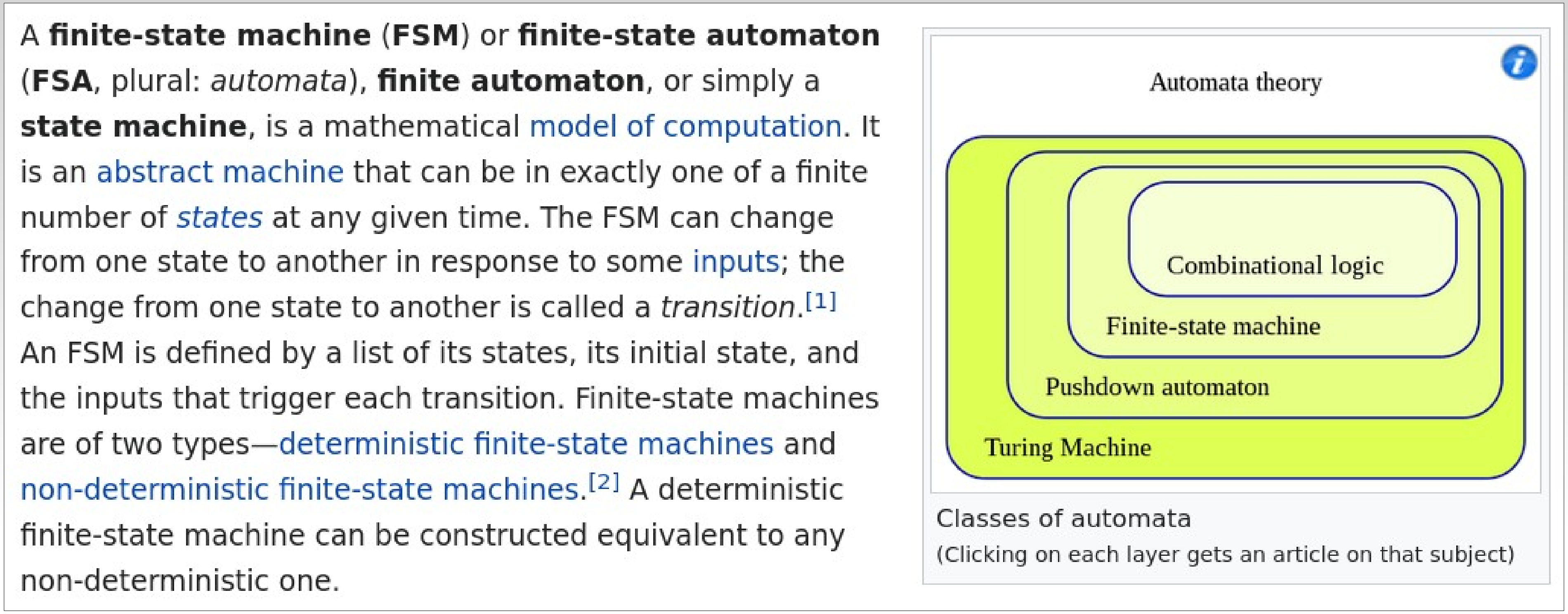
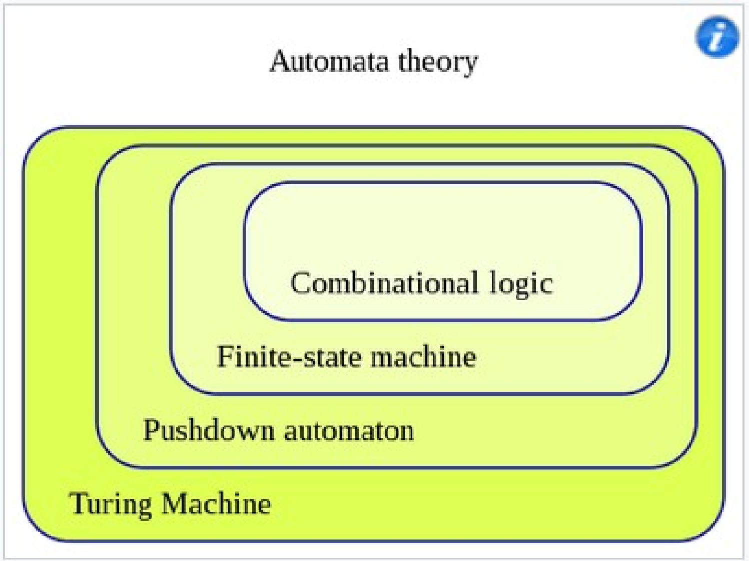
1. Máquinas de Estados

Programação de Jogos
`https://github.com/fsantanna-uerj/Jogos/`

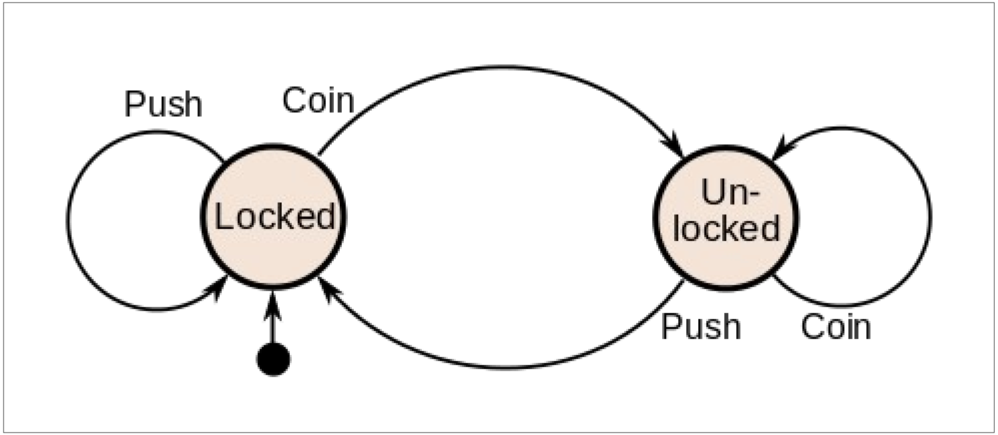
Francisco Sant'Anna
`francisco@ime.uerj.br`



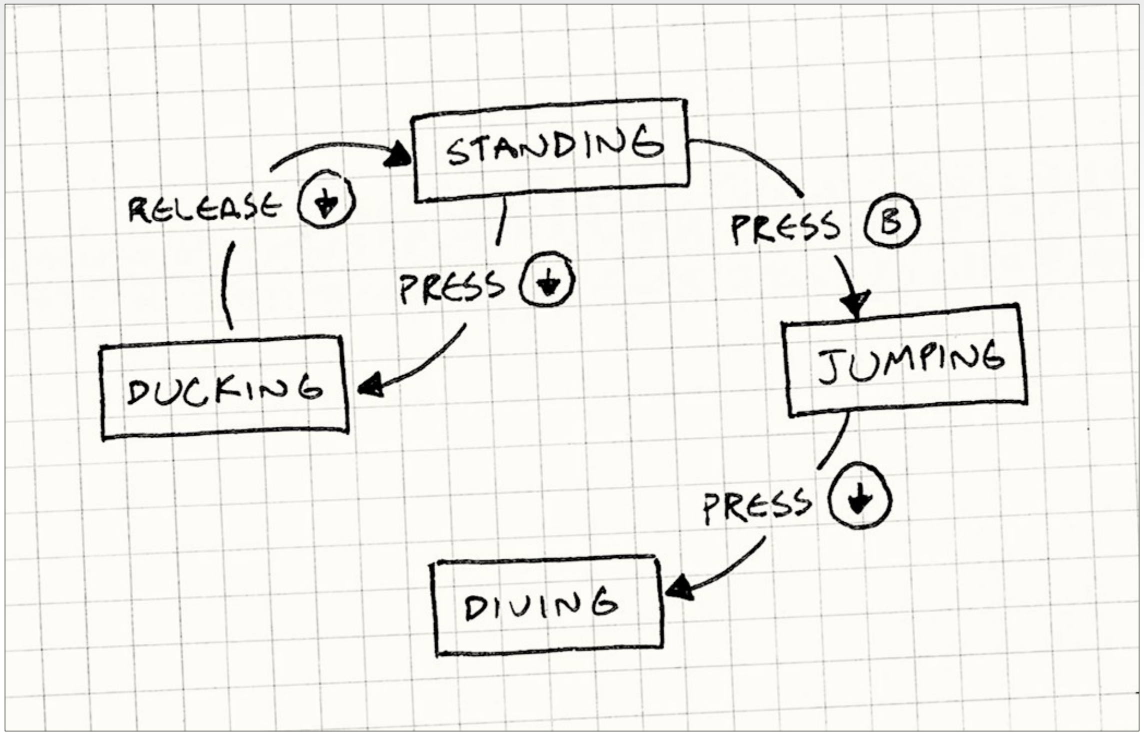
A **finite-state machine** (**FSM**) or **finite-state automaton** (**FSA**, plural: *automata*), **finite automaton**, or simply a **state machine**, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of *states* at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a *transition*.[1] An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines.[2] A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.



