Contents

[What are state machines & why would you use one? 2](#_Toc132718163)

[Why It is important for training: An Example Scenario 3](#_Toc132718164)

[Example of how you might already be handling flow in a VR application. 4](#_Toc132718165)

[Traditional coded state machines 5](#_Toc132718166)

[Visual Scripting State Machines (Previously Bolt) 6](#_Toc132718167)

[Graphs 6](#_Toc132718168)

[Script Graphs 6](#_Toc132718169)

[State Graphs 6](#_Toc132718170)

[Example 1: Enter, Exit and Update within a bolt state 7](#_Toc132718171)

[Enter 7](#_Toc132718172)

[Exit 7](#_Toc132718173)

[Update 7](#_Toc132718174)

[Example 3: UI Button Transition 8](#_Toc132718175)

[Example 4: VR Interaction, Waiting for grab 8](#_Toc132718176)

[Example 5: Custom Scripts inside a state 9](#_Toc132718177)

A screenshot of a video game

Description automatically generated

# What are state machines & why would you use one?

A ***state machine*** is a model consisting of a set of states, transitions, and actions. It can be used to represent the behaviour of a system that has a finite number of states, where the transitions between these states are based on a specific set of rules. Each state represents a specific condition or mode of the system, and the transitions between states define how the system responds to inputs or events. Actions are executed when the system transitions from one state to another.

Think of a state machine as a set of rules that defines the behaviour of an object or character in a game or simulation. It's like a flowchart where each box represents a specific behaviour or state, and arrows connect the boxes to show how the object can transition from one behaviour to another.

*For example, imagine a VR training simulation where you're learning how to operate a complex machine. The machine might have different states, such as idle, preparing to operate, operating, and shutting down. Each of these states has specific behaviours, like displaying instructions or warning messages, and responding to user input. The transitions between states might be triggered by events, such as the user pressing a button or the machine completing a task.*

Now, let's compare it to ***Unity Animator***, which is a tool used to create animations for game characters. In Unity Animator, you define a set of animation states, such as idle, walking, or jumping, and transitions between these states. For example, when the character is idle, and the user presses the jump button, the character transitions to the jump state and starts playing the jump animation. State machines in Unity work similarly to Unity Animator, but instead of defining animations, you define behaviours. By breaking down the behaviour of a game object or character into a set of states and transitions, it becomes easier to manage and modify its behaviour over time, just like how Unity Animator makes it easier to manage and modify animations.

Graphical user interface, application

Description automatically generated

<https://docs.unity3d.com/Manual/StateMachineBasics.html>

# Why It is important for training: An Example Scenario

Imagine you're designing a VR training simulation for a car maintenance procedure, where the user must change the oil in a car engine. To ensure that the user follows the correct steps, you could use a state machine to manage the behaviour of the VR environment and the user's interactions with it. In this case, the state machine would define a set of states that correspond to the different steps of the oil changing procedure, such as "drain oil," "replace oil filter," and "add new oil." Each state would have a specific set of behaviours associated with it, such as displaying instructions and prompting the user for input.

For example, in the "drain oil" state, the user would be prompted to locate the oil drain plug and remove it to drain the old oil. Once the user has completed this step, the state machine would transition to the "replace oil filter" state, where the user would be prompted to locate and replace the old oil filter. Once the user has completed this step, the state machine would transition to the "add new oil" state, where the user would be prompted to add new oil to the engine.

Throughout the procedure, the state machine would ensure that the user cannot progress to the next step until they have completed all the necessary steps in the current state. For example, the state machine would not allow the user to add new oil until they have completed both the "drain oil" and "replace oil filter" steps.

One of the main benefits of using a state machine in this scenario is that it ensures that the user follows the correct sequence of steps and cannot accidentally skip or exit a state. This makes the training procedure more effective and robust.



# Example of how you might already be handling flow in a VR application.

Text

Description automatically generatedThe simplest method for handling the flow of an application is a simple if/switch statement within the Unity Update() function.

In this example, the if statement checks the value of "currentStep" and updates the behavior of objects based on the current step of the training. For example, if the value of "currentStep" is 1, specific code can be executed.

