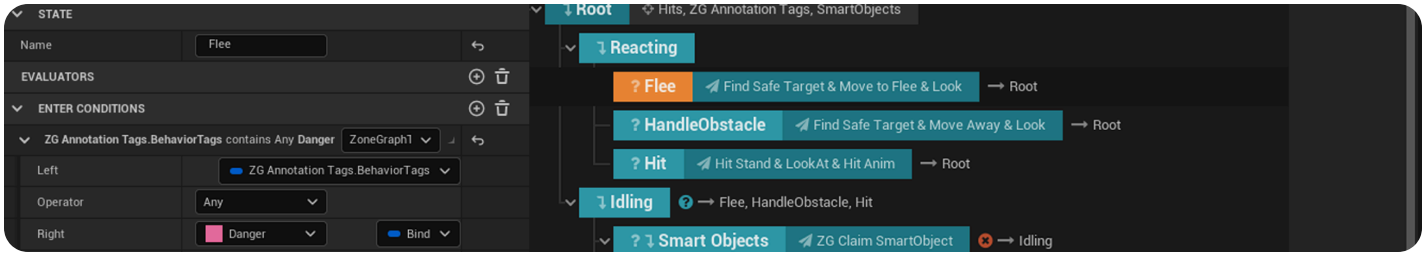
StateTree Overview

Overview of the StateTree system.



Overview

StateTree is a general-purpose hierarchical state machine that combines the **Selectors** from behavior trees with **States** and **Transitions** from state machines. Users can create highly performant logic that stays flexible and organized.

StateTree contains States that are arranged in a tree structure. The State selection can be triggered at any location in the tree, but it initially starts from the root. During the selection process, each State's **Enter Conditions** are evaluated. If they pass, the selection advances to the State's child States, if available. If no child States are available, the current State is activated.

Selecting a State will activate all of the States from the root to the leaf State. Each state consists of **Tasks** and **Transitions**.

When a State is selected, the selected State and all its parent States become active. All Tasks are executed for all active States, starting from the root, down to the selected State.

Each Task provides an output to the StateTree. Common output examples include selecting a target, playing an animation, and looking at an object. Each State can have multiple Tasks,

and all Tasks in a State run concurrently as long as the State remains active. The first Task that completes its execution triggers a Transition which can result in the selection of a new State.

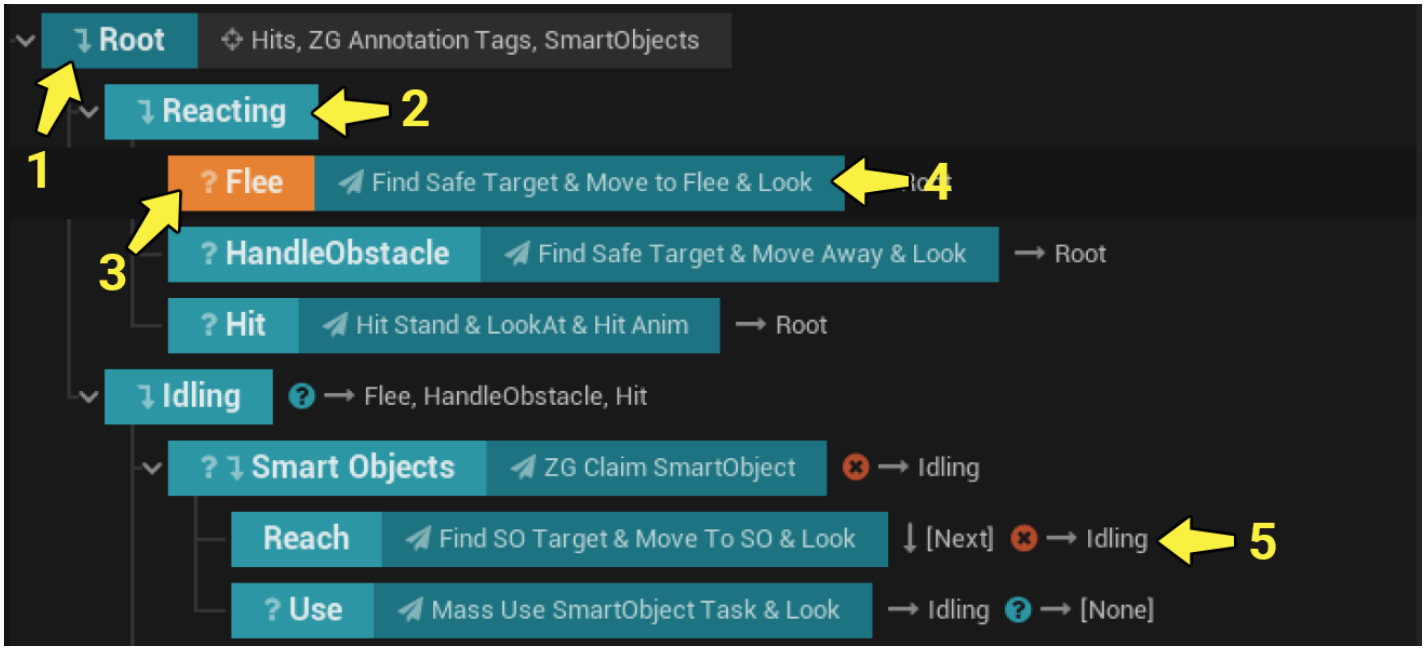
As a simple example, you could create a time of day system using StateTree. You would create a State for each time of the day, and underneath each State you would create separate Tasks. Each Task could handle different elements, such as fog density, sky sphere color, and so on.

