Kara Effector 3.2:

El **Kara Effector** es un archivo **LUA** que tiene la finalidad de hacer más simple la labor de hacer **Efectos Karaoke** y de **Líneas de Traducción**. Contiene Efectos prediseñados que pueden ser aplicados directamente o modificados según uno quiera. Dichos Efectos prediseñados pueden tomarse como un punto de partida para crear nuevos y diferentes Efectos.

El **Kara Effector** nos da la posibilidad de aplicar uno o más Efectos a la vez las veces que queramos ya que posee una amplia librería con variables y funciones que pueden hacer prácticamente cualquier tipo de Efecto.

Se recomienda evitar cualquier tipo de modificación tanto en las Librerías de **Kara Effector** como en su archivo principal, a menos que sea una recomendación nuestra o estén completamente seguros de los que están haciendo, ya que lo más seguro es que deje de funcionar de manera correcta.

El **Kara Effector** ofrece dos tipos de lenguajes en los cuales desarrollar y modificar los Efectos, se pueden hacer en lenguaje **LUA** si ya lo dominan o en **Automation Auto-4** que es el lenguaje de los efectos que normalmente se hacen en el **Aegisub**.

El **Kara Effector** cuanta con múltiples herramientas que harán de la experiencia de hacer Efectos en **Aegisub** una tarea simple, al mismo tiempo que irán aprendiendo a hacer sus propios y originales Efectos y a la vez ingeniosas y elegantes combinaciones.

Todo aquello que deban saber del **Kara Effector** estará consignado en este libro, lo mismo que en un par de canales en **You Tube** que hemos dispuesto con dicho fin, lo mismo que en su **Blog Oficial**:

http://www.KaraEffector.blogspot.com

http://www.facebook.com/KaraEffector

http://www.youtube.com/user/Victor8607

http://www.youtube.com/user/NatsuoDCE

El Autor: **Vict8r Kara**

Descarga Kara Effector 3.2

En la anterior versión del **Kara Effector** (3.1), en link de descarga solo era dado a aquellos que enviaban un correo privado en mi canal de **YT**. En esta ocasión la descarga será totalmente libre, ya sea desde el **Blog Oficial** o en el siguiente link:

Kara Effector 3.2

Instala Kara Effector 3.2

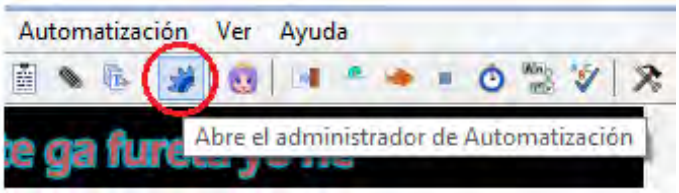
Hay dos formas distintas de hacer uso del **Kara Effector** en el **Aegisub**. El **Kara Effector 3.2** consta de 3 archivos **LUA** y un archivo .ass que es simplemente un karaoke test:

Effector-3.2.lua	330.475
Effector-utils-lib-3.2.lua	194.959
Effector-newfx-3.2.lua	8.923
Effector-3.2-test.ass	5.251

PRIMER MÉTODO DE INSTALACIÓN:

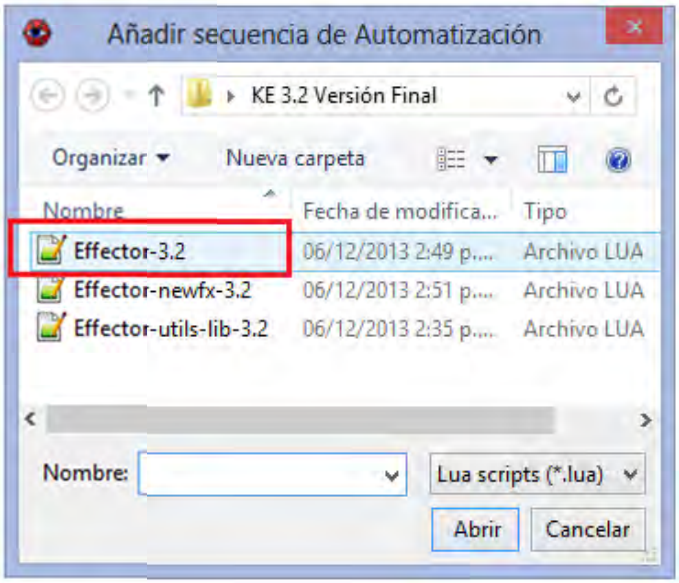
Al descargar el archivo, descomprimimos la carpeta y la ubicamos en la posición que más nos guste dentro de nuestro equipo, por lo general la ubicación recomendada es en el **ESCRITORIO**, pero cualquier lugar está bien, siempre que no olviden el lugar en donde la pegaron.

El primer paso es abrir el **Aegisub** y pulsamos el siguiente botón:

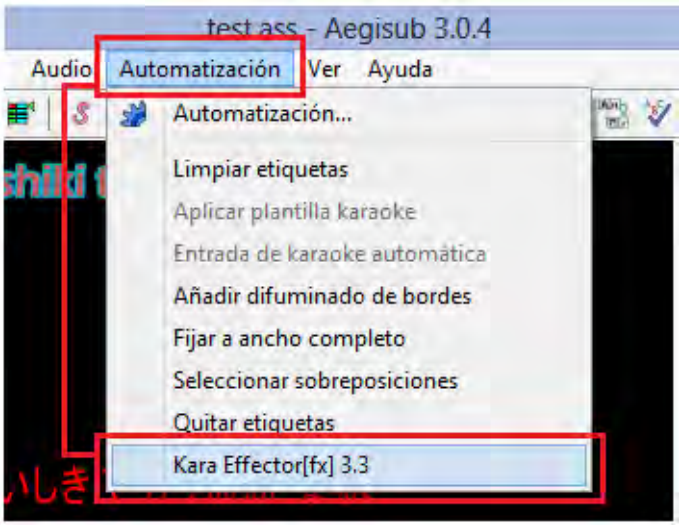


Y luego el botón donde pone: "**Añadir**"

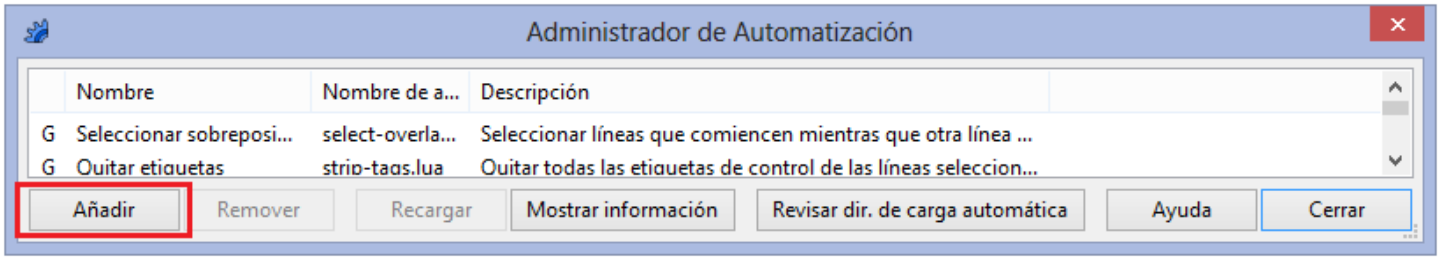
Una vez que pulsamos el botón "Añadir" se abre una ventana en donde podremos buscar la carpeta del **Kara Effector**, y de los tres archivos **LUA** solo cargaremos el archivo que pone: **Effector-3.2.lua**



Hecho esto, ya podemos visualizarlo en el **Aegisub** y empezar a usar el **Kara Effector**:



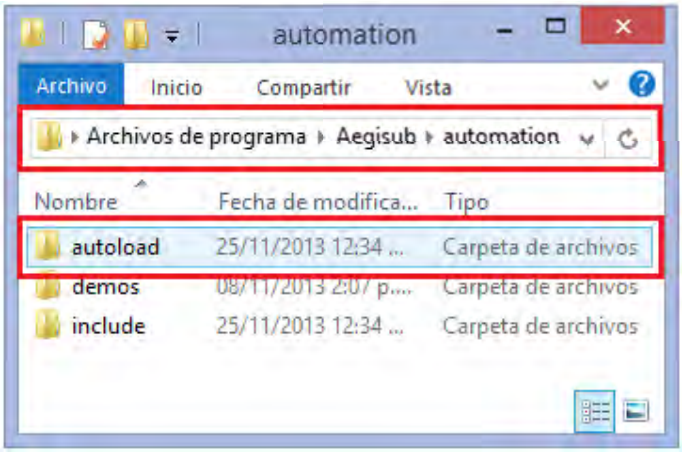
Este es la primera forma de instalar en **Kara Effector 3.2** en el **Aegisub**, a continuación mostraré la segunda forma de hacerlo ya que a algunos les parece molesto repetir este procedimiento cada vez que queramos usarlo.



SEGUNDO MÉTODO DE INSTALACIÓN:

Este segundo método es la forma de instalarlo de forma permanente en el **Aegisub** y consta de dos simple pasos que mostraré a continuación:

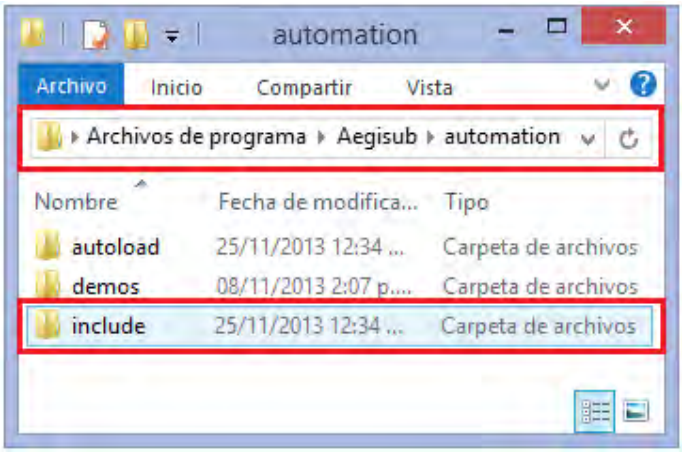
1. Se copia el archivo **Effector-3.2.lua** y se pega en la siguiente carpeta del Aegisub:



2. Se copian los otros dos archivos LUA del Kara Effector:

   Effector-utils-lib-3.2.lua
   Effector-newfx-3.2.lua

   Y se pegan en la siguiente carpeta del Aegisub:



Con estos dos simples pasos ya queda instalado el **Kara Effector** de forma permanente en el **Aegisub**, para poder hacer uso de él las veces que lo necesitemos para hacer nuestros Efectos Karaokes, o aplicar algún Efecto o Estilo especial en nuestras Líneas de Subtítulos.

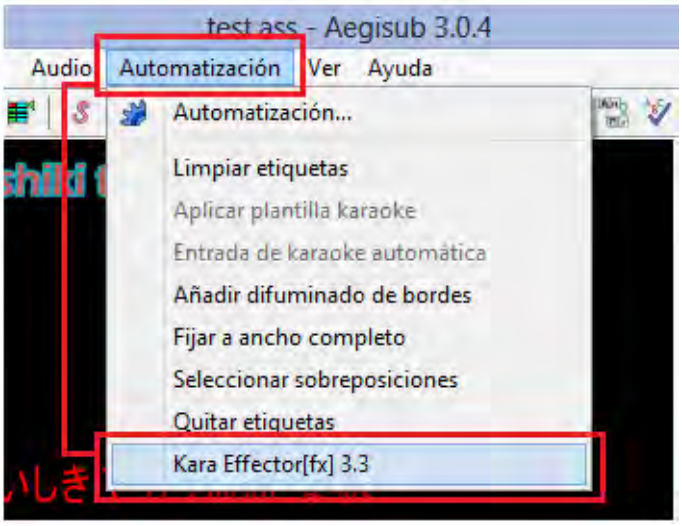
Ya es decisión de cada uno la forma en la que instalará el **Kara Effector** en el **Aegisub**, pero sea cual sea el método que elijan, éste funcionará de igual forma en ambos casos.

A partir de este punto ya podemos empezar a ver al **Kara Effector** como una de las opciones de Efectos del **Aegisub** automatizados y solo es cuestión de elegir los Efectos.

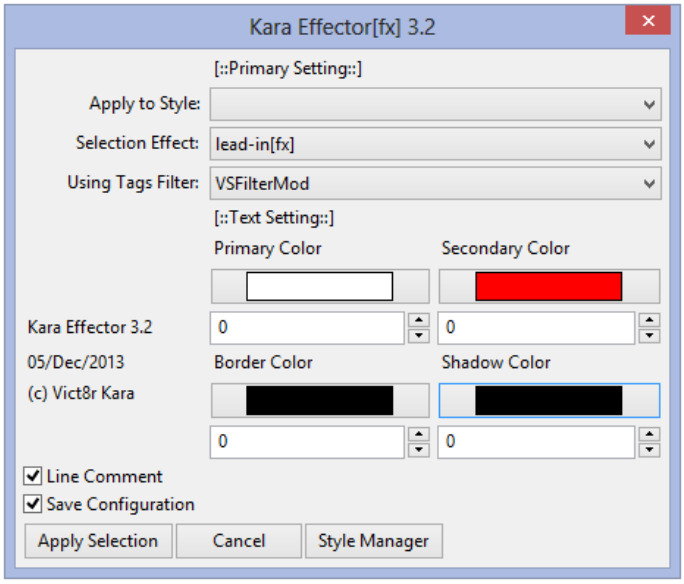
Usa Kara Effector 3.2

Para empezar a usar el **Kara Effector 3.2** debemos abrir el **Aegisub**, y en comparación con el **Kara Effector 3.1** que solo podía ser usado en las versiones 2.1.8 y 2.1.9 del **Aegisub**, la versión 3.2 es compatible con todas las versiones del **Aegisub** existentes hasta la actualidad. (**Aegisub 3.1.3**)

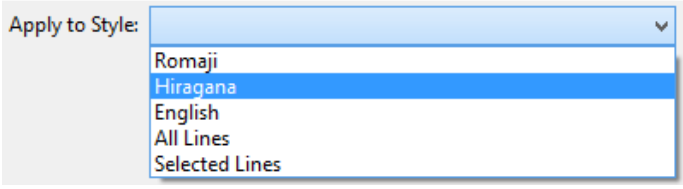
Una vez hayamos abierto el **Aegisub**, pulsamos el botón de **Automatización:**



Y se abrirá la primera ventana **del Kara Effector**:



Apply to Style:



La opción "**Apply to Style**" es una ventana desplegable (**dropdown**), que al pulsarla saldrán los listados de los estilos que contiene el archivo .ass que abrimos en el **Aegisub**, excepto los dos últimos de esa lista. En la imagen anterior, solo los tres primeros de esa lista son los estilos de las líneas del archivo .ass, o sea:

Estilo: **Romaji**

Estilo: **Hiragana**

Estilo: **English**

Se puede ver más claramente en esta captura:

#	Inicio	Final	Estilo	Texto
1	0:00:34.30	0:00:39.31	Romaji	✱Su✱re✱chi✱ga✱u ✱i✱shi✱
2	0:00:40.70	0:00:45.79	Romaji	✱Tsu✱ka✱ma✱e✱ru ✱yo ✱
3	0:00:45.92	0:00:54.56	Romaji	✱Mo✱to✱me ✱a✱u ✱ko✱ko
4	0:00:02.43	0:00:08.16	Hiragana	✱こ✱ど✱く ✱な ✱ほ✱ほ ✱を
5	0:00:08.33	0:00:13.19	Hiragana	✱よ✱あ✱け ✱の ✱け✱は✱い
6	0:00:13.54	0:00:18.39	Hiragana	✱わ✱た✱し ✱を ✱そ✱ら ✱え
7	0:00:02.43	0:00:08.16	English	Mis mejillas están manchadas
8	0:00:08.33	0:00:13.19	English	Pero puedo sentir la llegada d
9	0:00:13.54	0:00:18.39	English	Que me atrae hacia los cielos

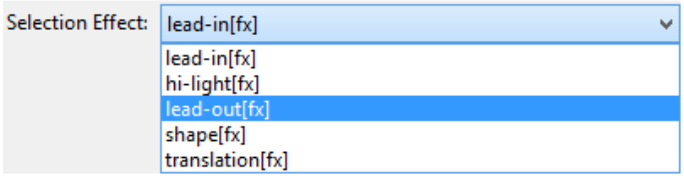
En pocas palabras, en "**Apply to Style**" escogemos el Estilo de las Líneas a las que le aplicaremos los Efectos. Al escoger un estilo, el Efecto se aplicará a todas las Líneas en nuestro archivo que tengan ese Estilo seleccionado.

La opción "**All Lines**", como es evidente al traducirlo al español (**Todas las Líneas**), aplica el efecto a todas las Líneas de nuestro archivo .ass, excepto aquellas Líneas que estén comentadas y que no sean de tipo "**Dialogo**", es decir que el Efecto no se aplicará a aquellas líneas que sean "**template**" como la "**template line**" o las "**code**" como las "**code once**" o "**code syl**".

La opción "**Selected Lines**" como su nombre nos lo hace saber (Líneas Seleccionadas), aplica el Efecto a todas aquellas líneas que previamente hayamos seleccionado. Las líneas se seleccionan "iluminando" las líneas como tradicionalmente lo hacemos al seleccionar un texto o simplemente manteniendo presionado **Ctrl + Clip Derecho** de este modo podemos escoger a nuestra conveniencia aquellas líneas que queremos que se aplique nuestro efecto.

Una vez ya tengamos claro a qué Estilo aplicaremos un Efecto o a cuáles líneas en especial, el siguiente paso es seleccionar el tipo de Efecto que le aplicaremos a dichas líneas. A continuación veremos cómo podemos hacerlo.

Selection Effect:



Como previamente había comentado, en esta pestaña desplegable es dónde podemos seleccionar el tipo de Efecto que aplicaremos a las líneas seleccionadas de nuestro archivo .ass. Son 5 tipos distintos de Efectos y a continuación los explicaré:

lead-in[fx]: son aquellos Efectos considerados como de "**Entrada**", es decir que son los efectos que se aplican antes del Efecto de la Sílaba Activa, en el caso de los Efectos Karaoke, o los efectos de antelación justo antes de que una línea quede estable en el caso de las líneas de traducción del mismo karaoke.

hi-light[fx]: son los Efectos que le aplicamos a la Sílaba Activa, son los Efectos de Karaoke que llevan el ritmo de la canción y podría decirse que son efectos exclusivos de las Líneas Karaokes.

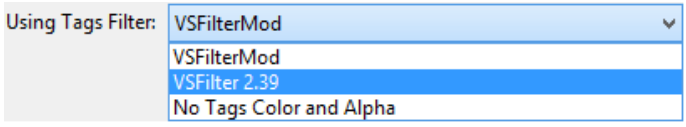
lead-out[fx]: son los Efectos de "**Salida**" son la contraparte de los Efectos tipo lead-in, y son los Efectos que se le hacen a las Sílabas luego de haber sido activadas en las Líneas de Karaoke o los Efectos que se hacen al final de una Línea de Traducción normal.

shape[fx]: son los Efectos que incluyen figuras hechas en el **AssDraw3**, conocidas como "**shapes**" y dependiendo de los tiempos del Efecto, pueden ser de **Entrada**, de **Salida** o de **Sílaba Activa**. También se pueden usar efectos tipo shape[fx] en las **líneas de traducción** y en los **subtítulos normales**.

translation[fx]: Son los Efectos que se pueden aplicar a todas las Líneas no comentadas de tipo Dialogo de nuestro archivo .ass. Generalmente son usados en las Líneas de Traducción, pero también pueden ser usados en las Líneas de Karaoke ya que este tipo de Efectos no exige que las Líneas estén separadas por Sílabas, por Palabras o por Carateres.

En este momento ya tenemos elegidos tanto el **Estilo** como **tipo de Efecto** que usaremos. El siguiente paso es:

Using Tags Filter:



En esta pestaña despegable seleccionamos el formato en el que saldrán en nuestro Efecto los Tasg de Colores y de Transparencias (alpha).

VSFilterMod: al elegir esta opción, hacemos que los tags de colores y transparencias saldrán en el formato del Filtro que lleva este nombre (**VSFilterMod**), ejemplo:

\1vc(&HFFFFFF&,&HFFFFFF&,&HFFFFFF&,&HFFFFFF&)

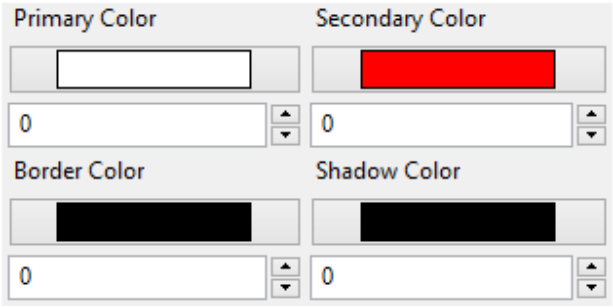
\3va(&H00&,&H00&,&H00&,&H00&)

VSFilter 2.39: esta opción hace que los tags de colores y transparencias en nuestro Efecto queden en el formato que por Default lo expresa el **Aegisub**, ejemplo:

\1c&HFFFFFF&

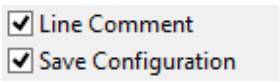
\3a&H00&

No Tags Color and Alpha: lo que hace esta opción es evitar que los colores y transparencias de la sección "**Text Setting**" salgan en nuestro Efecto. Esta opción solo afecta a estos ocho valores que se encuentran en la primera ventana del **Kara Effector**:



Estos ocho valores del "**Text Setting**" son cuatro colores y cuatro transparencias en donde están los colores primario y secundario, el color del borde y el color de la sombra, además de los respetivos valores de transparencia de estos colores. Los valores de las transparencias van desde 0 hasta 255, sabiendo que 0 es totalmente visible y 255 es completamente invisible.

