



Game State Machine

HowTo

[Build a framework for games simple and complex](#)

© 2016 Todd D. Vance



Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

1 Introduction	2
1.1 Vending Machine Model	3
1.2 What is a State Machine	4
1.3 Exercise: Draw Your Own State Machine	4
1.4 Exercise: Make it Precise	5
1.5 Exercise: Do Some Research	5
2 Game State Machine	5
2.1 Classic Arcade State Machine	6
2.2 Exercise: Draw Your Own Game State Machine	8
2.3 Exercise: Code the Framework of Pac-Man	8
2.4 Exercise: Code the Framework of Any Game	8
3 Coding a State Machine in Unity3d C#	9
3.1 Exercise: Fix the State Machine Class	10
3.2 "Boss Level" Exercise: Design and Build a Better State Machine Asset for Unity 3D	10
4 Using the GameStateMachine asset	11
4.1 Exercise: Set up the GameStateMachine project in Unity3D	12
4.2 Exercise: Build a Simple GameStateMachine in Unity3D	12
4.3 Exercise: Build the Pac-Man GameStateMachine in Unity3D	12
4.4 Exercise: Build a Game	12
4.5 Exercise: Improve the GameStateMachine assets	12
5 Appendix	13

1 Introduction

A Finite State Machine, more technically called a Deterministic Finite Automaton is a theoretical concept that models a simple robot that follows simple rules. It has been "invented" many times in many different forms over the centuries (Euclid's straightedge and compass constructions are probably the earliest), but the most common model used in computer programming, and the one used here, was first described in detail in the mid 20th century scholarly paper: *Moore, Edward F (1956). "Gedanken-experiments on Sequential Machines". Automata Studies, Annals of Math. Studies. Princeton, N.J.: Princeton University Press (34): 129–153.* (A "Gedanken-experiment" is a thought experiment, done in the head instead of in a laboratory).

In real life, finite state machines occur in lots of everyday products, nowadays controlled by computers but in earlier times, controlled by a mechanical process. An example would be a typical vending machine.

1.1 Vending Machine Model

A typical vending machine spends most of its time in the “Ready” state, with lights on and (at least for soft drinks) refrigeration running.

The possible inputs are: press a product button (of which there are several), activate the “coin return” (though nowadays it returns bills too), or insert coins (or bills).

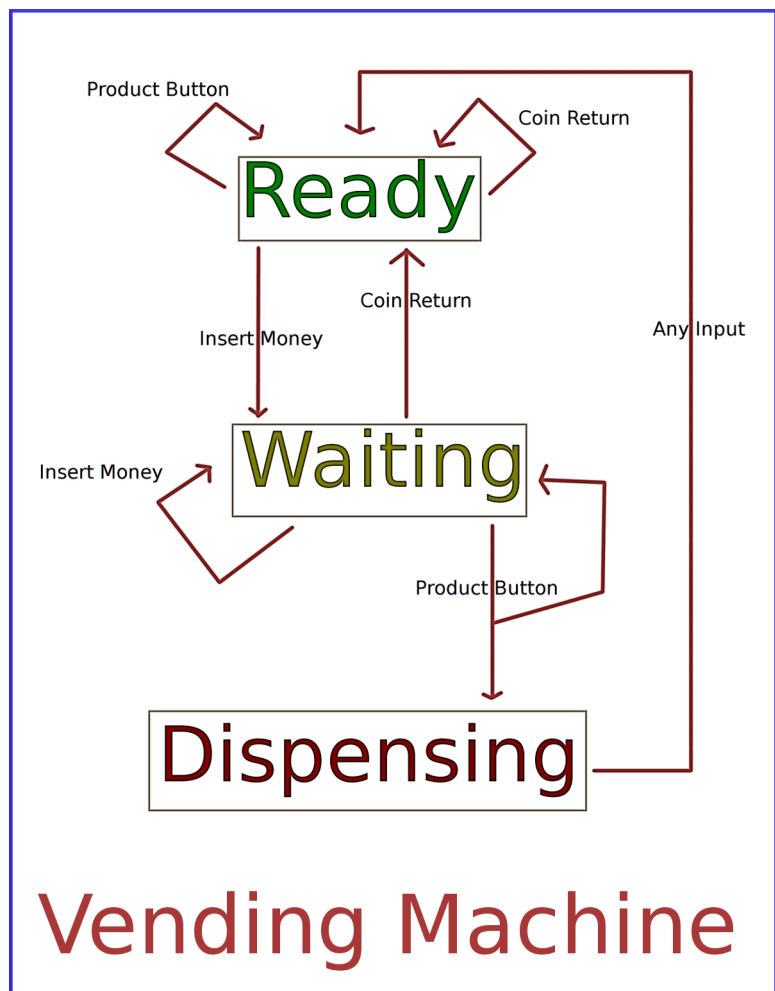
When in the ready state, if you press a product button, a typical machine would show the price of the product on a small display. Note that what makes a state machine a “state machine” is that the behavior depends not just on the input, but the state it is in. If the machine were in another state, it might dispense a product instead of just showing the price when the product button is pressed.