Classes of automata
(Clicking on each layer gets an article on that subject)



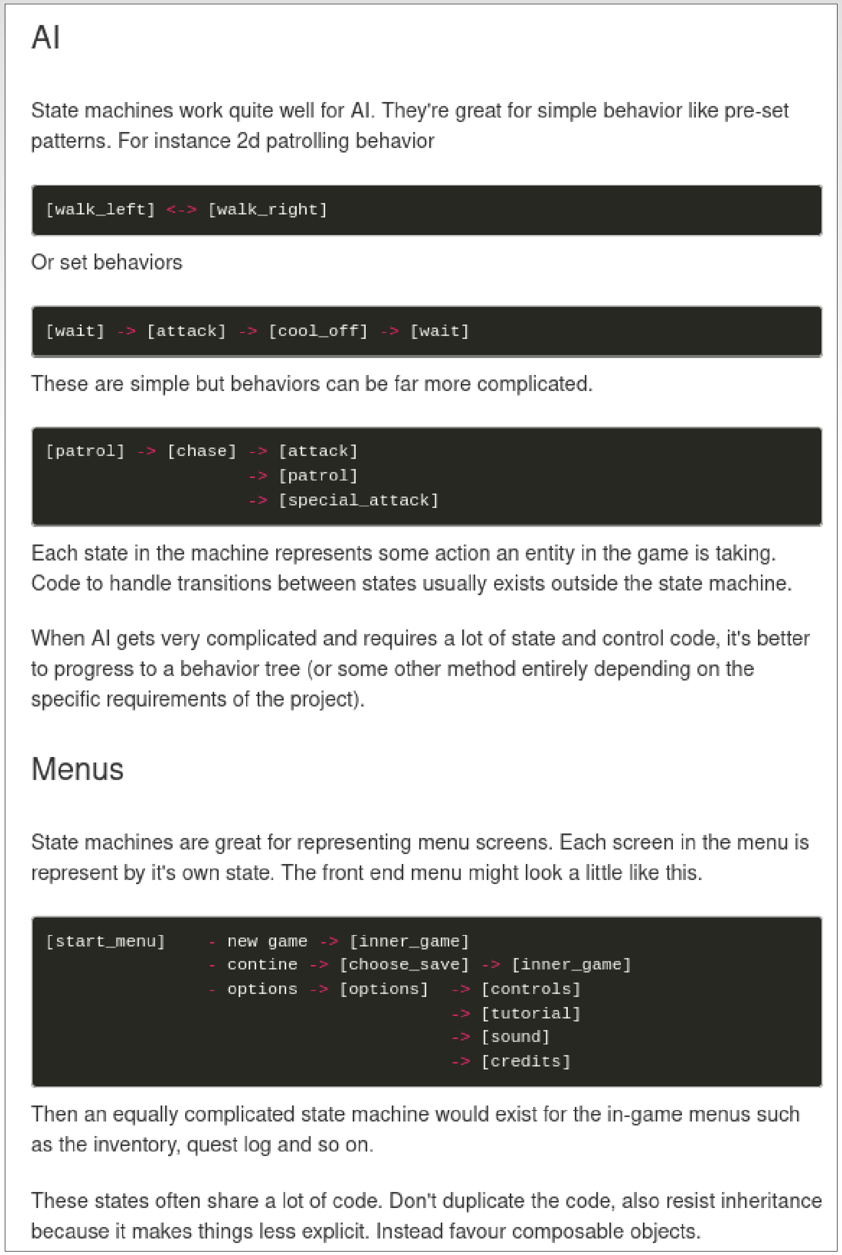
Jogos e FSMs



```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // Jump...
