

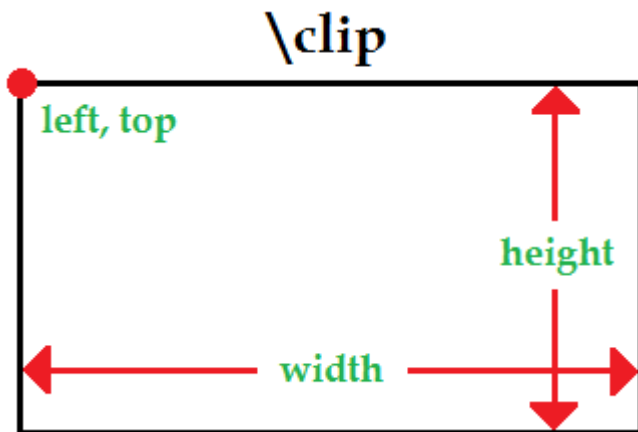
Librería: tag [KE] Parte 1

Esta librería contiene una serie de funciones enfocadas en los Tags que usamos a la hora de hacer Efectos.

tag.clip(left, top, width, height, mode): crea uno o más clip's rectangulares dependiendo del **loop** asignado, con posición y medidas específicas. Los cinco parámetros a ingresar en la función son opcionales, ya que cada uno de ellos tiene valor por default en caso de ser necesario.

left y top son las coordenadas 'x' y 'y' respectivamente del punto de origen del clip, que es el punto superior izquierdo del rectángulo que lo generará.

width y height son el ancho y el alto del rectángulo que generará al clip, respectivamente:



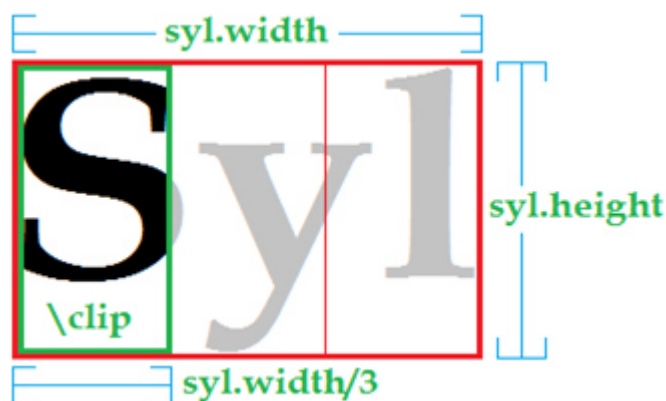
Veamos algunos ejemplos para empezar a aclarar los conceptos hasta acá vistos:

```
Add Tags: Add Tags Language: Lua
tag.clip( syl.left, syl.top, syl.width/3, syl.height )
```

- **left** = syl.left
- **top** = syl.top
- **width** = syl.width/3
- **height** = syl.height



Solo queda visible un tercio del ancho de la sílaba, ya que para el ejemplo se usó **syl.width/3** como el ancho del clip:



Si nuestro efecto es un **Template Type: Syl**, y queremos que toda la sílaba sea totalmente visible dentro del clip, debemos usar los siguientes valores:

Add Tags:	Add Tags Language:	Lua
<pre>tag.clip(syl.left, syl.top, syl.width, syl.height)</pre>		

Los anteriores valores también son los valores por default en el **Template Type: Syl**, o sea que lo podemos usar con el mismo resultado de la siguiente forma:

Add Tags:	Add Tags Language:	Lua
<pre>tag.clip()</pre>		

Es decir, que los valores por default que usará la función dependen del Template Type.

Para Template Type: **Line** y **Translation Line**, los valores por default de **tag.clip()** son: line.left, line.top, line.width y line.height

Para el Template Type: **Translation Word**, los valores por default de **tag.clip()** son: word.left, word.top, word.width y word.height

Para el Template Type: **Syl**, los valores por default de **tag.clip()** son: syl.left, syl.top, syl.width y syl.height

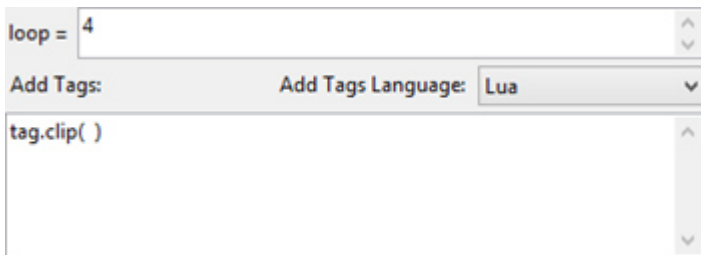
Para el Template Type: **furi**, los valores por default de **tag.clip()** son: furi.left, furi.top, furi.width y furi.height

Y para Template Type: **Char** y **Translation Char**, los valores por default de **tag.clip()** son: char.left, char.top, char.width y char.height

Hasta este punto pareciera que la función **tag.clip** no tiene nada de especial, o al menos que no tiene algo distinto a lo que el tag “**\clip**” podría hacer por sí solo, pero es aquí en donde la celda de texto “**loop**” entra en escena y nos muestra las ventajas de la función, dependiendo de los resultados que deseemos en nuestros efectos.

A continuación veremos las distintas combinaciones que podemos hacer con la función **tag.clip** a partir de los valores que usamos en la celda de texto “**loop**”. Ejemplo:

- **loop**: 4
- Template Type: **Translation Word**



loop = 4

Add Tags: Add Tags Language: Lua

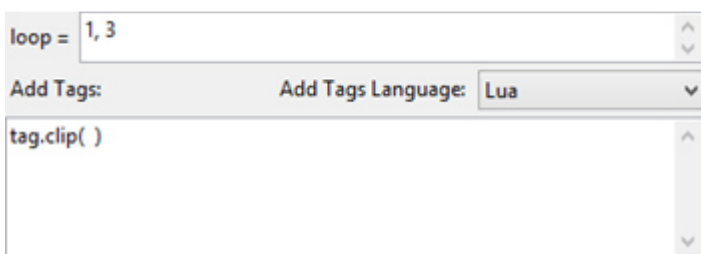
tag.clip()

La función **tag.clip** creará 4 clip’s horizontales dentro de rectángulo, según los parámetros que hayamos ingresado a la función, en este ejemplo está por default y como es un Template Type: **Translation Word**, entonces tiene las dimensiones exactas de cada una de las palabras en cada una de las líneas:



Con solo colocar cualquier valor en la celda de texto “**loop**” obtendremos tantos **clip’s horizontales** como los necesitemos. Para el siguiente ejemplo veremos cómo obtener clip’s verticales con la función **tag.clip**

- **loop**: 1, 3
- Template Type: **Char**



loop = 1, 3

Add Tags: Add Tags Language: Lua

tag.clip()



Lo que hace el loop usado de ese modo es que la función **tag.clip** cree **clip’s verticales** dentro del rectángulo hecho por las medidas ingresadas y como es un **Template Type**: Char y usamos los valores por default de la función, hará que cada carácter quede dentro de los tres clip’s creados.

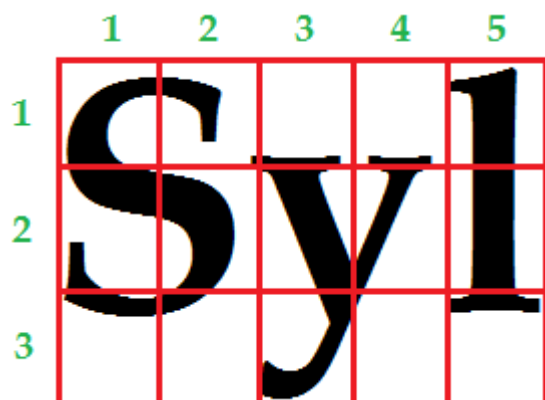
En el próximo ejemplo veremos la forma de hacer **clip's reticulares** (como en forma de grilla o cuadrícula):

- **loop:** 3, 5
- Template Type: Syl

loop = 3, 5

Add Tags: Add Tags Language: Lua

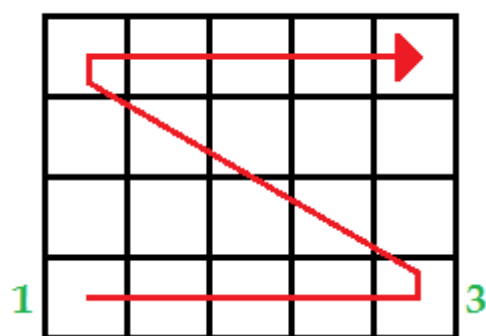
tag.clip()



La retícula creada por la función **tag.clip** será de **3 clip's** horizontales por **5** verticales y la cantidad total del **loop** será: $3 \times 5 = 15$

