A Lightweight Finite State Machine C++ Library aimed at Seamless Integration with Robotic Middlewares

David Estevez, Juan G. Victores and Carlos Balaguer

***Abstract*— Robotics is being heavily integrated in a great number of daily-life applications, such as domestic assistance, autonomous driving, and healthcare. Other applications, such as entertainment, are beginning to integrate robotic elements. A robotic behavior, however complex, can in general be modelled and implemented as a Finite State Machine (FSM). While these robotic behaviors may typically be implemented using high-level scripting languages such as Lua or Python, or even proprietary solutions, a set of use cases where the efficiency of C++ must be invoked exist. The use case presented in this paper is Robot Devastation, a video-game that combines Augmented Reality elements with robots to emulate combats between robots. Robot Devastation strives for code efficiency to enhance the end-user game play experience. In this work, we present the lightweight reusable C++ library that has been developed for implementing its FSM, reluctantly called StateMachineLib within the rd:: namespace. Aimed at seamless integration with robotic middlewares, let its current loosely coupled integration with the well-known robotic platform YARP serve as a living example its flexibility and usefulness.**

I. INTRODUCTION

Although robotics and automation have been traditionally focused on improving industrial production performance and capabilities, there is a rising trend of applying robotics in domestic environments and tasks. The main goal of these robots, such as robotic vacuum cleaners or cooking helpers, is to leverage the amount of everyday chores that humans have to perform at home. Nonetheless, an increasing number of robotics devices are beginning to be used for more diverse applications, such as education and entertainment.

In entertainment, robotics has the potential to impact with novelty and expand the currently available or traditional games in ways never seen before. Physical interaction with the real world allows a more natural interaction with the game elements. However, when using real-world robots as avatars for the players instead of virtual characters, several issues arise that challenge the enjoyment of the players. Robots are complex systems that are affected by real-world issues such as noise, communication latency, battery capacity, camera resolution, etc.

In our previous work [1] we proposed an architecture that combines Augmented Reality (AR) elements with robots to enhance the gaming experience while mitigating some of the aforementioned issues. One key element of a game is a state machine to control the game flow and actions available at each game situation. Whereas Robotic Middlewares (such as ROS[2] or YARP[3]) are typically focused on interfacing with robot hardware and can provide support for advanced data processing algorithms for navigation, collision avoidance, etc; a lightweight FSM C++ library aimed at being losely coupled with different robotic middlewares was not found within the current state of the art.

In this work, we present the FSM architecture we have developed within our Robot Devastation augmented reality robot video-game, and integrated with YARP in a losely coupled manner. It provides a simple and straightforward way of describing states and integrating them in a FSM.

All of the authors are members of the Robotics Lab research group within the Department of Systems Engineering and Automation, Universidad Carlos III de Madrid (UC3M). `destevez@ing.uc3m.es`.

II. STATEMACHINELIB

The Robot Devastation StateMachineLib, currently part of the rd:: namespace, is composed by three main classes: State, StateDirector, and FiniteStateMachine.

A. State

State is the base class for each state of the FSM. To create custom states, one must inherit from this class. The state functionality is implemented by overriding the following virtual member functions:

- **setup()**: Function executed right before the loop function, when the state is enabled. Return true/false upon success/failure.
- **loop()**: Function executed periodically when the state is active. Return true/false upon success/failure.
- **cleanup()**: Function excuted when the state is going to be stopped (due to an error or a transition). Return true/false upon success/failure.
- **evaluateConditions()**: This function is called after each call to loop() in order to know the transition to make. An integer value is assigned to each possible transition to identify them. This function must return the transition selected depending on the conditions of the state.

This architecture provides flexibility to the user, allowing him to create both Moore and Mealy Machines, as well as other more exotic state machine architectures, if required. For instance, to create a Moore Machine, whose outputs depend only on the current state, the user would only implement the setup() function with the corresponding outputs and leave the loop() and cleanup() empty. Conditions would then be periodically checked in the evaluateConditions() function.

B. StateDirector

Execution flow of the different states is controlled through a StateDirector class attached to a State. It is not necessary to implement a custom StateDirector for a custom State, as the StateDirector just provides the implementation to control how States are executed and how transitions are performed.