When in the ready state, if you press the coin return, nothing happens. (if something happens, such as money being returned, it means a previous customer walked away when the machine was in a different state and lost their money).

When in the ready state, if you start inserting money, now something useful happens. It changes to the “Waiting for Selection” state. (technically, there is additional state information here: the amount of money inserted so far).

When in the waiting for selection state, if you insert money, the amount of money inserted increases but nothing else happens.

When in the waiting for selection state, if you press the coin return, you get your money back



(if the machine is working...strange how many are not working...) and the machine returns to the Ready state.

When in the waiting for selection state, if you press a product button, there are two possibilities: you have inserted enough money for the product or you didn't. If you inserted enough money, it goes to the Dispense state. Otherwise, you get some kind of error message and it stays in the Waiting for Selection state.

When in the dispense state: none of the inputs do anything. It just dispenses the product, returns any change if any, and goes back to the Ready state.

Note that this description of the Vending Machine state machine is a bit simplified. To make it a more accurate model, there would be additional states like "enough money", "not enough money", "return change state", and so on. That a state and input can have two possible results (enough money to buy product or not enough) means more states are needed to be truly accurate, since this is not allowed with the technical definition of a pure deterministic automaton. But this should be enough to illustrate what a state machine is: a simple robot that follows simple rules.

1.2 What is a State Machine

A state machine has a finite set of "states" and a set of possible "inputs" and "behaviors". The state machine is always in one state. If it receives an input, then depending on which input it receives and which state it is in, it will perform one or more behaviors, and then either stay in the same state or transition to another state. Then it waits again for input and continues in the same way.

1.3 Exercise: Draw Your Own State Machine

Consider these exercises to be "additional fun," but also, a way of solidifying knowledge and enhancing future experience with state machines. Even a good but unsuccessful attempt will benefit the reader. Some exercises have Googleable solutions (or partial or even complete solutions may occur later in this document...), but the author suggests making a good try before looking. Remember, there are no grades—the goal of these exercises is to "think," and to gain some knowledge as a result.

The first exercise is: think of some system you are familiar with (like the Vending Machine example) and write up a model of it as a state machine. Possible ideas

include: other kinds of vending machines, an Automated Teller Machine (ATM), an espresso machine, or even some games can be modeled as state machines. Note text adventure games are really large state machines.

1.4 Exercise: Make it Precise

Make your knowledge of state machines more precise. For example, if you are a mathematician, come up with a precise mathematical definition of state machine, say, around the level of precision of the definitions in Euclid's Elements. If you are a programmer, write code that implements a state machine. For example, you could try to beat Roller Coaster Simulator or The Sims or Flight Simulator or Sim City and create Vending Machine Simulator! The prototype would, in your favorite programming language, take input as entered text or button presses, and behaviors (and which state the machine is in) would be shown by printing text.