    }
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    if (isDucking_)
    {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

▪ Usos

http://howtomakeanrpg.com/a/state-machines.html



AI

State machines work quite well for AI. They're great for simple behavior like pre-set patterns. For instance 2d patrolling behavior

```
[walk_left] <-> [walk_right]
```

Or set behaviors

```
[wait] -> [attack] -> [cool_off] -> [wait]
```

These are simple but behaviors can be far more complicated.

```
[patrol] -> [chase] -> [attack]
                    -> [patrol]
                    -> [special_attack]
```

Each state in the machine represents some action an entity in the game is taking. Code to handle transitions between states usually exists outside the state machine.

When AI gets very complicated and requires a lot of state and control code, it's better to progress to a behavior tree (or some other method entirely depending on the specific requirements of the project).

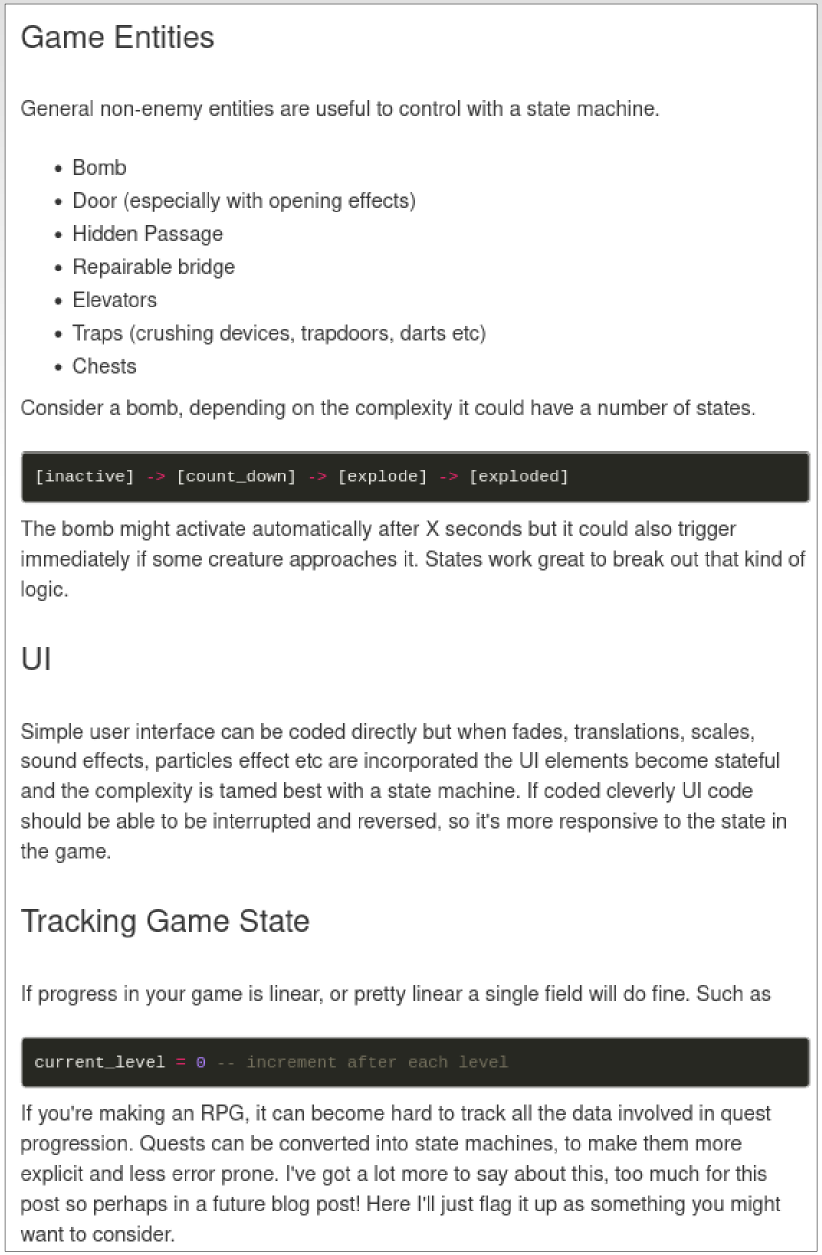
Menus

State machines are great for representing menu screens. Each screen in the menu is represent by it's own state. The front end menu might look a little like this.