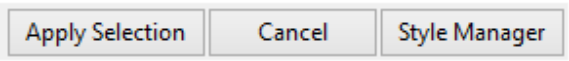
Line Comment – Save Configuration:



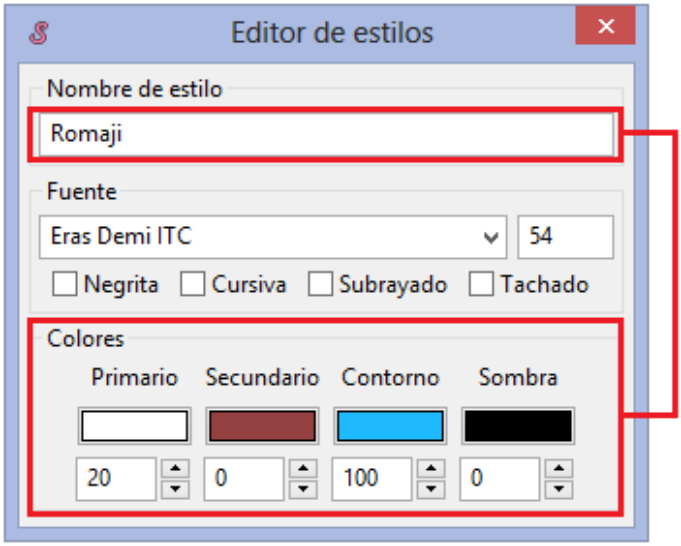
Son opciones tipo "**checkbox**", o sea un botón de marcar y desmarcar. Al marcar la opción de "**line Comment**" lo que hace es Comentar las Líneas a las que le hayamos aplicado el Efecto, es decir, en caso de que queramos aplicar más de un Efecto a un mismo Estilo o Líneas seleccionadas, debemos mantener esta opción sin marcar, para que al aplicar los Efectos, las Líneas no se comenten y puedan seguir activas para aplicarle nuevos Efectos.

Por otro lado, la opción "**Save Configuration**" hace posible que todas aquellas modificaciones que hayamos hecho en cuanto a la selección del Estilo, tipo de Efecto, colores y demás, se conserven tal cual, al momento de pretender aplicar más Efectos.

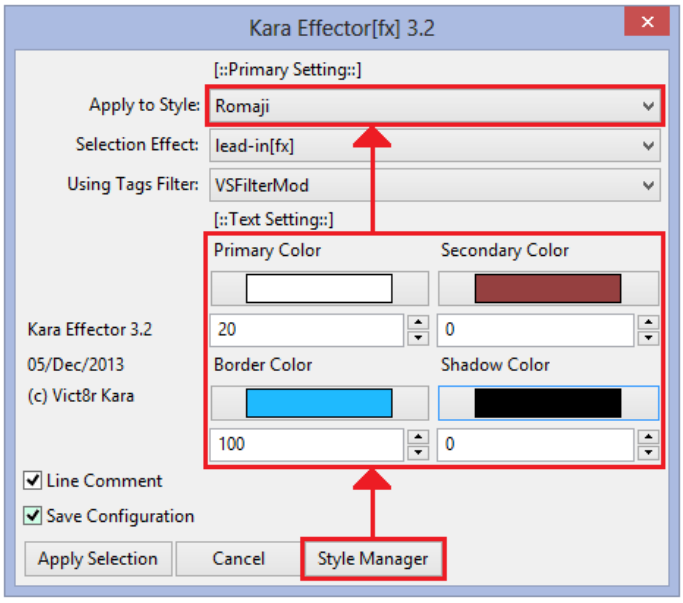
Y ya en la parte inferior de la primera ventana del **Kara Effector**, aparecen los botones de ejecución, los cuales explicaré de derecha a izquierda:



Style Manager: este botón hace que los valores de colores y transparencias del Estilo elegido, se copien en la ventana del **Kara Effector**. En el **Editor de Estilos** del **Aegisub** es en donde editamos generalmente estos valores:



Como se ve en la siguiente imagen, los valores de los colores y las transparencias del Estilo, son copiados:



Una duda razonable sería: ¿por qué la necesidad de copiar los valores de los colores y transparencias que ya había ajustado previamente?

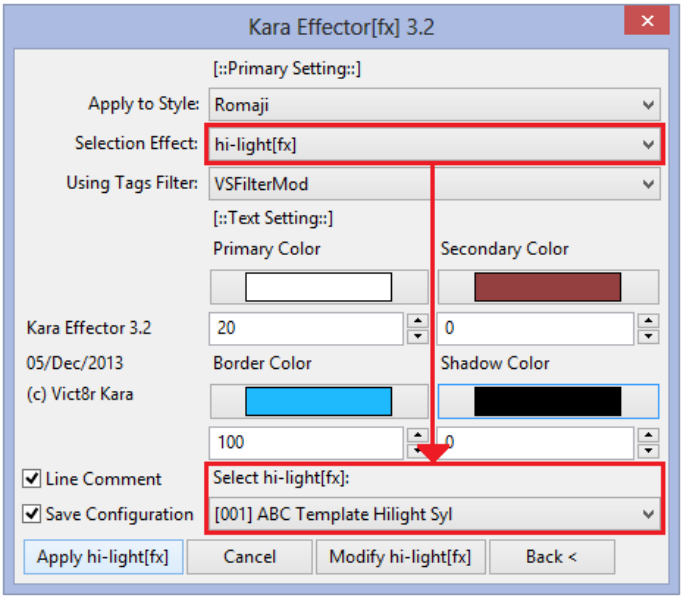
Y bueno, pasa que a veces creemos que los valores que elegimos en el Estilo en cuanto a colores y transparencias serán los definitivos, pero al aplicar un Efecto no quedamos conforme con los resultados y quizá haya la necesidad de modificar algunos de esos valores.

Otra razón sería la posibilidad de darle los valores de los colores y transparencias originales a un determinado Efecto y otros valores ligeramente modificados a un Efectos diferente. En fin, podría decir que a medida que se vayan familiarizando con el **Kara Effector**, se irán dando cuenta que no es simplemente un lujo, sino que puede llegar a ser de gran utilidad a la hora de modificar los Efectos o de crear otros nuevos.

Cancel: no hay mucho que decir de este botón, ya que es más que clara su función. Este botón cierra la ventana de inicio del **Kara Effector**.

Hasta este punto ya tenemos claro el Estilo o las Líneas a las que le aplicaremos un Efecto, el tipo de Efecto que haremos, el formato en el que quedarán los valores de los colores y las transparencias, incluso si queremos mantener dichos valores o los queremos modificar, pero lo que aún no hemos visto en es dónde podemos elegir el Efecto en particular que aplicaremos, y para ello es que está el siguiente botón:

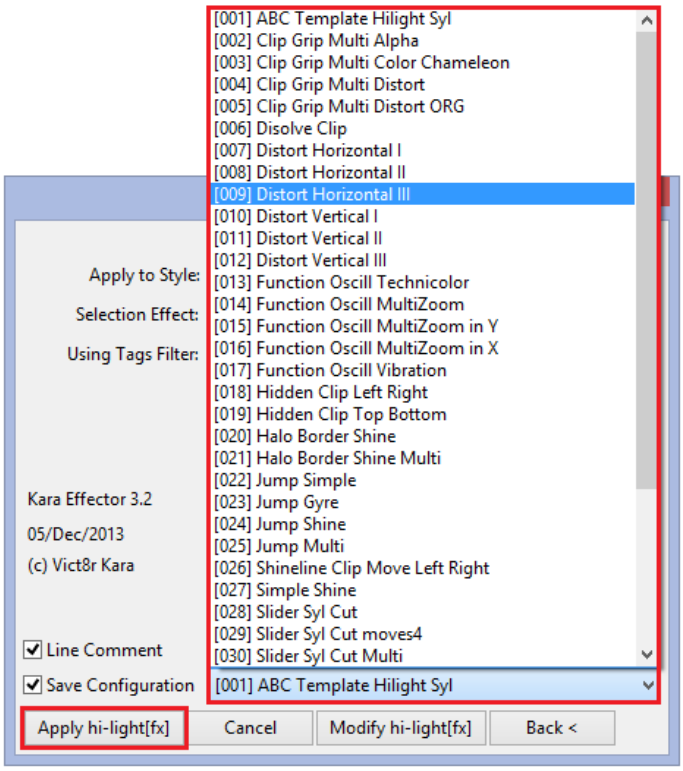
Apply Selection: este botón modifica la ventana de inicio del Kara Effector para poder visualizar el listado de Efectos según el tipo que hayamos elegido, también modifica los botones de ejecución para poder aplicar el Efecto elegido o para que podamos modificarlo:



En este ejemplo, el tipo de Efecto que elegí fue **hi-light** y por ello al pulsar el botón "**Apply Selection**", el nombre de la nueva pestaña desplegable que apareció luego de modificarse la ventana de inicio del **Kara Effector** es:

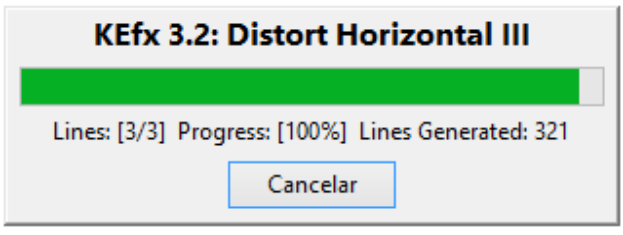
Select hi-light[fx]

Y como su nombre lo indica, es donde ya podemos elegir el Efecto tipo **hi-light** que aplicaremos al Estilo o a las Líneas seleccionadas:



El botón "**Back <**" hace que podamos volver a seleccionar el tipo de Efecto que usaremos, y ya por último, una vez escogido el Efecto que aplicaremos, pulsamos el botón "**Apply hi-light[fx]**" (el "**hi-light**" es porque fue el tipo de Efectos que había elegido, si por el contrario hubiera elegido por ejemplo el tipo de Efecto tipo shape, el botón pondría: "**Apply shape[fx]**").

Dependiendo del peso del Efecto, algunos de ellos se aplicarán más rápidos que otro. Al momento de aplicarse un Efecto veremos la siguiente ventana informativa:



Hay tres datos distintos que nos ofrece esta ventana, el primero nos dice a cuántas líneas ya se les ha aplicado el Efecto de todas en total. En este caso ya completó 3 de 3.

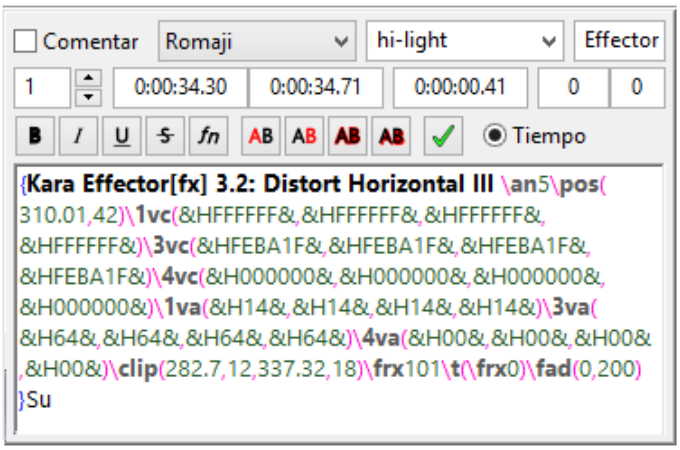
El siguiente dato es el progreso y está medido en el porcentaje total hasta que el Efecto se aplique en su totalidad.

Por último, el tercer dato nos muestra la cantidad de líneas que está generando el Efecto en cuestión.

Una vez que se haya aplicado el Efecto, ya podemos ver las líneas que generó, en donde el Actor pone el tipo de Efecto que seleccionamos (**hi-light** en este caso), y en la columna de Efecto siempre pondrá "**Effector [fx]**". De ese modo es fácil identificar cuáles fueron las nuevas líneas que se generaron:

#	L	Inicio	Final	Estilo	Actor	Efecto	Texto
6	0	0:00:13.54	0:00:18.39	Hiragana			✳わ✳た✳し ✳を ✳
7	0	0:00:02.43	0:00:08.16	English			Mis mejillas están
8	0	0:00:08.33	0:00:13.19	English			Pero puedo sentir
9	0	0:00:13.54	0:00:18.39	English			Que me atrae hac
10	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su
11	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su
12	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su
13	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su
14	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su
15	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su
16	1	0:00:34.30	0:00:34.71	Romaji	hi-light	Effector [Fx]	✳Su

Al mirar en unas de las líneas generadas, podemos ver claramente el nombre del programa, la versión y el nombre del Efecto que hayamos seleccionado:



En este caso en particular los valores de los colores y de transparencias están en formato del **VSFilterMod**, pero como ya había explicado anteriormente, todo depende de nuestra elección.

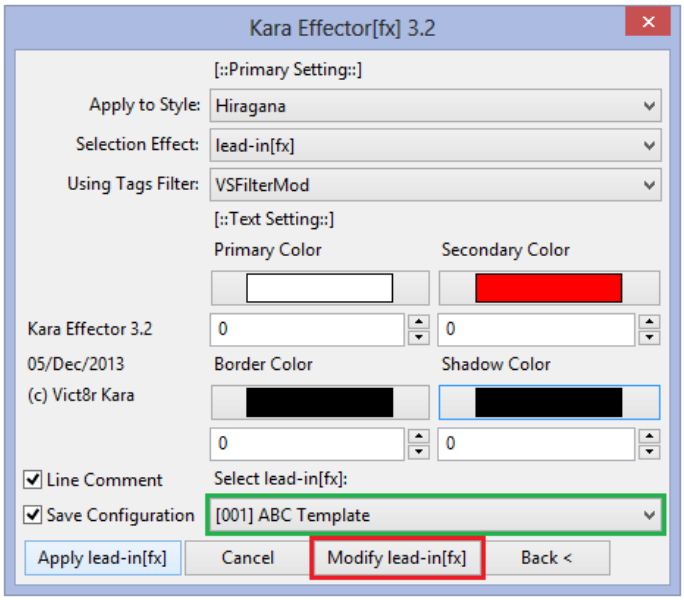
Siguiendo esta serie de instrucciones se pueden aplicar los Efectos que por default trae el **Kara Effector** con solo unas pocas modificaciones que son posible hacer desde la ventana de inicio. Para la próxima entrega veremos la ventana de modificación y creación de Efectos y también cada una de las partes que lo componen. Es todo por ahora y me despido con la ilusión de haber aclarado las dudas que hasta este momento hayan tenido. No olviden visitar el Blog Oficial del **Kara Effector**, lo mismo que los canales de **You Tube** que están en la primera página de esta entrega.

Para aclarar más dudas, nos pueden escribir en el Blog, lo mismo que en **YT**. Hasta la próxima.

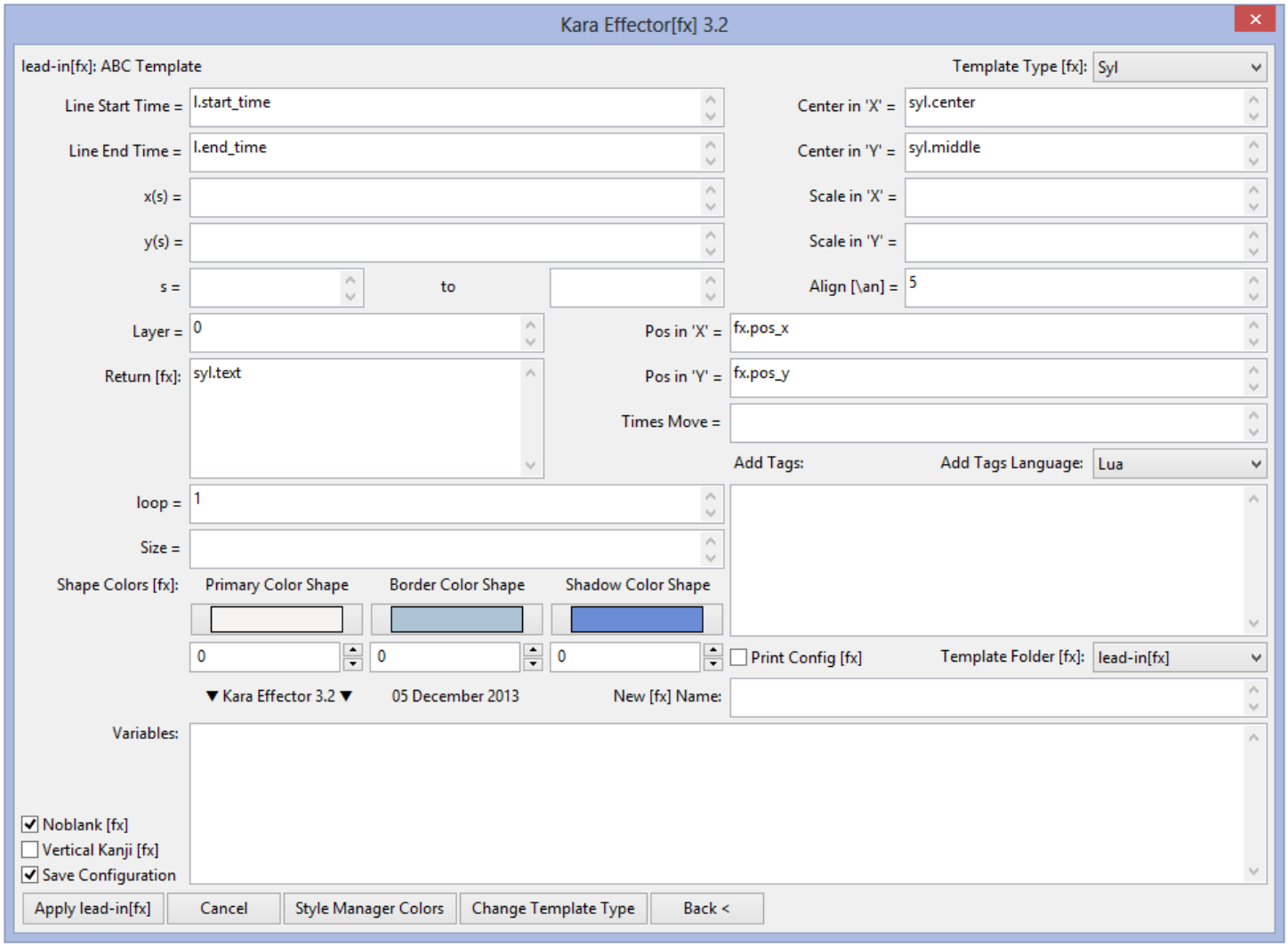
Kara Effector 3.2:

En la anterior entrega, vimos cómo descargar, instalar y aplicar los Efectos que vienen en el **Kara Effector** por Default. Las pocas modificaciones que se pueden hacer en la ventana de inicio no van más allá del cambio de los valores en los colores y la transparencia. En esta entrega veremos cómo abrir la ventana de modificación y creación de nuevos Efectos, que consta de muchas más opciones que la ventana de inicio pero no por ello es más compleja de usar o entender.

Esta segunda ventana se puede visualizar al pulsar el botón de "**Modify**" y el resto del nombre de este botón lo hereda del tipo de Efecto que hayamos seleccionado:



El pulsar el botón "**Modify**" hace que la ventana de inicio del **Kara Effector** de transforme en la segunda ventana:

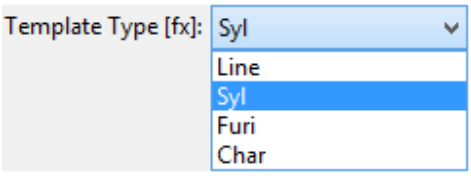


La **Ventana de Modificación** contiene toda la información concerniente al Efecto previamente seleccionado y a continuación haré una descripción de cada una de los elementos que lo conforman.

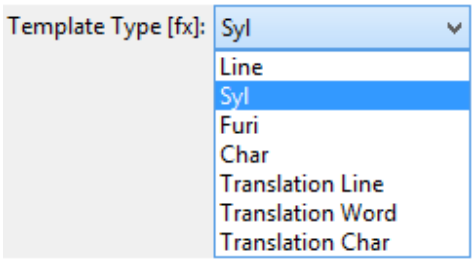
Template Type [fx]:

Es una pestaña desplegable en donde seleccionamos el modo en que se aplicará el Efecto y dependiendo el tipo de Efecto, las opciones de esta pestaña son las siguientes:

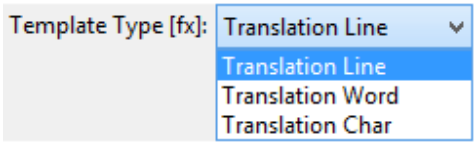
Para los Efectos tipo "**lead-in**", "**hi-light**" y "**lead-out**" son:



Para los Efectos tipo "**shape[fx]**" son:



Y para los Efectos tipo "**translation**" las opciones son:



Ahora veremos qué hace cada una de las opciones de esta pestaña desplegable y cómo sacarle el mejor provecho a la hora de decidirnos por una de ellas.

