Dealing with Automatic Reload on a Running Application

David Dinis, Henrique Freitas, João Pinto, Moisés Rocha
Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, 4200-465 PORTO
up{201706766, 201707046, 201705547, 201707329}@fe.up.pt

***Abstract*— This work aimed at developing an engine that software developers can use to build upon that ensures hot reload. Interaction based applications benefit greatly from instant debugging features, where reaching a state might be difficult. This is tested with a simple snake game. Problems faced were documented and reported at the end.**

I. INTRODUCTION

In terms of software development, live testing is constantly critical for programs heavily dependent on end-user input and feedback. Game development benefits significantly from this because a game flow is a continuous chain of states and actions, and some states are hard to achieve, taking time to reach there and adequately test.

Unity is a game engine with an editor that enables hot reload with significant success; when running the application inside the editor, the developer can freely change things and see them in action right away, without waiting for the game to restart.

Web Development is another area that is impractical to start every time the source code is changed; sometimes, all a developer wants to do is center a div element!

So, if a particular feature being implemented needs a considerable amount of time and/or effort to reach in the running program, testing and debugging might be impractical and an extremely tedious and slow process.

How have we decided to tackle this issue? At first glance, it needs to be context-agnostic to be a robust, generalized tool. Then, just caching the last state before closing and considering it the next time it runs is not a good solution because big applications might need a tremendous amount of time to start. States need to be contextualized, and the user needs to build the actions as he wants them to interact with the states. The array of actions will be applied to the states, composing a chain of consecutive states and actions; each state results from the previous action applied to the predecessor state.

This way, the first outline defines an engine that does not reload and can react accordingly when something is changed in the action-state space of the program by the user.

II. ENGINE

We decided to make an engine that was independent from any context the end user might use it for.

With this engine, the developer will be able to define a state model, as well as actions that modify the state. The engine will then receive sequences of actions and execute them sequentially over the state. It will enable users to modify the action sequence and apply rollback whenever the modification affects already executed actions. This allows users to quickly iterate and correct program behaviour without having to rerun the entire sequence over and over.

It was implemented as a tick-based engine, allowing the user to define when the engine should advance and investigate the state in between actions, this allows to user to, for example, create visualizations of the state evolution or apply state validation. It also provides facilities for the user to modify the state outside of the actions.

A. State

The state that the engine will work over is a type defined by the user of the engine library. The user is free to define it however they want, however, it is expected for state objects to be immutable, this allows for simpler logic and eliminates the worries of multiple objects owning the same state object.

B. Actions

Actions are operations executed over sate, they receive an instance of a state object and transform it into a new, different state. When implementing actions, the users must override their ”run” function, this function returns a status value that instructs the engine how to proceed:

- SUCCESS: the action ran successfully, the engine can skip to the next.
- FAIL: the action failed to run (e.g. an exception was thrown).
- RUN_AGAIN: the action needs to be run again, the engine shouldn't skip to the next one, good for repeating actions.
- INVALID_STATE: the state is or became invalid.

RUN_AGAIN means that actions may have their own internal state, while INVALID_STATE means that actions can have their own state validation.

C. Features

The engine is implemented with the following internal properties:

- Initial State: the initial state with which the engine is going to working, it must be provided when the engine is instantiated.
- Current State: the current instance of the application state, it is the state object over which the next action will be executed.
- Actions List: the list of actions the engine is working with, ordered in order of execution.

