# Multimedia Fusion's runtime engine.

**Getting the most of the Multimedia Fusion's runtime engine.**

This tutorial is about what goes on inside the Multimedia Fusion runtime engine. What exactly does it do... and when... and how? How does it deal with conditions and actions, etc.? We lift the hood and take a look!

If you're an advanced user - or are about to become one - you'll will find this especially valuable: here some explanations for the sometimes mysterious behaviours of Fusion's runtime engine as well as ways to make your applications run faster. But make sure you fasten your seat belts, this tutorial is a bit more complex than the previous ones!

(Actually, the following information is valid for the whole click range: since Klik&Play, the main principles of the runtime engine has not changed - for compatibility reasons.)

## True and fake events.

Fusion runtime makes a distinction between a **true** event and a **fake** event.
A true event starts with an **event-driven condition**. A event-driven condition is called by the system when the event actually happens, for example a click on the mouse button. Non-event-driven conditions are simply evaluated at each loop.
Only a small number of conditions are truly event-driven. Here is the full list :

*Player object*
Number of lives reaches 0
Player presses ...

*Keyboard object*
User clicks...
User clicks on object...
User click within zone...
User press any key...
( Note that the condition User presses a key is not a true event condition!)

*Timer object*
Timer equals...
Timer is lower than...
Timer is greater than...

*Storyboard Object*
Start of frame
End of frame
End of application

*System object*
Menu option selected...

*Question objects*
  Answer is correct
  Answer is wrong
  Answer equals...

*Any objects in movements*
  Collision with another object
  Collision with the background
  End of path is reached
  Node is reached
  Object leaves the frame area
  Object enters the frame area

*Any object with an animation*
  End of animation

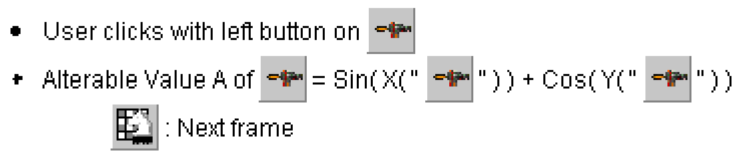
*Any object*
  Last object has been destroyed

**All of the other conditions are not event-driven.** A **fake** event starts with a **non-event driven** condition**.** The first conditions determines the behaviour of the event.

## ☑ What does this change in practice?

It can change a lot, specially the execution speed of your application. If a event-line starts with a true event-driven condition, Fusion only executes it when the event happens.

For example, let's imagine an event that checks for a click and a complex calculation to end the frame:

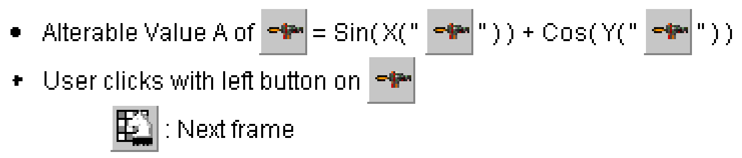


- User clicks with left button on
- Alterable Value A of = Sin( X(" ") ) + Cos( Y(" ") )

  : Next frame

The first condition is a true event-driven condition : Fusion *ignores it completely* until the Windows system indicates a click on the mouse. As a consequence, no processing power whatsoever is needed during 99.9% of the time.

Let's reverse the order of the conditions, and insert the calculation before the click.



- Alterable Value A of = Sin( X(" ") ) + Cos( Y(" ") )
- User clicks with left button on

  : Next frame

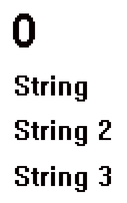
The calculation has to be evaluated for each loop of the application. In our example, Fusion looses valuable processing time each time round the loop calculating the sine and cosine. 99.9% of the time, the second condition is false, so Fusion does not execute the action: the result is of course the same than before, but the main loop has slowed down. The same is true with the collision detection: collision detection conditions should always come first in your lines of events.