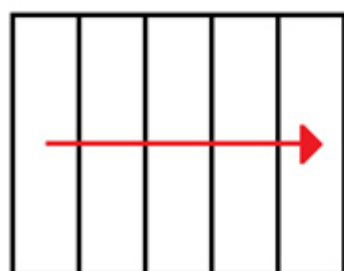
El parámetro **mode** en la función **tag.clip** es un número que asignamos con el fin de determinar el orden de los clip's. Son **8 modes** y los veremos a cada uno de ellos:

Modo: 13



- **mode 13:** la imagen anterior muestra el orden de los clip's en el **mode 13** de un **tag.clip** reticulado. En el caso de los clip's verticales, el **mode 13** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:

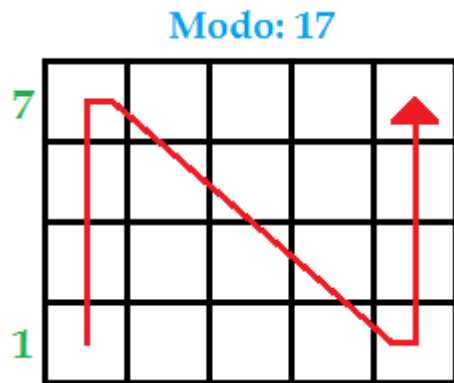
Modo: 13



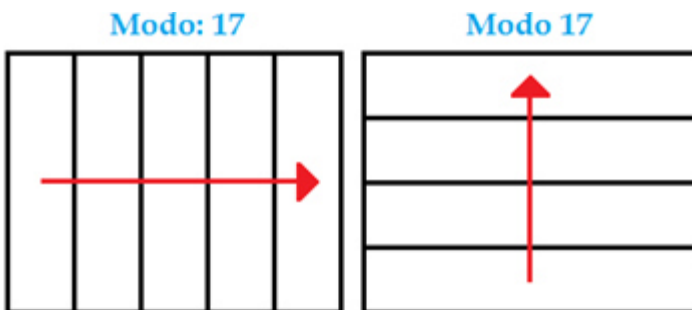
Modo: 13



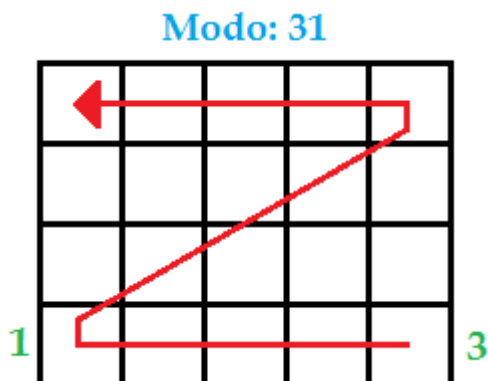
- **modo 17:** para un **tag.clip** reticulado, el orden de los clip's es como el de la siguiente imagen, es decir, el primre clip es el de la esquina inferior izquierda y el último es el de la esquina supeior derecha, siguiendo la trayectoria de la gráfica:



En el caso de los clip's verticales, el **modo 17** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:

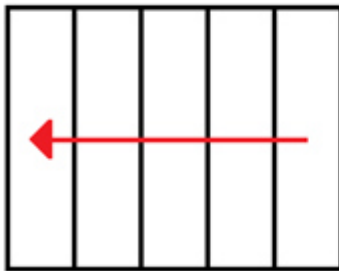


- **modo 31:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina inferior derecha y el último es el de la esquina supeior izquierda, siguiendo la trayectoria de la gráfica:



En el caso de los clip's verticales, el **modo 31** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:

Modo: 31

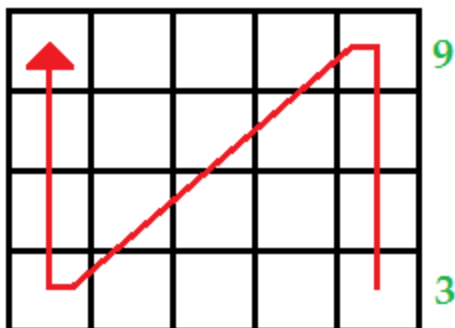


Modo: 31



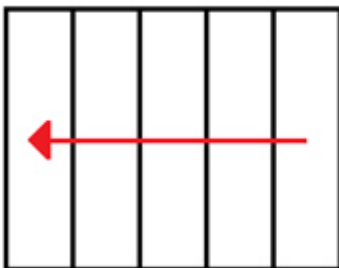
- **modo 39:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina inferior derecha y el último es el de la esquina superior izquierda, siguiendo la trayectoria de la gráfica:

Modo: 39



En el caso de los clip's verticales, el **modo 39** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:

Modo: 39

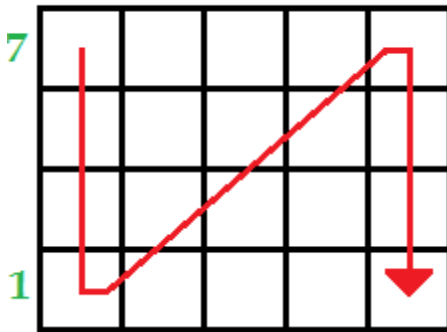


Modo: 39



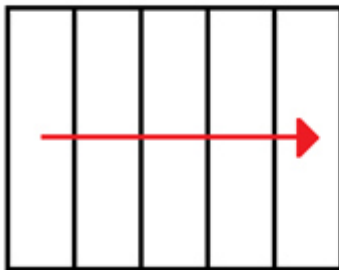
- **mode 71:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina superior izquierda y el último es el de la esquina inferior derecha, siguiendo la trayectoria de la gráfica:

Modo: 71



En el caso de los clip's verticales, el **mode 71** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:

Modo: 71

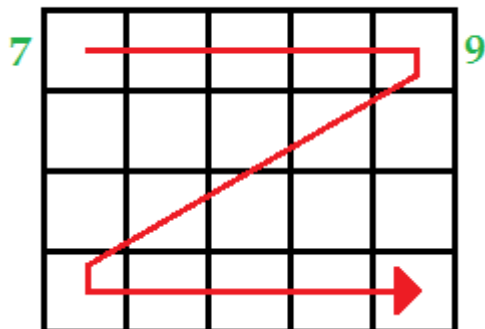


Modo: 71



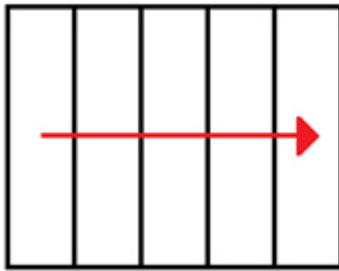
- **mode 79:** es el modo por default de la función. para un **tag.clip** reticulado, el orden de los clip's sería el siguiente; el primer clip es el de la esquina superior izquierda y el último es el de la esquina inferior derecha, siguiendo la trayectoria de la gráfica:

Modo: 79



En el caso de los clip's verticales, el **mode 79** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:

Modo: 79



Modo: 79



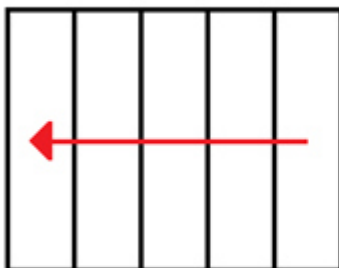
- **mode 93:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina superior derecha y el último es el de la esquina inferior izquierda, siguiendo la trayectoria de la gráfica:

Modo: 93



En el caso de los clip's verticales, el **mode 93** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:

Modo: 93

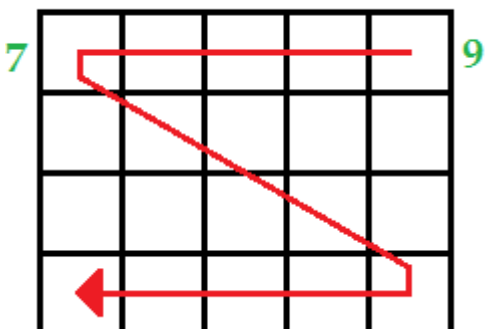


Modo: 93



- **mode 97:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina superior derecha y el último es el de la esquina inferior izquierda, siguiendo la trayectoria de la gráfica:

Modo: 97



En el caso de los clip's verticales, el **mode 97** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:



Y el **mode 97** sería el último de ellos. Así que hay para elegir según sea nuestra necesidad en un Efecto.