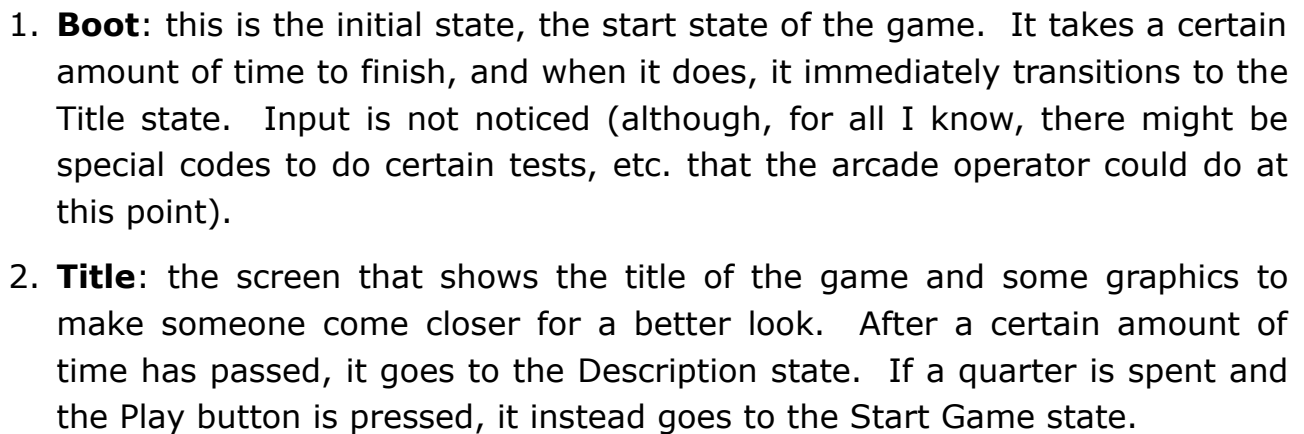
1.5 Exercise: Do Some Research

Using Google, Wikipedia, or even textbooks you have access to, read up on state machines. If you can, try to find the paper by Edward F. Moore. If not, google what you can find on his design. Find out things like, what other kinds of state machines there are and how they differ and how many of them are also essentially the same thing, why the Vending Machine "breaks the rules" of a deterministic automaton, and answer to your own satisfaction the question: "Is my computer a (really big) state machine?" Note there are very good reasons to answer either Yes or No to that one! Look up Turing Machines to find some good reasons the answer should be "No". For a bonus, answer to your own satisfaction, "Is the physical universe a (really, really big) state machine?". Note that Isaac Newton thought the answer to that was, "Yes," (but possibly an infinite state machine) but modern physicists tend toward "No". The idea is to find out why. And that doesn't even consider the religious element.... Many of these ideas would make good term papers for certain courses—check with your teacher.

2 Game State Machine

The particular state machine this paper is about is the overall game state machine. This doesn't have much to do with the details of game play, but of the organization of levels, cutscenes, title screens, and so on in a game. Let us begin with an example.

Consider the arcade game *Pac-Man*, the kind that was in a wooden box at an arcade one stood in front of and played after having fed it quarters. We shall discuss the state machine for it, noting that many classic arcade games had state machines that followed a similar pattern. When you walked up to the game, it was most likely in “attract” mode, the mode that tries to get you to spend a quarter. This is a big state that will be divided into smaller states (title, demo, and so on—note *Nameco*/*Balley* probably have their own names for the states; I’m making up names that are reasonably descriptive). But before the attract mode, there is another mode not seen often unless you pull the plug and plug it back in: booting up.



3. **Description:** here is a screen describing gameplay and points for various activities. After a certain amount of time has passed, it goes to the High Scores state (I might have Description and High Scores in the wrong order; it has been decades since I've played Pac-Man at an arcade). If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.
4. **High Scores:** the top ten scores, with initials, are displayed here. After a certain amount of time has passed, it goes to the Demo state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.
5. **Demo:** A sample game is shown, without sounds and possibility for player input. It doesn't last long before Pac-Man is eaten and the game returns to the Title state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.
6. **Start Game:** the Level 1 board is displayed, the intro music is played, Pac Man and the ghosts appear on the board, and then the game transitions to the Play Game state. Input is ignored during the Start Game state.
7. **Play Game:** now the input is connected to the game and a user can play. Depending on the events of the game, the next state is Level Up or Game Over (recall, this is just overall game state, we are not modeling the fine points of game play at his level).
8. **Level Up:** when the board is cleared, this state is entered. The board flashes, and there may be a cutscene. When this is done, the next level is loaded and the game goes back to Play Game.
9. **Game Over:** when the player runs out of lives, the game is over. A big Game Over message appears over top the game board while the ghosts dance in glee at having earned another quarter. After a time, one of two things happen. Either it returns directly to the Title state, or if the score achieved is in the top ten, it goes to the New High Score state.
10. **New High Score:** the player is given the opportunity to enter his initials to appear on the high score board. After the initials are entered (or if it times out, since some players just walk away), the game returns to the High Score state-- or rather, a duplicate of the High Score state (I'll call it High Scores Reprise) that transitions to the Title rather than to the Demo.