Another example would be having an AI Agent walk around the level and constantly check for hits while looking around. You could have a State for walking and looking around, and another for reacting to a hit.

A Transition in a StateTree can point to any other State in the tree. Transitions have **Trigger Conditions** that need to be met before the transition triggers the State selection process.

If the State selection succeeds, a new State will be selected. Some Transitions are monitored constantly, while others only execute on State completion. Transitions are evaluated starting from the leaf State and progressing upwards toward the root. During this process, the first Transition that succeeds and leads to State selection is selected. This hierarchy allows for grouping of common Transitions.

These are the different elements of the StateTree:



Legend	StateTree element	Description
1	Root	First State selected when the StateTree starts running.
2	Selector State	Refers to a State that has child States. This State will never be selected directly, and selection will continue to one of its child States.
3	State Enter Condition	Refers to the list of conditions that determine whether a State can be selected.
4	Task	Refers to a set of actions that belong to a State and are executed when the State becomes active.
5	Transition	Defines the conditions that trigger the State selection process. A Transition is triggered when a Task completes, succeeds, or fails, or when a monitored condition succeeds.

Selection Flow

Selecting a new State

StateTree selects active States similarly to a behavior tree. State selection starts from root on the first Tick, and it continues down the tree evaluating each State's Enter Conditions.

- If the Enter Conditions do not pass, the selection proceeds to the next sibling State.
- If the Enter Conditions pass, and the State is a leaf State, it is selected as a new State.
- If the State has child States, the same process continues on to the first child State until a leaf State is found.

> If a State has child States, yet none can be selected (their Enter Conditions fail), the State will not be selected even if all its Enter Conditions passed the test.

One of the big differences between behavior trees and state machines is that state machines usually commit State selection as the execution goes down the tree, compared to behavior trees which try to find a suitable leaf node.

In general terms, behavior trees keep executing the State selection logic even when a State has been selected. This is the only method for transitioning between States.

StateTree runs the State selection process on-demand, based on Transitions. On the first tick, there is an implicit transition to the root State, which will select the first State to run. Once that State has been selected, the Transitions dictate when and where the selection logic will be executed.

Executing the State Tasks

Once a State is selected, all its Tasks will begin executing concurrently. The Tasks will continue executing until a Transition triggers the selection process and a State is selected. The selected State can be the current State (the State continues executing) or a new State.

The most common Transition trigger is **completion**, which is executed once the first Task of the active State finishes. Other Transitions can be marked as **Tick** and are tested on each tick. If the conditional Transition passes the test, the State selection logic is executed and the selection process starts at the target State. If the target State has child States, the selection process will consider the child States as part of the selection logic.

Data Flow

State Trees use data binding to pass data within the tree. The data binding can be used to create conditions, or to configure Tasks to run. The data binding can access data passed into the State Tree, or between the nodes in a specific manner.