*By inserting the true event-driven conditions first in your line of events, you'll maximise the efficiency of the Fusion runtime engine.*

## Tips and trick in the event editor.

### The importance of the order of the events

All the events beginning with a non-event-driven condition are evaluated one after the other at runtime, in the original order of the list. This should be taken in consideration when using counters and testing them. Let's create an example to demonstrate this. In the frame editor, drop one counter object, and three different text objects, placing them where you want in the frame.


0

String

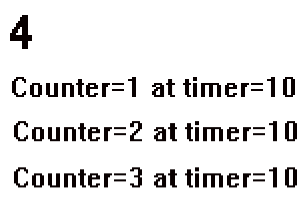
String 2

String 3

Enter the following events in the event editor.

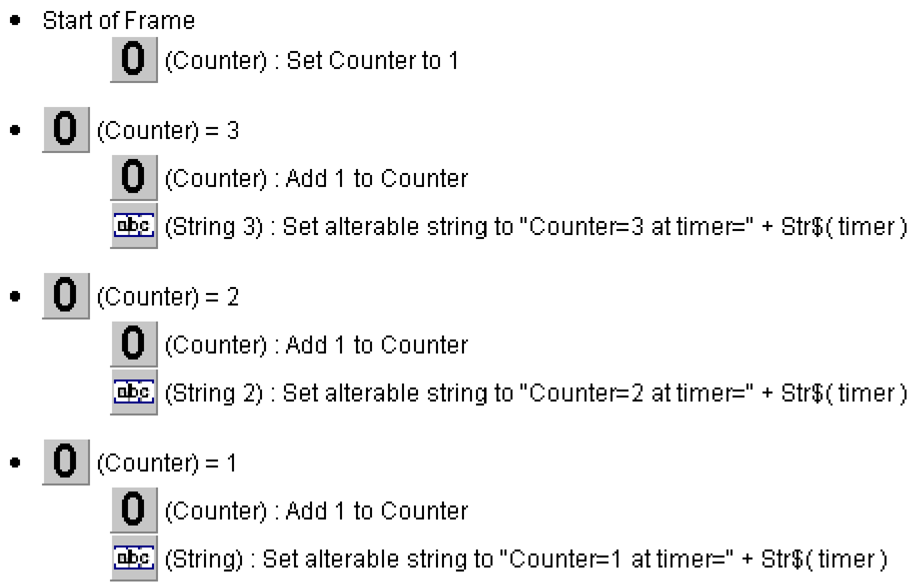
- Start of Frame
  - 0 (Counter) : Set Counter to 1
- 0 (Counter) = 1
  - 0 (Counter) : Add 1 to Counter
  - abc (String) : Set alterable string to "Counter=1 at timer=" + Str$( timer )
- 0 (Counter) = 2
  - 0 (Counter) : Add 1 to Counter
  - abc (String 2) : Set alterable string to "Counter=2 at timer=" + Str$( timer )
- 0 (Counter) = 3
  - 0 (Counter) : Add 1 to Counter
  - abc (String 3) : Set alterable string to "Counter=3 at timer=" + Str$( timer )

The example above initialises a counter at 1 when the frame starts. It then checks the value of the counter, and sets each of the string objects to the current value plus the position of the timer: this tells us exactly when the counter reached the given value during the run. Now run the frame, and you should get something like this:


```
4
Counter=1 at timer=10
Counter=2 at timer=10
Counter=3 at timer=10
```

Of course, you could well get different timer values on different machines. Let's see in detail what happens in our program :
- Start of frame, the counter is set to 1. The current value of the timer is 10 milliseconds (time necessary to perform the frame initialisations).
- As counter equals 1, the second line of events is true, and the **string** object is set to **Counter=1 at timer=10**. Then the counter is increased by 1.
- As Fusion evaluates the events one after the other, the third line of our example is immediately true, so the **string 2** object is set to **Counter=2 at timer=10**, and the counter increased by 1.
- The same happens for the last line of event : **string 3** is set to **Counter=3 at timer=10**. The counter reaches 3.
- No more events to evaluate. Fusion carries on with other tasks such as refreshing the display.
- 20 milliseconds later (exactly 1/50 second), Fusion runs the second loop of the runtime and evaluates the events again. As the counter is set to 4, no action is done.
You can see that the four lines of events were executed during the first loop of the program.