```
[start_menu]     - new game -> [inner_game]
                 - contine -> [choose_save] -> [inner_game]
                 - options -> [options]  -> [controls]
                                         -> [tutorial]
                                         -> [sound]
                                         -> [credits]
```

Then an equally complicated state machine would exist for the in-game menus such as the inventory, quest log and so on.

These states often share a lot of code. Don't duplicate the code, also resist inheritance because it makes things less explicit. Instead favour composable objects.



Game Entities

General non-enemy entities are useful to control with a state machine.

- Bomb
- Door (especially with opening effects)
- Hidden Passage
- Repairable bridge
- Elevators
- Traps (crushing devices, trapdoors, darts etc)
- Chests

Consider a bomb, depending on the complexity it could have a number of states.

```
[inactive] -> [count_down] -> [explode] -> [exploded]
```

The bomb might activate automatically after X seconds but it could also trigger immediately if some creature approaches it. States work great to break out that kind of logic.

UI

Simple user interface can be coded directly but when fades, translations, scales, sound effects, particles effect etc are incorporated the UI elements become stateful and the complexity is tamed best with a state machine. If coded cleverly UI code should be able to be interrupted and reversed, so it's more responsive to the state in the game.

Tracking Game State

If progress in your game is linear, or pretty linear a single field will do fine. Such as

```
current_level = 0 -- increment after each level
```

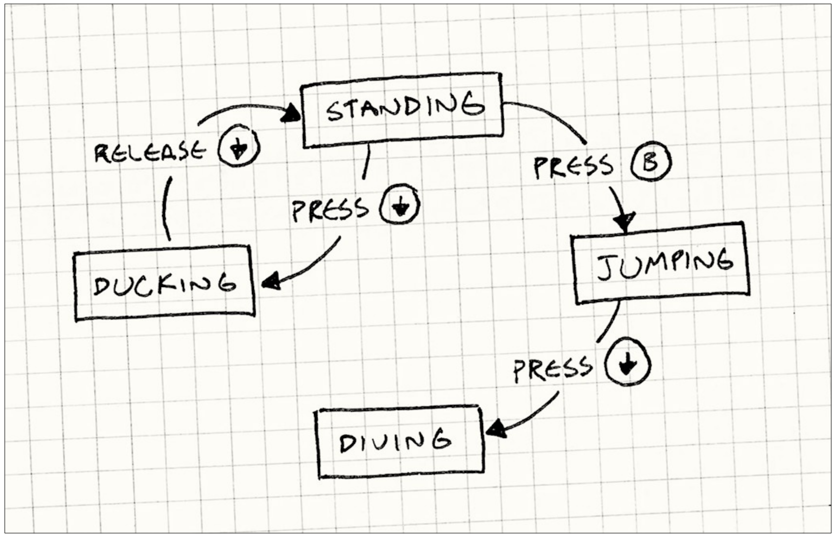
If you're making an RPG, it can become hard to track all the data involved in quest progression. Quests can be converted into state machines, to make them more explicit and less error prone. I've got a lot more to say about this, too much for this post so perhaps in a future blog post! Here I'll just flag it up as something you might want to consider.