Template Type [fx]: Line

La opción **Line** hace que se genere una única línea de Efecto por cada línea a la que se le aplique el mismo.

Template Type [fx]: Syl

La opción **Syl** genera una Línea de Efecto por cada Sílaba de cada Línea a la que se le aplique un Efecto.

Template Type [fx]: Furi

La opción **Furi** genera una Línea de Efecto por cada **Furigana** que tenga cada Línea que haya sido seleccionada para aplicarle un Efecto. Es necesario, más adelante, saber qué es y cómo se sincroniza un **Furigana**, para incluirlo como parte de nuestros Efectos.

De adelanto, los **furiganas** son los símbolos que están en rojo en la anterior imagen, y son **hiraganas** que indican cómo se pronuncia el **kanji** que está justo debajo de ellos. Esto genera más dudas, ya que para explicar un término, incluí otros dos nuevos, pero seguramente también harán parte de las próximas entregas. Se aclararán términos como: **Romaji, Kanji, Hiragana, Katakana, Kana, Furigana** y aquellos otros que de pronto puedan ser pertinentes a la hora de hacer Karaokes de gran calidad.

Template Type [fx]: Char

La opción **Char** genera una Línea de Efecto por cada **Caracter** (letra, número, signo de puntuación, espacios y tabulaciones, símbolos matemáticos y demás) que haya en cada Línea que haya sido seleccionada para aplicarle un Efecto.

Template Type [fx]: Translation Line

La opción **Translation Line** genera una Línea de Efecto por cada Línea de Traducción a la que se le aplique un Efecto. Esta opción al igual que las próximas dos, están pensadas para que sean usadas solo en las Líneas de Traducción.

Template Type [fx]: Translation Word

La opción **Translation Word** genera una Línea de Efecto por cada Palabra de cada Línea a la que se le aplique un Efecto.

Template Type [fx]: Translation Char

La opción **Translation Char** genera una Línea de Efecto por cada Carácter de cada Línea a la que se le aplica un Efecto.

Un Efecto en el **Kara Effector** no está necesariamente atado a un único Tipo de Plantilla (**Template Type**), ya que puede ser cambiado por otro a gusto de cada quien y dependiendo de los resultados que cada uno quiera obtener, o sea que es posible pasar un Efecto de **Template Syl** a **Template Char**, por ejemplo, si uno así lo quiere.

Tiempos del Efecto:

Para modificar los tiempos de un Efecto hay dos opciones, una de ellas es desde las celdas de texto ("**textbox**") dispuestas para ese fin:

Line Start Time: es el tiempo de inicio de las Líneas que genera un Efecto. En el ejemplo de la imagen el tiempo que pone es "**l.start_time**" que es equivalente al tiempo de inicio original de cada Línea que haya sido seleccionada para aplicarle un Efecto.

l.start_time = line.start_time

Por ejemplo l.start_time + 1000 quiere decir que el tiempo de inicio de las Líneas de Efectos generadas será 1000 milisegundos (1000 ms = 1 segundo) después del tiempo de inicio original de las Líneas seleccionadas para aplicarle el Efecto.

Line End Time: es el tiempo final de las Líneas que genera un Efecto. En el ejemplo de la imagen el tiempo que pone es "**l.end_time**" que es equivalente al tiempo final original de cada Línea que haya sido seleccionada para aplicarle un Efecto.

l.end_time = line.end_time

La otra forma de modificar los tiempos es con la función **retime**, que es una de las muchas funciones que pueden ser empleadas en el **Kara Effector** y que veremos en los próximos volúmenes.

Centros en los Ejes (Posición):

Los Centros en los Ejes son celdas de texto en donde las coordenadas "**x**" y "**y**" de la posición son los centros para ubicar las Líneas de Efecto generadas.

La ventaja de estas y de todas las celdas de texto del **Kara Effector** es poder usar todas las operaciones matemáticas para modificar a nuestro gusto los valores que queremos usar. Las anteriores dos variables son solo dos de una lista de variables que también veremos en los próximos tomos.

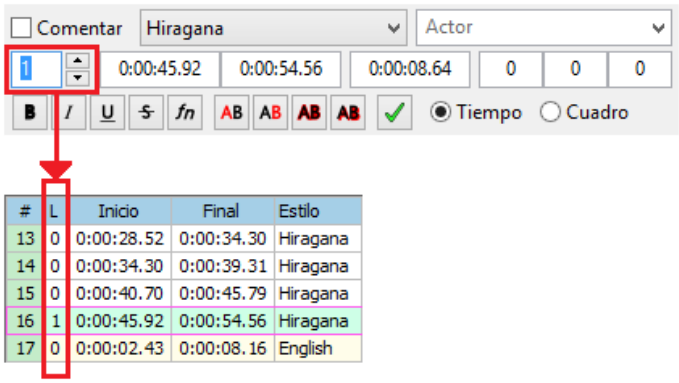
Alineación de las Líneas de Efecto:

Align [\an]: es la alineación del Efecto respecto a los centros de las Líneas seleccionadas para aplicar el mismo. Indica el punto de referencia que se alinea con los centros seleccionados en las dos celdas anteriores (**Center in 'X'** y **Center in 'Y'**).

En realidad la Alineación se ajusta a la posición final que tendrá la Línea de Efecto generada en el caso de las posiciones estáticas o a las trayectorias del movimiento.

Capa de las Líneas de Efecto:

Layer: es el número de la capa que tendrá la Línea de Efecto generada. Para modificar en número de la capa se puede hacer en la celda de texto asignada para ello o con la función **relayer**, que también veremos más adelante.



La función de la capa (**layer**) es determinar qué se verá encima de qué, en nuestro vídeo, entre dos o más objetos karaoke (Caracteres, Sílabas, Palabras, Líneas de Texto o Shapes) que coinciden total o parcialmente en su posición.

Posición y Movimiento:

Pos in 'X': es la o las coordenadas con respecto a la posición final o del movimiento de la línea de Efecto con respecto al eje "x". Para el caso de más de una coordenada, éstas se separan con comas (,) o con punto y coma (;).

fx.pos_x = coordenada en "x" de la línea de Efecto

Pos in 'Y': es la o las coordenadas con respecto a la posición final o del movimiento de la línea de Efecto con respecto al eje "y". Para el caso de más de una coordenada, éstas se separan con comas (,) o con punto y coma (;).

fx.pos_y = coordenada en "y" de la línea de Efecto

Times Move: para el caso del movimiento, se requiere un tiempo de inicio y un tiempo final para el mismo. Los tiempos son medidos en milisegundos (ms).

Si la celda de texto del **Times Move** está vacía, entonces el tiempo de inicio del movimiento es cero (0) y el tiempo final será el mismo que la duración total de la línea de Efecto (**fx.dur**). Los dos parámetros para el tiempo de inicio y final de un movimiento van separados por coma (,) o por punto y coma (;).

fx.pos_x y **fx.pos_y** son dos variables de la librería fx que hacen referencia a muchos de los valores del **Kara Effector** que podemos utilizar en nuestros efectos. En el próximo tomo veremos estas y más librerías que nos harán aún más simple la tarea de hacer un Efecto Karaoke.

Objetos Karaoke:

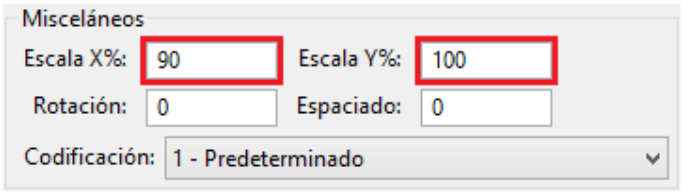
Return [fx]: es la parte visible en el vídeo que se verá de un Efecto Karaoke. Puede ser un Caracter, una Sílaba, una Palabra, una Línea de Texto, un Número o un Símbolo matemático, un Hiaraga, Furigana, Katakana, Kanji y/o Shape.

Repeticiones:

Loop: es la cantidad de repeticiones que se harán de un Efecto Karaoke. Si esta celda de texto está vacía, su valor por default es 1. Para algunas funciones del **Kara Effector** a veces se usan dos parámetros en el **loop**, para estos casos las repeticiones totales del Efecto será el producto de los dos parámetros. La cantidad de repeticiones de un Efecto se puede determinar de dos maneras, una es con la celda de texto **loop** y la otra es con la función **maxloop**, que también es otra de las funciones que veremos más adelante junto con las que ya se han mencionado antes.

Tamaño:

Size: es el porcentaje del tamaño del Objeto Karaoke. En el caso de que esta celda de texto esté vacía, se toman los porcentajes del tamaño que están en el estilo de las Líneas seleccionadas:



Lo que en tags sería: → \fscx90\fscy100

En el caso de que en **Size** haya un solo valor, éste se toma para las dos escalas, ejemplo:

Size = 45 → \fscx45\fscy45

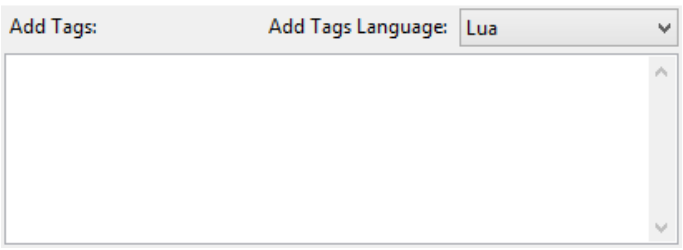
Y para cuando haya dos valores, entonces el primero es el porcentaje para la escala en "x" y el segundo para la escala en "y", ejemplo:

Size = 70, 92 → \fscx70\fscy92

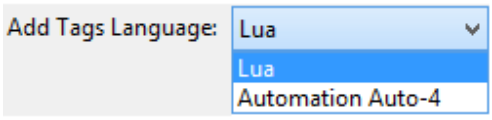
Colores y Transparencias de las Shapes:

Cuando en **Return [fx]** ponemos una **shape**, los colores y transparencias que se verán en nuestro Efecto serán los de la anterior imagen.

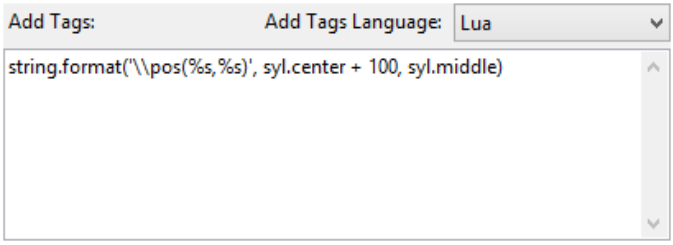
Agregar Tags:



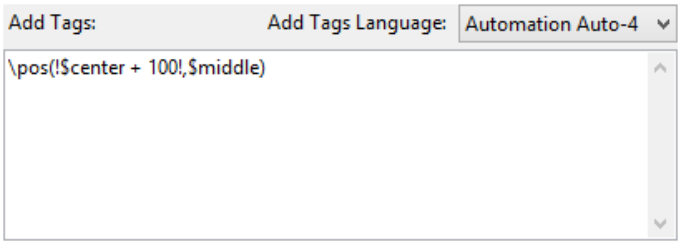
Add Tags: es una celda de texto que nos da la posibilidad de agregar nuevos tags a un Efecto, al igual que funciones. Ahora la versión 3.2 del **Kara Efector** ofrece la opción de añadir tags en el lenguaje que queramos, o sea, se pueden añadir en lenguaje **LUA** o en lenguaje **Automation Auto-4.**



Ejemplo en lenguaje **LUA**:

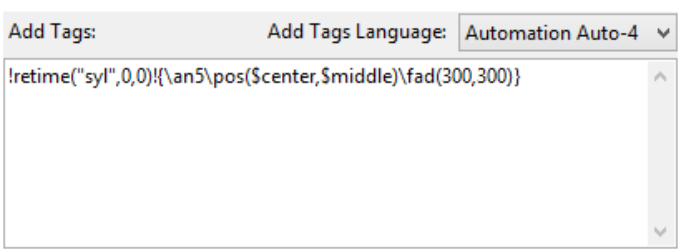


Este mismo tag en lenguaje **Automation Auto-4** sería:



La adaptación del **Kara Effector** para reconocer el lenguaje **Automation Auto-4** es ideal para todos aquellos que ya se familiarizan con ese método de hacer Efectos, ya que se usa exactamente igual que en el **Aegisub** cuando se hacen Efectos Karaokes. En el **Aegisub**, una plantilla de Efecto podría ser:

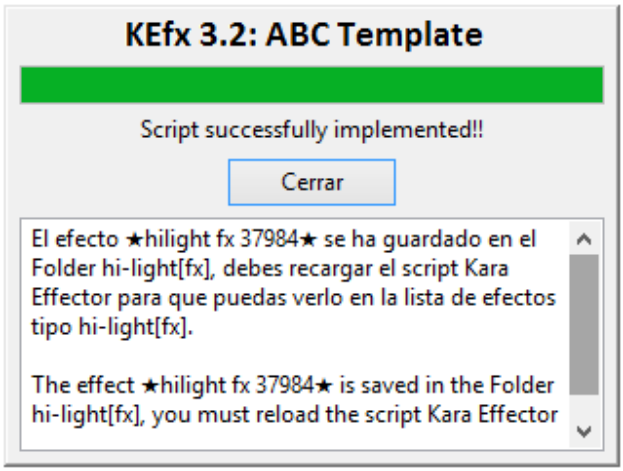
En el **Kara Effector** no cambiaría absolutamente nada:



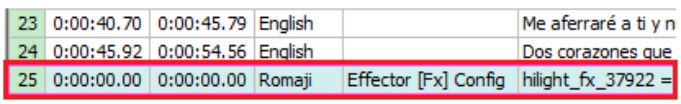
Generar y Guardar Efectos Nuevos:

Al hacer Efectos nuevos a partir de la modificación de alguno ya hecho en el **Kara Effector** o al crear uno nuevo partiendo desde ceros, genera la necesidad de guardarlos para luego ser usados posteriormente. Pensando en esto, están estas tres opciones que son muy simples de usar:

Print Config [fx]: es el **cheackbox** que decide sin un Efecto se genera normalmente o si en vez de ello, se guarda de forma permanente en el **Kara Effector**. Si esta opción no está marcada, el Efecto se genera en el **Aegisub**, de lo contrario, las configuraciones del Efecto se "**imprimen**" en el **Effector-newfx.lua** directamente o en el Aegisub para posteriormente ser copiado de forma manual en el **Effector-newfx.lua.** Al guardarlo directamente de salir un aviso como este:

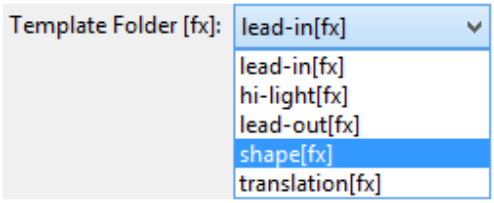


Este recuadro es la señal inequívoca de que el Efecto se ha guardado de forma satisfactoria en el archivo LUA del Effector, **Effector-newfx.lua**, que es el archivo que está destinado para los Efectos nuevos. De lo contrario, el Efecto se imprime en el **Aegisub** y se pega manualmente en el **Effector-newfx.lua**, se verá algo como esto:



El método para hacer que los Efectos nuevos se guarden directamente es muy simple, pero prefiero que quede explicado más claramente en los siguientes tomos.

Template Folder [fx]: es el lugar de destino del nuevo Efecto, o sea que al guardarlo aparece en el listado de tipos de Efectos que elegimos en esta opción:



New [fx] Name: es la celda de texto donde escribimos el nombre del nuevo Efecto. Se pueden usar letras y números, pero no signos de puntuación ni caracteres especiales del **ASCII**. Si no escribimos nada en esta celda e imprimimos el nuevo Efecto, el **Kara Effector** le asigna un nombre único por Default a cada Efecto que se imprima sin que escribamos el nombre.

Funciones y Variables:

Es una de las celdas de texto más importantes ya que aquí podemos declarar tanto variables como funciones y arreglos, en general se pueden definir el resto de cosas que no podamos hacer directamente desde el resto de las celdas.

Declarar un variable es tan simple como asignarle el nombre que queramos, seguido del signo "igual" (=) y por último el valor que tendrá dicha variable. Para los siguientes ejemplos inventé una variable llamada "**Color**" y el valor asignado es uno random.

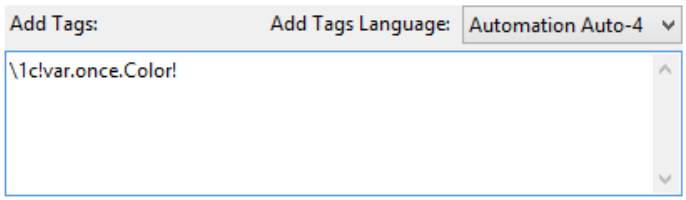
random.color es una función del **Kara Effector** y como su nombre ya lo hace prever, retorna un color al azar de entre todos los del espectro, excepto el blanco y el negro. Esta y las demás funciones del **Kara Effector** se verán con ejemplos en los próximos tomos.

Una misma variable declarada se puede "llamar" de diversas maneras, en caso de que una variable tenga un valor constante, no importaría de qué forma se llame, su valor seguirá siendo el mismo. Ejemplo:

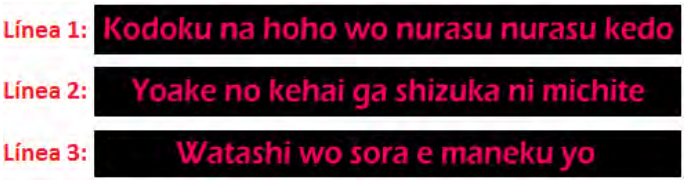
Angle = 2*math.pi

La variable "Angle" siempre tendrá el mismo valor sin importar de qué ni de cuántas veces se llame a ser usada. Caso distinto es cuando se usa un valor random como el del siguiente ejemplo:

Luego de declarar la variable ya puede ser llamada en cualquiera de las celdas de texto del **Kara Effector**, en este ejemplo lo hice en "**Add Tags**":



var.once: es la forma de llamar a una variable para que se ejecute una sola vez en todo el karaoke



El random del color se hizo una sola vez y por eso el color primario de todas las sílabas de todas las líneas es el mismo, como se puede ver en la imagen anterior.

Este ejemplo lo he hecho en lenguaje **Automation Auto-4** y el color random que generó la variable lo usé para el color primario:

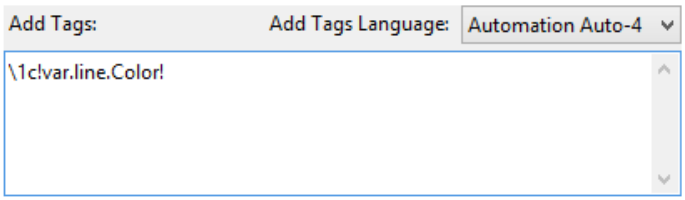
\1c!var.once.Color!

Este mismo ejemplo hecho, pero en lenguaje **LUA** tendría dos formas principales de hacerse:

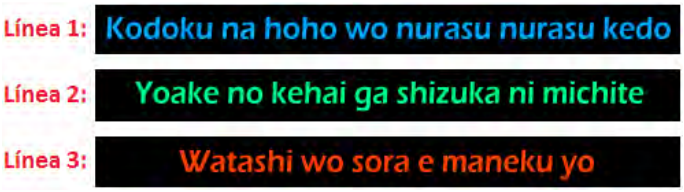
1. string.format("\\1c%s", var.once.Color)
2. "\\1v"..var.once.Color

Ambas hacen exactamente lo mismo y ya es decisión de cada uno elegir la que más se le facilite o guste.