Recordemos que el número **mode** lo escribiremos dentro de la función **tag.clip** como un valor numérico, ejemplos:

```
tag.clip( fx.pos_l, fx.pos_t, syl.width, syl.height, 31 )
```

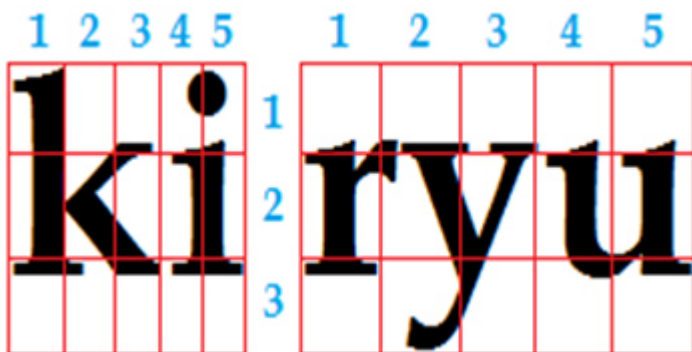
```
tag.clip( fx.pos_l, fx.pos_t, char.width, char.height, 97 )
```

```
tag.clip( fx.pos_l - 50, fx.pos_t, l.width + 100, l.height, 17 )
```

A continuación veremos la forma de hacer un **clip multiple cuadrado**, es decir que las dimensiones del clip sean las mismas. Ya sabemos cómo hacer un clip reticulado o de cuadrícula con la función **tag.clip** y es de la siguiente forma, ejemplo:



Pero este método no garantiza que los clip's tengan las mismas dimensiones, es decir que sean cuadrados:



En la imagen anterior vemos cómo para algunas sílabas las proporciones de los clip's no son para nada cuadradas y eso es porque la cantidad de clip's verticales es constante, lo que no es ideal dados los distintos anchos de la Líneas, Palabras, Sílabas, Caracteres y demás.

Entonces, para que los clip's sean cuadrados, les mostraré dos formas distintas de hacerlo. Y la primera forma de hacerlo es usando la variable **loop_h**, así:

loop = 3, loop_h

loop_h: (variable) es un número entero calculado por el **Kara Effector**, teniendo en cuenta la cantidad de clip's verticales (del ejemplo anterior: 3), que hace que el ancho y el alto de los clip's tengan las mismas dimensiones y sean cuadrados, ejemplo:

[loop_h = 10]

1
2
3

Word

[loop_h = 5]

1
2
3

ka

[loop_h = 8]

1
2
3

Ana

Entonces el **loop_h** varía en todos los casos con el fin de que los clip's den la función **tag.clip** sean cuadrados. La segunda forma de hacerlo es:

- Declaramos una variable con la medida en pixeles de las dimensiones de los clip's, ejemplo:

Variables: pix = 10

- Dependiendo del **Template Type**, por ejemplo un **Template Type: Char**, debemos escribir en la celda de texto **loop**, así:

loop = char.height/pix, char.width/pix

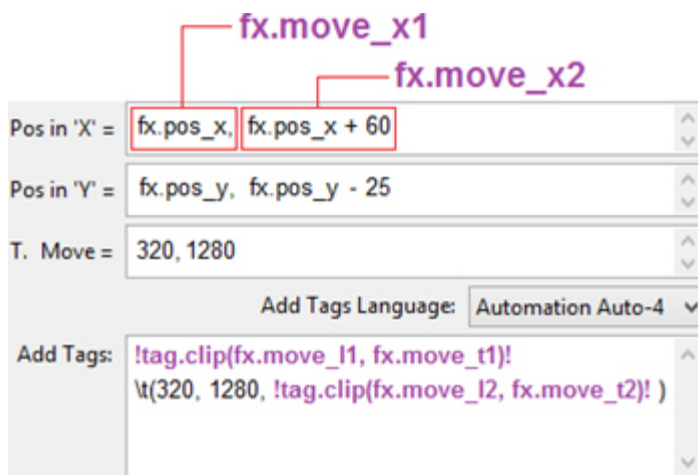
Si por ejemplo, un caracter mide 70 pixeles de ancho por 40 de alto, tendríamos:

- $40/\text{pix} = 40/10 = 4$
- $70/\text{pix} = 70/10 = 7$
- $4 \times 7 = 28$

O sea que obtendríamos 28 clip's de 10 x 10 pixeles de la función **tag.clip**, que a su vez sería la cantidad del **loop**, es decir que **maxj** = 28

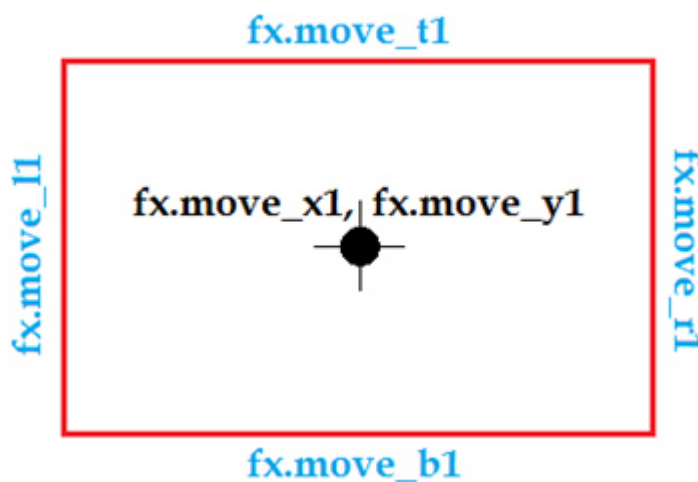
A menor tamaño de los clip's, mayor será la cantidad del loop, y a mayor cantidad de clip's, mayor será la cantidad de recursos consumidos por la memoria RAM y la PC se hará mucho más lenta.

A continuación veremos cómo mover los clip's generados por la función **tag.clip** en el **Kara Effector**, usando algunas cosas que ya hemos aprendido hasta el momento.



The screenshot shows the Kara Effector configuration window. The 'Pos in 'X'' field contains 'fx.pos_x, fx.pos_x + 60', with 'fx.move_x1' pointing to 'fx.pos_x' and 'fx.move_x2' pointing to 'fx.pos_x + 60'. The 'Pos in 'Y'' field contains 'fx.pos_y, fx.pos_y - 25'. The 'T. Move' field contains '320, 1280'. The 'Add Tags Language' dropdown is set to 'Automation Auto-4'. The 'Add Tags' field contains the code: `!tag.clip(fx.move_l1, fx.move_t1)!
\\t(320, 1280, !tag.clip(fx.move_l2, fx.move_t2)!)`

fx.move_l1 es la coordenada de la posición izquierda de **fx.move_x1**, y **fx.move_t1** es la superior de **fx.move_y1**:



Las dimensiones del rectángulo anterior dependerán del **Template Type**, por ejemplo, si es tipo Translation Word, entonces las dimensiones serán las de cada una de cada Palabra de cada Línea.

Desde y hacia dónde se mueve un clip es desición de cada uno, según un efecto así lo requiera, lo más importante es saber cómo hacerlo y el poder contar con una función como **tag.clip** que nos facilita un poco esa labor.

tag.iclip(left, top, width, height, mode): similar a **tag.clip**, pero con la leve diferencia que no hace clip's sino iclip's.

Para los que aún no están familiarizados con los iclip's y en qué consisten, recordemos qué son y para qué se usan.

- **clip:** es un rectángulo con posición y dimensiones específicas que hace visible únicamente a todo lo que esté dentro de dicho rectángulo. Ejemplo:



O sea que todo lo que está dentro del clip es lo que veremos y todo lo que esté por fuera de él quedará totalmente invisible.

- **iclip:** es un rectángulo con posición y dimensiones específicas que hace visible únicamente a todo lo que esté por fuera de dicho rectángulo. Ejemplo:



O sea que todo lo que está por fuera del iclip es lo que veremos y todo lo que esté por dentro de él quedará totalmente invisible.

La elección entre el clip y el iclip dependerá del efecto.

tag.clip2(left, top, width, height): esta función es también similar a la función **tag.clip**, pero con la gran diferencia que siempre genera el mismo clip sin importar el loop, es decir que no genera clip's verticales, horizontales ni reticulares, sino que siempre genera el mismo clip con las mismas dimensiones, es por eso que no necesita el parámetro **mode**.

tag.iclip2(left, top, width, height): es similar a **tag.clip2**, pero con la diferencia que no genera clip's sino iclip's y es la última función de la librería "**tag**" que emplea clip's rectangulares por medio de coordenadas.