Now, change the order of the lines like this :


```
• Start of Frame
      (Counter) : Set Counter to 1

• (Counter) = 3
      (Counter) : Add 1 to Counter
      (String 3) : Set alterable string to "Counter=3 at timer=" + Str$( timer )

• (Counter) = 2
      (Counter) : Add 1 to Counter
      (String 2) : Set alterable string to "Counter=2 at timer=" + Str$( timer )

• (Counter) = 1
      (Counter) : Add 1 to Counter
      (String) : Set alterable string to "Counter=1 at timer=" + Str$( timer )
```

As you can see, we check for the counters in *reverse order* : 3, 2 then 1. Run this frame - the result should be very different :

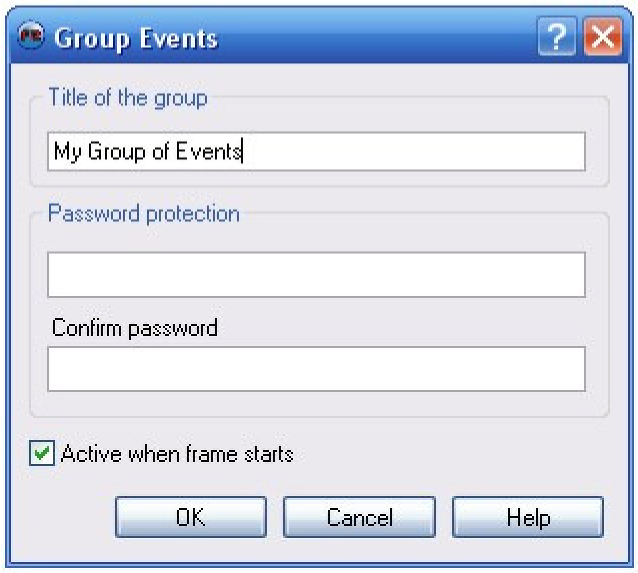
```
4
Counter=1 at timer=10
Counter=2 at timer=30
Counter=3 at timer=50
```

Here too, you may get different values for the timer depending on your machine. Let's see what happens in the program:
- Start of frame (timer equals 10 milliseconds due to initialisation process), the counter is set to 1.
- Fusion evaluates the events, only the last one is true: the **string** object is set to **Counter=1 at timer=10** and the counter's value is increased by one to reach 2.
- No more events to be evaluated: we have reached the end of the list. Fusion carries on with other tasks, such as refreshing the display. This is the end of the first loop.
- 20 milliseconds, exactly 1/50 of seconds later, Fusion runs the second loop of the runtime, and evaluates the list of events once again. As our counter is equal to 2, the **string 2** object will be set to **Counter=2 at timer=30**, and the counter will reach the value 3. Of course the last line (counter=1) is not true, and Fusion carries on with the other tasks.
- 20 milliseconds later, the same process happens, and the **string 3** object is set to **Counter=3 at timer=50**.

The same process as before now took 3 loops of the runtime to be executed, just by reversing the order of the events! You should take that into account when using counters in your list of events!

## A better use of groups.

Groups are certainly the best addition to the click range since Klik & Play. I'm not going to go into their use in complex applications to structure the list of events here - rather, I'll focus on the most handy feature of a group of events: its ability to be activated or deactivated. Careful use of this feature will dramatically reduce the complexity of your list of events. Let's see how....