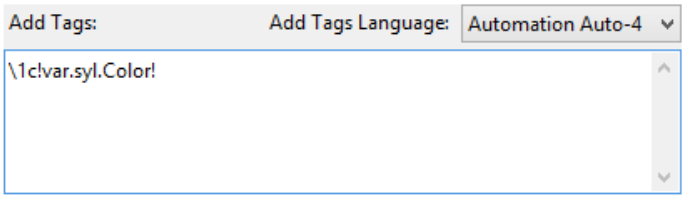
var.line: el random se ejecuta, pero una vez por cada línea a la que le apliquemos el Efecto. A diferencia del **var.once** que solo hizo el random una sola vez:



El random generó para cada línea un color diferente:

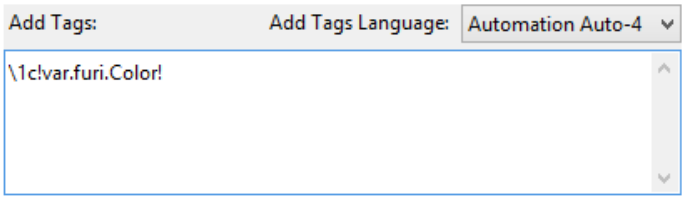


var.syl: el random se ejecuta una vez por cada sílaba de cada línea:

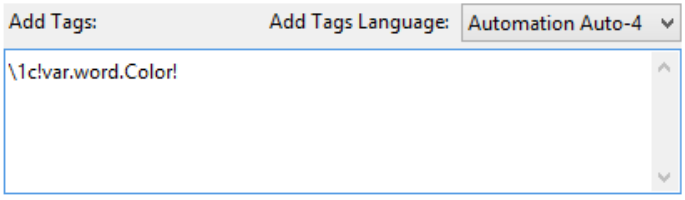


El random generó un color diferente para cada sílaba:

var.furi: cumple la misma función que **var.syl**, con la diferencia de que genera el random una vez por cada Furigana que tengan las líneas seleccionadas:

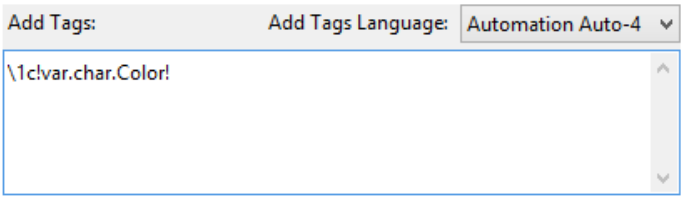


var.word: es similar al **var.syl**, pero genera el random una vez por cada palabra de cada línea seleccionada:

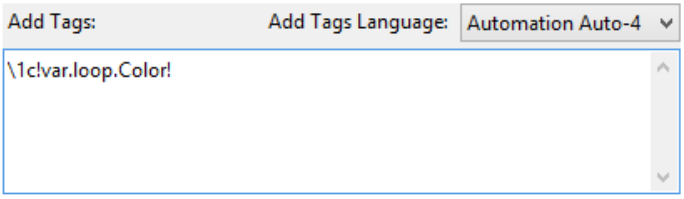


El random generó un color diferente para cada palabra:

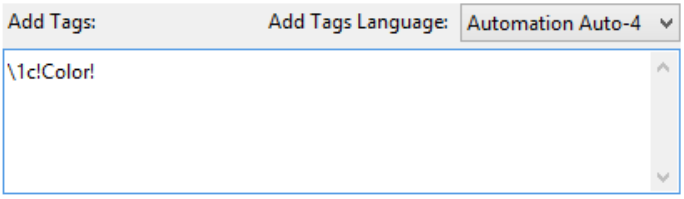
var.char: el random se ejecuta una vez por cada carácter de cada línea:



var.loop: el random se ejecuta una vez por cada loop que hayamos puesto como número de repeticiones para un Efecto:

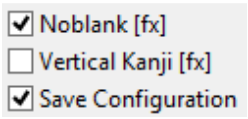


El **var.once**, **var.line**, **var.word**, **var.syl**, **var.furi**, **var.char** y **var.loop**; adquiere relevancia, como lo mencionaba antes, cuando declaramos una variable o alguna función que incluya uno o más valores random. De otra manera los valores de las variables serían constantes y el valor al llamar a ese tipo de variables sería el mismo sin importar el "**var**" que se use. Para variables constantes está la opción de llamar a la variable directamente por su nombre como se ve en el siguiente ejemplo:



La forma de llamar una variable declarada en "Variables" depende mucho del resultado que se quiera obtener. En esta celda de texto, las variables y funciones, lo mismo que los arreglos (array) pueden ir separados por comas (,) o por punto y coma (;), pero para que sea posible que sean llamadas por su nombre original (como en el ejemplo anterior), debe ir separadas por punto y coma (;).

Configuraciones "Check":

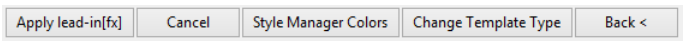


Noblank [fx]: al estar marcada, esta opción NO toma en cuenta a las Sílabas que están en Blanco o Vacías, al igual que Líneas en Blanco. Para poder tener en cuenta a estos valores en blanco, lo que debemos hacer es "desmarcar" esta opción.

Vertical Kanji [fx]: al marcar esta opción, las líneas de karaoke de un Efecto saldrán en pantalla de forma Vertical en vez de horizontalmente, que es la forma en que salen tradicionalmente. Es recomendado para los Kanjis o el resto de los pictogramas japoneses:

Save Configuration: cumple la misma función que su opción gemela de la ventana de inicio del **Kara Effector**, o sea que conserva las modificaciones hechas en la segunda ventana a un Efecto, siempre y cuando no recarguemos el **Kara Effector**.

Botones de Ejecución:



Apply [fx]: aplica el Efecto seleccionado

Cancel: cierra el **Kara Effector**

Style Manager Colors: asigna los valores de los colores y transparencias del Estilo de las Líneas seleccionadas a los colores y transparencias **Shape**, en el caso en que queramos usar dichos valores. Es decir que cumple la misma función que la opción **Style Manager** de la ventana de inicio del **Kara Effector**.

Change Template Type: cambia el tipo de Plantilla del Efecto. En **Template Type** elegimos el tipo de Plantilla a la que queremos pasar el Efecto, acto seguido, pulsamos el botón **Change Template Type** e inmediatamente todas las variables se convertirán, como por ejemplo, de syl.center a char.center, de line.duration a word.duration.

Esta opción facilita la tarea de cambiar un Efecto de un tipo de Plantilla a otra, ya que también podemos hacerlo de forma manual.

Esta opción debe tomarse con cierta calma ya que en algunas ocasiones no queremos cambiar alguna variable en especial y debemos estar atento cuando no queramos que esto suceda.

Back <: nos lleva de vuelta a la ventana de inicio del **Kara Effector** en caso de que hayamos pasado por alto alguna modificación en dicha ventana o que queramos cambiar de Efecto sin la necesidad de cancelar y volver a abrir.

Este es un breve repaso de los elementos que hacen parte de la segunda ventana del **Kara Effector**. No teman si hasta este punto aún tienen cosas que no has quedado del todo claras, de hecho cuento con ello, y por eso la necesidad de los próximos tomos, en donde veremos a profundidad los ítems anteriormente mencionados en este tomo y aquellos que aún ignoran.

Es todo por ahora, recordándoles que pueden escribirnos sus dudas y comentarios en el Blog Oficial del **Kara Effector** y/o en nuestros canales de **You Tube**.

Kara Effector 3.2:

Y ha llegado la hora de empezar a ver las Librerías con las que cuenta en **Kara Effector 3.2** y para ello he dispuesto una serie de ejemplos y listados para tratar que cada una de ellas quede lo más clara posible. El conocimiento y dominio de las Librerías es indispensable a la hora de hacer y modificar Efectos de alta calidad.

Librería "l":

Es un listado de variables y constantes que hacen referencia a valores de la "Línea", es decir, a cada una de las líneas del archivo .ass.

La mayoría de los valores de esta Librería son los que asignamos en el Estilo de Línea en el editor de estilos del **Aegisub**:

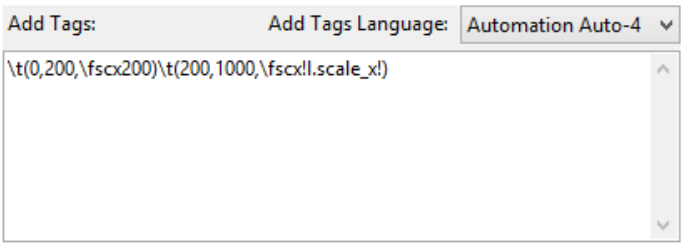
1. **l.fontname**
2. **l.fontsize**
3. **l.color1**
4. **l.color2**
5. **l.color3**
6. **l.color4**
7. **l.alpha1**
8. **l.alpha2**
9. **l.alpha3**
10. **l.alpha4**
11. **l.margin_l**
12. **l.margin_r**
13. **l.margin_v, l.margin_t, l.margin_b**
14. **l.align**
15. **l.outline**
16. **l.shadow**
17. **l.scale_x**
18. **l.scale_y**
19. **l.angle**
20. **l.spacing**

Los anteriores 20 valores de la Librería “l” no necesitan mucha explicación, ya que son los que siempre usamos cuando creamos y modificamos un nuevo Estilo de Línea.

Un pequeño ejemplo sería:

En el ejemplo, se aumenta la escala en el eje “x” en 200% desde 0 hasta 200 ms y luego desde 200 ms a 1000 ms la escala en el eje “x” vuelve a la proporción que tiene en el Estilo, es decir, 90% (como se puede apreciar en la imagen anterior del Estilo, ítem 17).

Veamos este mismo ejemplo, pero ahora en lenguaje **Automation Auto-4**. Aunque la forma de escribirlo cambie dado los dos tipos de lenguajes, el resultado es el mismo:



Y continuando con los valores de la Librería “l”, ahora veremos los que son referentes a las Líneas de Texto y a su posición en el vídeo.

l.width: es el Ancho medido en pixeles de cada una de las Líneas.

l.left: es la Distancia medida en pixeles desde la parte izquierda del vídeo hasta la parte izquierda de la Línea.

l.center: es la Distancia medida en pixeles desde la parte izquierda del vídeo hasta el centro de la Línea.

l.right: es la Distancia medida en pixeles desde la parte izquierda del vídeo hasta la parte derecha de la Línea.

l.height: es la Altura medida en pixeles de cada una de las Líneas.

l.top: es la Distancia medida en pixeles desde la parte superior del vídeo hasta la parte superior de la Línea.

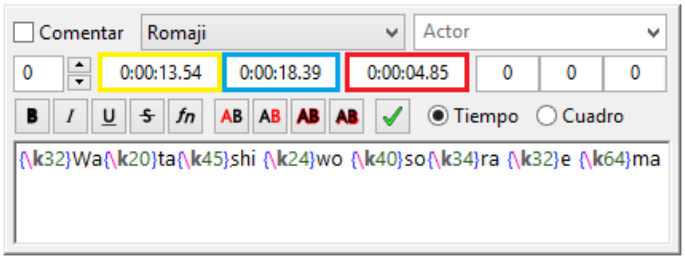
Notarán que la parte superior de la Línea no coincide con la parte superior de sus letras más grandes, esto se debe a que hay un espacio extra que sirve para separar las Líneas verticalmente cuando estas están una encima de la otra. Ya que de otro modo la Líneas se verían pegadas.

l.middle: es la Distancia medida en pixeles desde la parte superior del vídeo hasta la mitad de la altura de la Línea.

l.bottom: es la Distancia medida en pixeles desde la parte superior del vídeo hasta la parte inferior de la Línea.

l.descent: es la Distancia medida en pixeles desde la parte superior de la Línea hasta la parte superior Real de la misma. Es la Distancia que separa verticalmente una Línea de otra.

Para terminar, los valores de la Librería “l” que hacen falta son los que tienen referencia con el tiempo y su forma de escritura:

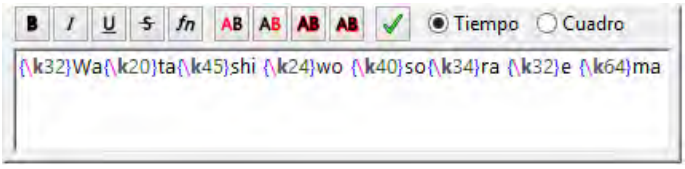


l.start_time: (recuadro amarillo de la anterior imagen) es el tiempo de inicio medido en ms (milisegundos) de cada Línea.

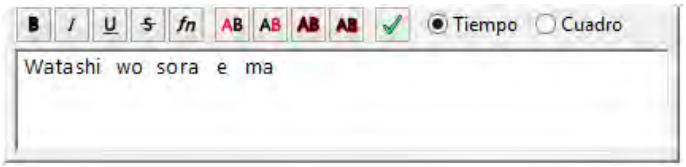
l.end_time: (recuadro azul) es el tiempo final medido en (ms) de cada Línea.

l.duration: (recuadro rojo) es la duración total medida en (ms) de cada Línea de Dialogo del archivo .ass.

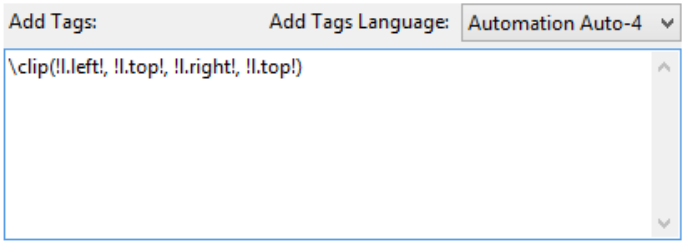
l.text: es todo lo que está escrito en la Línea, incluidos los tags:



l.text_stripped: es similar a **l.text**, pero con la diferencia que no tiene en cuenta a los tags que tenga dicha Línea:

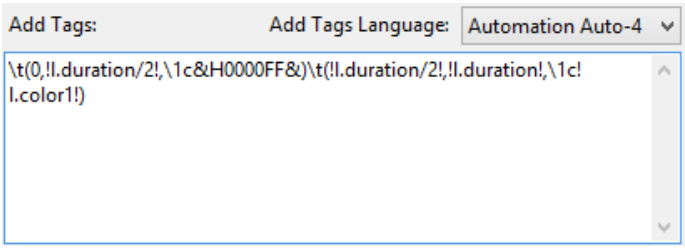


El uso de cada uno de los valores de la Librería "l" o de las próximas librerías que veremos, facilitará la forma de hacer Efectos que en principio serían mucho más complejos. Acá otro corto ejemplo de cómo poder usar la Librería "l":



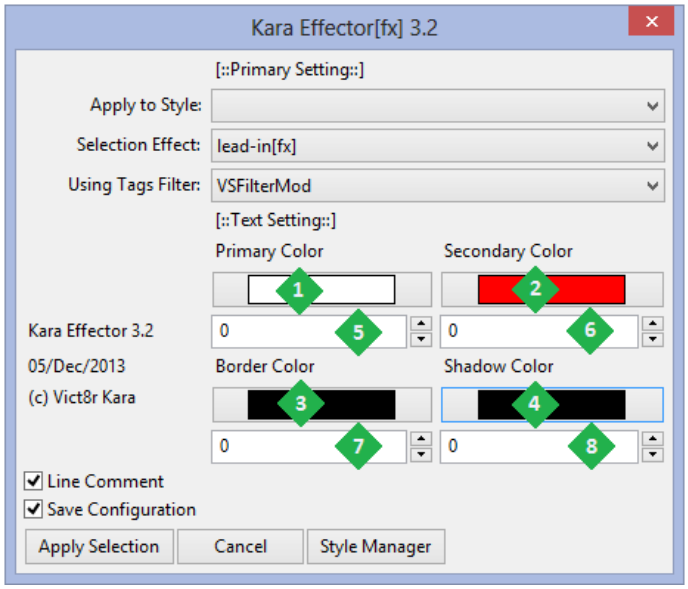
Es un clip en donde es visible toda la Línea.

En este otro ejemplo usamos **l.duration** para generar una transformación del color primario. Desde 0 hasta la mitad de la duración, el color primario se transforma en Rojo (**&H0000FF&:** Rojo en formato .ass) y desde la mitad de la duración hasta la duración total, regresa a su color original asignado en el Estilo (**l.color1**):

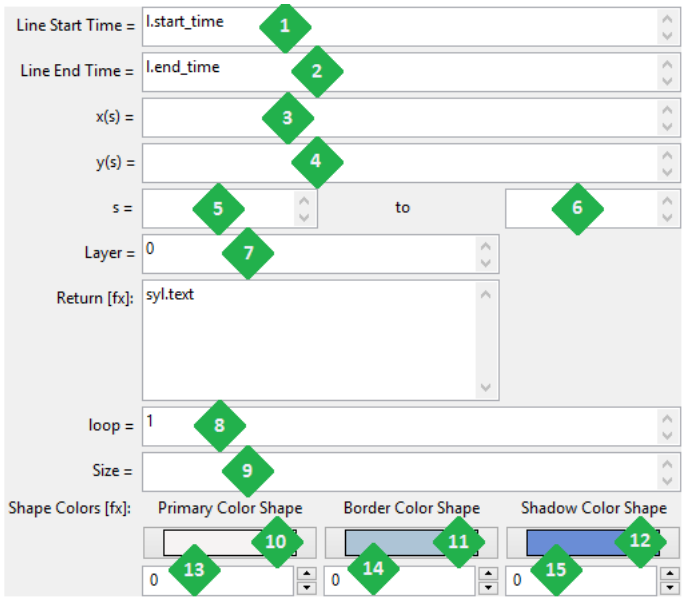


Librería "fx":

Es la Librería que contiene los valores del Kara Effector.

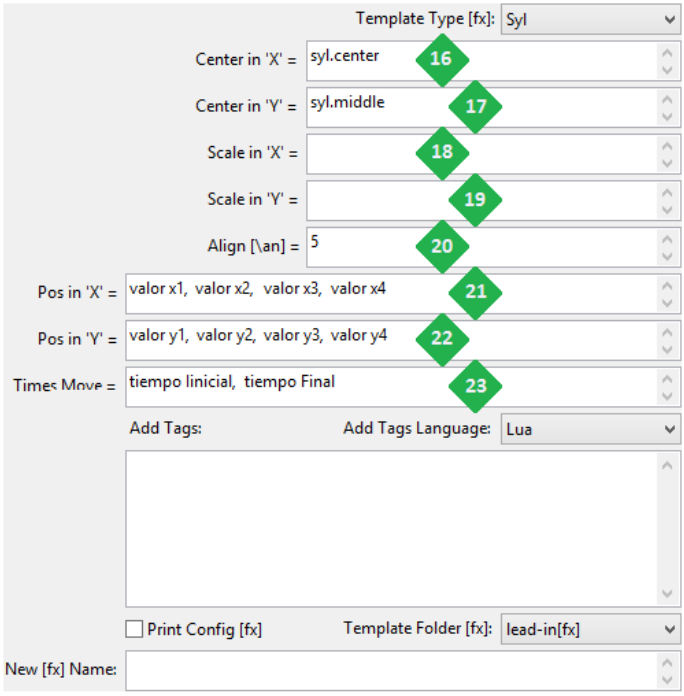


1. **text.color1**
2. **text.color2**
3. **text.color3**
4. **text.color4**
5. **text.alpha1**
6. **text.alpha2**
7. **text.alpha3**
8. **text.alpha4**



1. **fx.start_time**

2. **fx.end_time**
3. **fx.fun_x**
4. **fx.fun_y**
5. **fx.domain_i**
6. **fx.domain_f**
7. **fx.layer**
8. **fx.maxloop_fx**
9. **fx.sizex, fx.sizey**
10. **shape.color1**
11. **shape.color3**
12. **shape.color4**
13. **shape.alpha1**
14. **shape.alpha3**
15. **shape.alpha4**



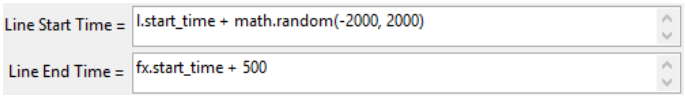
16. **fx.center_x**
17. **fx.center_y**
18. **fx.scale_x**
19. **fx.scale_y**
20. **fx.align**
21. **fx.move_x1, fx.move_x2, fx.move_x3, fx.move_x4**
22. **fx.move_y1, fx.move_y2, fx.move_y3, fx.move_y4**
23. **fx.movet_i, fx.movet_f**

No parecen ser muchas, pero son suficientes. Ahora veremos una pequeña explicación de cada una de ellas.

Bueno, considero que los valores de la ventana de inicio del **Kara Effector** no necesitan de mucha explicación ya que son los colores y transparencias que tendrán por default cada una de las Líneas de Efecto generadas.

fx.start_time: es el tiempo de inicio de cada una de las Líneas generadas por un Efecto. Puede ser modificado directamente desde su respectiva celda de texto o con la función **retime** desde **"Add Tags"** o alguna función hecha en **"Variables"**.

fx.end_time: es el tiempo final de cada una de las Líneas generadas por un Efecto. Este valor puede ser modificado directamente desde su respectiva celda de texto o con la función **retime** desde **"Add Tags"** o alguna función hecha en **"Variables"**. La variable **fx.start_time** puede ser usada en esta celda de texto, ejemplo:



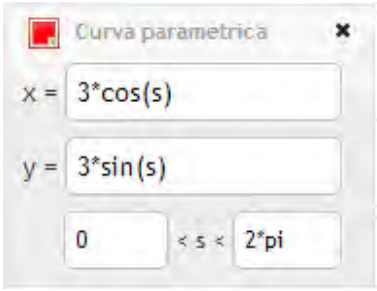
O sea que **fx.start_time** es un valor aleatorio entre -2000 ms y 2000 ms respecto al tiempo de inicio original de cada Línea seleccionada para aplicarle un Efecto (**l.start_time**).

Por otro lado, **fx.end_time** pone que será igual al tiempo en donde inició la Línea de fx (**fx.start_time**), sumado a 500 ms. Es fácil deducir que el **fx.dur** de todas las Líneas que se generen con estos dos tiempos, siempre será de 500 ms. (véase la siguiente variable).