Using an if statement like this can work for simple linear training procedures, but it can become difficult to manage for more complex procedures with multiple paths and branching logic. This is where state machines come in handy, as they provide a more robust and scalable solution to handle complex training procedures in VR.

**Pros:**

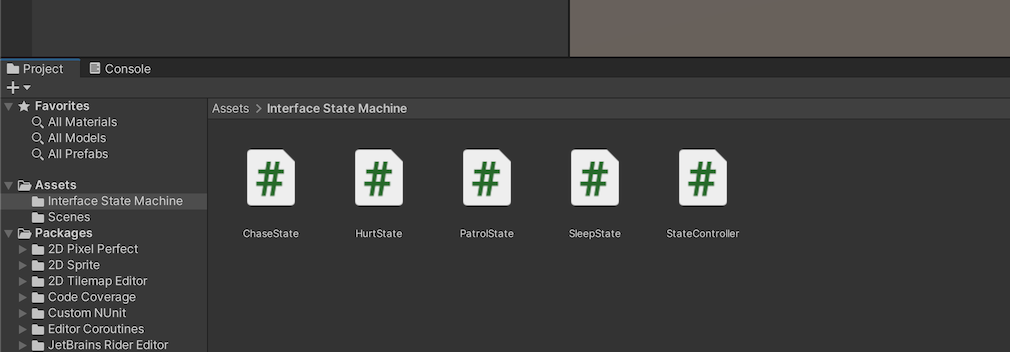
* Simple and easy to understand for linear procedures.
* Works well for small and simple apps
* No additional tools or plugins required.

**Cons:**

* Difficult to manage for more complex procedures with multiple paths and branching logic.
* Code can become cluttered and difficult to maintain as the number of steps increases.
* Limited scalability and flexibility
* Difficult to add additional functionality, such as error handling or automatic state transitions.
* Can result in a lot of duplicated code.

# Traditional coded state machines

Traditional coded state machines in Unity are implemented using scripts, where each state is represented by a separate class that inherits from a base state class. These classes define the behaviour for each state, including the conditions for transitioning to other states.

To manage the state machine, a controller script is used to handle the transitions between states based on the current conditions. The controller script updates the state based on the user's inputs, events, and other conditions.

<https://gamedevbeginner.com/state-machines-in-unity-how-and-when-to-use-them/#:~:text=Let's%20get%20started%E2%80%A6-,What%20are%20State%20Machines%20in%20Unity%3F,of%20an%20object%20or%20system>.

**Pros:**

* Powerful and flexible solution for managing complex VR training procedures.
* Fine-grained control over state behaviour and transitions
* Can be optimized for performance, particularly for large or complex state machines.
* Can be integrated with other programming systems and tools, such as AI or physics engines.

**Cons:**

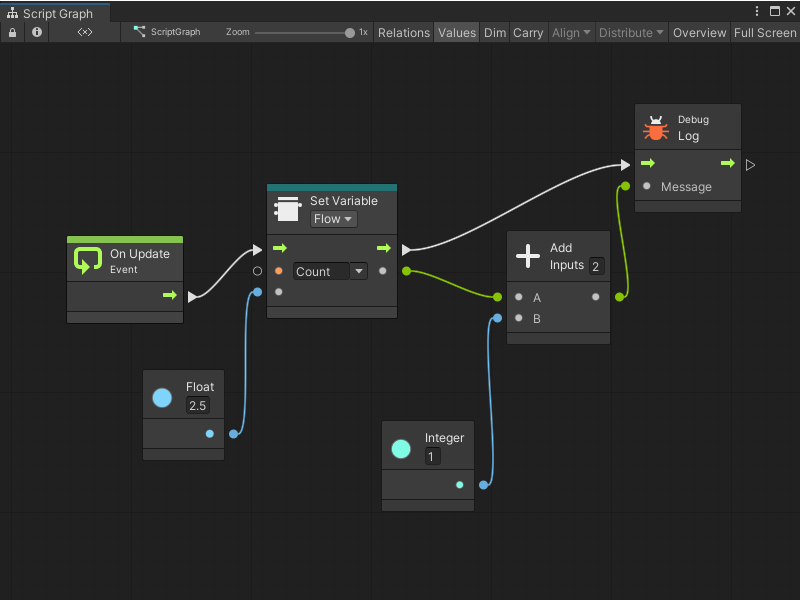
* Difficult for non-programmers and other project members to understand and contribute to
* Can be time-consuming and error-prone to modify or maintain, particularly for complex state machines.
* Debugging and testing can be difficult due to the complexity of the code and the potential for errors in state transitions.
* Can be less accessible for small or simple VR training applications where a simpler solution would suffice.