The common data types available to all nodes in the State Tree are:

Data Type	Description
State Tree Parameters	Users can add input parameters to the State Tree that can be referenced while the tree is running. These parameters provide the ability to customize the use of the tree depending on its usage. For

Data Type	Description
	example, a user could define an animation asset parameter that can be passed externally to the tree for use during gameplay.
Context Data	The Context Data refers to the predefined data available to the State Tree based on the selected State Tree Schema. This changes based on where the State Tree is used. For example, a State Tree used by an Actor will have that actor as its Context. However, if the same State Tree is used with a Smart Object behavior, the Context will refer to the Smart Object used and the Actor that uses it.
Evaluators	Evaluators provide a way to expose data to the State Tree that would otherwise not be possible to do with Parameters or Context Data. Evaluators are a separate class that can be executed in a State Tree during runtime. Evaluators contain variables and can execute custom code when the tree starts, stops, and on each Tick. The properties of an Evaluator can be bound to Parameters or Context Data, or other Evaluators that come before in the Evaluator list.
Global Tasks	Global Tasks provide a way to run StateTree Tasks that are active between the Tree Start and Stop events. Global Tasks can be used when you need to have permanent data available for State selection. For example, a time of day system could create a Global Task that determines the current time of day. Since the Global Tasks are started before the Root State in the tree, this information would be available when the tree starts and during the first State selection.

StateTree nodes can share data among themselves. The different elements in the StateTree can bind to data in the following manner:

StateTree Element	Data the element can bind to
Enter Conditions	Can bind to common data, and Tasks in all parent States.
Transition Conditions	Can bind to common data, and Tasks from the current State and all parent States.

StateTree Element	Data the element can bind to
Tasks	Can bind to common data, and earlier Tasks in the current State and all the parent States.

Blueprint Integration

StateTree is designed to be extended via Blueprint scripting. You can create custom Tasks, Evaluators, and Conditions by extending the following Blueprint classes:

Base Class	Description
UStateTreeTaskBlueprintBase	Base class for a StateTree Task.
UStateTreeEvaluatorBlueprintBase	Base class for a StateTree Evaluator.
UStateTreeConditionBlueprintBase	Base class for a StateTree Condition.

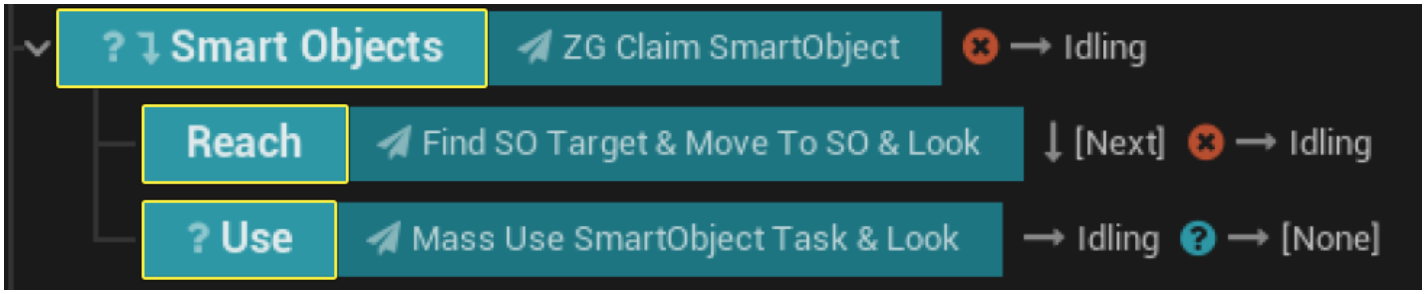
Following up with the example mentioned before, you could create a time of day system using StateTree and Blueprints. This system would change the fog density of the Level depending on the time. The system could also check for a storm spell to change the lighting conditions regardless of time.

For this example, you could create the following Blueprint classes:

Blueprint Class	Functionality
Task	Gradually changes the fog density over time. This Task can be added to all the States that represent a specific time of day.
Condition	Checks if a storm spell has been cast. This Condition can be evaluated at the root State. If true, it would move execution to a special Storm State.
Evaluator	Exposes the time of day to the Transitions and Enter Conditions.