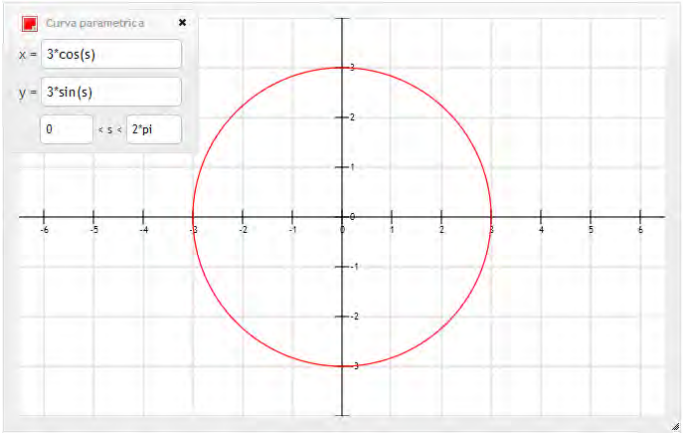
fx.dur: es la duración total de cada Línea fx que sea generada por un Efecto. Es equivalente a la siguiente diferencia:

fx.dur = fx.start_time – fx.end_time

fx.fun_x: es la ecuación paramétrica de "x" en términos de "s" (donde "s" es el dominio de la función). Ejemplo:

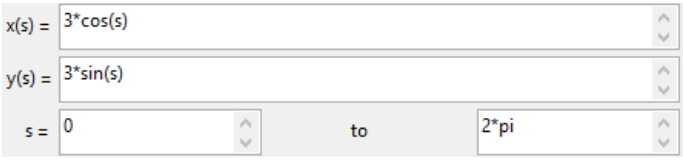


En la anterior imagen están las ecuaciones paramétricas de “x” y “y” de un Círculo de Radio 3. También podemos ver el dominio “s” que va desde 0 a 2*pi:



La anterior gráfica fue generada en http://fooplot.com

En el Kara Effector pondríamos así:



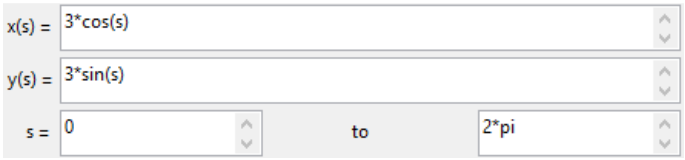
Las escalas en el **Kara Effector**, tanto en el eje “x” como en el “y” son por default 1, es decir que este círculo se verá en el vídeo muy pequeño, ya que el Radio del Círculo es solo de 3 pixeles. Hay dos formas de aumentar su tamaño: o se aumenta el Radio del Círculo directamente en las ecuaciones paramétricas o se aumentan las escalas en las celdas de texto que están al lado derecho de estas, todo depende del gusto y del nivel de habilidad adquirido de cada uno.

En estas celdas de texto podemos modificar las Escalas en ambos ejes de una determinada gráfica:

Olvidaba mencionar que este tipo de Efecto se hacen con Shapes y con un **loop** lo suficientemente grande para que se generen la gráficas.

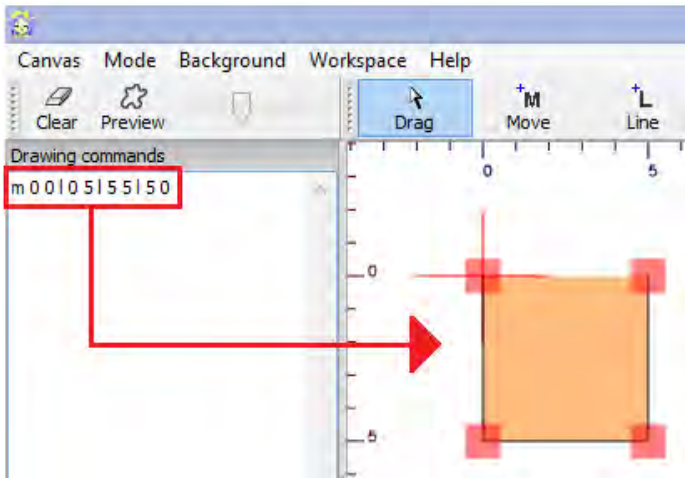
Para un ejemplo guiado en el **Kara Effector** tomando como base las ecuaciones paramétricas del Círculo, sería:

A. Definimos las ecuaciones paramétricas y el dominio de las funciones (“s”):

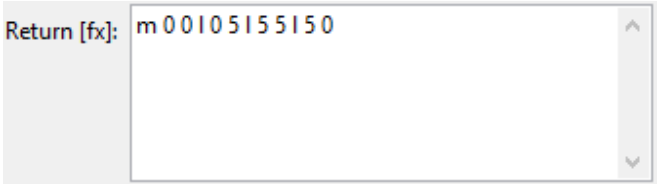


B. Aumentamos las escalas en 10 para que el Radio del Círculo pase de 3 pixeles a 3*10 = 30 pixeles. (Se pude experimentar con distintos valores en las escalas y ver cómo varían las gráficas):

C. Elegimos una **Shape** y la ponemos en **Return [fx]**:

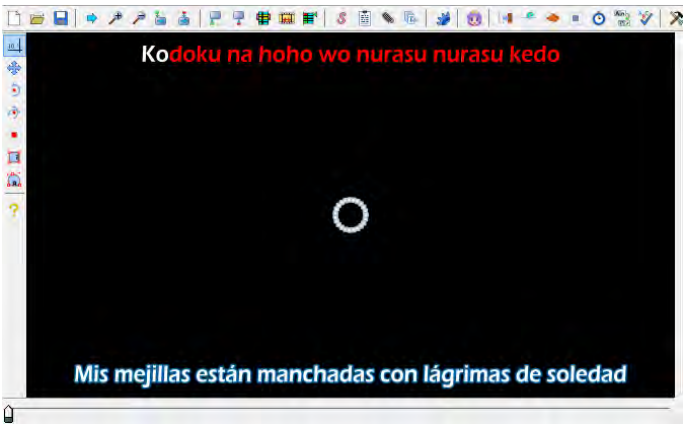


En mi caso, dibujé un Cuadrado en el **AssDraw3**, de 5 pixeles de lado y copié el código de la Shape en **Return [fx]**:



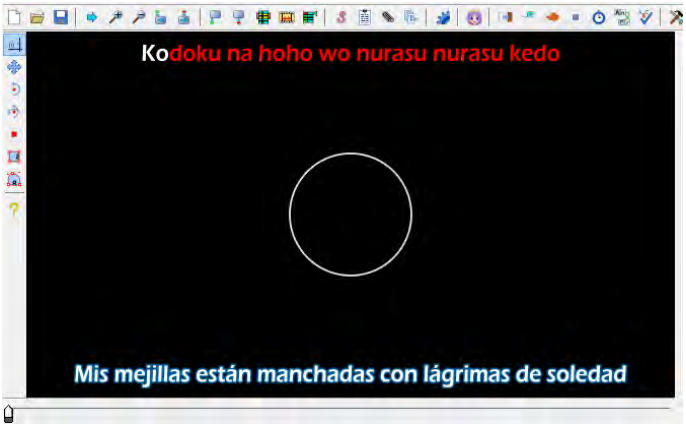
D. Aumentamos el **loop** para que generará la gráfica:

Para este ejemplo puse en **Template Type** la opción “**Line**” para que se genere un solo Círculo por cada Línea:



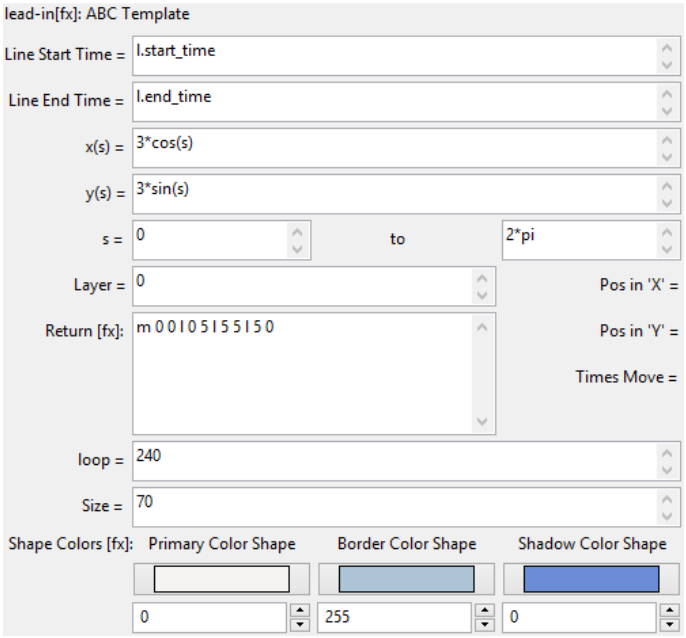
Ya se puede ver en el vídeo el Círculo que se generó, pero parece que 30 pixeles para el Radio es poco para poder ver los 24 cuadraditos (loop) de 5 X 5 pixeles que lo conforman, por ello aumentaré las escalas en ambos ejes a 50 y volveré a aplicar el Efecto:

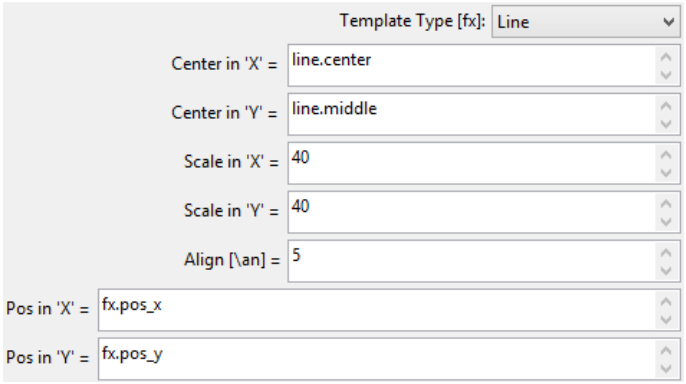
Ahora sí se pueden apreciar los cuadraditos que generan el Círculo. La cantidad de cuadrados, lo mismo que el valor de las escalas, tamaños (**size**) y colores, se pueden modificar para que la gráfica sea más continúa, ejemplo:



Es un ejemplo de cómo "graficar" en el **Kara Effector**.

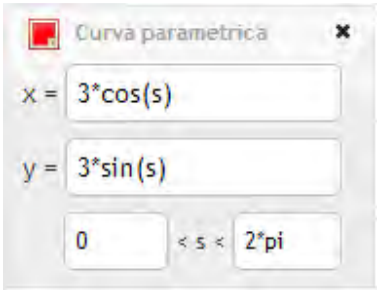
Las configuraciones para generar el anterior Círculo son:





En próximos ejemplos veremos cómo hacer un Efecto a partir de la gráfica de las ecuaciones paramétricas.

fx.fun_y: es la ecuación paramétrica de "y" en términos de "s" (donde "s" es el dominio de la función). Ejemplo:



Con el ejemplo anterior del Círculo, esta variable de la Librería "**fx**" no necesita muchas más explicación y por ello iremos directamente a las siguiente variables.

fx.domain_i: es el inicio del dominio “s” de las ecuaciones paramétricas. Si en esta celda de texto no pone nada, el inicio del dominio por default es 0.

fx.domain_f: es el final del dominio “s” de las ecuaciones paramétricas. Si en esta celda de texto no pone nada, el final del dominio por default es 1.

No necesariamente el inicio del dominio “s” debe ser menor que el final, lo convencional es que sí lo sea para que la gráfica se dibuje normalmente de menor a mayor, de lo contrario de dibujará al revés. Este orden cobra importancia cuando hacemos animaciones con las gráficas y las vemos en el vídeo.

fx.layer: es la capa de cada una de las Líneas de fx y decide la prioridad en el vídeo de dos o más Objetos Karaoke que coinciden de forma total o parcial en su posición o trayectoria de movimiento.

fx.maxloop_fx: este valor equivale a la cantidad total de repeticiones de cada una de las Líneas fx. Se puede usar con este nombre o con uno más conocido por aquellos que ya conocen algo de **Automation Auto-4**: **maxj**

maxj = fx.maxloop_fx

El valor de **maxj** (Máximo valor de “j”) se puede modificar directamente en la celda de texto destinada para ello o con la función **maxloop**.

En el caso de haber un solo valor en esta celda, el valor de **maxj** es ese mismo valor:

En este caso, **maxj** = 2

Para el caso de dos valores:

Para este caso, **maxj** = 2 x 5 = 10

Y para el caso de que haya tres valores:

Y para este último caso, **maxj** = 2 x 5 x 4 = 40

3 es el número máximo de valores que se pueden poner en esta celda de texto. Cuando alguno de ellos o todos, no están, su valor por default es 1.

Más adelante veremos por qué la necesidad de más de un valor en esta celda de texto, pero por ahora les mostraré la variable asignada a cada uno de ellos, ejemplo:

fx.loop_v: (el 2 en la imagen) es el loop vertical.

fx.loop_h: (el 5 en la imagen) es el loop horizontal.

fx.loop_3: (el 4 en la imagen) es un loop de respaldo.

Como lo mencionaba antes, si alguno de estos tres valores no está, por default es 1. Lo que quedaría:

fx.maxloop_fx = fx.loop_v * fx.loop_h * fx.loop_3

maxj = fx.loop_v * fx.loop_h * fx.loop_3

maxj = fx.maxloop_fx

fx.sizex: es el porcentaje del tamaño con respecto al eje “x” asociado al tag \fscx, ejemplo:

En este ejemplo, **fx.sizex** valdría 80 y en el Efecto se verán los siguientes tags: \fscx80\fscy80

Es decir que si solo hay un valor en esta celda de texto, el porcentaje en el eje “y” será el mismo que para el eje “x”, o sea 80% para los dos ejes.

fx.sizey: es el porcentaje del tamaño con respecto al eje “y” asociado al tag \fscy, ejemplo:

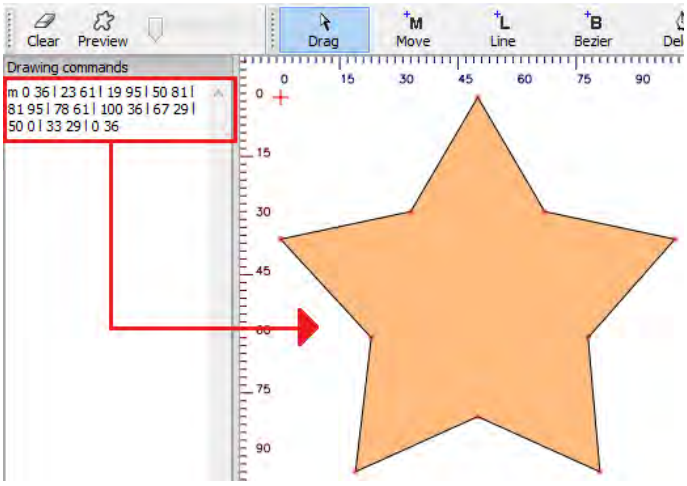
O sea: **fx.sizex** = 80 y **fx.sizey** = 125

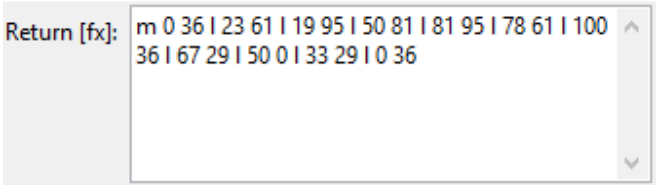
Y en tags: \fscx80\fscy125

Si esta celda de texto está vacía los valores por default de estas dos variables son:

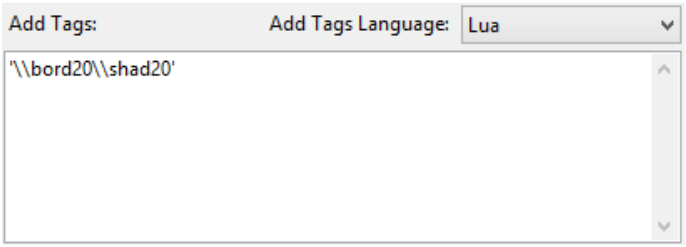
fx.sizex = l.scale_x y **fx.sizey = l.scale_y**

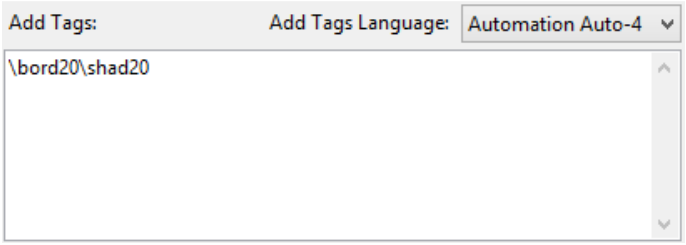
Para las variables de la Librería “fx” concernientes a las figuras hechas en el **AssDraw3** usaré la siguiente Shape:





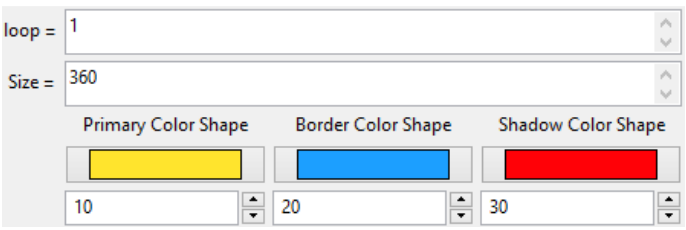
Y le agregaré los siguientes tags:



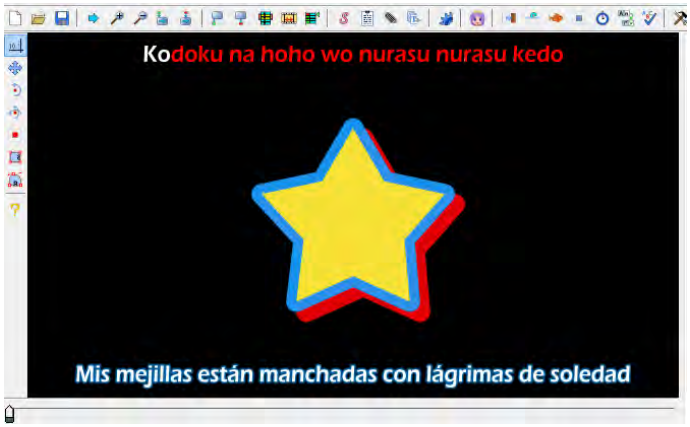


O sea, 20 pixeles para el tamaño del borde y otros 20 para el tamaño de la sombra.

Además de esto, las siguientes configuraciones:



Y al aplicar el Efecto, se verá en pantalla la Shape con las configuraciones que le di:



shape.color1: es el color Primario de la Shape (amarillo).

shape.color3: es el color del Borde de la Shape (azul).

shape.color4: es el color de la Sombra de la Shape (rojo).

shape.alpha1: es la transparencia del color Primario de la Shape (10 para este ejemplo).

shape.alpha3: es la transparencia del color del Borde de la Shape (20).

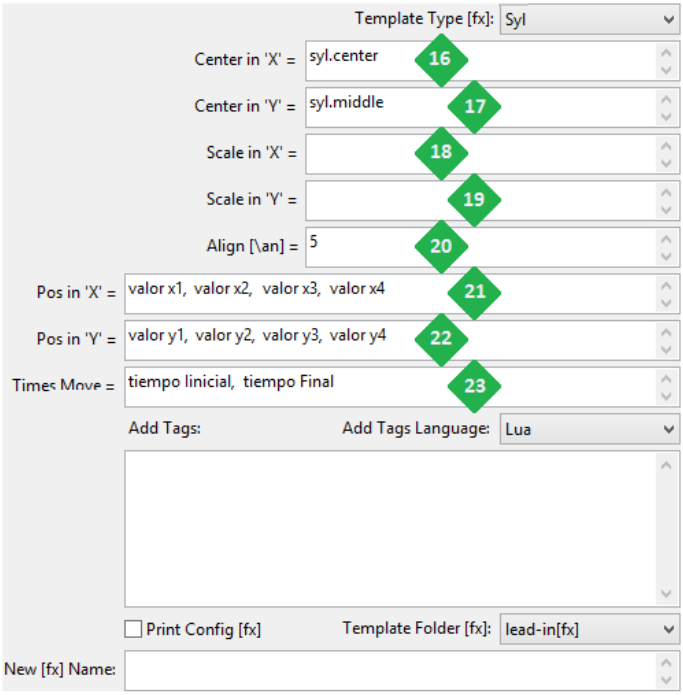
shape.alpha4: es la transparencia del color de la Sombra de la Shape (30).

Para el **Tomo IV** terminaremos con la explicación de las variables restantes de la Librería “**fx**” y veremos más de las librerías del **Kara Effector**. No olviden visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o si quieren comentar, exponer alguna duda o hacer alguna sugerencia.

Kara Effector 3.2:

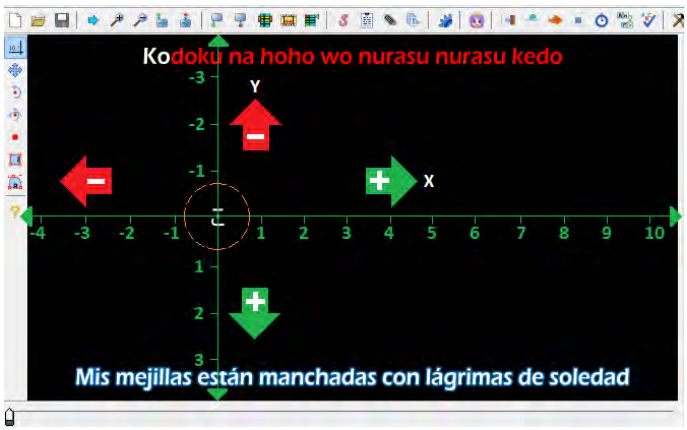
El inicio de este **Tomo IV** veremos la segunda parte de la Librería "**fx**" de **Kara Effector**. Estas son las variables que quedaron sin explicar del **Tomo** anterior y a continuación veremos un poco más de cada una de ellas:

Librería "fx":



16. **fx.center_x**
17. **fx.center_y**
18. **fx.scale_x**
19. **fx.scale_y**
20. **fx.align**
21. **fx.move_x1, fx.move_x2, fx.move_x3, fx.move_x4**
22. **fx.move_y1, fx.move_y2, fx.move_y3, fx.move_y4**
23. **fx.movet_i, fx.movet_f**

fx.center_x: es el Centro medido en pixeles con respecto al eje “x” y hace las veces de la Abscisa al Origen de un sistema de coordenadas cartesianas para cada Línea de fx.



