Game Logic & Finite State Machines

[Functions](#_m0kx58bktmys)

[Inputs and Outputs](#_jc3prbwro3za)

[Top-Down Approach](#_ao9lvw81o2fh)

[Finite State Machine](#_ufsz056bmd4v)

[Modelling Game Logic with FSM](#_kdlr1q3t8ajw)

[Enums](#_t4wtuzl0gpj9)

[Switch Statement](#_xvx9kfq9xdlg)

[Translating FSMs to C#](#_jwhq3zd3wlr1)

[From the Asset Store](#_bm0kpag1r5xj)

# Functions

One of the things that separates beginners programmers from experienced software developers is the ability to write structured, maintainable code.

As the level of complexity of a game increases, its scripts will grow longer and harder to read. For larger projects, organising code is not an option: is a requirement.

One such way is to group pieces of code into containers called **functions**. In Unity, both Start and Update are functions.

|  |
| --- |
| **📖 Methods**  Generally speaking, when a function is nested inside a **class**, it is referred to as **method**. In the context of Unity programming, you will generally see the terms *function* and *method* being used interchangeably.  The term *method* is commonly used in the context of **object-oriented programming**, which is the paradigm chosen by both C# and Unity. |

Functions are typically needed when there is a specific piece of code that is repeated or reused many times. In that case, it is often convenient to move it into a function, so that it can be called easily in a single line of code.

## Inputs and Outputs

In order to operate, each function can take some **inputs** and can produce some **outputs**. When a function is declared, its name is preceded by the **return type**, which indicates what we expect the function to produce as a result of its computation. If a function is not expected to produce (technically speaking: *to return*) anything, its return type is void. Start and Update are both preceded by the keyword void, because they do not produce any output.

The inputs that a function can take are indicated in round brackets. C# syntax requires to indicate the type and a name that will be used for each parameter a function can take. For instance: int F (float a) is a function called F that takes a number (which is referred to as a) and is expected to return an integer.

A function can return the result of its computation using the keyword return. The following function, for instance, can be used to add two integer numbers together:

|  |
| --- |
| int Add (int a, int b)  {  return a + b;  } |

It can be executed like this:

|  |
| --- |
| int result = Add(5, 10);  Debug.Log(result); // prints 15 |

## Top-Down Approach

The best way to structure and write complex code is a controversial topic. There are many strategies, methods and approaches used in the industry that are supposed to help write better code.

One such approach is called **top-down**, and is designed to help to break down complex problems into simpler parts which can be solved easily. It also allows seeing a problem at various different **levels of abstraction**. A recipe, for instance, talks about ingredients and how to mix them. However, a complete description of a recipe for a robot would also require instructions at a much lower level, explaining how to cut vegetables step-by-step. Both levels of abstractions are equally important, but it is easier to talk about recipes in terms of ingredients, rather than hand movements.

The top-down approach aims to write code in a similar way. When we write code, we do it assuming we have all the functionalities we need available, aiming to make the overall code as compact and easy to read as possible.

|  |
| --- |
| **📖 Bottom-Up**  The opposite approach is called **bottom-up**. It aims to solve a problem by using very simple and generic modules that can be attached together.  Construction blocks like LEGO bricks, for instance, are a typical example of bottom-up design. Everything is built starting from the very same blocks, which are designed to be as general and multi-purpose as possible. |

Let’s imagine the Update method for a hypothetical *Super Mario* clone. We are interested in coding the movement behaviour. The top-down approach would look like this:

|  |
| --- |
| void Update ()  {  if ( **MoveKeyPressed()** )  {  **Move();**  }  if ( **JumpKeyPressed()** )  {  **Jump();**  }  } |

The code above has been written assuming the existence of methods like MoveKeyPressed and Move, which are not available in Unity. While this code alone does not compile, it provides a *high-level* description of our problem. The complexity of the implementation details (*moving* versus *how to move*) is hidden inside those methods.

In order to make the code compile, one needs to revisit all of them to repeat the process. For instance, JumpKeyPressed could simply return true when the space bar is pressed:

|  |
| --- |
| bool JumpKeyPressed ()  {  return Input.GetKeyDown(KeyCode.Space);  } |

Likewise, Jump could simply apply an *impulse* (an instantaneous *force*) to make the character move upward:

|  |
| --- |
| Void Jump ()  {  Rigidbody.AddForce(Vector2.up \* JumpForce, ForceMode2D.Impulse);  } |

In this simple example, both JumpKeyPressed and Jump are a single line of code each. However, Jump() is much easier to read and to understand compared to Rigidbody.AddForce(Vector2.up \* JumpForce, ForceMode2D.Impulse).