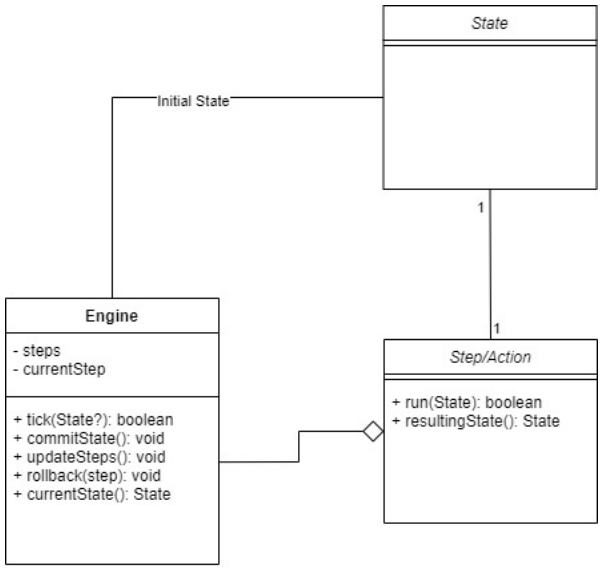

Fig. 1. Engine UML

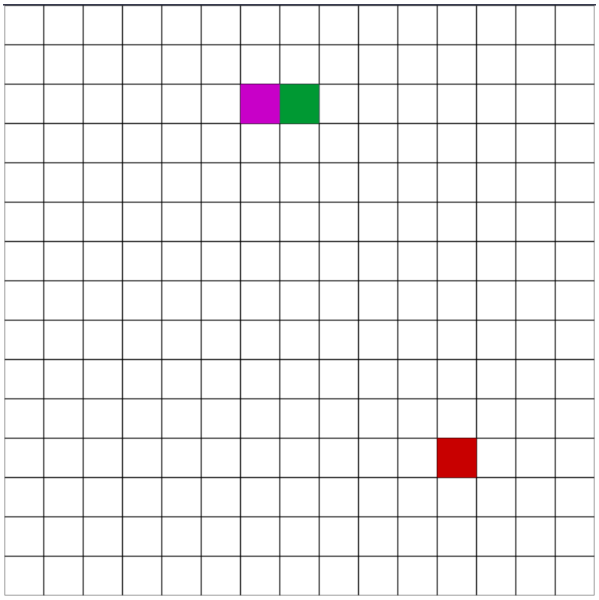

Fig. 2. Example Demo State

- Current Action: a pointer to the next action to be executed.

It then provides some features/functions that operate over these properties.

1) Tick: In order to progress in the program a tick functionality was needed. Each tick executes an action and updates the current state. Depending on the result of the action, as mentioned above, it advances the current action pointer.

2) Rollback: When needing to correct a mistake, changing the order of the action or simply undoing an action to test alternatives, there is a need to roll back the engine. The rollback function does precisely this. Given a past action, the engine undoes all actions taken after it, restoring the current state to how it was before the specified action was executed. It also sets the current action pointer accordingly.

3) Commit: When using the engine for a long time, the actions list starts to get lengthy and difficult to navigate and use. So, the commit function makes all executed actions final, turning the current state into the present initial state and removing from the actions list all actions that were already executed. This way the developer can focus on implementing the behaviour as a sequence of phases.

4) CRUD for action array: The user should not change the action array by himself. When editing the array, the engine needs to ensure that the application run-time is valid, so the program doesn't need to be restarted. The goal of this tool is precisely to avoid such situations, improving quality of life for the developer. The current state depends on the influence of all previous actions.

III. Demo Example - Snake Game

To test the engine a game was needed. We settled in a snake game using React that is fast and easy to develop showed in figure 2. Adjacent to this, an interface that communicates the actions to the engine was required. This

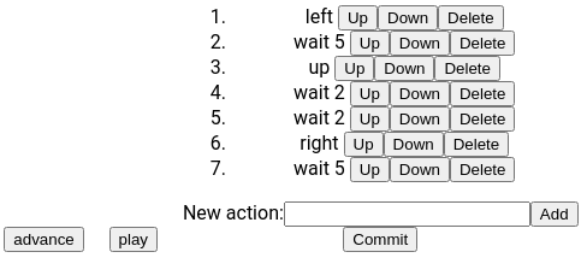

Fig. 3. Example Demo Interface

interface can change the actions position in the queue, add or remove actions, like shown in figure 3.

The game has 5 actions:

- Up
- Left
- Down
- Right
- Wait

The first 4 control the direction of the snake head, while the wait action takes an argument that represents the amount of ticks that the snake has no action: this results in it moving forward in the direction the snake is facing at the moment. This means that the engine is responsible only for controlling the snake's movement direction.

In between ticks, the game state is advanced: the snake moves forward, and collision with things like fruits and its own body are calculated. Resulting in there being state modification and validation outside the engine, like it was implemented to allow.

When playing the game the player will then define the moves they want the snake to take in order to eat the most

fruit possible while avoiding a collision with itself. The commit feature is available, an example of its usage is to commit every time the snake eats a fruit, as each fruit that appears can be thought of as a phase of the game.

IV. Problems We Faced

A. State immutability

Since we are allowing the user to access the state in between actions, we need to make sure the state isn't adulterated which can lead to problems like rolling back to an incorrect state. To mitigate this, every time the user asks to engine for the current state, it returns a clone of it, this makes sure that changes to the returned state don't affect the internal logic of the engine.

B. State modifications outside the engine

Another problem arises with allowing the user to modify the state outside the engine, if this happens it will break the tick, rollback and commit features since these depend on the current state and it can be outdated due to modifications outside the engine. To fix this, the tick and commit functions allow the user to provide an optional state object to replace the current state. Actions also keep a copy of the state they initial received so, when rollback is applied, the current state is reset to that one.

C. Rollback of repeating actions

Since actions are defined by the user, they may have internal state that evolves with the tick progression, especially repeating actions that may return RUN_AGAIN. In the snake demo, the wait action has internal state to control how many ticks to wait before proceeding to the following actions. This means that, when rolling back, each action needs to be reset so it behaves the same way every time, if the same state is provide.

D. Integration of the Engine

The engine doesn't have a way to display the array of actions, requiring the user to know which actions are inside the engine's actions list in order to edit them, this means that the user needs to keep track of which action they append to the list, as well as all the modifications they do to the list. A way to fix this, however not implemented, is to allow the actions to identify and distinguish themselves through a parsable value, e.g. a description string or a structured JSON object.

V. CONCLUSIONS

With all that said, we gained a different respect for the applications that have hot reload implemented, that make any developer's life a lot easier regarding software development.