasks in random order as long as they return `SUCCESS`.

79.1 Description

BTRandomSequence executes its child tasks in a random order as long as they return `SUCCESS`. If a child task results in `SUCCESS`, BTRandomSequence will immediately execute the next child task until one of them returns `FAILURE` or all of them result in `SUCCESS`.

Returns `RUNNING` if a child task results in `RUNNING`. BTRandomSequence will remember the execution order and the last child task that returned `RUNNING`, ensuring it resumes from that point in the next tick.

Returns `FAILURE` if a child task results in `FAILURE`.

Returns `SUCCESS` if all child tasks result in `SUCCESS`.

CHAPTER

EIGHTY

BTRANDOMWAIT

Inherits: *BTAction* < *BTTask* < *BT*

BT action that waits for a randomized duration to elapse and then returns `SUCCESS`.

80.1 Description

BTRandomWait action waits for a randomized duration to elapse and then returns `SUCCESS`. The duration is randomized within the specified range, defined by *min_duration* and *max_duration*.

80.2 Properties

`float`	*max_duration*	`2.0`
`float`	*min_duration*	`1.0`

80.3 Property Descriptions

`float` **max_duration** = `2.0`

- VOID **set_max_duration**(value: `float`)
- `float` **get_max_duration**()

Maximum duration for the wait.

`float` **min_duration** = `1.0`

- VOID **set_min_duration**(value: `float`)
- `float` **get_min_duration**()

Minimum duration for the wait.

CHAPTER

EIGHTYONE

BTREPEAT

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that repeats its child a specified number of *times*.

81.1 Description

BTRepeat iterates its child task a specified number of times, as defined by *times*. If *forever* is `true`, the child's execution will be repeated indefinitely.

Returns `RUNNING` if the child task results in `RUNNING`. If *forever* is `true`, BTRepeat will always return `RUNNING`.

Returns `SUCCESS` if the specified number of successful executions is reached. If *abort_on_failure* is `false`, a `FAILURE` status returned by the child is also considered a successful execution.

Returns `FAILURE` if the child task results in `FAILURE` when *abort_on_failure* is `true`.

81.2 Properties

`bool`	*abort_on_failure*	`false`
`bool`	*forever*	`false`
`int`	*times*	`1`

81.3 Property Descriptions

`bool` **abort_on_failure** = `false`

- VOID **set_abort_on_failure**(value: `bool`)
- `bool` **get_abort_on_failure**()

If `false`, `FAILURE` status returned by the child task is also considered as a successful execution.

`bool` **forever** = `false`

- VOID **set_forever**(value: `bool`)
- `bool` **get_forever**()

If `true`, the child's execution will be repeated indefinitely, always returning `RUNNING`.

`int` **times** = `1`

- VOID **set_times**(value: `int`)
- `int` **get_times**()

The number of times to repeat execution of the child task.

CHAPTER

EIGHTYTWO

BTREPEATUNTILFAILURE

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that repeats its child task until `FAILURE`.

82.1 Description

BTRepeatUntilFailure repeats its child task until it results in `FAILURE`.

Returns `RUNNING` if the child task results in `RUNNING` or `SUCCESS`.

Returns `SUCCESS` if the child task results in `FAILURE`.

CHAPTER

EIGHTYTHREE

BTREPEATUNTILSUCCESS

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that repeats its child task until SUCCESS.

83.1 Description

BTRepeatUntilSuccess repeats its child task until it results in SUCCESS.

Returns RUNNING if the child task results in RUNNING or FAILURE.

Returns SUCCESS if the child task results in SUCCESS.

CHAPTER

EIGHTYFOUR

BTRUNLIMIT

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that restricts the execution of its child a limited number of times.

84.1 Description

BTRunLimit restricts the execution of the child task to a maximum number of times, defined by *run_limit*.

Returns `FAILURE` if the limit on executions is exceeded; otherwise, it returns the status of the child task.

84.2 Properties

CountPolicy	*count_policy*	`0`
`int`	*run_limit*	`1`

84.3 Enumerations

enum **CountPolicy**:

CountPolicy **COUNT_SUCCESSFUL** = `0`

Count only successful runs towards the limit.

CountPolicy **COUNT_FAILED** = `1`

Count only failed runs towards the limit.