A continuación veremos dos funciones más basadas en **clip's**, pero esta vez no serán rectangulares sino basadas en **shapes**, es decir que están enfocadas en las figuras que dibujamos en el **AssDraw3**.

tag.movevc(shape, x, y, Dx, Dy, t_i, t_f): similar a la función **tag.clip** con la diferencia que el clip que genera es una **shape**. Todos los parámetros de esta función, excepto el primero (**shape**), pueden tener valores por default. Veamos qué son y en qué consisten:

- **x:** coordenada con respecto al eje "x" que hace referencia a la posición en donde estará el centro de la **shape**, su valor por default es **fx.move_x1**
- **y:** coordenada con respecto al eje "y" que hace referencia a la posición en donde estará el centro de la **shape**, su valor por default es **fx.move_y1**
- **Dx:** cantidad de desplazamiento en píxeles con respecto al centro de la **shape**, con referencia al eje "x". Su valor por default es **fx.move_x2 - fx.move_x1**

De no existir un **fx.move_x2**, recordemos que su valor por default es **fx.move_x1**, por lo que el valor por default del parámetro **Dx**, en este caso, sería: **fx.move_x1 - fx.move_x1 = 0**

- **Dy:** cantidad de desplazamiento en píxeles con respecto al centro de la **shape**, con referencia al eje "y". Su valor por default es **fx.move_y2 - fx.move_y1**

De no existir un **fx.move_y2**, recordemos que su valor por default es **fx.move_y1**, por lo que el valor por default del parámetro **Dy**, en este caso, sería: **fx.move_y1 - fx.move_y1 = 0**

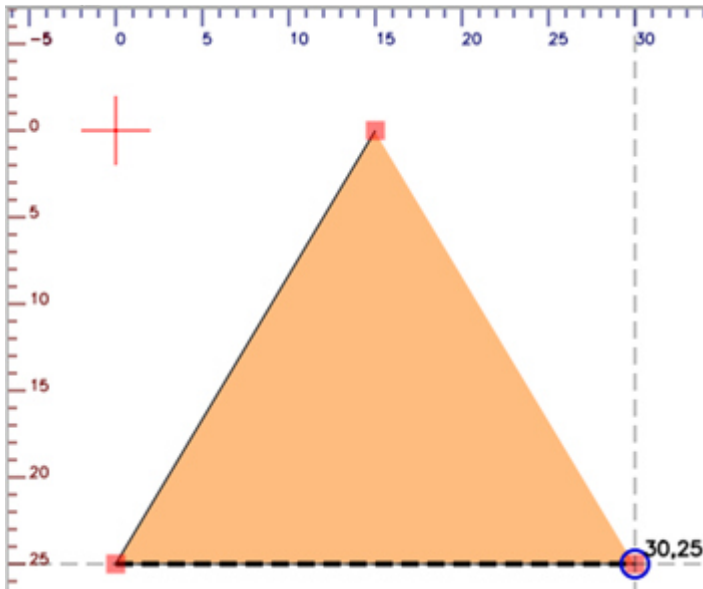
- **t_i:** es el tiempo en que iniciará el movimiento del clip, en caso que decidamos que se mueva. Su valor por default es **fx.movement_i**

De no existir un **fx.movement_i**, recordemos que su valor por default es 0, por lo que en este caso el valor por default de **t_i** sería 0.

- **t_f**: es el tiempo en que finalizará el movimiento del clip, en caso que decidamos que se mueva. Su valor por default es **fx.movet_f**

De no existir un **fx.movet_f**, recordemos que su valor por default es **fx.dur**, por lo que en este caso el valor por default de **t_f** sería **fx.dur**

Para el siguiente ejemplo usaré un simple triángulo hecho en el **AssDraw3**, como lo podemos ver en esta imagen:



El triángulo tiene 30 pixeles de ancho por 25 de alto. El siguiente paso es copiar el código de esa **shape** y pegarlo dentro de la función **tag.movevc** y dejaremos el resto de los parámetros por default. He usado un **Template Type: Syl**, pero pueden usar cualquiera de los de la lista:

Pos in 'X' =

Pos in 'Y' =

T. Move =

Add Tags: Add Tags Language:

La función **tag.movevc** siempre retorna dos tags, un **\clip** que contiene dentro de sí a la **shape** y un **\movevc** que le da la posición al **clip** y lo mueve si ese fuere el caso:

☐ Comment

0 0 0 0

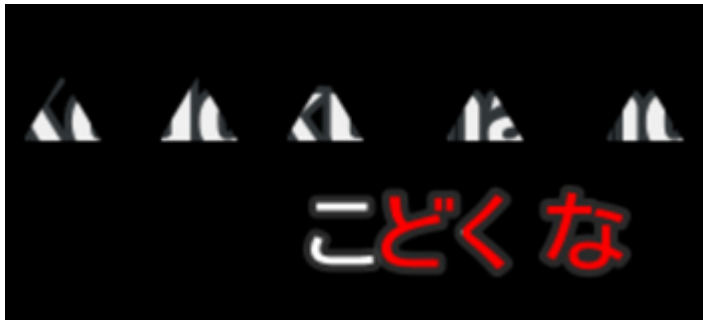
☐ B ☐ I ☐ U ☐ S ☐ fn ☐ AB ☐ AB ☐ AB ☐ AB ☒ Time ☐ Fram

[Kara Effector[fx] 3.2: ABC Template \an5\pos(269.97, 314.1)\clip(m 15 0 | 0 25 | 30 25 \movevc(254.97,301.6, 254.97,301.6,0,5730))Ko

Como el tag de posición es un **\pos** entonces el **\movevc** solo posiciona al clip. Ahora veremos cómo queda el clip del triángulo en cada una de las Sílabas de la Línea:



Como el triángulo es mucho más pequeño que el tamaño de cada una de las sílabas, entonces éstas no alcanzan a ser totalmente visibles:



En cuanto el clip y el objeto karaoke al que afecta (ya sea una sílaba, una línea, carácter, palabra, shape o demás) se muevan al mismo tiempo, desde el mismo punto de inicio y hacia el mismo punto final; lo recomendable es dejar el resto de los parámetros de la función **tag.movevc**, por default. Ejemplo:

Hacemos un **\move** con posiciones iniciales random:

Pos in 'X' =	<input type="text" value="fx.pos_x + R(-30,30), fx.pos_x"/>	^
Pos in 'Y' =	<input type="text" value="fx.pos_y + R(-40,40), fx.pos_y"/>	^
T. Move =	<input type="text" value="0, 360"/>	^

Usaremos la función **tag.movevc** con la misma shape y el resto de los parámetros por default:

Add Tags:	Add Tags Language:	<input type="text" value="Lua"/>
<input data-bbox="209 1406 778 1536" type="text" value="tag.movevc('m 15 0 10 25 30 25 ')"/>		

De este ejemplo tendríamos que:

- $x = fx.pos_x + R(-30,30)$
- $y = fx.pos_y + R(-40,40)$
- $Dx = fx.pos_x - fx.pos_x + R(-30,30) = R(-30,30)$
- $Dy = fx.pos_y - fx.pos_y + R(-40,40) = R(-40,40)$
- $t_i = 0$
- $t_f = 360$

Y el resultado sería:



Que luego de 360 ms las sílabas y los clip's quedaran en su posición natural:



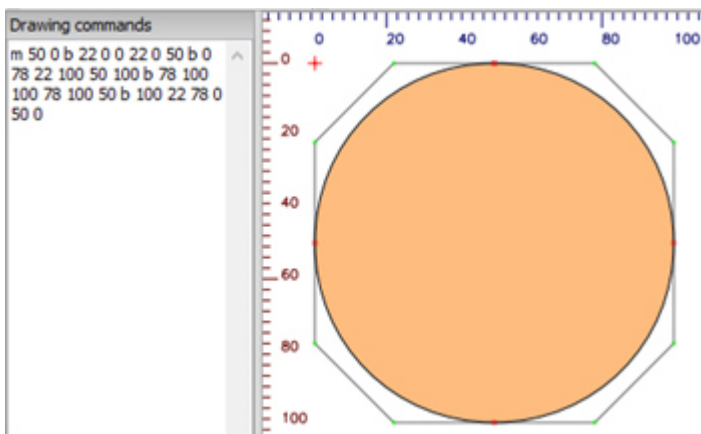
El tag `\movevc` solo es aplicable con el filtro **VSFilterMod**, de otro modo no se verá el efecto. El `\movevc` solo funciona de la mano de los tags `\pos` y `\move`, pero no lo hace con los tags `\moves3`, `\moves4` y `\mover`. En pocas palabras, el tag `\movevc` solo puede mover a un clip en línea recta, sin importar la dirección.