11. **High Scores Reprise:** the top ten scores, with initials, are displayed here, with the just-made high score highlighted. After a certain amount of time has passed, it goes to the Title state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.

Note that one can “factor” this state machine. The outer state machine has for major states: Boot, Attract, Play, and Post Game. Within each major state is a refinement into more states.

Note also that because the Pac-Man state machine is such that each major state can be entered from another major state at only one inner state (namely, Boot, Title, Start Game, and New High Score), one could actually program this with 5 state machines: the outermost major state machine with 4 major states, and within each major state, start up a minor state machine (the Boot one is trivial, just one state) for the inner minor states. Even if not every major state can only be entered at one minor state, this can be done with a little extra code.

2.2 Exercise: Draw Your Own Game State Machine

Pick a computer game, whether arcade, mobile, desktop, console, or even hand-held (like GameBoy or the old single-game machines like Football), and draw a game state diagram. Remember, this is the top level state machine for the game and doesn’t include finer game play details.

2.3 Exercise: Code the Framework of Pac-Man

Code up, in your favorite language, the Pac-Man state machine. Either use the full 11-state machine or factor it into 5 state machines, the outermost one being the controlling one having 4 states. Keep it simple, accepting input like “play game” and printing out how it responds and what state it is now in. This code could then be used as a framework for the full game.

2.4 Exercise: Code the Framework of Any Game

For the state machine you made in Exercise 2.2, code up a simulator in your favorite programming language. Keep it simple, accepting input like “play game” and printing out how it responds and what state it is now in. This code could then be used as a framework for the full game.

3 Coding a State Machine in Unity3d C#

[Unity3d](#) is a popular, general purpose game engine that is free to use for hobbyists, though some premium pay features are also available. The [online Udemy course](#) is not free, but is for Unity Version 4, and a Version 5 course is in the works (to be sold at a discount to those who purchased Version 4). In addition, the [Unity3d](#) site has some tutorials and manuals and a forum for learning Unity.

A basic state machine can be coded as a C# class called `StateMachine` whose main responsibility is to keep track of which state the machine is in. A state is then another C# class, called `State`, which maintains its name and a list of states it can transition to. The simplest possible state machine code would thus look something like this, in a single file called [SimpleStateMachine.cs](#). This code is specific to Unity3d, but it could easily be adapted to other systems and languages. In Unity, one can fill in state machine details using the Inspector. In other languages, it might be necessary to write code to fill in the details of a specific state machine. This version uses strings to label states, as well as a string array to list states that can be transitioned to, making it easy to fill in using plain text. In addition to the previous link, the state machine code occurs in the first Appendix[13].

To use the simple state machine in Unity, one creates an empty `GameObject` in the hierarchy and attaches the `SimpleStateMachine` script to it as a component. Then, when the empty `GameObject` is selected, one can add new states in the Inspector. When running the game, changing the value of “currentState” in the inspector causes a transition to the new state (or an error if the state does not exist). Another script can also call the `Transition` function with an integer argument to select which state to transition to.

There are several problems with this simple state machine. First of all, all it does is show what state it is in. There is no additional behavior (though that can easily be added) and no means for transitioning in response to events. The second is a small issue since one only need put the `StateMachine` method `Transition` into an event (such as a button click) to cause the transition to happen upon that event.

A more serious problem is it is currently impossible to load a new scene without losing the state machine instance. This means the state machine should be a `Singleton` class. That means, any game has only one state machine (we are

specifically dealing with overall game state machines here—a game can have many smaller state machines) and it remains active across scene changes.

3.1 Exercise: Fix the State Machine Class

At a minimum, think of how you would do this, Googling for “Unity DontDestroyOnLoad” to find out how to make a Singleton. If able, modify the code to make it robust and useful and test it well. As a bonus, redo some of the previous exercises with this new code. As an extended bonus, actually create a complete, playable game using this new code. The reader may wish to compare what they wrote with the [author’s State Machine code](#).

3.2 “Boss Level” Exercise: Design and Build a Better State Machine Asset for Unity 3D.

Start with a list of features it should have (for example, automatically transition after a certain amount of time has elapsed, or—this is very advanced—create a graphical Editor interface to build the state machine). Prioritize them into important, very useful, nice to have, and optional.