CountPolicy **COUNT_ALL** = `2`

Count successful and failed runs towards the limit.

84.4 Property Descriptions

CountPolicy **count_policy** = `0`

- void **set_count_policy**(value: *CountPolicy*)
- *CountPolicy* **get_count_policy**()

Which runs should be counted towards the limit: successful, failed, or all?

`int` **run_limit** = 1

- VOID **set_run_limit**(value: `int`)
- `int` **get_run_limit**()

The maximum number of times the child is permitted to be executed.

CHAPTER

EIGHTYFIVE

BTSELECTOR

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that sequentially executes tasks until first `SUCCESS`.

85.1 Description

BTSelector executes its child tasks sequentially, from first to last, until any child returns `SUCCESS`. If a child task results in `FAILURE`, BTSelector will immediately execute the next child task until one of them returns `SUCCESS` or all of them result in `FAILURE`. BTSelector and *BTSequence* are two of the most common building blocks of behavior trees. Essentially, while *BTSequence* is similar to a boolean AND operation, BTSelector is similar to a boolean OR operation. Selectors enable the behavior tree to respond to changes in the environment and trigger transitions between various fallback behaviors.

Returns `RUNNING` if a child task results in `RUNNING`. BTSelector will remember the last child task that returned `RUNNING`, ensuring it resumes from that point in the next execution tick.

Returns `SUCCESS` if a child task results in `SUCCESS`.

Returns `FAILURE` if all child tasks result in `FAILURE`.

CHAPTER

EIGHTYSIX

BTSEQUENCE

Inherits: *BTComposite* < *BTTask* < *BT*

BT composite that sequentially executes tasks as long as they return `SUCCESS`.

86.1 Description

BTSequence executes its child tasks sequentially, from first to last, as long as they return `SUCCESS`. If a child task results in `SUCCESS`, BTSequence will immediately execute the next child task until one of them returns `FAILURE` or all of them result in `SUCCESS`. BTSequence and *BTSelector* are two of the most common building blocks of behavior trees. Essentially, while *BTSelector* is similar to a boolean OR operation, BTSequence is similar to a boolean AND operation. Sequences enable the behavior tree to compose complex behaviors from a chain of simpler tasks.

Returns `RUNNING` if any child task results in `RUNNING`. BTSequence will remember the last child task that returned `RUNNING`, ensuring it resumes from that point in the next execution tick.

Returns `SUCCESS` if all child tasks result in `SUCCESS`.

Returns `FAILURE` if a child task results in `FAILURE`.

CHAPTER

EIGHTYSEVEN

BTSETAGENTPROPERTY

Inherits: *BTAction* < *BTTask* < *BT*

BT action that assigns a value to the specified agent's property.

87.1 Description

BTSetAgentProperty assigns the specified *value* to the agent's property identified by the *property* and returns `SUCCESS`. Optionally, it can perform a specific *operation* before assignment.

Returns `FAILURE` if it fails to set the property.

87.2 Properties

Operation	*operation*	`0`
`StringName`	*property*	`&""`
BBVariant	*value*	

87.3 Property Descriptions

Operation **operation** = `0`

- VOID **set_operation**(value: *Operation*)
- *Operation* **get_operation**()

Specifies the operation to be performed before assignment. Operation is executed as follows:

```
property = property OPERATION value
```

`StringName` **property** = `&""`

- VOID **set_property**(value: `StringName`)
- `StringName` **get_property**()

Parameter that specifies the agent's property name.

BBVariant **value**

- VOID **set_value**(value: *BBVariant*)
- *BBVariant* **get_value**()

Parameter that specifies the value that will be assigned to agent's property.

CHAPTER

EIGHTYEIGHT

BTSETVAR

Inherits: *BTAction* < *BTTask* < *BT*

BT action that assigns *value* to the *variable* and then returns `SUCCESS`.

88.1 Description

BTSetVar assigns *value* to the *variable* and then returns `SUCCESS`. Optionally, it can perform a specific *operation* before assignment.

Returns `FAILURE` if it fails to set the *variable*.