Group Events

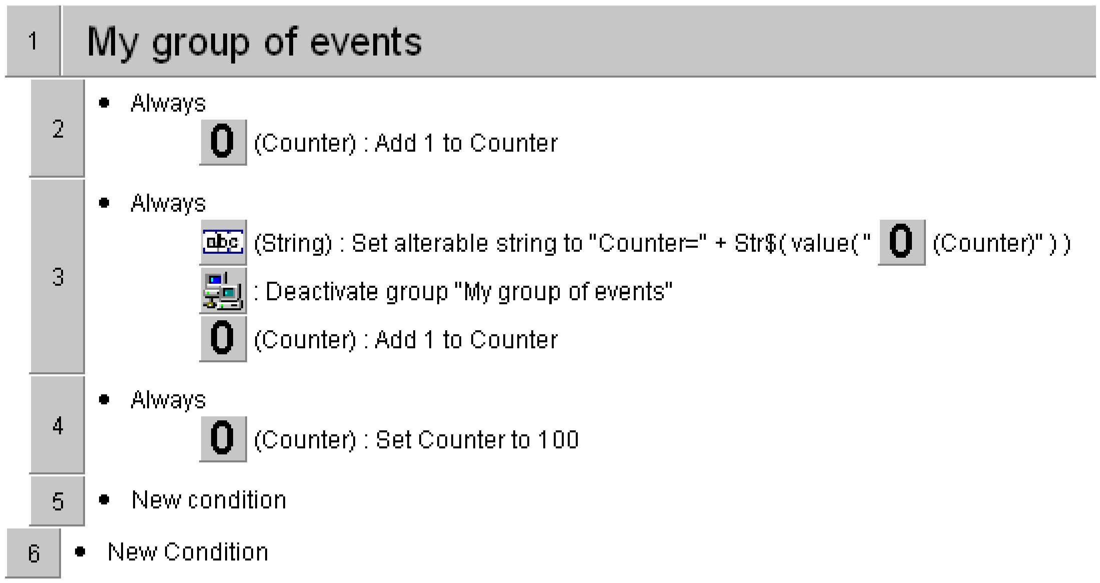
Title of the group: My Group of Events

Password protection:

Confirm password:

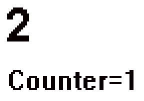
[✓] Active when frame starts

When creating a group of events, you have the choice of activating it or deactivating it prior to running the frame. A deactivated group of events will be completely ignored by Fusion. This will not take any processing time (or very little). Passwords are used in combination with the frames Run-Time option of Password. (which is not part of this tutorial, sorry)
A simple action can activate or deactivate any group of events. Let's see an example: delete the entire list of events of our previous example leaving the objects in the frame, and enter the following events.



```
1  My group of events
   2  • Always
            (Counter) : Add 1 to Counter
   3  • Always
            (String) : Set alterable string to "Counter=" + Str$( value( "  (Counter)" ) )
            : Deactivate group "My group of events"
            (Counter) : Add 1 to Counter
   4  • Always
            (Counter) : Set Counter to 100
   5  • New condition
6  • New Condition
```

Run the frame, you should get the following result.



```
2
Counter=1
```

- The group of events is initialised at first: the first line of the group is executed and the counter is set to 1
- The second line is done immediately: the **string** object is set to **Counter=1**, and the group is deactivated. When an action deactivates the current group it is still executed and Fusion **carries on** with that action's line of events. Therefore the counter will be incremented to 2. Fusion only exits a deactivated group at the next line: the counter will **never** be set to 100.

-> The position of the deactivate action has no importance in the list of actions of the line
-> Events sitting **after** a line with deactivate will **not** be executed.

As you can see, one can very easily create complex lists of tasks to executed **only once** by using **groups**, **always** and **deactivate**. Imagine the creation of a complex character made of many different active objects and counters linked together. Place them in the same group of events, **not activated** at start of frame. Make sure that the last action of this group is to **deactivate** itself. To create one complex character, your program simply has to **activate** this group: it will perform the task only once and quit. It's a kind of **routine** inside events.

Note: as we have seen before, if the **activate** action is placed **before** the group, the group will perform its task during the same loop of the runtime. If the **activate** action sits **after** the group, it will only be executed on the next loop...