Ahora veremos un ejemplo de objeto estático y clip móvil, es decir, en donde tengamos que usar el resto de los parámetros de la función **tag.movevc**:

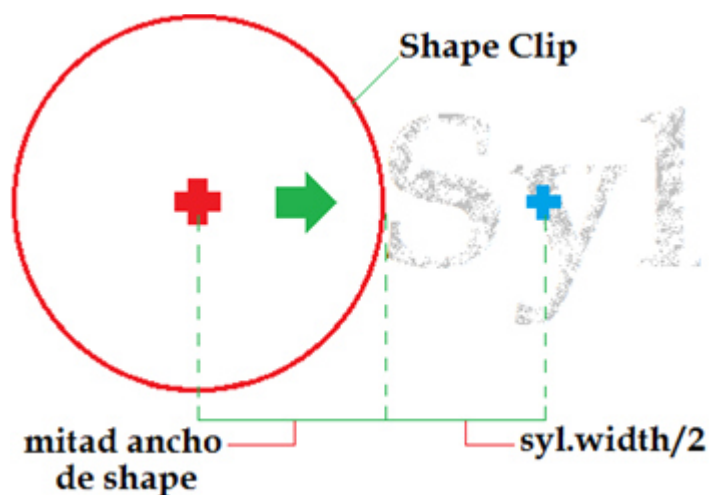
En un **Template Type: Syl**, le dejamos las posiciones por default que equivalen a **syl.center** y **syl.middle**:

Pos in 'X' =	<input type="text" value="fx.pos_x"/>	^ v
Pos in 'Y' =	<input type="text" value="fx.pos_y"/>	^ v

La **shape** que usaremos para el clip es un círculo de 100 px de ancho, dicha **shape** hace parte de las librerías del **Kara Effector** y se llama: **shape.circle**



Y la posición inicial del clip será justo al lado izquierdo de cada sílaba, como muestra la siguiente imagen:



Dicha posición inicial de clip sería:

- $fx.pos_x - syl.width/2 - 50, fx.pos_y$

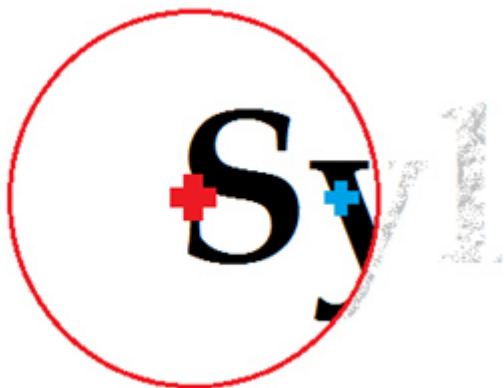
Y la posición final del clip será:

- $fx.pos_x, fx.pos_y$

```
Add Tags:      Add Tags Language:  Lua
tag.movevc(shape.circle, fx.pos_x - syl.width/2 - 50, fx.pos_y,
syl.width/2 + 50, 0)
```

- **shape = shape.circle**
- **x = $fx.pos_x - syl.width/2 - 50$**
- **y = $fx.pos_y$**
- **Dx = $syl.width/2 + 50$**
- **Dy = 0**

Entonces, el círculo que hace de clip irá avanzando desde donde estaba inicialmente hasta que su centro coincida con el de la sílaba:



Y cuando se cumpla el tiempo total de la línea fx, la shape y la sílaba tendrán el mismo centro:



La función **tag.movevc** nos será de gran ayuda para hacer efectos con clip's de alto nivel, ya que de otra forma sería muy complicado lograr el control en cuanto a posición y movimiento cuando de clip's se trata.

tag.movevci(shape, x, y, Dx, Dy, t_i, t_f): similar a **tag.movevc**, pero con la diferencia que no retorna un clip, sino un iclip.

tag.only(condition, exit_t, exit_f): esta función retorna alguno de los dos parámetros (**exit_t** o **exit_f**) según el valor de verdad del parámetro **condition**. Si el valor de verdad de **condition** es verdadero (**true**), retorna al parámetro **exit_t**, si es falso (**false**) retorna **exit_f**.

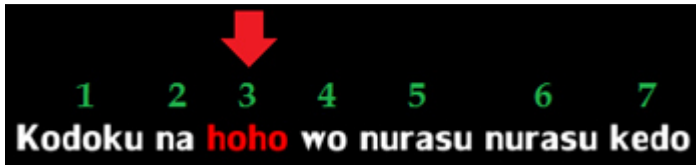
Veamos los tipos de condiciones que podemos usar en la función **tag.only**:

- **==** igual a
- **~=** diferente a
- **<** menor que
- **>** mayor que
- **<=** menor o igual que
- **>=** mayor o igual que

Ejemplo:



Esto quiere decir que, si la Palabra en la línea es la número tres, su color primario será **Rojo** ('\\1c&H0000FF&'), pero si dicha condición es falsa, su color primario será **Blanco** ('\\1c&HFFFFFF&'):



En la función **tag.only** solo el tercer parámetro (**exit_f**) puede tener un valor por default, y este valor depende del tipo de objeto que sea el segundo parámetro (**exit_t**). Si **exit_t** es un **string**, entonces el valor por default de **exit_f** es vacío (''), y si **exit_t** es un **número**, el valor por default de **exit_f** es cero (0). Ejemplos:

- **tag.only(syl.i <= 5, "\\blur3")**

Como **exit_t** es un **string** ("\\blur3"), entonces el valor por default de **exit_f** será vacío. Esto quiere decir que, si la sílaba es una de las cinco primeras de la línea, la función retornará "\\blur3", de lo contrario no retornará nada.

- **l.end_time + tag.only(char.i == char.n, 1200)**

Como **exit_t** es un **número** (1200) entonces el valor por default de **exit_f** será cero (0). Esto quiere decir que, si el caracter es el último de la línea (char.n), se sumarán 1200 ms al tiempo final de la línea, de lo contrario se le sumará cero.

Los seis tipos de condiciones vistos anteriormente tienen un único valor de verdad, ya sea verdadero (**true**) o falso (**false**), pero no puede tener ambos valores al mismo tiempo. Estas seis condiciones son conocidas como **condiciones simples** y al combinar dos o más de ellas se crean las **condiciones compuestas**.

Condiciones simples:

- **==** igual a
- **!=** diferente a
- **<** menor que
- **>** mayor que
- **<=** menor o igual que
- **>=** mayor o igual que

Y para obtener las **condiciones compuestas** es necesario usar **conectores**, la **conjunción** (**and**) o la **disyunción** (**or**). Cada una de las dos mencionadas también tiene un único valor de verdad que dependerá del valor de verdad de cada una de las condiciones que la conforman.

Tabla de verdad de la **conjunción**:

Caso	p	q	p and q
1	V	V	V
2	V	F	F
3	F	V	F
4	F	F	F

Tabla de verdad de la **disyunción**:

Caso	p	q	p or q
1	V	V	V
2	V	F	V
3	F	V	V
4	F	F	F

Ejemplos:

- (syl.i >= 7) **and** (syl.dur < 300)
- (R(2) == 1) **or** (char.width > 80)
- (syl.text ~= "ke") **and** (syl.i < 5)
- (word.n < 10) **or** (line.width > 600)

Las posibles combinaciones son muchas, así que el tener claro qué es lo que queremos en nuestro efectos nos ayudará a decidimos por alguna de ellas.

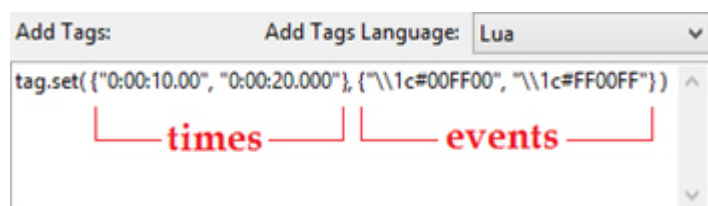
A continuación están las posibles **combinaciones** entre solo dos **condiciones** por medio de alguno de los dos **conectores** (**and** u **or**):

Caso	Cond. 1	Conector	Cond.2
1	==	and // or	==
2	==	and // or	~=
3	==	and // or	<
4	==	and // or	>
5	==	and // or	<=
6	==	and // or	>=
7	~=	and // or	~=
8	~=	and // or	<
9	~=	and // or	>
10	~=	and // or	<=
11	~=	and // or	>=
12	<	and // or	<
13	<	and // or	>
14	<	and // or	<=
15	<	and // or	>=
16	>	and // or	>
17	>	and // or	<=
18	>	and // or	>=
19	<=	and // or	<=
20	<=	and // or	>=
21	>=	and // or	>=

21 posibles combinaciones, pero no se preocupen, no es necesario memorizarlas todas, ya que el solo hecho de saber el valor de verdad de los **conectores** (**and** u **or**) es más que suficiente y el resto es solo cuestión de aplicar un poco de lógica al asunto.

tag.set(times, events): esta función asume que todas las líneas habilitadas para aplicarle un Efecto son una sola y luego aplica una transformación de un 1 ms de duración de cada uno de los elementos de la **tabla “events”**, según los tiempos de la **tabla “times”**.

Las tablas **“times”** y **“events”** pueden ser ingresadas directamente en la función o declararlas a modo de variables en la celda de texto **“Variables”**. Veamos un ejemplo de los dos casos:



No está de más decir que ambas tablas deben tener la misma cantidad de elementos. En este ejemplo las tablas fueron ingresadas directamente.