# Visual Scripting State Machines (Previously Bolt)

One of the key advantages of using visual scripting for VR training state machines is that it allows non-programmers and other project members to easily create and modify state machines using a graphical interface, without needing to understand advanced programming concepts. Visual scripting tools in Unity, such as Bolt, have some similarities to Unity's Animator, which is a tool for creating and managing animations in Unity. Both tools use a similar node-based interface and allow users to easily define behaviours and transitions without writing any code.

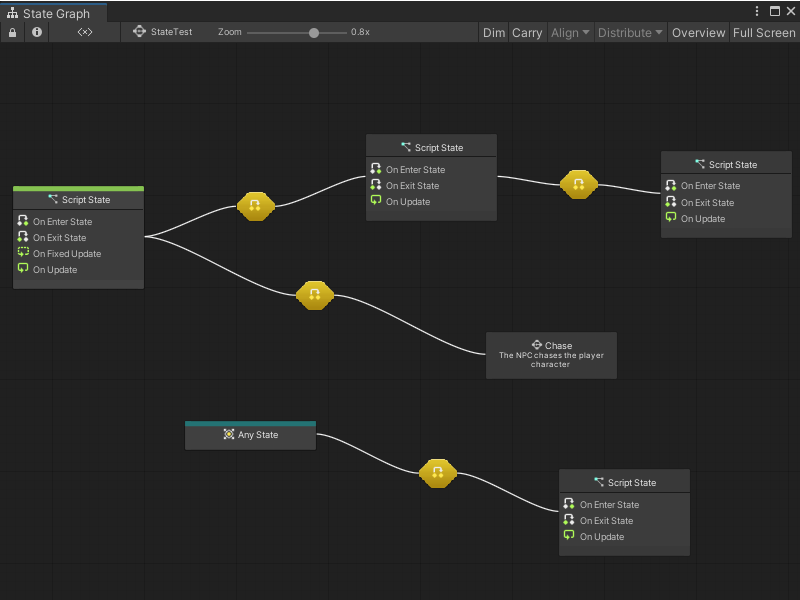
## Graphs

A graph is a Visual Scripting asset that contains a visual representation of logic in an application. Visual Scripting has two different types of graphs: Script Graphs and State Graphs.



## Script Graphs

Script Graphs control and connect specific actions and values. The actions in a Script Graph happen in a specific order. Actions can happen every frame, or when a specific event occurs.



## State Graphs

A State Graph has states and gives the logic for when your application moves between states, through connections called transitions.

# Example 1: Enter, Exit and Update within a bolt state

In Bolt, each state within a state machine has three distinct sections: Enter, Exit, and Update.

## Graphical user interface, text, application, chat or text message Description automatically generatedEnter

The Enter node is called when the state is first entered. This section is typically used to initialize any variables or data needed for the state and can also be used to trigger any actions that should happen when the state is entered.

## Graphical user interface, text, application, chat or text message Description automatically generatedExit

The Exit node is called when the state is exited, either by transitioning to another state or by ending the state machine. This section is typically used to clean up any variables or data used in the state and can also be used to trigger any actions that should happen when the state is exited.