Links

- Wikipedia: Finite State Machine
  - https://en.wikipedia.org/wiki/Finite-state_machine
- Livro: Game Programming Patterns
  - https://gameprogrammingpatterns.com/state.html
- Blog:
  - http://howtomakeanrpg.com/a/state-machines.html

Exercícios

1. Desenhe uma FSM para o retângulo do Exercício 2.0
2. Desenhe uma FSM para a principal entidade do seu jogo.
    - Não é necessário implementar nada
    - Use como base o diagrama abaixo



1. Máquinas de Estados

Programação de Jogos
`https://github.com/fsantanna-uerj/Jogos/`

Francisco Sant'Anna
`francisco@ime.uerj.br`

2. Representação de FSMs

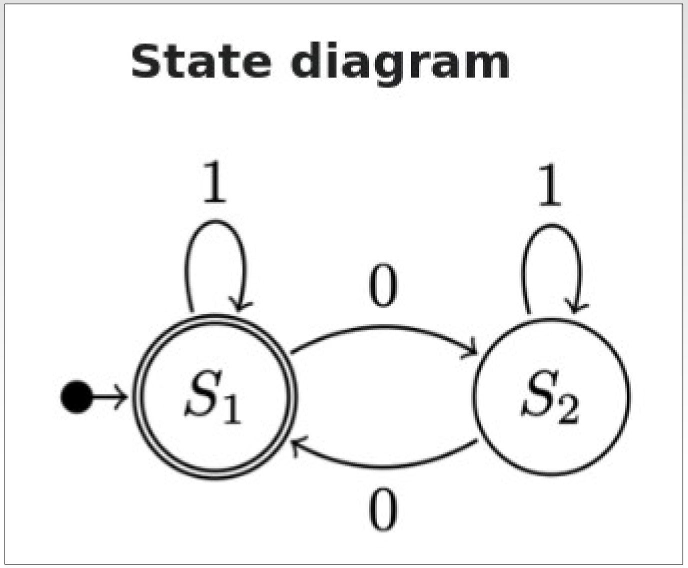
Programação de Jogos
`https://github.com/fsantanna-uerj/Jogos/`

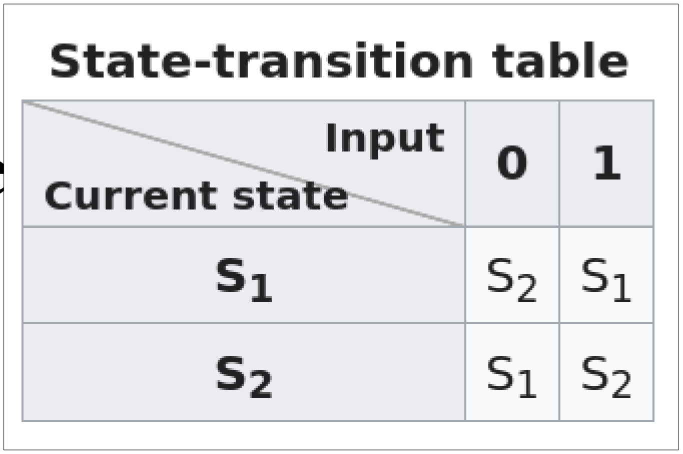
Francisco Sant'Anna
`francisco@ime.uerj.br`

FSMs

- Entidades
  - Estados + Estado Inicial
  - Eventos (Entradas)
  - Transições
  - Efeitos (Saídas, Ações)

- Representações