Using the priorities as a guide, built a state machine asset having as many of those features as you are able to do, understanding that what looks easy at the design stage might turn out to be hard when you actually do it.

Test the result. Fix any bugs you find. Are there non-bugs that are still annoyances? Are there things that could be made to work better? Are there new features that appear useful in retrospect not in the original list? Amend the design and create a better version.

Test again. Be paranoid and test for anything that could possibly go wrong. Give it the acid test: build some games using the state machine and notice what goes wrong or what slows you down and think of how it can be improved.

Test again.

When all is working magnificently, go to the [Unity3d](#) website and research how to submit an asset to the [Asset Store](#)—the procedure is not trivial! But it can be done. The author has done it. Good luck and hope it gets accepted.

Also look at comparable assets in the store. If yours is significantly better than the others available, consider asking for money! That requires even more work and set up. If it isn't, can you amend the design and improve it? Some of the ones there have some advanced, hard-to-code features.

4 Using the GameStateMachine asset

The [author's State Machine code](#) can be used as an overall state

machine. The StateMachine.cs class is found in the Assets/StateMachine folder. In addition are Unity3d scenes for testing: ClassicArcadeStart and MobileStyleStart. The first is similar (but not exactly like) the Pac-Man example above. The second is a style similar to what might be found on a mobile game like Angry Birds. To use these, all scenes in subfolders must be in the Build Settings. The two scenes mentioned give two examples of how to set up the state machine for a game. The key object is the StateMachine prefab (in the Assets/StateMachine folder). Put this in the start scene of a game. It is recommended the start scene be a special scene that only loads Singletons (like the StateMachine) and is never returned to again.

The screenshot shows a grid of 12 asset listings from the Unity Asset Store. Each listing includes an icon, the asset name, a brief description, a star rating, the number of reviews, and the price. The assets are sorted by relevance. The first row includes 'C# State Machine Scripting By the Tale' (Free), 'Finite State Machi... Scripting Fairuz' (\$12), and 'Finite State Machine Scripting Nerdius Rex LLC' (Not enough ratings, \$10). The second row includes 'Finite State Machine Scripting MorphVGX' (Not enough ratings, \$4), 'Animator State M... Scripting Paul Hayes' (4.15 stars, Free), and 'White Cat's Tool... Scripting White Cat' (4.37 stars, \$29). The third row includes 'SimpleFSM - A po... Scripting/AI coAdjoint Limited' (4.4 stars, \$10), 'State & Task Man... Scripting/AI JekelSkaz' (4.7 stars, \$19), and 'Lost Zombie Studi... Scripting Daniel Gallagher' (4.19 stars, \$15). The fourth row includes 'Behaviour Machin... Editor Extensions/... Anderson Cardoso' (4.27 stars, Free), 'Behaviour Machin... Editor Extensions/... Anderson Cardoso' (4.27 stars, \$250), and 'Behaviour Machin... Editor Extensions/... Anderson Cardoso' (4.20 stars, \$50). The fifth row includes 'The Spaghetti Ma... Editor Extensions/... Method in the Mad...' (4.9 stars, \$100), 'MXD Vending Mac... 3D Models/Props/... Mixed Dimensions' (Not enough ratings, \$5), and 'Retro Sci-Fi Machi... Audio/Ambient/Sci-Fi cabled_mess' (Not enough ratings, \$10). The sixth row includes 'American Gas Stat... 3D Models/Enviro... Gabro Media' (4.3 stars, \$10), 'MBS Core Scripting myBad Studios' (4.9 stars, \$5), and 'Subway Station 3D Models/Enviro... Realusion' (4.9 stars, \$15). The seventh row includes 'Underground Station 3D Models/Enviro... Realusion' (Not enough ratings, \$15), 'Nottorus Editor Extensions/... Stridemann' (4.17 stars, \$250), and 'Action Game Syst... Editor Extensions/... Mystik Digital' (4.4 stars, \$37.50). The eighth row includes 'AGS Core Scripting Mystik Digital' (4.8 stars, Free), 'Subway / Metro / ... 3D Models/Enviro... PM assets' (Not enough ratings, \$19.99), and 'NodeCanvas Editor Extensions/... Paradox Notion' (4.215 stars, \$75). The ninth row includes 'Rocket Jump Proc... Complete Projects... Rainbirth SLU' (4.17 stars, \$40), 'Logic Forge Editor Extensions/... Wlax Production' (Not enough ratings, \$50), and 'Apex Mecanim Tu... Complete Projects... Apex Game Tools' (4.33 stars, Free). The tenth row includes 'TF: Mobile Third P... Complete Projects... Tairna Foundation' (Not enough ratings, \$19.99), 'Playmaker Editor Extensions/... Hutong Games LLC' (4.2693 stars, \$65), and 'Pro-TEK Sci-Fi PB... 3D Models/Enviro... Level One Games' (Not enough ratings, \$14.90). The eleventh row includes 'SAGE ANIMATOR 1.6 Sage: Anim Graph ... Editor Extensions/... Altered Reality Ent...' (4.27 stars, \$50), 'Simple Option Sta... Scripting/AI Imre Angelo' (Not enough ratings, \$20), and 'EASY FARM MOBIL... 3D Models/Props JR' (Not enough ratings, \$7). The twelfth row includes 'Attack Zombie Mo... Complete Projects... SURIYUN' (Not enough ratings, \$29.50), 'GDG-FSM Scripting/AI Game Developers ...' (Not enough ratings, \$20), and 'SimpleSM Scripting G-Squared Games' (Not enough ratings, \$5). At the bottom, it says 'All prices are exclusive of VAT'.