## Fusion's runtime main loop.

Every game or demo programmer knows about it: a game is build around the main loop. Funnily enough, the main loop is a programmer's concept that the Click range of products hides completely, as its way of working is based on events and parallel programming. Despite the fact that the Click range does not mention the loop anywhere, it still exists! It is very much there - and it controls the way the runtime engine works!

**Every 1/50 of second, the Fusion's engine does the following :**

1. Checks the player inputs
2. Animates the objects one by one
3. Checks the timer events
4. Performs the other events
5. Destroys the marked objects
6. Displays the screen, updates the sound buffers
7. Handles Windows messages, and regulates loop speed
8. Checks for end of loop, and jumps to 1 if frame not finished.

**1. Checks the player inputs.**
The first task of the main loop is to explore the position of the joystick, and to store it for further use (for example by the movements of the player driven objects). At this time of the loop, the **Player presses ...** condition may become true if it is the first condition in a line of events.

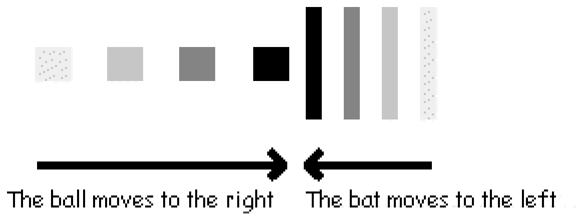
**2. Animates the objects one by one.**
A simple loop explores all the objects of the frame one after the other. Actually, the order of that exploration is at the same time both simple and complicated!

- All the objects are created one after the other, following the order in which they were dropped on the frame in the frame editor. So if you drop object A, object B and object C and immediately run the frame, that order will hold.
- If you delete object B, and insert an object D immediately after, object D will fill the hole left by object B in the database, therefore when running the frame, the order of creation for the frame will be A D C.
- If you work a little longer on the frame, and do some UNDO / REDO steps ... then there is no way of knowing the order of creation! You can only know that all the objects that were dropped on the frame are created prior to starting the frame!
- When starting the frame, you can be sure of one thing though: the list of objects in the runtime has no hole in it, so any object created while running the frame will be positioned immediately after the last object present when the frame was started. It will be the last to be animated.
- Of course, a destroy action will create a hole in the middle of the list, and the next object to be created will fill this hole at that position.

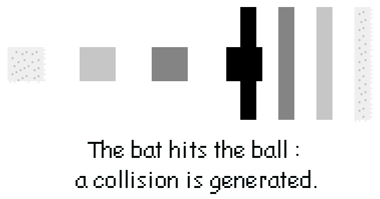
Who cares, you might well ask? Well, this could explain some of the strange behaviours you might notice from time to time with the click range: sometimes an object bounces just fine and sometimes it just won't!

**Explanation of bizarre bounces with small objects...**

This kind of problems only occurs with small objects moving at fast pace.

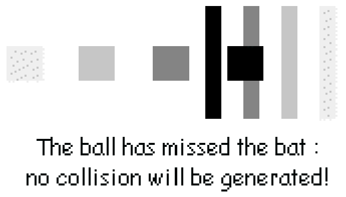


The picture above represents two small objects moving toward each other at a fast pace. The scales of grey represent the last positions of the objects in movement. Let's imagine that the bat comes first in the animation loop. It will be the first to move, to the left, at its speed (slower than the ball).



As you can see above, after being moved to the left, the bat hits the ball. Fusion detects a collision, and the game works fine: the ball bounces on the bat.

Now, let's image that the ball comes first in the animation loop. We will find ourselves with the following problematic situation :

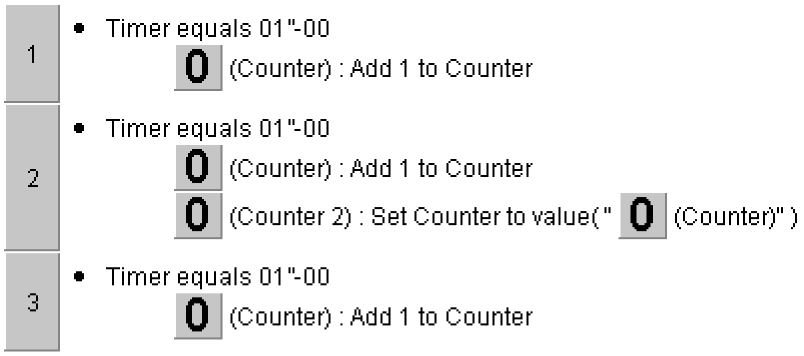

The ball has missed the bat :
no collision will be generated!