En la imagen anterior dibujé las coordenadas cartesianas con el origen en el centro del Hiragana “ko”:

La coordenada en “x” de ese origen sería **syl.center** que como su nombre lo indica, es el centro de la Sílaba. Nótese que en los archivos .ass, con respecto al eje “y”, hacia arriba es negativo y hacia abajo es positivo, es decir que este eje está invertido.

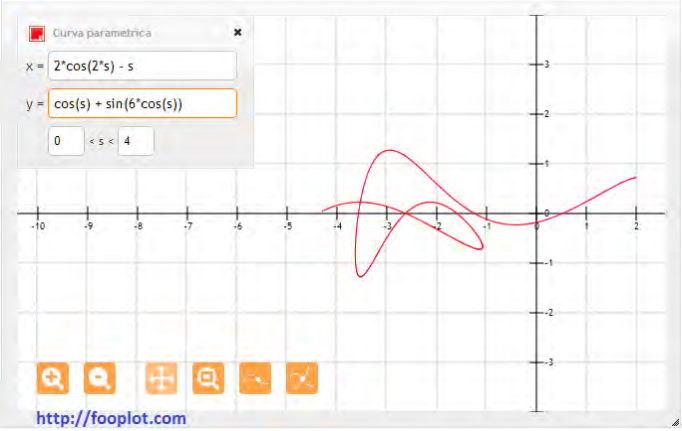
fx.center_y: es el Centro medido en pixeles con respecto al eje “y” y hace las veces de la Ordenada al Origen de un sistema de coordenadas cartesianas para cada Línea de fx.

Como podemos ver en el ejemplo del Hiragana “ko”, sería **syl.middle**, que es el centro de la Sílaba, pero medido verticalmente.

syl.center y **syl.middle** son variables de la Librería “syl” que originalmente ya viene en el Aegisub por default, pero que también explicaré en los próximos tomos, ya que he agregado algunas variables más a esta Librería y es necesario que sepamos en qué consisten dichas variables.

fx.scale_x: junto a **fx.scale_y**, es la Escala con respecto al eje “x” de las gráficas que se generen con ecuaciones paramétricas.

fx.scale_y: es la Escala con respecto al eje “y” de las gráficas que se generen con ecuaciones paramétricas.



Acá hay otro ejemplo de una Gráfica generada por ecuaciones paramétricas, y son este tipo de gráficas las que serán afectadas por las anteriores Escalas. El valor por default de ambas Escalas es 1.

Una vez conocidas las variables **fx.center_x**, **fx.center_y**, **fx.fun_x**, **fx.fun_y**, **fx.scale_x** y **fx.scale_y**; es importante que veamos las siguientes variables:

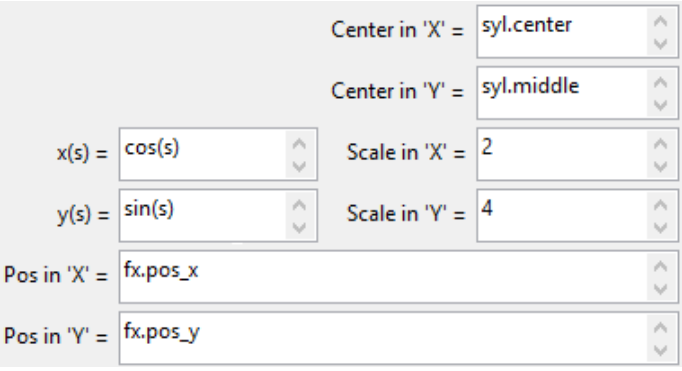
fx.pos_x: es la coordenada en “x” de la posición final de la Línea fx luego de haber asignado valores a las variables **fx.center_x**, **fx.center_y**, **fx.fun_x**, **fx.fun_y**, **fx.scale_x** y **fx.scale_y**.

fx.pos_x = fx.center_x + fx.fun_x * fx.scale_x

fx.pos_y: es la coordenada en “y” de la posición final de la Línea fx luego de haber asignado valores a las variables **fx.center_x**, **fx.center_y**, **fx.fun_x**, **fx.fun_y**, **fx.scale_x** y **fx.scale_y**.

fx.pos_y = fx.center_y + fx.fun_y * fx.scale_y

Veamos un ejemplo para tener claro lo de la Posición final:

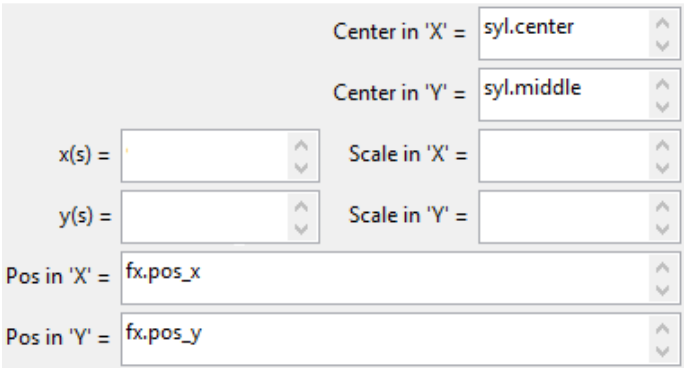


Para este ejemplo obtendríamos los siguientes valores:

fx.pos_x = **syl.center** + cos(s) * 2

fx.pos_y = **syl.middle** + sin(s) * 4

El valor por default de **fx.fun_x** y **fx.fun_y** es 0, teniendo en cuenta esto, para el siguiente ejemplo tenemos:

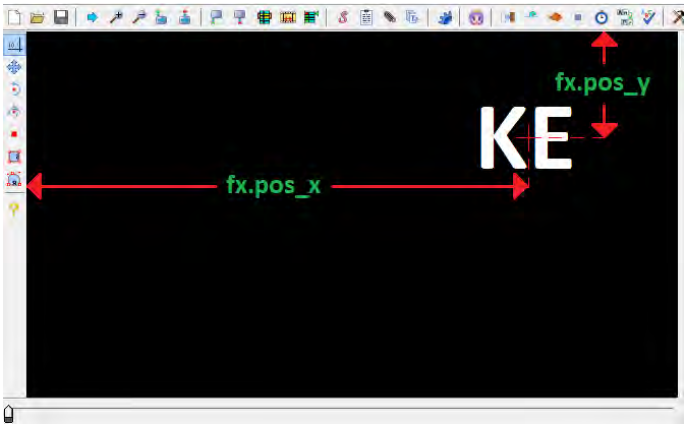


fx.pos_x = **syl.center** + 0 * 1 = **syl.center**

fx.pos_y = **syl.middle** + 0 * 1 = **syl.middle**

Los ceros de las anteriores ecuaciones corresponden a los valores por default de **fx.fun_x** y **fx.fun_y**, ya que en la imagen ambas celdas están vacías. Los unos son los valores por default de **fx.scale_x** y **fx.scale_y**, dado que también están vacías sus respectivas celdas de texto.

Entonces concluimos que **fx.pos_x** y **fx.pos_y** son los centros de la posición final de cada una de las Líneas fx:

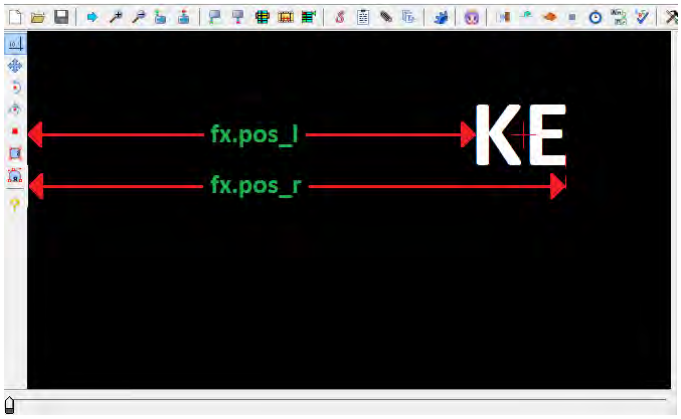


Una vez aclarados los calores de las anteriores dos variables, será más simple entender las cuatro siguientes, ya que dependen directamente de ellas.

fx.pos_l y fx.pos_r: (left y right) son la parte izquierda y derecha de **fx.pos_x** respectivamente.

La izquierda y la derecha de **fx.pos_x** dependen del "**Template Type**", es decir que depende del tipo de plantilla del Efecto. Ejemplo:

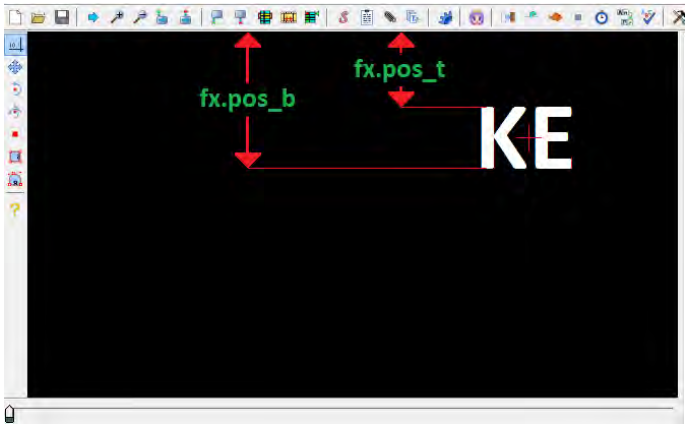
Si la plantilla es tipo "**Syl**" entonces fx.pos_l y fx.pos_r serán la izquierda y la derecha de la Sílaba, sin importar en dónde esté esa Sílaba:



Para **Template Type** "**Line**", entonces **fx.pos_l** y **fx.pos_r** serán la izquierda y la derecha de la Línea de Texto. Lo mismo pasaría para los demás tipos de Plantillas: "**Word**", "**Furi**", "**Char**" y las demás.

fx.pos_t y fx.pos_b: (top y bottom) son la parte superior e inferior de **fx.pos_y** respectivamente.

Como en el caso de las dos variables anteriores, la parte superior e inferior depende únicamente de la posición final:



fx.align: es la alineación de la Línea fx. Debe ser un valor entero entre 1 y 9, inclusive. El valor por default de esta variable es 5. 5 es la alineación recomendada para aquellos que aún no tienen mucha experiencia a la hora de hacer Efectos Karaoke, pero todas son importantes.

fx.move_x1, fx.move_x2, fx.move_x3, fx.move_x4: son la cuatro posibles coordenadas con respecto al eje "x" de las posición o de la trayectoria del movimiento de la Línea fx. Son las coordenadas en "x" de los tags: \pos, \move, \moves3, \moves4 y \mover.

fx.move_y1, fx.move_y2, fx.move_y3, fx.move_y4: son la cuatro posibles coordenadas con respecto al eje "y" de las posición o de la trayectoria del movimiento de la Línea fx. Son las coordenadas en "y" de los tags: \pos, \move, \moves3, \moves4 y \mover.

Veamos algunos ejemplos de la anteriores ocho variables y de a poco las iremos aclarando:

fx.move_x1 = **fx.pos_x** – 45

fx.move_y1 = **fx.pos_y**

Los valores por default de **fx.move_x1** y **fx.move_y1** son, **fx.pos_x** y **fx.pos_y**, respectivamente. Del ejemplo de la anterior imagen, su resultado en la Línea fx sería el tag \pos, ya que solo hay una coordenada para ambos ejes.

Este ejemplo también retornaría un tag \pos:

fx.move_x1 = **fx.pos_x** ← por Default

fx.move_y1 = **fx.pos_y**

Para el caso de que ambas celdas de texto estén vacías, no retornaría ningún tag de posición ni de movimiento:

El valor por default de **fx.move_x2**, **fx.move_y2** y todas las superiores, es siempre la variable inmediatamente a cada una de ellas:

fx.move_x1 = **fx.pos_x** ← por Default

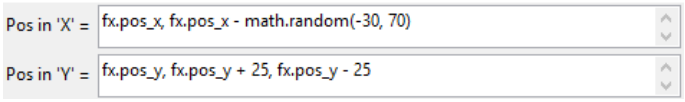
fx.move_x2 = **fx.move_x1** ← por Default

fx.move_y1 = **fx.pos_y**

fx.move_y2 = **fx.pos_y** + 20

El ejemplo anterior daría como resultado un tag \move, ya que habría dos coordenadas para cada eje.

Para el próximo ejemplo, veremos el caso cuando ambos tienen tres coordenadas o al menos una celda de texto tiene tres coordenadas:



fx.move_x1 = **fx.pos_x**

fx.move_x2 = **fx.pos_x – math.random(-30,70)**

fx.move_x3 = **fx.move_x2** ← por Default

fx.move_y1 = **fx.pos_y**

fx.move_y2 = **fx.pos_y** + 25

fx.move_y3 = **fx.pos_y** - 25

El tag que resultaría de este ejemplo sería un \moves3. Para el caso en el que al menos una de las dos celdas de texto tenga cuatro coordenadas, veamos este ejemplo:

fx.move_x1 = 50

fx.move_x2 = 120

fx.move_x3 = -80

fx.move_x4 = **syl.center**

fx.move_y1 = 200

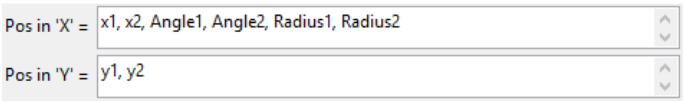
fx.move_y2 = 400

fx.move_y3 = 680

fx.move_y4 = **syl.middle**

Para el caso del ejemplo anterior obtendríamos como tag un \moves4, lo que quiere decir que el tag que resulta de la combinación de **Pos in 'X'** y **Pos in 'Y'** depende de cuál de estas dos celdas de texto tenga más coordenadas.

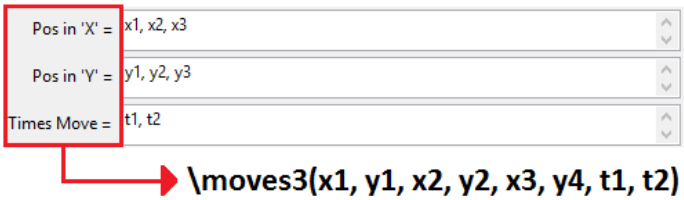
Y para el caso del tag \mover hacemos lo siguiente:



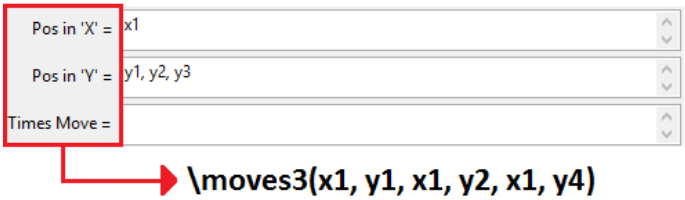
fx.movet_i, fx.movet_f: son los tiempo de inicio y final para los tags de movimiento: \move, \moves3, \moves4 y \mover:

"**t1**" representa el tiempo de inicio del movimiento y "**t2**" el tiempo final. Sus valores por default son 0 y **fx.dur**, respectivamente.

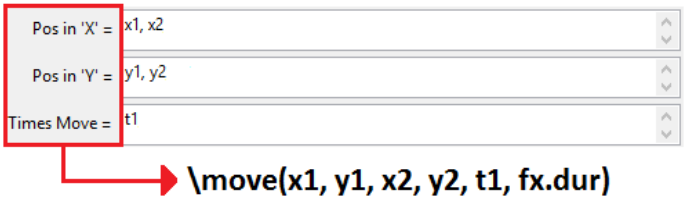
Veamos algunos ejemplos:



El anterior es un ejemplo sencillo de cómo obtener un \moves3 junto con los parámetros de tiempo incluidos.



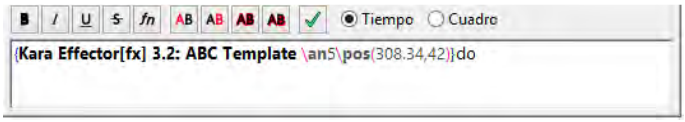
Este ejemplo ilustra cómo **fx.move_x2** y **fx.move_x3** son **fx.move_x1** por default, y al no haber parámetros de tiempo, entonces no salen en el tag.



Y en este otro ejemplo vemos que **fx.movet_f** equivale a **fx.dur** (duración de la Línea fx) por default.

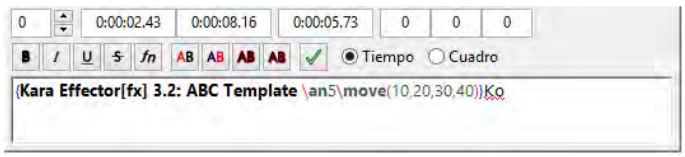
Todo tag de posición o de movimiento que hagamos utilizando estas tres celdas de texto:

Se verá reflejado en las Líneas fx:

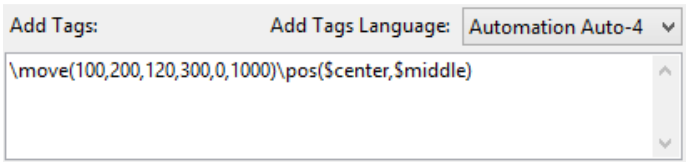


Pero no es la única forma de usar los tags de posición y movimientos, ya que desde **Add Tags** también lo podemos hacer, de hecho, se si hace desde **Add Tags**, entonces cualquier otro tag de posición o de movimiento es anulado:

El **Kara Effector** detecta de forma automática que hay un tag de movimiento (\move) en **Add Tags**, entonces anula el tag \pos que se iba a generar por las configuraciones de **Pos in 'X'** y **Pos in 'Y'**:



Si por algún motivo se coloca más de un tag de posición o de movimiento en **Add Tags**, el **Kara Effector** solo tomará en cuenta el último de ellos:

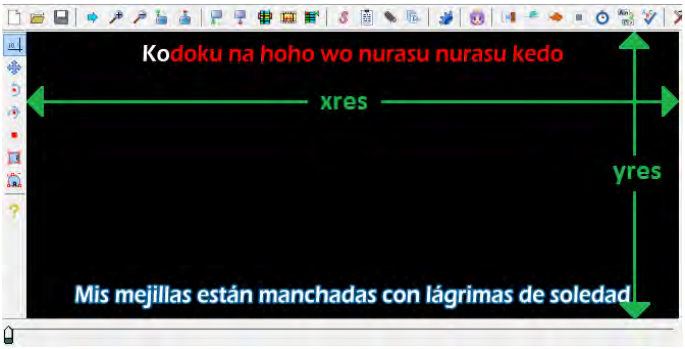


En este caso solo se toma en cuenta el tag \pos ya que es el último y el tag \move no saldrá en la Línea de fx.

Con los anteriores ejemplos concluye la explicación de la Librería "fx" pero hay otra serie de variables y valores propios del **Kara Effector** que veremos a continuación:

xres y **yres:** son las dimensiones medidas en pixeles del vídeo que se esté usando al momento de aplicar un Efecto y sus valores por default son 1280 y 720 p.

xres es el ancho del vídeo y **yres**, el alto:



ratio: es la Proporción que usa el **Kara Effector** para que un Efecto funcione exactamente igual sin depender de las dimensiones del vídeo. Al aplicar un Efecto con un vídeo abierto, el valor del **ratio** es: **xres**/1280, para el caso contrario su valor por default es 1.

frame_dur: es la duración medida en milisegundos de cada uno de los cuadros (frames) del vídeo que se esté usando al momento de aplicar un Efecto. Su valor por default es de 40 ms.

line.i: es el Contador numérico de cada una de las Líneas seleccionadas a las que le aplicaremos un Efecto. Es similar a la variable **syl.i**, salvo que esta última es el contador numérico de las Sílabas que contiene una Línea.

line.n: es la cantidad total de Líneas seleccionadas para aplicar un Efecto. Es similar a la variable **syl.n**, salvo que esta última es la cantidad total de Sílabas que contiene una Línea.