Mastering the top-down approach is tricky. It is important to try and keep the code as balanced as possible. The objective is to iterate on the problem in order to break it down into progressively smaller, low-level tasks that are easier to tackle.

# Finite State Machine

One of the most common ways to structure complex code is to rely on **finite state machines**. A finite state machine (*FSM*) is a diagram in which there are **states** (sometimes called *nodes*) and **transitions** which allow changing from a state to another.

Each state can have an *operation* that is performed while that state is active. And each transition is associated with a *condition* (which triggers the state change) and an action that is performed when the transition is used.

FSMs can be used to model virtually anything. The one below, for instance, explains the transition between *Small Mario* and *Super Mario*:



Finite State Machines are often used in combination with a top-down approach to writing very complex code.

|  |
| --- |
| **💡 Implementing FSMs**  There are countless ways to implement finite state machines in your code. This tutorial is based on **switch statements**, although many implementations in C# rely on more advanced and flexible techniques such as **delegates**. |

## Modelling Game Logic with FSM

For the purpose of this tutorial, let’s imagine a simple *Crush the Castle* clone, in which the player can:

* Use the arrow keys to rotate a cannon
* Charge the projectile by pressing and holding down the space bar
* Fire the projectile by releasing the spacebar
* Wait for the projectile to reach a halt before being able to fire again

A possible FSM could be the following:



Having a correct representation of the states and their transitions is essential before moving to the next step.

|  |
| --- |
| **⚠ Unity and FSM**  While designing a FSM, is important to keep in mind the architecture and limitation of the framework that is being used.  If you are developing a Unity game, for instance, collisions are usually detected in special methods such as OnCollisionEnter2D. If one of your transitions is based on such an event, the code for your FSM might have to be split across different methods, becoming much harder to understand. |

## Enums

Variables can hold many different **types** of data. For instance, float variables can store numbers, while bool ones can only store *boolean* values (either true or false). C# also allows creating special variables that can have only certain specific values. They are called **enums**, and are often used to represent the state of a game.

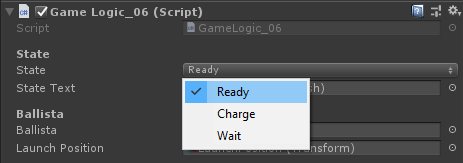
For instance, the following code creates a new type called GameState, which can have only three values: Ready, Charging and Waiting.

|  |
| --- |
| public enum **GameState**  {  Ready,  Charging,  Waiting  } |

Once declared, GameState can be used as a variable type, like float, int or bool:

|  |
| --- |
| public **GameState** State = GameState.Ready; |

Unity works very well with enums. In fact, any enum variable exposed in the inspector will appear as a drop-down menu.



|  |
| --- |
| **💡 Why enums?**  Enums do not really add anything new to the language. One could have simply used an integer variable to store the state of the game, assuming (for instance) Ready to be 0, Charging to be 1 and Waiting to be 2.  That is definitely possible, but the risk is that such a variable could potentially assume values that do not represent a valid state, like 3. Enums remove such a problem, ensuring the value can only have the designated values. |

## Switch Statement

Enums are designed to work together with another feature of C#: the **switch statement**. It is a way to write conditions based on the value of an enum.

To continue the example of the *Crush The Castle* game:

|  |
| --- |
| public GameState State = GameState.Ready;  void Update ()  {  **switch** (State)  {  **case** GameState.Ready:  ...  **break**;  **case** GameState.Charging:  ...  **break**;  **case** GameState.Waiting:  ...  **break**;  }  } |

Within the curly brackets of the switch statement, there is a **case statement** for each possible value of the State variable. The code is executed until the break keyword.