Los tiempos de la tabla **“times”** son copiados de la parte inferior del vídeo, en el Aegisub y posteriormente pegados entre comillas, no importando si son sencillas o dobles.

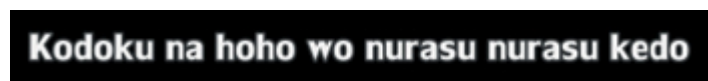
Los tags en la **tabla “events”** también se deben escribir entre comillas, así como se puede apreciar en la anterior imagen. Como la función **tag.set** retorna transformaciones de los tags que están en la **tabla “events”**, entonces éstos deben ser tags que puedan ser **“animados”** por el tag **“\t”**, por ejemplo: **\bord**, **\blur**, **\3c**, **\alpha**, **\1a**, **\jitter**, **\fsvp**, **\clip**, **\iclip**, **\fscx**, **\fsc**, **\fscy**, **\fsp**, **\shad**, entre otros.

Ahora veamos algunos ejemplos de tags que no pueden ser **“animados”** por el tag **“\t”**: **\pos**, **\move**, **\org**, **\moves3**, **\moves4**, **\movevc**, **\mover**, **\fad**, **\t**, entre otros.

Lo que hará la función en este ejemplo será que a los 10 s (**“0:00:10.000”**) contados desde el inicio del vídeo (cero absoluto), convertirá al color primario (**“\1c”**) de su color por default a Verde (**“#00FF00”** está en formato **HTML**, pero no es obligación que esté en este formato, también se podría hacer en formato **.ass**, o sea **“&H00FF00&”**):

#	L	Start	End	Style	Text
1	1	0:00:02.43	0:00:08.16	Romaji	*Ko*do*ku *na *ho*ho *wo *
2	1	0:00:08.33	0:00:13.19	Romaji	*Yo*a*ke *no *ke*ha*ni *ga *
3	1	0:00:13.54	0:00:18.39	Romaji	*Wa*ta*shi *wo *so*ra *e *
4	1	0:00:18.67	0:00:27.22	Romaji	*Ki*bo*u *ga *ka*na*ta *de *
5	1	0:00:28.52	0:00:34.30	Romaji	*Ma*yo*ni *na*ga*ra *mo *ki *

En la imagen, como la línea 1 va desde los 2.43 s hasta los 8.16 s, entonces no se verá afectada por la función y el color primario de ésta será el que ya tiene por default, que en este ejemplo es Blanco:



Pero la línea 2, como va desde los 8.33 s hasta los 13.19 s, sí se verá afectada, a partir de los 10 s. O sea que el color primario de la línea 2 será Blanco desde que inicia (8.33 s) hasta los 10 s:



Y justo cuando el vídeo llegue a los 10 s, el color primario cambiará de forma automática a Verde:



Como la línea 3 inicia en 13.54 s y finaliza en 18.39 s, su color primario será siempre el Verde:



En el caso de la línea 4 pasará algo similar a la línea 2, ya que su tiempo de inicio está en 18.67 s, entonces su color primario iniciará siendo Verde, pero al llegar a los 20 s (“0:00:20.000”) cambiará a Lila (“#FF00FF” en HTML), ya que su tiempo final está en 27.22 s:

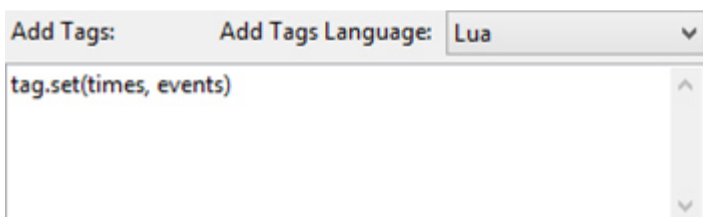


Y desde la línea 4 en adelante, el color primario será siempre Lila, ya que eso es lo que dicta la función.

Si queremos declarar las variables de las dos tablas de la función (“**times**” y “**events**”), haríamos algo como:



Y dentro de la función, en **Add Tags**:



Y sin importar cuál de los dos métodos usemos, los resultados serán los mismos. Es decir, que los tags que coloquemos en la **tabla** “**events**” sucederán de forma inmediata, según los tiempos que hayamos registrado en la **tabla** “**times**”. Esta función es ideal para hacer que nuestros efectos hagan cosas puntuales a medida que en el vídeo al que le hacemos un karaoke, sucedan cambios que nos llamen la atención, como un cambio de color o de escena, cambios de estilos por un personaje y demás.

El **lead-in** de la línea que se ve en la imagen, incluye una **shape** de Plumas para imitar el estilo de las plumas que salen en el vídeo:



La función **tag.set** nos serviría para hacer que las plumas de nuestro efecto desaparezcan exactamente en el mismo momento que lo hacen las del vídeo, haciendo algo como:

```
tag.set( { acá el tiempo exacto }, { "\\alpha&HFF&" } )
```

Entonces en ese preciso momento las plumas quedaran invisibles gracias al tag `\\alpha`. Como se podrán imaginar, las posibilidades son prácticamente infinitas y a gusto de cada uno de nosotros.

Un tag en la **tabla** “**events**” no necesariamente debe ser uno solo, pueden ser varios:

```
events = { "\\fscxy125\\3c&H000000&\\blur4", "\\shad2" }
```

Si queremos que una o más de las transformaciones no suceda de forma inmediata (ya que por default cada una de la transformaciones en esta función tarda solo 1 ms), el **Kara Effector** nos da dos opciones para ello:

- **Opción 1:** duración de la transformación en ms. Ej:

```
times = { "0:00:10.000", { "0:00:20.000", 460 } }
```

Esto hará que la segunda transformación ya no dure 1 ms, sino que ahora tardará 460 ms.

- **Opción 2:** tiempo final de la transformación. Ej:

```
times = { "0:00:10.000", { "0:00:20.000", "0:00:21.810" } }
```

Esto hará que la segunda transformación ya no dure 1 ms, sino que ahora tardará 1810 ms, ya que:

0:00:21.810 – 0:00:20.000 = 1810 ms

De lo anterior es fácil deducir que como la duración por default de cada una de las transformaciones de la función **tag.set** es 1 ms, se vería de la siguiente forma, ejemplo:

```
times = { { "0:00:10.000", 1 },  
{ "0:00:20.000", 1 } }
```

Con este método haremos que una transformación tarde exactamente el tiempo que necesitemos y no siempre 1 ms como lo hace por default. Recuerden que la cantidad de tiempos en la **tabla** “**times**” es ilimitada, y que por cada uno de esos tiempos debe haber un tag o serie de tags que le correspondan, para que las transformaciones sean posibles y la función no nos arroje un error.

tag.glitter(time, add_i, add_f): esta función hace transformaciones al azar en un tiempo determinado “time” que consisten en cambios bruscos de las dimensiones del objeto karaoke. Los parámetros “add_i” y “add_f” son opcionales.

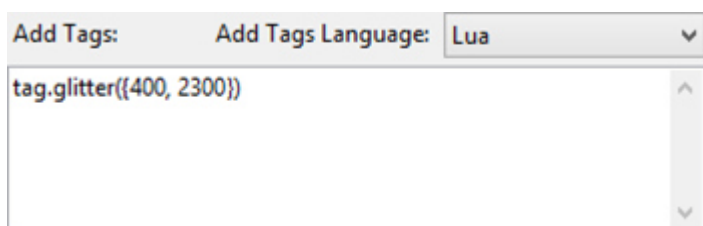
El parámetro “time” puede ser un tiempo en ms, lo que hará que las transformaciones al azar se hagan en el lapso de tiempo entre 0 y dicho valor. Ejemplo:

```
tag.glitter(1200)
```

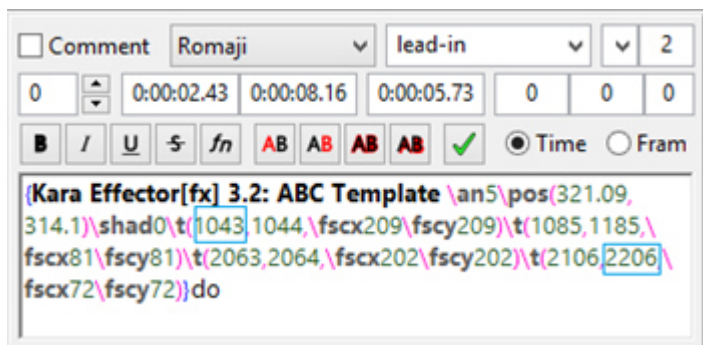
La otra forma que puede tener el parámetro “time” es en forma de **tabla**, en donde el primer elemento de la misma sea el tiempo en ms del inicio y el segundo elemento sea el tiempo final. Ejemplo:

```
tag.glitter({400, 2300})
```