The execution flow (Fig. 1) is the following. When a StateDirector is started, it becomes the active StateDirector, and the associated state's setup() function is called. Then, it enters in a loop, in which, periodically, the StateDirector executes the state's loop() method. Each time the loop() method is executed, the state's evaluateConditions() function is called, obtaining the ID of the next state to be run. If the next state is not the current state, the current state is stopped and the next one is started. Otherwise, the loop proceeds with the next iteration.

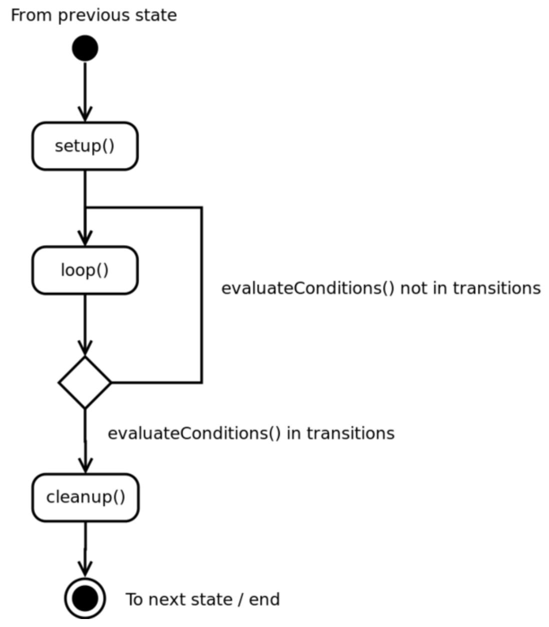
From previous state
setup()
loop()
evaluateConditions() not in transitions
evaluateConditions() in transitions
cleanup()
To next state / end

Fig. 1: Execution flow of Robot Devastation FSM

C. *FiniteStateMachine*

The StateMachine class provides a nice interface to manipulate the StateMachine. States are added and configured and, then, the FiniteStateMachine is executed with the start() function. From that point, the FiniteStateMachine takes care of the execution flow and the deletion of the different states once the execution has finished.

To create a FiniteStateMachine, a helper class called StateMachineBuilder is additionally provided. The StateMachineBuilder follows the builder programming pattern, encapsulating and decoupling the creation of the FiniteStateMachine from its implementation. New states and transitions can be added to configure the FSM through its simple interface. Once configured, the FiniteStateMachine is created and returned to the user, which can then start it.

III. CONCLUSIONS

To validate the concept presented in this work, the a FSM was implemented and integrated in Robot Devastation using the proposed framework. Robot Devastation is a multiplayer Augmented Reality game that uses robots as avatars for the players, emulating robot battles [4]. The FSM is in charge of the game flow control, with 4 main states: init, game, dead and exit. In the init state, the game shows the initial screen and waits for user input to log in the game server. Once it is logged in, the state machine starts the game state, which shows the game display with all the info relevant to the current combat: health, ammo, and health of the other players. Once the health arrives to 0, the player is dead, and the state machine moves to the dead state for 10 seconds. Then, the player can choose to respawn in the game or exit the game. When the player requests to exit the game, either during the game or in the dead state, the exit state is started, where the player is properly logged out from the game server. Each of the states was developed and tested individually, and the later integration of states in the FSM was simple due to the modular nature of the framework. While simple mobile robots and turrets are currently used in game-play, advanced humanoid robotics and unmanned aerial vehicles remain within the Robot Devastation agenda.

ACKNOWLEDGMENT

This work was supported by RoboCity2030-III-CM project (S2013/MIT-2748), funded by Programas de Actividades I+D in Comunidad de Madrid and EU.

REFERENCES

[1] D. Estevez, J. G. Victores, and C. Balaguer, "A New Generation of Entertainment Robots Enhanced with Augmented Reality," in *RoboCity 2016*, 2016.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[3] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006.

[4] D. Estevez, J. Victores, S. Morante, and C. Balaguer, "Robot Devastation: Using DIY Low-Cost Platforms for Multiplayer Interaction in an Augmented Reality Game," *EAI Endorsed Transactions on Collaborative Computing*, vol. 15, no. 3, pp. 1–5, 2015. [Online]. Available: http://dx.doi.org/10.4108/icst.intetain.2015.259753