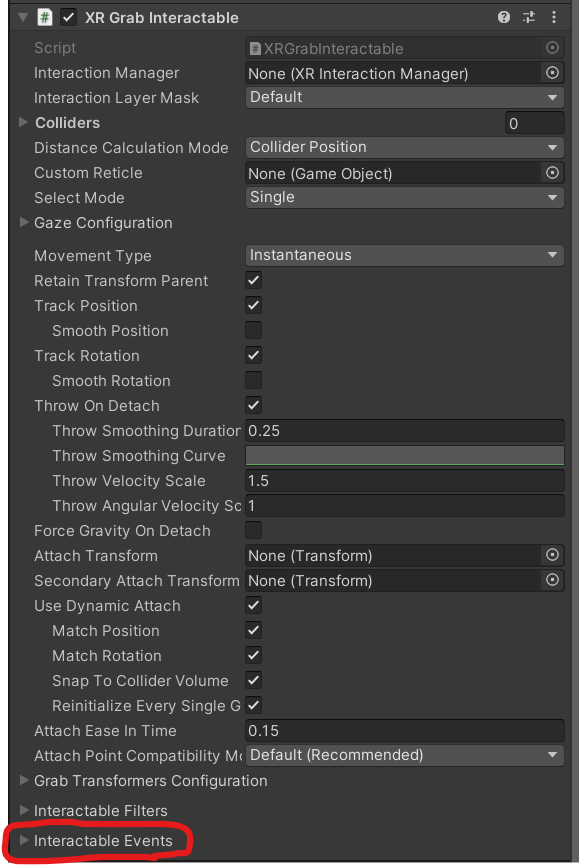
## A screenshot of a computer Description automatically generated with low confidenceUpdate

The Update node is called repeatedly while the state is active. This section is typically used to update the behaviour of the state based on any changes in the environment or user input. This is exactly the same as when you use Update() in your own scripts.

# A screenshot of a video game Description automatically generated with medium confidenceA screenshot of a computer Description automatically generated with medium confidenceExample 3: UI Button Transition

1. In your state transition, add a unity event node (right click), and give it a unique event name.
2. Connect this to the trigger transition node.
3. Add a standard UI button to your scene.
4. Go to the button, you will see the “On Click” unity event exposed in the inspector.
5. Click the small plus to add an action to fire when the button is clicked.
6. Populate the action by dragging in the gameobject which has your state machine on it, select the “***State Machine Trigger Unity Event***” method.
7. In the string field, populate the name of the unique event you want to fire when the button is clicked.

# Example 4: VR Interaction, Waiting for grab

A screenshot of a computer

Description automatically generated with medium confidenceSimilarly to a UI button, we need to invoke the uniquely named event when the user has grabbed the object.

On an XR grab interactable, there is a foldout called “Interactable Events”, In here there are exposed events that will run based on different conditions.

On grab, the select entered event Is fired. So we can add a listender of our state machine and get it to trigger out named unity event in our visual script.

# Example 5: Custom Scripts inside a state

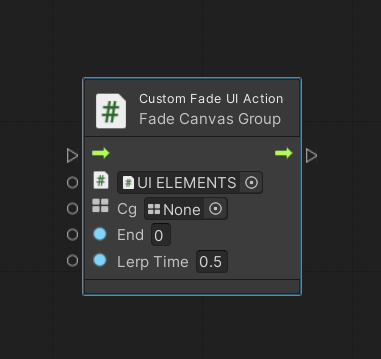
Text

Description automatically generated with medium confidenceText

Description automatically generatedIt is possible to introduce your own scripts as nodes within visual scripting graphs. This is useful when you have custom logic that you need to call within certain states. As an example, I have created a node that fades canvas groups to a set alpha value over time.

1. Go to project settings, visual scripting.
2. If your script is using a custom namespace, you will need to add it to the “Node Library” list in the settings.
3. Click “Regenerate nodes”
4. Now, if you search for your script inside of a state – you can add any of the public functions from it into your state machine.

* Any parameters of your public function are exposed within the node, so you can populate these.



When using custom scripts, you need to have it on a gameobject in your scene. Here is where you can drag and populate that ***reference***.

Here is the ***canvas group*** parameter that has come directly from our public function.

Both floats to control the ***alpha value*** and the ***time to fade***, again directly from the public function in our custom script.