O sea que las transformaciones al azar sucederán entre los 400 ms y los 2300 ms. Pongamos a prueba este mismo ejemplo:



Y verificamos si las transformaciones se realizaron en el rango de 400 a 2300 ms:

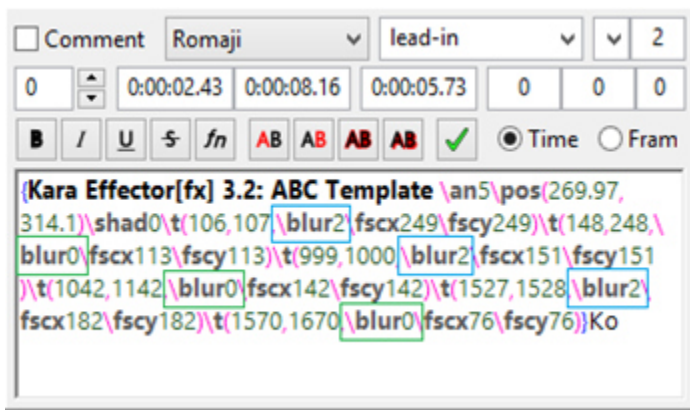


El valor por default de “time” es **fx.dur**, o sea, la duración total de cada una de las líneas fx:

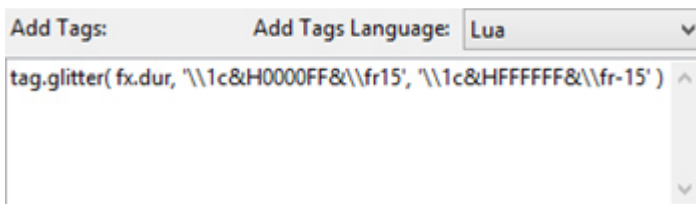
```
tag.glitter( ) = tag.glitter( fx.dur )
```

Las transformaciones que retorna la función **tag.glitter** vienen en cantidades pares. El parámetro “add_i” es un tag o una serie de ellos que a la postre se agregarán a las transformaciones impares retornadas. De forma similar, el parámetro “add_f” se agregará a las transformaciones pares retornadas. Ejemplo:

```
tag.glitter( {0,1700}, “\\blur2”, “\\blur0” )
```

El truco radica en que los tags de “**add_i**” sean opuestos a los de “**add_f**” para que una transformación “contradiga” a la otra. Si por ejemplo en “**add_i**” colocamos un tag \lc, entonces debemos usar en “**add_f**” otro tag \lc, pero con un color distinto, ejemplo:



Del anterior ejemplo, notamos cómo los dos colores son distintos (Rojo y Blanco), y cómo los ángulos son opuestos entre sí (\fr15 y \fr-15).

La función **tag.glitter** es muy útil para asemejar efectos de brillo en **shapes** pequeñas, ya que asemeja un efecto de “titileo” como el de las estrellas en el firmamento. Así que la pueden poner a prueba con la **shape** de una estrella con un tamaño en pixeles entre 5 y 10.

tag.oscill(time, delay, ...): esta función genera transformaciones de duración “**delay**” en un lapso de tiempo “**time**”.

El parámetro “**time**” puede ser un valor en ms, lo que hará que las transformaciones se ejecuten desde cero hasta dicho valor, ejemplo:

- **time** = 1000
- **delay** = 200

Estos dos valores hacen que la función **tag.oscill** retorne cinco transformaciones:

1. de 0 a 200 ← Duración = 200
2. de 200 a 400 ← Duración = 200
3. de 400 a 600 ← Duración = 200
4. de 600 a 800 ← Duración = 200
5. de 800 a 1000 ← Duración = 200

El parámetro “**time**” también puede ser una **tabla**, en donde el primer elemento es el tiempo inicial y el segundo elemento es el tiempo final, ejemplo:

- **time** = {400, 1120}
- **delay** = 180

Estos dos valores hacen que la función retorne cuatro transformaciones:

1. de **400** a 580 ← Duración = 180
2. de 580 a 760 ← Duración = 180
3. de 760 a 940 ← Duración = 180
4. de 940 a **1120** ← Duración = 180

Entonces, el parámetro “**time**” puede ser tanto un **valor** numérico, como una **tabla** con dos valores que limiten el rango de tiempo en el cual se ejecutará la función.

El parámetro “**delay**” tiene las dos mismas cualidades del parámetro “**time**”, de poder ser tanto un **valor** numérico como una **tabla** de valores.

Para un “**delay**” con valor numérico, nos sirve un ejemplo similar a los dos anteriores:

- **time** = 300
- **delay** = 100

Lo que generará tres transformaciones:

1. de 0 a 100 ← Duración = 100
2. de 100 a 200 ← Duración = 100
3. de 200 a 300 ← Duración = 100

cuando “**delay**” es una **tabla**, ampliamos las posibilidades de manipular más a la función **tag.oscill** y a los resultados que nos ofrece. Recordemos que una transformación se basa en el tag \t, y uno de los modos de uso del tag \t es:

\t(t1, t2, **accel**, \...

En donde el parámetro “**accel**” es la aceleración de la transformación, cuyo valor por default es 1. Si queremos modificar la aceleración en las transformaciones de la función **tag.oscill**, lo que debemos hacer es especificarla en el segundo elemento de la tabla “**delay**”, ejemplo:

- **time** = 450
- **delay** = {150, 0.8}

Lo que generará tres transformaciones, pero con la aceleración de 0.8:

1. \t(0, 150, 0.8, \... ←Duración = 150
2. \t(150, 300, 0.8, \... ←Duración = 150
3. \t(300, 450, 0.8, \... ←Duración = 150

Entonces decimos que el primer elemento de la **tabla** “**delay**” será la duración de cada transformación, y el segundo elemento equivale a la aceleración “**accel**” de las transformaciones.

El “**delay**” como **tabla** nos da otra tercera posibilidad, que es el “dilatar” la duración de cada transformación que se genere. El valor de dicha dilatación es el tercer elemento de la **tabla** “**delay**“, y este valor puede ser positivo, lo que hará que cada transformación dure más tiempo que la inmediatamente anterior; o si el valor es negativo, hará que cada transformación dure cada vez menos tiempo que la transformación inmediatamente anterior. Ejemplo:

- **time** = {500, 1110}
- **delay** = {100, 1.2, 10}

o sea que:

- **Duración** = 100
- **Aceleración** = 1.2
- **Dilatación** = 10

Este ejemplo generará cinco transformaciones con las siguientes características:

1. \t(500, 600, 1.2, \... ←Duración = 100
2. \t(600, 710, 1.2, \... ←Duración = 110
3. \t(710, 830, 1.2, \... ←Duración = 120
4. \t(830, 960, 1.2, \... ←Duración = 130
5. \t(960, 1100, 1.2, \... ←Duración = 140

Es notorio cómo cada transformación dura 10 ms más que la transformación inmediatamente anterior, ya que ese fue el valor asignado como “dilatación”.

Hasta este punto, en la función **tag.oscill**, el “**delay**” no solo decide la duración de cada transformación, sino también su frecuencia. Si por ejemplo tenemos un “**delay**” de 200 ms, lo que significaría que la duración de las transformaciones generadas será de 200 ms, y que cada transformación empezará 200 ms después de haber iniciado la transformación inmediatamente anterior.

Para modificar la frecuencia de las transformaciones generadas, el “**delay**” en modo de **tabla** también nos da esa posibilidad. Ejemplo:

- **time** = 500
- **delay** = {{100, 25}, 1}

Notamos que el primer elemento de la **tabla** “**delay**” es otra **tabla**, en donde el primer elemento de esta otra **tabla** (100) marca la **frecuencia**, y el segundo (25), marca la **duración** de las transformaciones:

1. \t(0, 25, 1, \... ←Duración = 25
2. \t(100, 125, 1, \... ←Duración = 25
3. \t(200, 225, 1, \... ←Duración = 25
4. \t(300, 325, 1, \... ←Duración = 25
5. \t(400, 425, 1, \... ←Duración = 25

Y en las cinco transformaciones vemos que cada una de ellas empieza a 100 ms después de haber iniciado la transformación inmediatamente anterior (dado que 100 ms es la frecuencia asignada) y, la duración total de cada transformación es de 25 ms.