As the ball moves faster than the bat, it jumps over the bat and misses the collision detection! The ball doesn't bounce on the bat and the game doesn't work! We have programmed some specific routines in the Fusion engine to prevent this kind of problem: fast movements are "cut" into smaller chunks and a collision detection performed at each step. But as this process is very time consuming, we had to limit it to objects bigger than approximately 10 pixels for full speed.
**Conclusion** : the order of the animation is important for critical case like this one.
**Conclusion 2** : make bigger objects 8-) !

**The object animation loop can call the following event-driven conditions if they come first in a line of events :**
- Collision between Object and OtherObject

- Object collides with the background
- Object is bouncing
- Object has reached a node in its path
- Object has reached the end of its path
- Object leaves the play area
- Object enter the play area

**3. Checks the timer events.**
After animating the objects, Fusion's main loop explores all the timer-related conditions (if they come first in the line of events):
- Start of frame. The start of frame conditions are obviously run once during the life of a frame.
- Timer smaller than.
- Timer equals
- Timer greater than
If multiple events in the event list are true in the same loop, they will be executed on after the other, in the order of the event list. The following example will store the value 2 in the **counter 2** object - not 1, or 3!



```
1  • Timer equals 01"-00
         (Counter) : Add 1 to Counter
2  • Timer equals 01"-00
         (Counter) : Add 1 to Counter
         (Counter 2) : Set Counter to value(" (Counter)" )
3  • Timer equals 01"-00
         (Counter) : Add 1 to Counter
```

- Every is **not** called at this stage, but in the next stage, with the other events : it is not a true event-driven condition.

**4. Performs the other events.**
At this stage of the loop, Fusion runtime explores the list of events from the beginning to the end, and evaluates all the lines of events that begin with a non event-driven condition.

**5. Destroys the objects.**
When a Destroy action is called, the object is **not** destroyed immediately: a flag is set in the internal object datazone, and the object continues its life until stage 5 of the loop, where all the marked objects are physically destroyed. At this stage, the condition "Last object has been destroyed" is called if it is the first condition in a line of events. If this condition is inserted in the middle of the events, it will only be true at the **next** loop!

**6. Updates the display**
Now that all the internal handling of the objects is done, Fusion can concentrate on displaying the result of this loop on the screen, and send the sound samples to the speaker. Please note that in **Machine independent speed**, this stage of the loop may be skipped for synchronisation reasons (read the previous tutorial, Getting the most of the application properties).

**7. Handle Windows messages**

Fusion now returns the control to the Windows operating system, allowing multi-tasking to perform smoothly. This is all handled so as to synchronise the main loop at 50 Hz, not more.

During this period, the Windows system checks the keyboard and mouse input, and may call Fusion in return if the user presses a key or clicks with a mouse button. The following conditions may then be true (if they come first in a line of events)

- User presses a key
- User clicks

**8. Checks for end of loop**

Actions that stop the current frame (End of application, Next frame, etc.) do not immediately end the frame: they just set a flag in the Fusion runtime datazone to indicate the action. Fusion performs the main loop until the end, where it checks the flag, and decides whether or not to go back to step 1. As a consequence, events positioned after a **End of frame** action will be executed before the frame quits!

Conditions like **End of frame** or **End of application** are called at this time in the loop.

I hope this tutorial has shone some light on hidden aspects of the runtime engine. Knowing the way it works internally will allow you to program better and faster applications, and reduce the number of lines in the event editor.