  - Grafo Direcionado (Diagrama de
  - Tabela de Transições

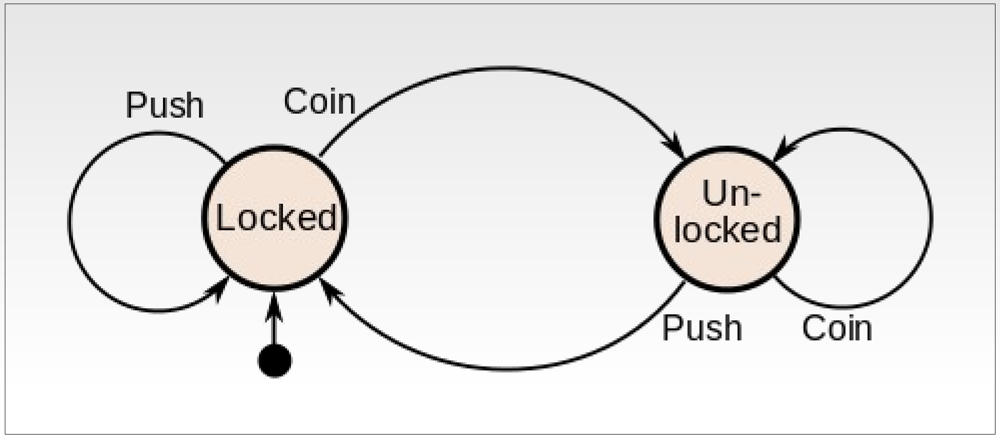




State-transition table

Current state \ Input	0	1
S_1	S_2	S_1
S_2	S_1	S_2

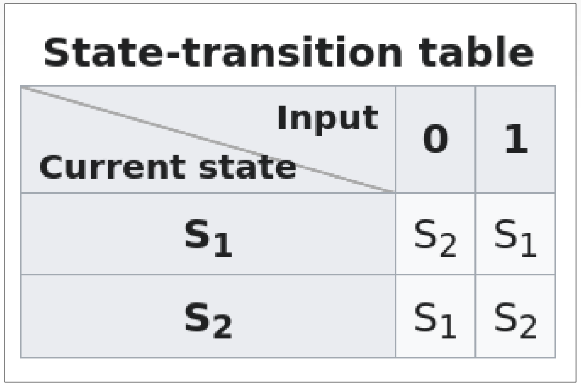
Exemplo: Roleta



Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

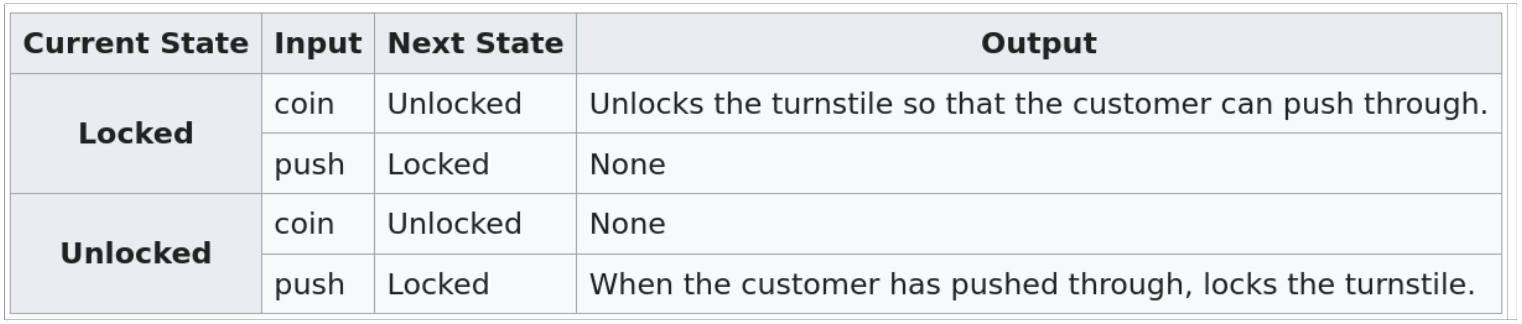
Exercícios

1. Construa uma Tabela de Transições para 2.1.1
2. Construa uma Tabela de Transições para 2.1.2



State-transition table

Input / Current state	0	1
S_1	S_2	S_1
S_2	S_1	S_2



Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

2. Representação de FSMs

Programação de Jogos
`https://github.com/fsantanna-uerj/Jogos/`

Francisco Sant'Anna
`francisco@ime.uerj.br`

4. Extensões de FSMs

Programação de Jogos
https://github.com/fsantanna-uerj/Jogos/

Francisco Sant'Anna
francisco@ime.uerj.br

Extensões de FSMs

- Concurrent State Machines
- Hierarchical State Machines
- Pushdown Automata