En resumen, el parámetro “**delay**” en la función **tag.oscill** tiene los siguientes cuatro modos:

1. **delay** = dur
2. **delay** = { dur, accel }
3. **delay** = { dur, accel, dilatation }
4. **delay** = { { frequency, dur }, accel, dilatation }

Los valores por default de las anteriores variables son de la siguiente forma:

Modo	frequency	accel	dilatation
1	dur	1	0
2	dur		0
3	dur		0
4		1	0

Lo que en resumen sería:

delay = dur = { { dur, dur }, 1, 0 }

El tercer parámetro de la función **tag.oscill** pone (...), ya sabemos que los tres puntos seguidos hacen referencia a una cantidad indeterminada de parámetros, que para este caso son los tags que necesitamos que se **alternen** en las transformaciones.

Como el objetivo de la función **tag.oscill** es alternar tags, se debería usar por lo menos con dos de ellos. De ahí en adelante podemos usar la cantidad de tags que deseemos. Ejemplo:

tag.oscill(1000, 200, “\\blur1”, “\\blur3”, “\\blur5”)

Obtendríamos las siguientes transformaciones:

1. \t(0, 200, \blur1) ←Duración = 200
2. \t(200, 400, \blur3) ←Duración = 200
3. \t(400, 600, \blur5) ←Duración = 200
4. \t(600, 800, \blur1) ←Duración = 200
5. \t(800, 1000, \blur3) ←Duración = 200

Los tres tags ingresados en la función se alternaron en las transformaciones, a manera de ciclo. Este procedimiento será el mismo sin importar la cantidad de tags, lo que hace que esta función tenga muchas utilidades, como alternar cambios de tamaños, de colores, de blur, de ángulos o lo que nos podamos imaginar.

Los tags a alternar se pueden ingresar en la función como lo hecho en el anterior ejemplo, pero también hay otra forma de hacerlo, y es en forma de **tabla**. Ejemplo:

Definimos nuestra **tabla** con los tags a alternar, que para este ejemplo, es una tabla de tres colores primarios:

Variables:

```
colores = {"\\1c&H1D1EF0&", "\\1c&HCB8422&", "\\1c&HD803F3&"}
```

Y en **Add Tags** ponemos:

Add Tags:	Add Tags Language:	Lua
<code>tag.oscill(fx.dur, 300, colores)</code>		

Lo que haría que esos tres colores primarios se alternen cada 300 ms durante la duración total de cada línea de fx.

La última opción que nos da la función **tag.oscill** es poder decidir cuál será el primer elemento de la **tabla** de tags, con el que empezará la primera transformación entre todas las retornadas. Ejemplo:

Primero declaramos la siguiente **tabla**, con un Template Type: **Translation Word**

Variables:
<code>colores = table.make("color", word.n, 30, 90, "\\1c")</code>

Recordemos que la función **table.make** crea una **tabla**, en este caso de colores, de word.n de tamaño, con colores equidistantes entre los ángulos 30° y 90° con un tag \1c.

Ahora usamos la **tabla** creada en la función **tag.oscill** de la siguiente manera:

Add Tags:	Add Tags Language:	Lua
<code>tag.oscill(fx.dur, 240, colores)</code>		

Así la primera transformación de todas las generadas en cada palabra (**word**) empezarán con el mismo color:



Y la opción que ya había mencionado, de decidir con cuál tag empezarán las transformaciones es:

Add Tags:	Add Tags Language:	Lua
<code>tag.oscill({0, fx.dur, word.i}, 240, colores)</code>		

- **time = { 0, fx.dur, word.i }**

El tercer elemento de la **tabla** “**time**” es el que nos ayuda a decidir el primer tag con el que empezará la primera transformación. Para este ejemplo es **word.i**

Kodoku na hoho wo nurasu nurasu kedo

こどく な ほほ を めらす めらす けど

Hecho esto, el tag de la primera transformación para la primera palabra, será el primer color de la **tabla “colores”** ya que **word.i** = 1; para la segunda palabra será el color 2 de la **tabla**, porque **word.i** = 2; y así sucesivamente con las demás palabras.

Nuevas opciones en la función **tag.oscill** :

Esta es una de las funciones más amplia y completa de la **librería tag** y aun así se puede seguir expandiendo. Las actualizaciones que veremos a continuación nos dan más posibilidades en el tercer parámetro de la función **tag.oscill**, que es donde añadimos los tags.

Ejemplo:



Lo que debemos hacer es adjuntar unos paréntesis luego del tag para poder poner dentro de él la función **R** con los parámetros que queramos.

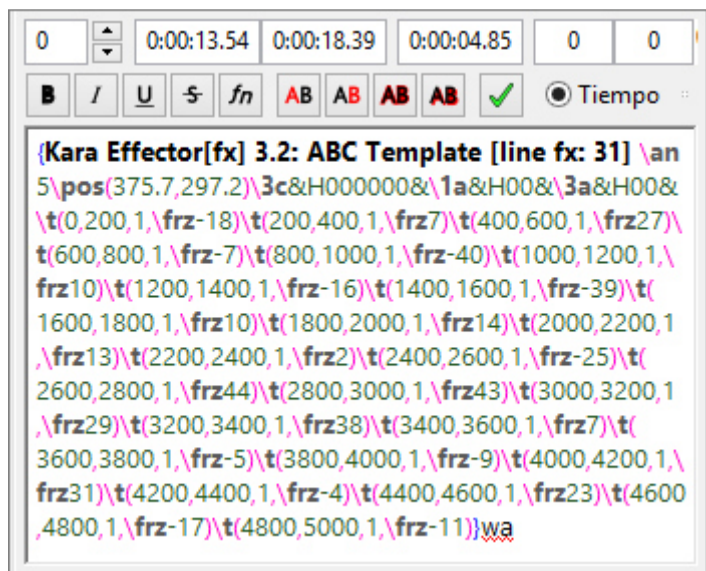
Para este ejemplo en particular la función generará una serie de transformaciones de 200 ms de duración y en cada una de ellas aparecerá el tag `\frz` con un valor aleatorio entre -45° y 45° . Es decir que esta actualización hace que la función **R** se lleve a cabo una y otra vez dentro de cada una de las transformaciones que genera la función **tag.oscill**:

Una vez aplicado el efecto veremos una sucesión de giros respecto al eje “z”, en transformaciones de 200 ms de duración:

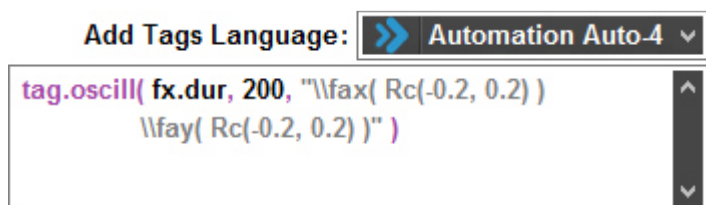
watashi wo sora e mane ku yo

わたし を そら え まね く よ

Así se vería una de las líneas generadas:



Ejemplo:



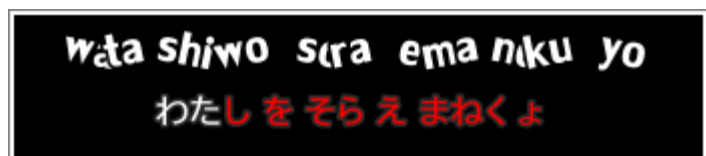
Entonces, entre comillas simples o dobles ponemos el o los tags que deseemos, y adjunto a cada uno de ellos abrimos paréntesis para que dentro de ellos pongamos la función **R** con los valores que más se adecuen al efecto necesitado.

Y la segunda actualización de la función **tag.oscill** consiste en poder utilizar dentro de los tags al contador “i” de las transformaciones generadas al aplicar. Este contador parte desde 0 y va aumentando progresivamente de uno en uno según las transformaciones que genere la función.

Ejemplo:



En este caso el contador “i” hará que los valores se vayan alternando:



tag.ipol(valor_i, valor_f, index_ipol): esta función interpola los parámetros “valor_i” y “valor_f” teniendo como referencia al parámetro interpolador “index_ipol”. Retorna el valor que deseemos entre todos los que existan entre “valor_i” y “valor_f”, inclusive ellos mismos.

Los parámetros “**valor_i**” y “**valor_f**” pueden ser colores, transparencias (alpha) o números reales y ambos deben ser del mismo tipo. Y el parámetro “**index_ipol**” es un número real entre 0 y 1, ya que si es menor que 0 la función lo tomará como 0, si es mayor que 1, la función lo tomará como 1, y su valor por default es 0.5 para que retorne el valor promedio entre “**valor_i**” y “**valor_f**”.

Ejemplos:

- **tag.ipol**(“&HFFFFFF&”, “&H000000&”, 0.7)
- **tag.ipol**(“&HFF&”, “&HAA&”, j/maxj)
- **tag.ipol**(200, 100, char.i/char.n)
- **tag.ipol**(text.color3, “&H00FFFF&”)

Proximamente actualizaciones en este artículo, se está documentando y organizando mejor la información, muchas gracias por seguirnos.