Common Patterns

Grouping Similar Tasks

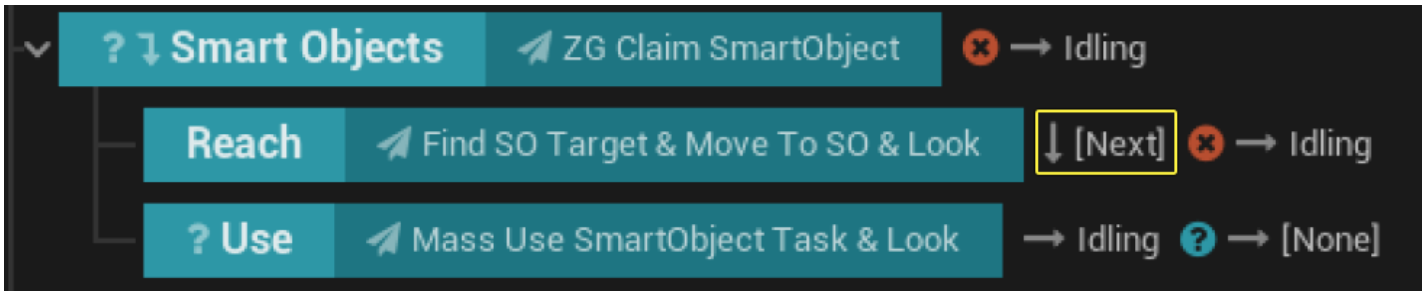


Tasks can be grouped under a common State. In the example above, there is a State that handles Smart Objects in the world. This State contains child States that handle Reaching the Smart Object and Using the Smart Object.

Each of these child States contains Tasks that execute when that State is selected. The Reach child State contains Tasks that find the Smart Object and move the AI Agent to the Smart Object.

When you use this grouping strategy, all Tasks share the same Transitions via the State.

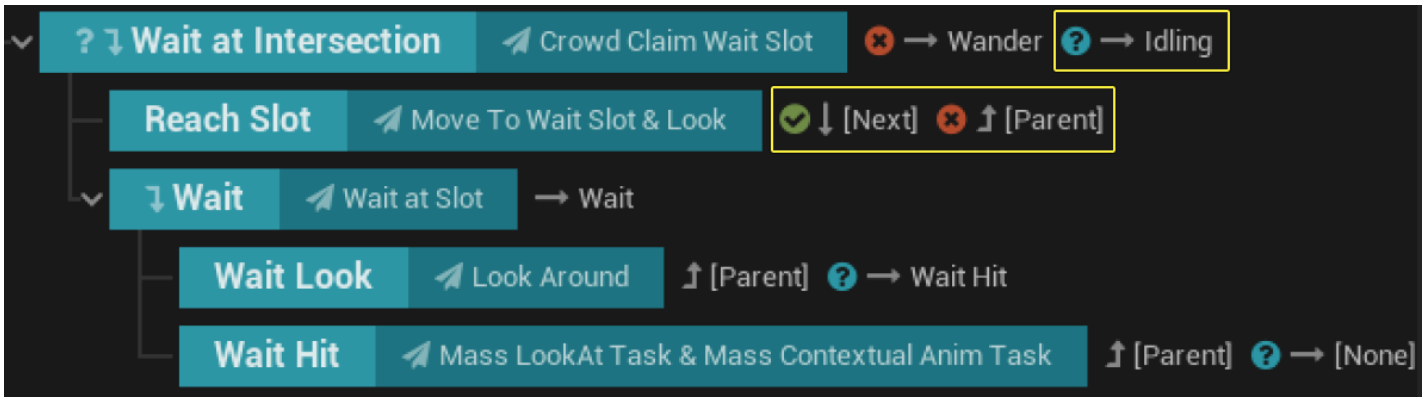
Sequences



StateTree comes with the **Next** Transition, which simplifies creating and arranging a sequence of States.

In the example above, when the **Reach** State is selected, the **Find SO Target**, **Move to SO**, and **Look** Tasks are executed. Once the Tasks are completed, the **Next** Transition moves execution to the **Use** State below, where its Tasks can begin executing.