Use the inspector to add new states and fill in details, such as transitions, auto transitions (after a certain number of seconds), scenes to load, and so on. Three events are defined for each state: entering, exiting, and loading the scene. The `DefaultActions.cs` file is a simple class that simply logs the events. A game programmer will likely provide their own classes for actions to be invoked upon events happening.

4.1 Exercise: Set up the GameStateMachine project in Unity3D

Download the entire folder structure from [GitHub from the link given](#). Also ensure you have Unity version 5.5 or higher (though it should still be Unity 5). Then open the base folder (which contains the Assets directory and other files) in Unity. It should import everything. Make sure all scenes are in the Build Settings. Test-run the `ClassicArcadeStart` and the `MobileStyleStart` scenes and make sure all the possible button combinations work. Read through the `StateMachine` in the Inspector and try to understand how it works.

4.2 Exercise: Build a Simple GameStateMachine in Unity3D

Now, start with a new scene. Put the `StateMachine` prefab in it and build the Vending Machine simulator that was done in a previous exercise. Use UI buttons for simulating the various inputs.

4.3 Exercise: Build the Pac-Man GameStateMachine in Unity3D

Again, start with a new scene and build the Pac-Man system described above. Compare this with the `ClassicArcadeStart`.

4.4 Exercise: Build a Game

Create a complete, playable game of your choice using the `GameStateMachine` assets.

4.5 Exercise: Improve the GameStateMachine assets

The author left plenty of room for improvement. What can you improve? Redo the exercise[10] from the last section using this asset package.

5 Appendix

[SimpleStateMachine.cs](#) file is shown here:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Maintain the current state that the state machine is in
/// </summary>
public class SimpleStateMachine : MonoBehaviour {

    #region Public interface

    /// <summary>
    /// A state in the state machine
    /// </summary>
    public struct State {
        [Tooltip("Name of the State")]
        public string name;

        [Tooltip("List of states this state can transition to")]
        public string[] canTransitionTo;

        //constructor for convenience: needed only to create a default "Start"
state.
        public State(string name) {
            this.name = name;
        }
    }

    [Tooltip("Changing this changes the state the machine is in")]
    public string currentState = startState;

    [Tooltip("All the available states")]
    public State[] states = { new State(startState) };

    public void Transition(int which) {
        currentState = activeState.canTransitionTo[which];
    }
    #endregion

    //the name of the start state, what the machine is in upon awakening
    private static string startState = "Start";

    //the currently active state
    State activeState;

    // Use this for initialization
    void Start() {
        //set the current state
        activeState = FindState(currentState);
    }
}
```

```

// Update is called once per frame
void Update() {
    if (currentState != activeState.name) {
        //if state changes, reset the current state
        activeState = FindState(currentState);
    }
}

//find state having given name
State FindState(string name) {
    foreach (State state in states) {
        if (state.name == name) {
            return state;
        }
    }
    Debug.LogError("Missing state: " + name);
    return null;
}
}

```