Updates, Timers and Messages

These exist in system/support/update.lua, system/support/messages.lua and system/support/timers.lua/. Together with the tags, queries and the core these are the basis of the System.

Updates

Updates are the core of the system. They are driven by the Corona enterFrame event. Objects that are updated have their *onUpdate()* method called every frame. They are driven by a singleton class known as system.support.update which is created by the source file.

There are two kinds of update. One that happens every frame, this is known as “raw update”, and one that can be turned off and on, known as “update”. They are applied to “rawupdate” and “update” tags respectively. To get a raw update, an object needs to be tagged to both as it uses either one tag list or the other.

Normally, code that forms part of a scene should be driven using update, so it can be paused (all activity save for Corona transitions is driven by this code) and also so that when a game scene is being transitioned the game will not happen.

void :onUpdate(deltaTime<number>,isRawUpdate<boolean>)

This method is provided for objects tagged with update, and is called when the system is not paused. deltaTime is the time elapsed in seconds since the last update call (these are done as a batch) in seconds, with a maximum of 0.1s if a pause or some other system causing a stop has happened.

If firing this causes an error (e.g. __execute fails) then no more updates of any type will be sent.

boolean :__setTimerRawUpdateOnly(isOn<boolean>) private

This controls whether updates are performed or not (raw updates are always performed). The method takes a boolean, whether *only* rawupdate tagged objects are updated, and returns the previous state.

This method effectively selects which tag index “update” or “rawupdate” is used as a source list for objects to be updated. Setting this on effectively pauses the game as no updates will be sent.

Note that messages can still be sent from display objects. Messages and Timers are paused likewise, as these are driven by being tagged with update, so this can be handled by configuring display objects to send messages to their data representations if required - so instead of a touch/tap event directly being handled directly, it sends a message to itself with the event data.

Because the messaging system is update driven (not raw update) and asynchronous it will not be sent until updates are being performed again (e.g. __*setTimerRawUpdateOnly(false)*)

Messages

Objects can communicate with each other (and themselves) by sending themselves messages. These are asynchronous, so sending a message puts it in queue, from which it is dispatched. This is done by a singleton instance of class system.support.messages which is tagged update. A consequence of this is that messaging is turned off when raw updates only is on.

Any message which is sent to an object which no longer exists is lost.

There is a single message sending function, and a message receiving method.

void :send([name <any type>],[target <instance|string>],[body <table>],[delay <number>])

This sends a single message via the asynchronous queue. All parameters are optional.

The name parameter is the identifier for the message. This is determined by the specific class implementation (and can be nil, if there is only one message), but should ideally be a string or named constant for readability

The target parameter can either be a single instance reference, in which case the message is sent only to that, or a string query, in which case the message is sent to all objects satisfying that query. The query is evaluated at the point the message is sent, not when it is created. This defaults to ‘self’ - so just send() sends a message ‘nil’ to the object sending (with a body nil, and zero delay)

The body parameter can contain any data you like, but must be a table. It is optional. The delay allows messages to be sent after a delay time in seconds, so as to ‘do something in 5 seconds’. This defaults to zero.

void :onMessage(name <any type>,body <table>,msgInfo <table>)

This method must be implemented by any class designed to receive a message. The parameters are the name, and body, as sent and also a reference to the object that sent it. These parameters do not of course have to be defined.

The msgInfo table contains information about the message - currently the only member is 'sender' which is a reference to the object that originally sent the message. This table should be considered read only.

Timers

Objects can schedule future events using timers. The code for timers reposes in system.support.timers and has a singleton driven by update which generates events. As with messages, they are asynchronous and are only dispatched and down-counted when raw events only is false.

Timers can only be sent to their originator, and cannot be used for messaging. Timer events are not sent to disposed objects, and if an object has been found to be disposed all subsequent timer events are cancelled automatically.

<any> multipleTimer(target<instance>, delay <number>,[repeatCount <number>])

This is the main timer method. It sends the object a timer event after a given period of time. A timer can repeat as many times as you wish by setting repeatCount (this defaults to 1) or continually until stopped, by setting the repeat count to zero.

A timerID is derived from an internal counter which is incremented with every create, producing an ID like "timer1103" for example. Its type is not guaranteed.

The method returns the timerID, which was internally generated.

These following four functions are attached to all objects via Root, so can be used by any object to generate timer events. Corona events - transitions, timer.performs and so on are permissible but will not be stopped during rawUpdateOnly events unless this is specifically coded.

void cancelTimer(timerID <any>)

Cancels all timers with that ID; this can be none at all or all of them.

<id> :multipleTimer(delay <number>,repeatCount<number>)
<id> :singleTimer(delay <number>)
<id> :repeatingTimer(delay <number>)

These last two methods are shortcuts for readability. singleTimer is multipleTimer call with a repeatCount of 1. repeatingTimer is a multipleTimer call with a repeat count of 0. All return the id of the timer in question.

void onTimer(timerID <any>)

This method is called when a timer event occurs. If the object has been disposed the timer is not sent.

First version 26 Nov 2014
Second version 26 Nov 2014 - removed passing in of timerIDs - they are now all automatically generated, so as to keep uniqueness across the object range.