Failure Handling

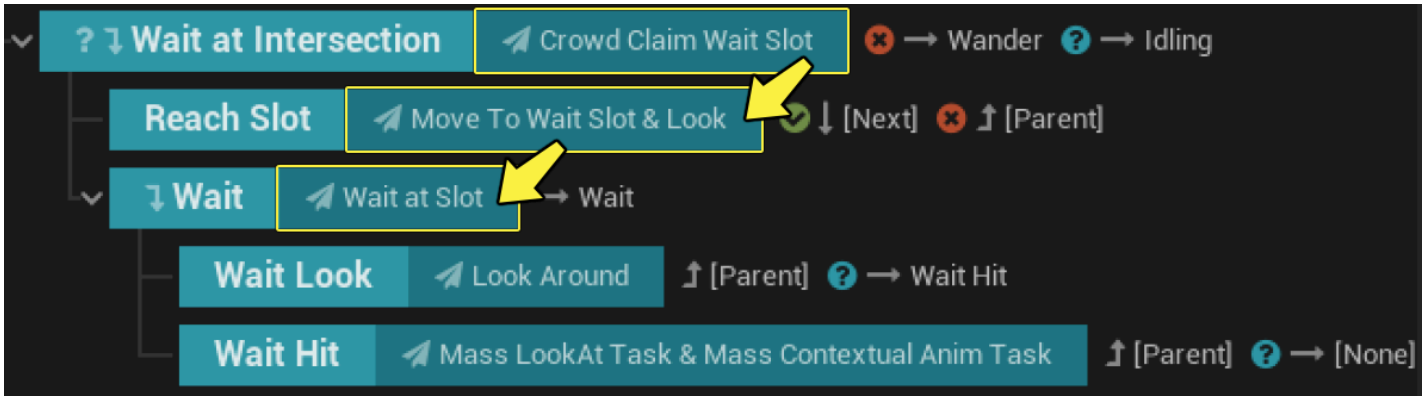


StateTree handles Task completion failure in a hierarchical way, starting with the active Task and going up the tree.

In the example above, the **Reach Slot** State will move execution to the next State on success (Wait), or it will move execution to its parent State on failure (Wait at Intersection). The **Wait at Intersection** State will trigger a Transition to the **Idling** State if any of its child States fail.

The **Wait** State will move execution to itself indefinitely on success or failure until its parent State selects a different State.

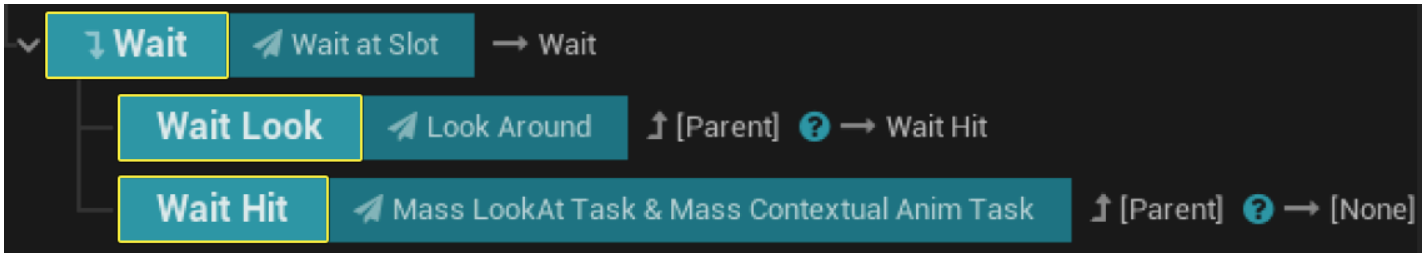
Hierarchical Data



Tasks can share data between each other. The data exposed by a Task will be available to any other Task that belongs to an active State. This makes resource handling in the StateTree more efficient.

In the example above, the **Crowd Claim Wait Slot** Task will try to claim a Smart Object Slot for the AI Agent, and if it is successful, it will pass execution to the **Move To Wait Slot** Task. This Task will use the Slot location from the parent Task. If successful, it will pass execution to the **Wait At Slot** Task, which will use the Slot Location from its parent Task as well.

Refining Behavior



StateTree provides a way to organize Tasks such that contextual behaviors can be achieved easily.

In the example above, the **Wait** State handles the AI Agent's standing locomotion - the AI Agent looks around and reacts when hit. By default, the **Wait Look** State will be executed. If the State succeeds, it will return execution to its parent. However, if it fails, it will move execution to the **Wait Hit** State.

The **Wait Hit** State will execute the **Mass LookAt** and **Mass Contextual Anim** Tasks. These Tasks will then play the appropriate animation to the hit.