88.2 Properties

Operation	*operation*	`0`
BBVariant	*value*	
`StringName`	*variable*	`&""`

88.3 Property Descriptions

Operation **operation** = `0`

- void **set_operation**(value: *Operation*)
- *Operation* **get_operation**()

Specifies the operation to be performed before assignment. Operation is executed as follows:

```
variable = variable OPERATION value
```

BBVariant **value**

- void **set_value**(value: *BBVariant*)
- *BBVariant* **get_value**()

Parameter that specifies the value to be assigned to the variable.

`StringName` **variable** = `&""`

- void **set_variable**(value: `StringName`)
- `StringName` **get_variable**()

Name of the variable to which the value will be assigned.

CHAPTER

EIGHTYNINE

BTSTOPANIMATION

Inherits: *BTAction* < *BTTask* < *BT*

BT action that stops the playback of an animation on the specified `AnimationPlayer` node.

89.1 Description

BTStopAnimation action stops the playback of an animation on the specified `AnimationPlayer` node and returns `SUCCESS`. If *animation_name* is set, it will only stop the playback if the specified animation is currently playing.

Returns `FAILURE` if the action fails to get the `AnimationPlayer` node.

89.2 Properties

`StringName`	*animation_name*	`&""`
BBNode	*animation_player*	
`bool`	*keep_state*	`false`

89.3 Property Descriptions

`StringName` **animation_name** = `&""`

- VOID **set_animation_name**(value: `StringName`)
- `StringName` **get_animation_name**()

Animation's key within the `AnimationPlayer` node. If not empty, BTStopAnimation will only stop the playback if the specified animation is currently playing.

BBNode **animation_player**

- VOID **set_animation_player**(value: *BBNode*)
- *BBNode* **get_animation_player**()

Parameter that specifies the `AnimationPlayer` node.

`bool` **keep_state** = `false`

- VOID **set_keep_state**(value: `bool`)
- `bool` **get_keep_state**()

If `true`, the animation state is not updated visually.

CHAPTER

NINETY

BTSUBTREE

Inherits: *BTNewScope* < *BTDecorator* < *BTTask* < *BT*

BT decorator that instantiates and runs a subtree within the larger tree.

90.1 Description

BTSubtree instantiates a *BehaviorTree* and includes its root task as a child during initialization, while also creating a new *Blackboard* scope.

Returns the status of the subtree's execution.

Subtree blackboard variables can be mapped to the main tree blackboard plan variables. Check out mapping section in the inspector.

Note: BTSubTree is designed as a simpler loader, and does not support updating *subtree* at runtime. A custom subtree decorator is better suited:

```
extends BTDecorator
## MyCustomBranch

func _setup() -> void:
    var bt: BehaviorTree = load("res://...")
    var my_branch: BTTask = bt.get_root_task().clone()
    my_branch.initialize(agent, blackboard)
    add_child(my_branch)

func _tick(delta) -> Status:
    var child: BTTask = get_child(0)
    return child.execute(delta)
```

This allows for the *BehaviorTree* to load a specific behavior tree from a variable in the blackboard, or from some other location, and use that new behavior tree at runtime.

90.2 Properties

BehaviorTree	*subtree*

90.3 Property Descriptions

BehaviorTree **subtree**

- VOID **set_subtree**(value: *BehaviorTree*)
- *BehaviorTree* **get_subtree**()

A *BehaviorTree* resource that will be instantiated as a subtree.

CHAPTER

NINETYONE

BTTIMELIMIT

Inherits: *BTDecorator* < *BTTask* < *BT*

BT decorator that sets a time limit for its child's execution.

91.1 Description

BTTimeLimit allocates a limited time for the child's execution and aborts it, returning `FAILURE` if the *time_limit* is exceeded.

Returns `FAILURE` if the *time_limit* is exceeded; otherwise, it returns the status of the child task.

91.2 Properties

`float`	*time_limit*	`5.0`

91.3 Property Descriptions

`float` **time_limit** = `5.0`

- VOID **set_time_limit**(value: `float`)
- `float` **get_time_limit**()

Time allocated for the child task's execution.

CHAPTER

NINETYTWO

BTWAIT

Inherits: *BTAction* < *BTTask* < *BT*

BT action that waits for a specified *duration* to elapse and then returns `SUCCESS`.

