RFC - Game loop

Metacosm Development Team

Revision History

Revision 0.1April, 15th 2000Revised by: Elkine, Horus, Ruffy

The goal of this document is to define the algorithm used in the main loop of the game server in Metacosm project.

1. License

2. Metacosm

For more information on Metacosm, check our site: Metacosm (http://metacosm.org).

3. Why turns?

Let's begin with a fact: about all the RPG or MUD use turn-based system. There are few exceptions (like the Amber™ RPG for example).

Turn-based means synchronous system and non turn-based asynchronous system.

Asynchronous systems are a mess to manage. They are hard to code and even more hard to understand when they run. Nevertheless, they seem closest to our perception of reality. It's also more flexible (at least because synchronous systems are a sub-class of asynchronous ones) and it's said quite adapted to object designed entities (like distributed agents behavior).

Synchronous systems are easy to implement a coherent and persistent time system. They are compliant to the largest number of RPG systems. They are better for equity between players (connection speed or lag are less important), and between players and AI (reacting faster than a one million times faster AI is really hard). They also have a high reliability.

It can seem that a turn-based system will be more complex to set up than an event-driven one, but that's only at the beginning. Dealing with asynchronicity will require to put more safeguards, and it will become more difficult to foresee what could possibly happen. We are here talking of thousands of Entities here, not just five or ten, and the complexity'll reach an unmanageable threshold.

For all there reasons, we choose, at least for now, to use a turn-based system.

4. Turn notion

The turn is the smallest time unit. Each action duration is a multiple of this unit. Actions have a default duration. An action can be interrupted at each turn.

The turn has a duration (TURN_DURATION), but there is no guarantee this duration will be respected. if it doesn't, an error is logged.

5. Interruption of actions series

A special keyword can be used to interrupt the current action and to flush the actions buffer. The I/O thread receives it and purges the actions buffer of the Player. At the next turn, the current action (if there is any) will be interrupted.

6. I/O thread

The I/O thread manages connections with Players (input from Players and outputs to Players). It also deals with spam. Commands sent are immediately executed for social commands (discussions outside the game) or actions outside the game (exit, reading help, etc), or stored in a buffer and delayed. This is this buffer that can be purged.

7. The raw algorithm

```
while the game is not stopped
  // Order for acting
  sort Players/Controllers by priority

  // Actions
  for each Player/Controller
    if command available
      execute the command
      resolve action (can throw some Stimuli and Events)
    else
      do nothing


  // StimuliDispatchers
  for each StimuliDispatcher
    send Stimuli to its listeners
```

```
  // EventsDispatchers
  for each EventsDispatcher
    send Events to its listeners

  // Plan new actions
  // nc = number of Controllers
  // X  = number of threads for planning
  for i from 1 to (nc div X)
    for j from 1 to N
      start thread j with Controller (i * X + j) as parameter
      wait THREAD_DURATION or all threads finish
      for each thread
        if unfinished
          stop thread
          log error for Controller
          Controller does nothing next turn
  do the same thing for (nc mod X) remaining Controllers

  // End of turn
  log if turn duration is greater than TURN_DURATION
  wait some time if necessary to make the turn dura-
tion equal to TURN_DURATION
```

8. Order for acting

We can use the Reaction capability defined in Entity-RFC. The most you have, the more quickly you act.

9. Actions

At its reaction level, the current action of a Controller is executed. Command name, source and target entities, and parameters are used to resolve the action. It can throw some Stimuli and/or Events.

10. StimuliDispatchers

StimuliDispatchers collect Stimuli during actions resolution phase, and send expected Stimuli to their listeners.

11. EventsDispatchers

EventsDispatchers collect Events during actions resolution phase, and send expected Events to their listeners.

12. Action planning

Each Controller determines its action for the next turn in function of its own memory, of its last actions, of the received Stimuli and Events and sometimes of randomness. It has only a limited time (THREAD_DURATION) to do it. If it doesn't respect this limit, we suppose it went into trouble and decide it does nothing next turn.

To quicken this part, we use several threads in parallel. We use X threads, so we group Controllers by X and the remaining will be treated at the end. For each group of Controllers, we link one Controller with each thread (passed as parameter), start the thread (it'll call the Controller method to determine next action).

The main loop thread will wait until the THREAD_DURATION or the end of all threads (we can use a periodic check). If some threads are not finished, we stop them, log the error and decide the Controller will do nothing next turn.

13. End of turn

If the turn duration is greater than TURN_DURATION, an error is logged. If the turn duration is lesser than TURN_DURATION, we'll wait a bit more. It the turn duration is equal to the TURN_DURATION, damn' we have the force with us! (or perhaps we cheat... :)

14. How to choose TURN_DURATION and THREAD_DURATION?

These two values are highly important. They determine how much time the good guy in front of a display will wait...

It can be fixed values.

Perhaps it's better to determine them dynamically to avoid spam with log errors... We can use a checker which will count errors and increase TURN_DURATION and/or THREAD_DURATION to decrease the errors number. Sometimes, if there are no error during some time (NO_ERROR_DURATION :o)), we can decrease these numbers (after a high overload for example, or if many Controllers disappear). It can be a good method to adapt the game to an overloaded machine or to calibrate automatically the game for the machine.

Joke: can we have a vicious enough guy which will implement this type of vicious bot: the bot do nothing except writing the time spent since its turn activation. So the next time, this bot knows the value of THREAD_DURATION. So it can optimize and use always the max duration to think about what it can do. And even the worst idea: the bot really does nothing. So it creates an error and the THREAD_DURATION will increase. But it continues to do nothing... DoS

To avoid these values become too high (really slow game, too slow so unfriendly for Players) or too low (really quick game, too quick for Players, bots will be like Superheroes for Players and move like Flash (do you imagine the scene? A bot (let's call it Stanley Ipkins) can put your underpants on your head before you can react :)), we should bound these values (MIN_THREAD_DURATION, MAX_THREAD_DURATION, MIN_TURN_DURATION and MAX_TURN_DURATION).

The MAX_* and MIN_* values are the max and min acceptable values for Players.

And no, we won't use a MIN_MIN_TURN_DURATION, a MAX_MIN_TURN_DURATION, ... :p