Everything that can be done with switch statements can actually be done with traditional if statements. For instance:

|  |  |
| --- | --- |
| public GameState State;  void Update ()  {  **switch** (State)  {  **case** GameState.Ready:  ...  **break**;  **case** GameState.Charging:  ...  **break**;  **case** GameState.Waiting:  ...  **break**;  }  } | public GameState State;  void Update ()  {  **if** (State == GameState.Ready)  {  ...  } **else**  **if** (State == GameState.Charging)  {  ...  } **else**  **if** (State == GameState.Waiting)  {  ...  }  } |

The advantage of preferring switches over ifs is not really *technical*, but *intentional*. Switch statements are usually used to highlight the intention of covering all possible states of an enum. C# will, in fact, raise an error if you are trying to have two cases over the same value. By comparison, you can indeed have multiple branches of an if statement with the same condition.

## Translating FSMs to C#

We now have all the necessary tools to translate a finite state machine into C# code. The idea is to use a switch statement to execute a different piece of code for each state. The transitions between states are implemented using if statements.

Since FSMs are used to represent game logic, they should refer to concepts that are connected to the logic of the game. This is done by using a top-down approach, grouping all low-level code into higher level abstractions.

The rules are simple:

* Create an enum to store the current state of the FSM
* Use a switch statement in Update
* Write the *instructions* to be executed in each state in a function (return type: void)
* Write the *condition* of each transition in a function (return type: bool)
* Write the instructions to be executed when a transition is triggered in a function (return type: void)
* Inside each case branch of the switch statement, add calls to the previously created functions and an if statement for each transition

The following piece of code shows a possible implementation of the FSM introduced before.

At this stage, we are only interested in capturing the logic of the game, not the details of its implementation. For this reason, we write the code assuming we have already a function for each *state*, *condition* and *transition*:

|  |  |
| --- | --- |
| public GameState State;  void Update ()  {  **switch** (State)  {  **case** GameState.Ready:  Aim();  if ( SpacePressed() )  {  StartCharging();  State = GameState.Charging;  }  **break**;  **case** GameState.Charging:  Charge();  if ( SpaceReleased() )  {  FireProjectile();  State = GameState.Waiting;  }  **break**;  **case** GameState.Waiting:  if ( ProjectileStopped() )  {  State = GameState.Ready;  }  **break**;  }  } | |

By using this approach, the code written maps *one to one* to the FSM.

The next step is to actually revisit those newly created functions, implementing them:

|  |  |
| --- | --- |
| // Rotates the cannon using the vertical axis (up/down keys)  **void Aim ()**  {  float v = Input.GetAxis("Vertical");  Cannon.Rotate(0, 0, v \* RotationSpeed \* Time.deltaTime);  }  // Return true when the spacebar is pressed  **bool SpacePressed ()**  {  return Input.GetKeyDown(KeyCode.Space);  }  // Charges the power of the projectile  // counting the time (in seconds) the spacebar was being pressed  **void Charge ()**  {  CurrentCharge += Time.deltaTime;  }  // Returns true when the spacebar is released  **bool SpaceReleased ()**  {  return Input.GetKeyUp(KeyCode.Space);  }  // Instantiate a new projectile where the cannon is,  // and launches the projectile based on the current charge  **void FireProjectile ()**  {  Projectile = Instantiate (ProjectilePrefab,  Cannon.position, Cannon.rotation);  Projectile.velocity = Projectile.right \* CurrentCharge;  }  // Returns true when the velocity of the Projectile is small enough  **bool ProjectileStopped ()**  {  return Projectile.velocity.magnitude < 0.05f;  } | |

# From the Asset Store

Finite State Machines are one of the most common tools used model and represent game logic. There are several professional assets available online that you can use in your future commercial games. These assets might allow you adding complex logic to your game, without having to write actual C# code.

Below you can find some of the most popular assets. Please, be aware that no assets are necessary for this module, but they could be a good starting point if you are planning to work on something more professional.

|  |
| --- |
| **💰 Behaviour Machine PRO**  There are many assets on the Unity Asset Store that can help you drawing state machines visually. The most popular is [Behaviour Machine PRO](https://assetstore.unity.com/packages/tools/visual-scripting/behaviour-machine-pro-16224?aid=1100l45Ay), which also converts the diagrams into code that is very easy to read. |

|  |
| --- |
| **💰 Bolt Visual Scripting**  One of the most versatile tools for visual scripting in Unity is [Bolt](https://assetstore.unity.com/packages/tools/visual-scripting/bolt-87491?aid=1100l45Ay). It allows creating not just the structure of a state machine, but also the code that runs inside it. |