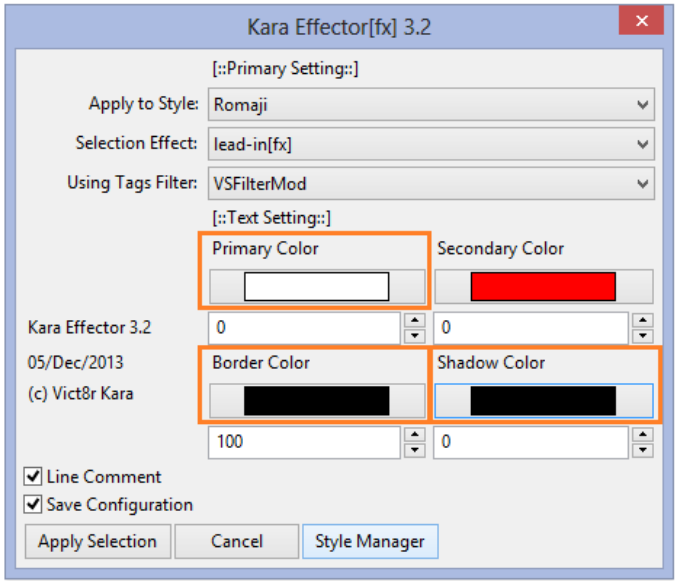
line.index: es el contador de todas las Líneas de Dialogo de un archivo .ass y también puede ser llamado como: **ii**

text.color: son los tres tag de color de la ventana de inicio del Kara Effector. En Lenguaje **Automation Auto-4** sería:

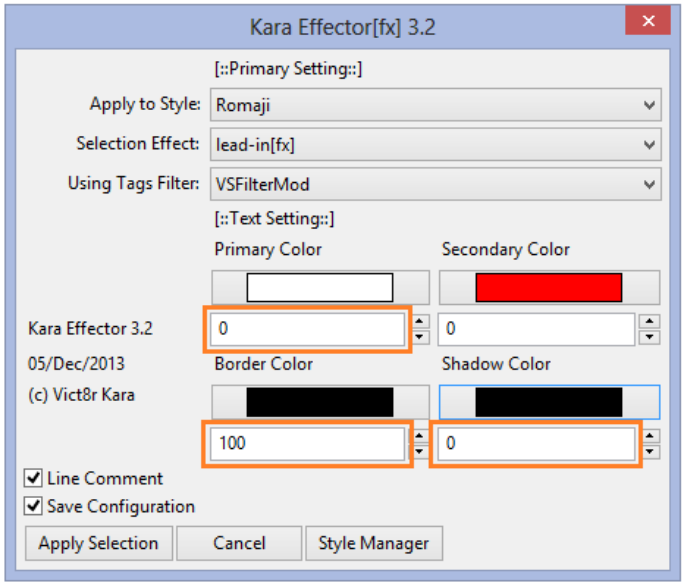
\1c**!text.color1!**\3c**!text.color3!**\4c**!text.color4!**

El formato en que se ven depende del filtro seleccionado.

Es decir que **text.color** equivale a estos tres colores:



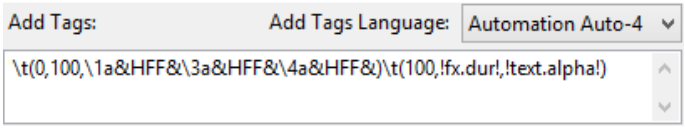
text.alpha: es similar a **text.color**, pero hace referencia a las transparencias de los tres colores anteriormente mencionados:



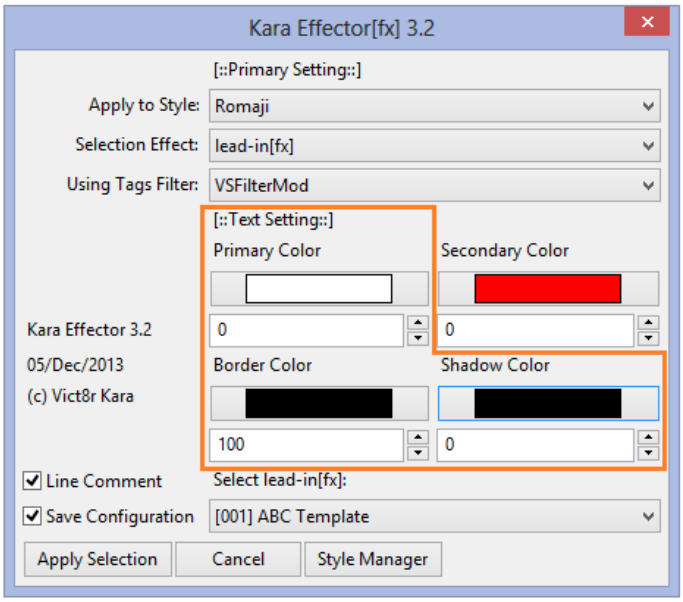
En Lenguaje **Automation Auto-4** sería:

\1a**!text.alpha1!**\3a**!text.alpha3!**\4a**!text.alpha4!**

Tanto **text.color** y **text.alpha** pueden servir para volver a las configuraciones de la ventana de inicio luego de alguna transformación de colores y/o de transparencias, ejemplo:

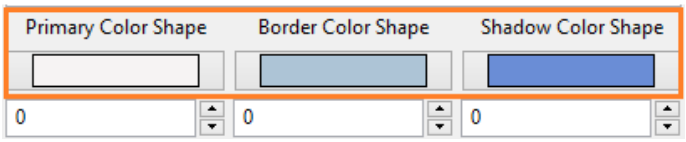


text.style: es la unión de **text.color** y **text.alpha**, o sea que hace referencia a estos seis valores de la ventana de inicio del **Kara Effector**:



text.alpha0: equivale a \alpha&HFF&.

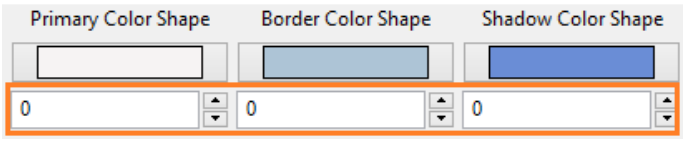
shape.color: son los tres tag de color de las Shapes:



En Lenguaje **Automation Auto-4** sería:

\1c**!shape.color1!**\3c**!shape.color3!**\4c**!shape.color4!**

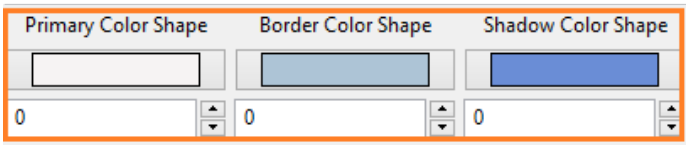
shape.alpha: son los tres tag de transparencia de las Shapes:



En Lenguaje **Automation Auto-4** sería:

\1a**!shape.alpha1!**\3a**!shape.alpha3!**\4a**!shape.alpha4!**

shape.style: es la unión de **shape.text** y **shape.alpha:**



shape.alpha0: equivale a \alpha&HFF&.

module: es la interpolación equidistante de los valores numéricos entre 0 y 1 con respecto al loop de un Efecto.

module = (**j** - 1)/(**maxj** - 1)

module1: es la interpolación equidistante de los valores numéricos entre 0 y 1 con respecto a la cantidad de sílabas de las Líneas seleccionadas para un Efecto.

module1 = (**syl.i** + **module** - 1)/**syl.n**

module2: es la interpolación equidistante de los valores numéricos entre 0 y 1 con respecto a la cantidad de Líneas seleccionadas para un Efecto.

module2 = (**line.i** + **module1** - 1)/**line.n**

Y hemos llegado al final de las variables de la Librería "fx" sin antes mencionarles que aún hay algunas funciones de la misma que he decido explicar en futuros Tomos para una mejor comprensión. De a poco vamos revelando los secretos del **Kara Effector** y espero que no se pierdan las próximas entregas.

A medida que avanzamos en los **Tomos**, profundizamos cada vez más en el mundo del **Kara Efector** y contamos con más herramientas para hacer nuestros propios Efectos. No olviden visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o si quieren comentar, exponer alguna duda o hacer alguna sugerencia.

Kara Effector 3.2:

En este **Tomo V** nos adentramos más en el mundo de las Librerías, tanto de **LUA** como las del **Kara Effector**. Algunas funciones solo serán mencionadas y otras más tendrán ejemplos para una mayor comprensión, de todas maneras aquellas que no queden claras, más adelante las veremos con mayor detenimiento.

No es importante memorizarlas todas, pero ayuda el hecho de que tengamos una idea de qué hace cada función y para qué nos podría servir una variable en especial.

Librería "math" [LUA]:

Es la librería de las funciones matemáticas de **LUA** y es de mucha utilidad. A continuación veremos las funciones y valores más usados, al menos para hacer Efectos. Omití algunas funciones y valores que consideré que no serían relevantes para usar en el **Kara Effector**, pero de todas maneras son fáciles de conseguir en la web.

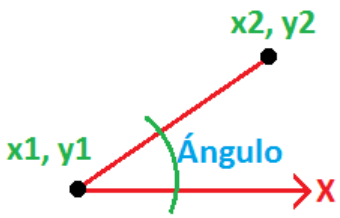
math.abs(x)	Retorna el Valor Absoluto de x
math.acos(x)	Retorna el Arco Coseno de x en un ángulo medido en radianes
math.asin(x)	Retorna el Arco Seno de x en un ángulo medido en radianes
math.atan(x)	Retorna el Arco Tangente de x en un ángulo medido en radianes
math.atan2(y, x)	Retorna el Arco Tangente de y/x en un ángulo medido en radianes
math.ceil(x)	Retorna el número entero mayor más cercano a x
math.cos(x)	Retorna el Coseno del ángulo x medido en radianes
math.cosh(x)	Retorna el Coseno Hiperbólico de x
math.deg(x)	Retorna el valor de x, convertido de radianes a sexagesimal. Ej: **math.deg(math.pi)** = 180

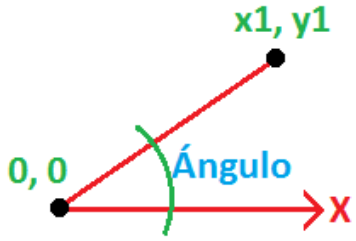
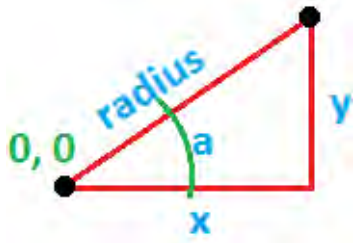
math.exp(x)	Retorna el valor de e^x. e = 2.718281…
math.floor(x)	Retorna el número Entero Menor más cercano a x
math.mod(x, y)	Retorna el Residuo de x/y, o sea que retorna el Modulo entre x e y
math.log(x)	Retorna el Logaritmo Natural (base e) de x
math.log10(x)	Retorna el Logaritmo Decimal (base 10) de x
math.max(x, …)	Retorna el número mayor de entre todos los parámetros
math.min(x, …)	Retorna el número menor de entre todos los parámetros
math.pi	Retorna el valor de pi. pi = 3.14159…
math.pow(x, y)	Retorna el valor de x^y. o sea x elevado a la y
math.rad(x)	Retorna el valor de x, convertido de sexagesimal a radianes
math.random(m, n)	Retorna un Número Aleatorio. En el caso de haber dos valores (m, n), retorna un número aleatorio entre esos dos valores. En el caso de un valor (m), retorna un número aleatorio entre 1 y ese valor. Y para el caso de no tener ningún valor, retorna un número decimal aleatorio entre 0 y 1
math.sin(x)	Retorna el Seno del ángulo x medido en radianes
math.sinh(x)	Retorna el Seno Hiperbólico de x
math.sqrt(x)	Retorna la Raíz Cuadrada de x. se puede remplazar con x^0.5
math.tan(x)	Retorna la Tangente del ángulo x medido en radianes
math.tanh(x)	Retorna la Tangente Hiperbólica de x

Librería "math" [KE]:

La anterior Librería es la que ya viene por default el lenguaje **LUA**, pero para hacer alguno de los Efectos parece que ésta se queda corta, es por ello que hubo la necesidad de "ampliar" la Librería "**math**". Las siguientes funciones y variables hacen parte de la Librería "**math**", pero son solo de uso exclusivo del **Kara Effector**.

También he omitido algunas funciones de la ampliación de la Librería "**math**" del **Kara Effector**, pero es porque no son para usar en los Efectos, sino que son para ayudar a otras funciones a hacer su trabajo.

math.R(m, n)	Cumple prácticamente la misma función que **math.random**, pero con la diferencia que genera un número aleatorio por cada vez que se aplica el Efecto. La forma abreviada es: **R(m, n)**
math.Rfake(m, n, i)	Parecida a **math.random**, pero el random se genera una única vez para todos los Efectos que se use, además consta de un tercer parámetro opcional, que hace que el random haga más operaciones antes de retornar un resultado. La forma abreviada es: **Rf(m,n,i)**
math.round(n, dec)	Redondea un número n según las cantidad de cifras decimales que indiquemos (dec). Si no está el parámetro "dec" entonces retorna el entero más cercano
math.distance (x1, y1, x2, y2)	Retorna la Distancia entre dos puntos P1=(x1, y1) y P2=(x2, y2). También se puede usar así: **math.distance(x1, y1)** En este caso retorna la Distancia entre P1=(0, 0) y P2(x1, y1)
math.angle (x1, y1, x2, y2)	Retorna el Ángulo que forma el segmento formado por los dos puntos, y el tramo positivo del eje "x":  También se puede usar así: **math.angle(x1, y1)** En este caso retorna el Ángulo que forma el segmento formado

	por los puntos P1 = (0, 0) y P=(x1, y1), y el tramo positivo del eje “x”: 
math.polar **(a, radius, Return)**	Retorna las coordenadas Polares que forma el ángulo (**a**) y el radio (**radius**). El parámetro **Return** puede ser “x” o “y” según la coordenada que queremos que retorne. Si **Return** no está, retorna ambas coordenadas. 
math.point **(n, x, y, xi, yi, xf, yf)**	Retorna un conjunto de Puntos. **n:** es el tamaño del conjunto **x:** es el rango horizontal **y:** es el rango vertical **xi, yi:** es el primer punto **xf, yf:** es el último punto
math.factk(n)	Retorna **n!**, es decir que retorna el factorial de **n**
math.bezier **(Return, ...)**	Retorna una **Curva Bezier** a partir de una serie de puntos o el trazado de una Shape. Es una función exclusiva para Efectos con Shapes.

Para ver la siguiente Librería es importante que antes tengamos claro el concepto de “**tabla**”. Una **tabla** es un conjunto de elementos, también se le conoce como “arreglo”.

El nombre de una **tabla** es asignado por uno mismo y sus elementos están dentro de llaves (“{ }”). Veamos un corto ejemplo:

Letras = {“A”, “B”, “C”, “D”}

En este ejemplo, la **tabla** se llama “Letras” y tiene cuatro elementos. Los elementos de una **tabla** están separados por coma (,) o por punto y coma (;), no importa cuál de las dos se use.

Para usar los elementos de la **tabla** “Letras”, se hace con el nombre de la **tabla** seguido del número de la posición del elemento entre corchetes:

Letras = {“A”, “B”, “C”, “D”}

Letras**[1]** = “A”

Letras**[2]** = “B”

Letras**[3]** = “C”

Letras**[4]** = “D”

Otra forma de definir la tabla de este ejemplo y aun así tener el mismo resultado sería:

Letras = {**[1]** = “A”, **[2]** = “B”, **[3]** = “C”, **[4]** = “D”}

Y por si fuera poco, todavía hay una tercera forma de definir la **tabla** “Letras”. Se define vacía o con algunos de sus elementos y luego se le “insertan” el resto:

Letras = { }

Letras**[1]** = “A”

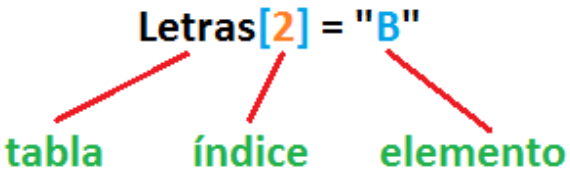
Letras**[2]** = “B”

Letras**[3]** = “C”

Letras**[4]** = “D”

Letras = {“A”, “B”, “C”, “D”}

El número de la posición que ocupa un elemento dentro de una **tabla**, se le llama **índice**:

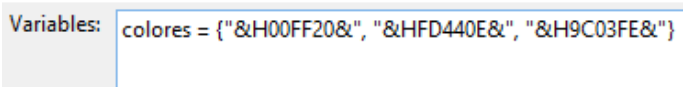


El tamaño de una **tabla** es la cantidad de elementos que tenga la misma. Para obtener el tamaño de una **tabla** se escribe el signo número seguido del nombre de la **tabla**:

#Letras = 4

Veamos un ejemplo en el **Kara Effector** con este tipo de **tabla**:

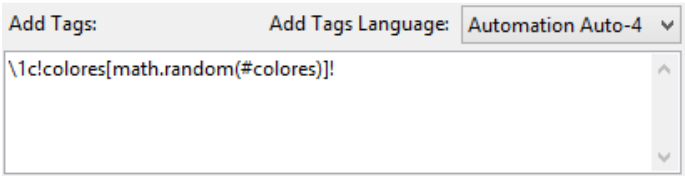
Paso 1. Definimos una **tabla** en la celda de texto Variables:



Le di por nombre "colores" y contiene tres colores: verde, azul y lila. De esta **tabla** se infiere que su tamaño es:

#colores = 3

Paso 2. Usar los elementos de la **tabla** en Add Tags:



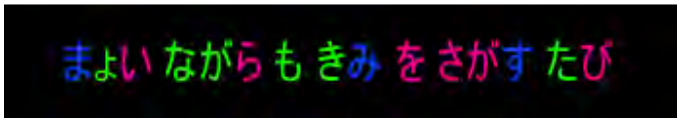
Para este ejemplo, usé los elementos de la **tabla** con un random. Una pequeña explicación de la expresión usada sería:

- \1c!**colores[math.random(#colores)]!**
- #colores = 3
- \1c!**colores[math.random(3)]!**
- \1c!**colores[math.random(1,3)]!**

Como es un random, los posibles resultados son:

- \1c!**colores[1]!** ← verde
- \1c!**colores[2]!** ← azul
- \1c!**colores[3]!** ← lila

Y en el vídeo veríamos algo más o menos como esto:



Este es solo un pequeño ejemplo de cómo usar los tres elementos de esta **tabla**, pero hay muchas formas más y en futuros ejemplos veremos esas formas de hacerlo. Si al momento de hacer este, no se ve en el vídeo como se ve en la imagen anterior, les recomiendo volver a descargar el **Kara Effector** en el **Blog Oficial**.

Los elementos de la **tabla** "colores" del ejemplo anterior también pueden ser declarados por medio de un nombre, lo que me da pie para mostrarles otra forma de declarar una **tabla**:

colores = {

verde = "&H00FF20&",

azul = "&HFD440D&",

lila = "&H9C03FE&"

}

Y la forma de "llamar" a los elementos declarados de esta forma es:

- **colores.verde**
- **colores.azul**
- **colores.lila**

O sea, el nombre de la **tabla** seguido de un punto (.) y luego el nombre que le hayamos dado a cada elemento.

El nombre que se le da a cada elemento de una **tabla**, en este caso la **tabla** "colores", también es asignado a gusto propio. Lo pude haber hecho de esta otra forma y sería lo mismo:

colores = {

v = "&H00FF20&",

a = "&HFD440D&",

l = "&H9C03FE&"

}

Veamos un ejemplo de cómo usar los valores de esta **tabla**: (en lenguaje **Automation Auto-4**)

\3c!**colores.v!**

O sea que el color del **Borde** (\3c) sería verde. Este mismo ejemplo pero en lenguaje **LUA** sería: (recordemos que hay dos formas de hacerlo)

- **string.format**("\\3c%s", colores.v)
- "\\3c"..colores.v

Lo anterior es solo una pequeña introducción al mundo de las **tablas**, pero el concepto es mucho más amplio de lo que hasta ahora hemos visto y de apoco les mostraré lo que aun reste por ver.

Ya con el concepto de **tabla** un poco más claro, veremos la siguiente Librería:

Librería "table" [LUA]:

Es la librería de las funciones de LUA que hacen referencia a las **tablas**. Esta Librería también tiene una ampliación en el **Kara Effector** y por ahora solo explicaré las funciones que son útiles para hacer Efectos:

table.concat (t, separador)	Retorna los elementos de una tabla, pero concatenados. Si el parámetro "**separador**" no está, simplemente concatena a los elementos. Ejemplo: T = {"a", 7, "FF"} **table.concat**(T) = "a7FF" Ejemplo con "**separador**": T = {"a", 7, "FF"} **table.concat**(T, " +-+ ") = "a +-+ 7 +-+ FF"
table.insert (t, i, elemento)	Inserta un elemento asignado a una tabla. Si el parámetro "**i**" no está, inserta al elemento al final de la tabla. Ejemplo: A = {2, 4, 6, 8, 10} **table.insert**(A, "&HFF&") Entonces: A = {2, 4, 6, 8, 10, "&HFF&"} Si para este mismo ejemplo ponemos el parámetro "**i**", sería: **table.insert**(A, 3, "&HFF&") Entonces: A = {2, 4, "&HFF&", 6, 8, 10} Nótese cómo el elemento se insertó en la posición 3.
table.maxn(t)	Cumple la misma función que **#t**, o sea que retorna el tamaño de la tabla **t**.
table.remove(t, i)	Remueve el elemento de la posición "**i**" de la tabla "**t**". Ej: B = {1, 3, 5, 7} **table.remove**(B, 2) Entonces: B = {1, 5, 7}
table.sort(t)	Organiza de menor a mayor los elementos de una tabla, sí y solo sí todos los elementos son números. Ejemplo: D = {5, 4, 7, 9, 2, 1} **table.sort**(D) Entonces: D = {1, 2, 4, 5, 7, 8} Los elementos de la tabla D ahora están organizados de menor a mayor.