92.1 Properties

`float`	*duration*	`1.0`

92.2 Property Descriptions

`float` **duration** = `1.0`

- VOID **set_duration**(value: `float`)
- `float` **get_duration**()

There is currently no description for this property. Please help us by contributing one!

CHAPTER

NINETYTHREE

BTWAITTICKS

Inherits: *BTAction* < *BTTask* < *BT*

BT action that waits for a specified number of ticks to elapse and then returns `SUCCESS`.

93.1 Properties

`int`	*num_ticks*	1

93.2 Property Descriptions

`int` **num_ticks** = 1

- VOID **set_num_ticks**(value: `int`)
- `int` **get_num_ticks**()

The number of ticks to wait before returning `SUCCESS`.

CHAPTER

NINETYFOUR

LIMBOUTILITY

Inherits:

Helper functions for LimboAI.

94.1 Methods

`String`	*decorate_output_var*(variable: `String`) CONST
`String`	*decorate_var*(variable: `String`) CONST
`String`	*get_check_operator_string*(check: *CheckType*) CONST
`String`	*get_operation_string*(operation: *Operation*) CONST
`String`	*get_status_name*(status: `int`) CONST
`Texture2D`	*get_task_icon*(class_or_script_path: `String`) CONST
`bool`	*perform_check*(check: *CheckType*, a: `Variant`, b: `Variant`)
`Variant`	*perform_operation*(operation: *Operation*, a: `Variant`, b: `Variant`)

94.2 Enumerations

enum **CheckType**:

CheckType **CHECK_EQUAL** = `0`

Equality Check.

CheckType **CHECK_LESS_THAN** = `1`

Less Than Check.

CheckType **CHECK_LESS_THAN_OR_EQUAL** = `2`

Less Than or Equal To Check.

CheckType **CHECK_GREATER_THAN** = `3`

Greater Than Check.

CheckType **CHECK_GREATER_THAN_OR_EQUAL** = `4`

Greater Than or Equal To Check

CheckType **CHECK_NOT_EQUAL** = `5`

Inequality Check.

enum **Operation**:

Operation **OPERATION_NONE** = `0`

No operation.

Operation **OPERATION_ADDITION** = `1`

Arithmetic addition.

Operation **OPERATION_SUBTRACTION** = `2`

Arithmetic subtraction.

Operation **OPERATION_MULTIPLICATION** = `3`

Arithmetic multiplication.

Operation **OPERATION_DIVISION** = `4`

Arithmetic division.

Operation **OPERATION_MODULO** = `5`

Produces the remainder of an integer division.

Operation **OPERATION_POWER** = `6`

Multiply `a` by itself `b` times.

Operation **OPERATION_BIT_SHIFT_LEFT** = `7`

Bitwise left shift.

Operation **OPERATION_BIT_SHIFT_RIGHT** = `8`

Bitwise right shift.

Operation **OPERATION_BIT_AND** = `9`

Bitwise AND.

Operation **OPERATION_BIT_OR** = `10`

Bitwise OR.

Operation **OPERATION_BIT_XOR** = `11`

Bitwise XOR.

94.3 Method Descriptions

`String` **decorate_output_var**(variable: `String`) CONST

Just like *decorate_var*, produces a string with a *Blackboard* variable name that is formatted for display, and also adds an additional symbol to indicate that the variable is used as an output.

`String` **decorate_var**(variable: `String`) CONST

Produces a string with a *Blackboard* variable name that is formatted for display or console output.

`String` **get_check_operator_string**(check: *CheckType*) CONST

Returns an operator string for a *CheckType* enum value. For example, *CHECK_EQUAL* returns "==".

`String` **get_operation_string**(operation: *Operation*) CONST

Returns a string representation of an *Operation* enum value.

`String` **get_status_name**(status: `int`) CONST

Returns a name of a *BTTask* status code.

`Texture2D` **get_task_icon**(class_or_script_path: `String`) CONST

Returns the icon texture associated with a task based on its class name or script resource path.

`bool` **perform_check**(check: *CheckType*, a: `Variant`, b: `Variant`)

Performs a `check` on two values, `a` and `b`, and returns `true` if the check passes.

`Variant` **perform_operation**(operation: *Operation*, a: `Variant`, b: `Variant`)

Performs an `operation` on two values, `a` and `b`, and returns the result.