Es momento de mencionar que las funciones de la Librería "**table**" se usan para crear nuevas funciones. Crear una nueva función es una herramienta para hacer Efectos que el **Aegisub**, el lenguaje **LUA** y **Kara Effector** nos ofrecen. Hacer funciones será la parte final de este curso, ya que cuando cada uno pueda hacer sus propias funciones, le resta muy poco por aprender acerca de Efectos Karaoke, por eso es importante ir viendo todos estos conceptos, para que cuando llegue el momento de usarlos a pleno, no quedaremos tan perdidos.

Librería "table" [KE]:

Es la ampliación de la Librería "**table**" con que cuenta el **Kara Effector** y algunas de sus funciones nos sirven para hacer Efectos y las restantes para crear nuevas funciones.

table.inside(t, e)	Retorna **true** (verdadero) en el caso que el elemento "**e**" esté en la tabla "**t**". En caso contrario retorna **false** (falso). Ejemplo: P = {"a", "b", "c"} **table.inside**(P, "c") = **true** **table.inside**(P, "e") = **false**
table.index(t, e)	Retorna el **índice** del elemento "**e**" en el caso de que dicho elemento pertenezca a la tabla "**t**". En caso contrario retorna de nuevo al elemento. Ejemplo: R = {13, 42, 63, 34, 25} **table.index**(R, 42) = 2 Retorna 2, porque R[2] = 42 **table.index**(R, "AA") = "AA" Retorna "AA", porque: **table.inside**(R, "AA") = **false** O sea que "AA" no está en R

table.compare (t1, t2)	Retorna **true** (verdadero) en el caso de que la tabla **t1** sea igual que la tabla **t2**, de otro modo retorna **false** (falso). Ejemplo: A = {1, 2, 3} B = {1, 2, 3, 4} **table.compare**(A, B) = **false** C = {7, "a"} D = {7, "a"} **table.compare**(C, D) = **true**
table.disorder(t)	**t** puede ser una tabla o un número entero positivo mayor o igual que 2. Para el caso en que **t** sea una tabla, retorna a la misma tabla con sus elementos en desorden. Los elementos se desordenan de forma aleatoria (random). Ej: H = {"a", "b", "c", 3, 9} G = **table.disorder**(H) Un posible resultado sería: G = {"b" , 9, "a", 3, "c"} Si **t** es un entero positivo, entonces la función crea una tabla de números consecutivos desde 1 hasta **t**, para luego desordenar esos números. Ej: G = **table.disorder**(6) Un posible resultado sería: G = {3, 5, 6, 2, 1, 4} El número total de resultados es **#t!**, o sea el factorial del tamaño de la tabla. Para este último ejemplo sería: 6! = 720 posibilidades

He decidido hacer un pequeño paréntesis acá para aclarar algunos conceptos que nos ayudarán a entender mucho de lo que se viene. El lenguaje **Automation Auto-4** es una modificación del lenguaje **LUA** para que reconozca las variables **Dólar** (**$**) y las operaciones hechas dentro de los signos de admiración (**!!**). En pocas palabras, el lenguaje **Automation Auto-4** está basado en el lenguaje **LUA**.

Otro de los conceptos que quería mencionar es uno que en ocasiones nos encontramos en los parámetros que requiere una función, y son tres puntos seguidos (…). Los tres puntos seguidos pueden ser una variable, una tabla, un elemento o un listado de elementos y/o una combinación de éstos. Estos tres puntos se colocan para indicar que podemos poner cuantas cosas queramos.

table.concat1(t, …)	Retorna la tabla **t** con todos sus Elementos concatenados a (…). Ejemplo 1: (tabla y un elemento) A = {"a", "b", "c"} B = **table.concat1**(A, 9) B = {"9a", "9b", "9c"} Ejemplo 2: (tabla y más de un elemento) F = {1, 2, 3} G = **table.concat1**(F, a, b) G = {a1, b1, a2, b2, c1, c2} Ejemplo 3: (tabla con tabla) M = {4, 6, 8} N = {f, g} O = **table.concat1**(M, N) O = {f4, g4, f6, g6, f8, g8} En estos ejemplos vemos cómo los tres puntos pueden ser un solo elemento o varios, también pueden ser una tabla
table.concat2(t, …)	Es similar a **table.concat1** y la diferencia se notará en los ejemplos. Ejemplo 1: (tabla y un elemento) A = {"a", "b", "c"} B = **table.concat1**(A, 9) B = {"9a", "9b", "9c"} Ejemplo 2: (tabla y más de un elemento) F = {1, 2, 3} G = **table.concat1**(F, a, b) G = {a1b1, a2b2, c1c2} Ejemplo 3: (tabla con tabla) M = {4, 6, 8} N = {f, g, h} O = **table.concat1**(M, N) O = {f4g4h4, f6g6h6, f8g8h8}
table.replay(n, …)	Retorna una tabla con **n** veces repetidas a (…). Ejemplo 1: A = **table.replay**(4, "a") A = {"a", "a", "a", "a"} Ejemplo 2: B = **table.replay**(3, f, g, h) B = {f, g, h, f, g, h, f, g, h} Ejemplo 3: C = {7, 8, 9} D = **table.replay**(2, C) D = {7, 8, 9, 7, 8, 9} Se nota cómo se repiten **n** veces el o los elementos ingresados

table.count(t, e)	Retorna el número de veces en el que el elemento **e** aparece en la tabla **t**. en el caso en que el elemento **e** no esté en **t**, retorna 0. Ejemplo: A = {a, b, a, 7, a, 8, 9, a} **table.count**(A, a) = 4 **table.count**(A, c) = 0 **table.count**(A, b) = 1
table.pos(t, e)	Retorna una tabla con el o los **índices** del elemento **e** en la tabla **t**. En el caso de que el elemento **e** no esté en **t**, retorna una tabla vacía. Ejemplo: A = {a, b, a, 7, a, 8, 9, a} B = **table.count**(A, a) B = {1, 3, 5, 8} C = **table.count**(A, c) C = { } D = **table.count**(A, b) D = {2}
table.retire(t, …)	Retorna la tabla **t**, pero retira a (…) de la tabla en el caso en que están en ella. Ejemplo 1: A = {a, b, a, 7, a, 8, 9, a} B = **table.retire**(A, a) B = {b, 7, 8, 9} Ejemplo 2: C = **table.retire**(A, a, 7, 8) C = {b, 9} Ejemplo 3: D = {7, 8, 9} E = **table.retire**(A, D) E = {a, b, a, a, a}
table.inserttable (t1, t2, i)	Inserta los elementos de la tabla **t2** en la tabla **t1** a partir del **índice i**, o en el caso de no estar el parámetro **i**, se insertan al final de la tabla **t1**. Ejemplo 1: A = {6, 7, 8} B = {a, b, c, d} C = **table.inserttable**(A, B, 3) C = {6, 7, a, b, c, d, 9} Los elementos de la tabla B se insertaron en la tabla A, a partir del **índice** 3 (la tercera posición). Ejemplo 2: D = {5, 4, 1} E = {f, g} F = **table.inserttable**(D, E) F = {5, 4, 1, f, g}

table.reverse(t)	Retorna a la tabla t, pero con el **índice** invertido. Ejemplo 1: A = {3, 4, 5, 6, 7} B = **table.reverse**(A) B = {7, 6, 5, 4, 3} Ejemplo 2: C = {f, 5, 3, a, m, 2, 9} D = **table.reverse**(C) D = {9, 2, m, a, 3, 5, f}
table.cyclic(t)	Genera un "**ciclo**" con todos los elementos de la tabla t. Ejemplo 1: A = {2, 4, 6, 8} B = **table.cyclic**(A) B = {2, 4, 6, 8, 6, 4, 2} Ejemplo 2: C = {a, 4, 2, d, 5, b} D = **table.cyclic**(C) D = {a, 4, 2, d, 5, b, 5, d, 2, 4, a}
table.op(t, mode)	Genera operaciones con los elementos de la tabla **t**. Dichas operaciones dependen del modo "**mode**". Esta función es exclusiva para las tablas con elementos numéricos. Ejemplo 1: **mode** = "sum" Suma los elementos de la tabla A = {9, 1, 16, 4} **table.op**(A, "sum") = 9 + 1 + 16 + 4 = 30 Ejemplo 2: **mode** = "concat" Une a los elementos de la tabla **table.op**(A, "concat") = 14916 Ejemplo 3: **mode** = "average" Obtiene un promedio de la tabla **table.op**(A, "average") = (9 + 1 + 16 + 4) / #A = 30 / 4 = 7.5 Ejemplo 4: **mode** = "min" Da al menor de los elementos **table.op**(A, "min") = 1 Ejemplo 5: **mode** = "max" Da al mayor de los elementos **table.op**(A, "max") = 16 Ejemplo 6: **mode** = "add" Adiciona un tercer parámetro a cada uno de los elementos B = **table.op**(A, "add" -2) B = {9 - 2, 1 - 2, 16 - 2, 4 - 2} B = {7, -1, 14, 2}

Omití algunas funciones de la ampliación de la Librería "**table**" por un par de motivos: para ver algunos conceptos previos y para poder darle más espacio y mayor número de ejemplos para total comprensión.

Este ha sido un **Tomo** cargado con mucha teoría, teoría que no necesariamente deben memorizar ni dominar toda al 100%, pero es mejor tener a la mano el medio para consultar alguna duda, que necesitar un concepto y no saber en dónde buscar.

Lo que con certeza les puedo asegurar es que el tener claro cuáles son y para qué sirve cada una de las variables y funciones de las Librerías vistas en esta series de tomos, les dará las herramientas necesarias para crear sus propias funciones, en donde las posibilidades son infinitas y los Efectos que se pueden lograr con cada una de ellas no tienen comparación.

En el **Kara Effector** la teoría es tan importante como la práctica, ambas deben ir de la mano, ya que sin una de ellas la otra no sería suficiente. De a poco iremos haciendo un equilibrio entre ellas y por eso en los próximos tomos irá aumentando el número de ejemplos para ponerlos en práctica y aumentar nuestras destrezas.

Espero que en este tramo del camino ya hayan podido ver algunos de los Efectos que por default vienen en el **Kara Effector** y hayan podido entender un poco mejor en qué consiste cada uno de ellos, y eso gracias a los conceptos, variables y funciones vistas. No olviden visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia.

Kara Effector 3.2: Effector Book Vol. I [Tomo 5.1]

Kara Effector 3.2:

Este **Tomo 5.1** es la expansión del **Tomo V**, ya que hubo un par de funciones de la Librería "**table**" del **Kara Effector** que no quedaron explicadas en él. También debo agregar que desde la fecha de publicación del **Tomo V**, hasta la actualidad (**04 de Junio de 2014**), la Librería se ha venido ampliando con otras funciones que también explicaremos en este **Tomo**.

Para garantizar que todos los ejemplos planteados en este **Tomo** les funcionen tal cual están, es recomendable usar el **Kara Effector 3.2.7** o superior, ya que la mayoría de las funciones nuevas fueron diseñadas a partir de dicha versión. Para las versiones anteriores simplemente les arrojará un error.

Librería "table" [KE]:

table.make(Objet, Size, limit_i, limit_f, ...): esta función crea una **tabla** de un tamaño determinado "**Size**" y que contiene elementos equidistantes entre sí, del tipo "**Objet**".

El parámetro **Objet** indica el tipo de elementos que tendrá la **tabla** retornada, y tiene cuatro opciones:

- "**number**"
- "**color**"
- "**alpha**"
- **Tags string**

Cada uno de ellos crea elementos distintos en nuestra **tabla** y más adelante veremos ejemplos de cada.

El parámetro **Size** indica el tamaño de la **tabla**, es decir, la cantidad de elementos que contendrá. **Size** debe ser un número entero mayor a cero.

Los parámetros **limit_i** y **limit_f** indican los límites inferior y superior que tendrán los elementos de la **tabla** creada. El primer elemento de la tabla sería **limit_i** y el último vendría a ser **limit_f**, y en medio de ellos todos los elementos equidistantes dependiendo del parámetro Size.

Los tres puntos seguidos (**…**) hacen referencia a uno o más tags que queramos añadirle a cada uno de los elementos que harán parte de la **tabla** retornada. Este parámetro es opcional.

- **Ejemplo 1. Objet** = "**number**":

mi_tabla = **table.make("number", 8, 20, 55)**

mi_tabla = {
[1] = 20, ← **limit_i**
[2] = 25,
[3] = 30,
[4] = 35,
[5] = 40,
[6] = 45,
[7] = 50,
[8] = 55 ← **limit_f**
}

Del anterior ejemplo vemos que la anterior **tabla** contiene 8 elementos (**#mi_tabla** = 8), y cada uno de ellos es un número equidistante entre 20 (**limit_i**) y 55 (**limit_f**).

- **Ejemplo 2. Añadir un tag**:

mi_tabla = **table.make("number", 8, 20, 55, "\\fr")**

mi_tabla = {
[1] = **\fr**20, ← **limit_i**
[2] = **\fr**25,
[3] = **\fr**30,
[4] = **\fr**35,
[5] = **\fr**40,
[6] = **\fr**45,
[7] = **\fr**50,
[8] = **\fr**55 ← **limit_f**
}

Ahora en este ejemplo, como en el quinto parametro colocamos "**\\fr**", entonces la función añade este tag al inicio de cada elemento de la **tabla**.

- **Ejemplo 3. Añadir más de un tag**:
  Para añadir dos o más tags a los elementos hay dos diferentes métodos. **Método 1**:

mi_tabla = **table.make("number", 8, 20, 55, "\\fr","\\b")**

mi_tabla = {
[1] = **\fr**20 **\b**20, ← **limit_i**
[2] = **\fr**25 **\b**25,
[3] = **\fr**30 **\b**30,
[4] = **\fr**35 **\b**35,
[5] = **\fr**40 **\b**40,
[6] = **\fr**45 **\b**45,
[7] = **\fr**50 **\b**50,
[8] = **\fr**55 **\b**55 ← **limit_f**
}

Cada elemento de la **tabla** se multiplica para corresponder a cada uno de los tags ingresados en el quinto parámetro de la función ("**\\fr**" y "**\\b**").

- **Ejemplo 4. Añadir más de un tag**:
  Método 2:

Tags = {"**\\frx**", "**\\fry**", "**\\frz**", "**\\fscy**"}
mi_tabla = **table.make("number", 8, 20, 55, Tags)**

mi_tabla = {
[1] = **\frx**20 **\fry**20 **\frz**20 **\fscy**20,
[2] = **\frx**25 **\fry**25 **\frz**25 **\fscy**25,
[3] = **\frx**30 **\fry**30 **\frz**30 **\fscy**30,
[4] = **\frx**35 **\fry**35 **\frz**35 **\fscy**35,
[5] = **\frx**40 **\fry**40 **\frz**40 **\fscy**40,
[6] = **\frx**45 **\fry**45 **\frz**45 **\fscy**45,
[7] = **\frx**50 **\fry**50 **\frz**50 **\fscy**50,
[8] = **\frx**55 **\fry**55 **\frz**55 **\fscy**55
}

En resumen, cuando **Objet** = "**number**", los parámetros **limit_i** y **limit_f** deben ser números también, de modo que la función cree a cada uno de sus elementos de forma equidistante entre esos dos valores.

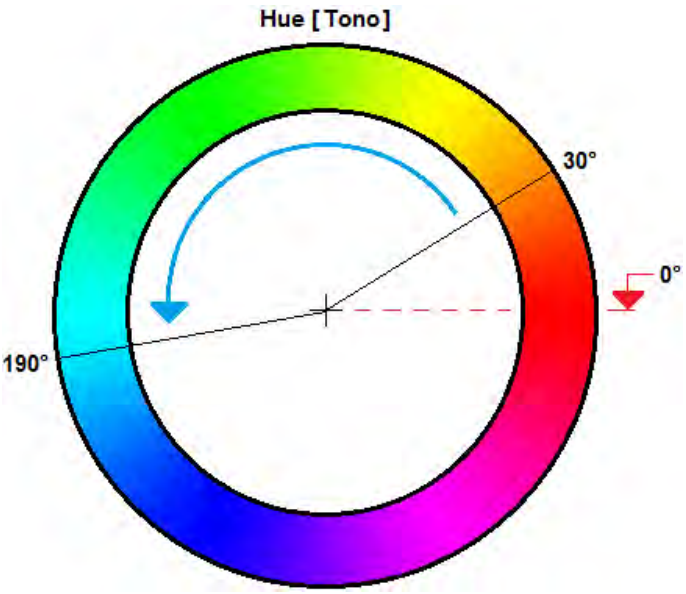
Para los siguientes ejemplos, usaremos la siguiente opción del parámetro **Objet**: "**color**"

- **Ejemplo 5. Objet** = "**color**":
  Límites Numéricos:

mi_tabla = **table.make("color", 15, 30, 190)**

- **Objet** = "**color**"
- **Size** = 15 (tamaño de la **tabla**)
- **limit_i** = 30 (límite inferior)
- **limit_f** = 190 (límite superior)

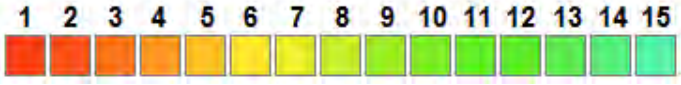
Los valores 30 y 190 hacen referencia a un ángulo entre 0° y 360° del Círculo Cromático de la **Teoría del Color HSV** (**Hue**, **Saturation**, **Value**):



Entonces la función creará una **tabla** de 15 colores entre los 30° y los 190°, equidistantes entre sí:

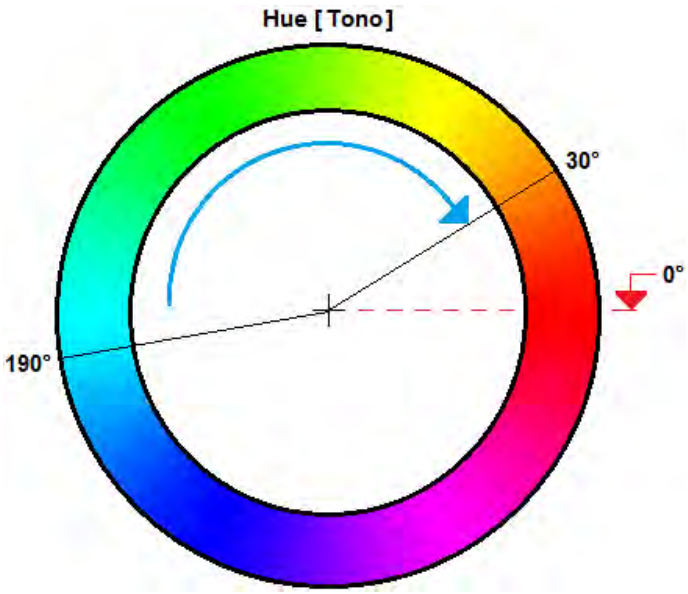
```
mi_tabla = {
                [1]  = "&H007FFF&",
                [2]  = "&H00B0FF&",
                [3]  = "&H00E0FF&",
                [4]  = "&H00FFEC&",
                [5]  = "&H00FFBC&",
                [6]  = "&H00FF8B&",
                [7]  = "&H00FF5B&",
                [8]  = "&H00FF2A&",
                [9]  = "&H06FF00&",
                [10] = "&H36FF00&",
                [11] = "&H67FF00&",
                [12] = "&H97FF00&",
                [13] = "&HC8FF00&",
                [14] = "&HF8FF00&",
                [15] = "&HFFD400&"
        }
```

Se entiende un poco más si se ven los tonos de los 15 colores de la **tabla** generada. Noten que equivalen a los colores entre 30° y 190°:



No necesariamente el valor de **limit_i** debe ser menor que el de **limit_f**. Ejemplo:

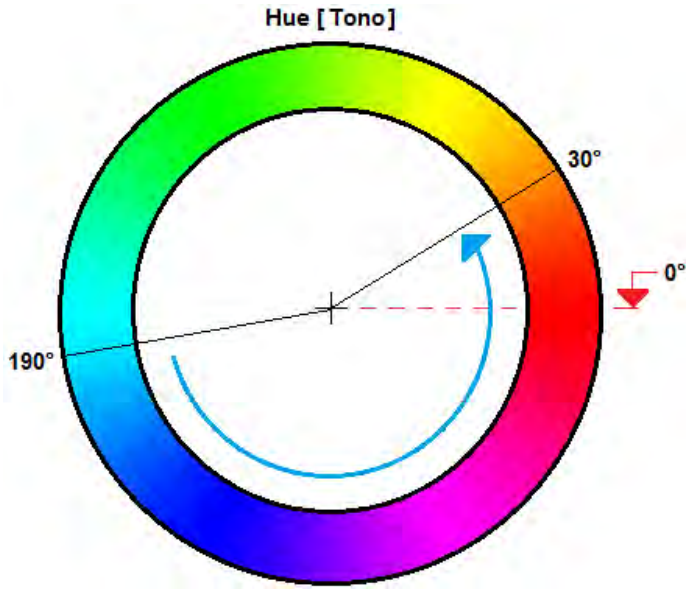
mi_tabla = **table.make(“color”, 20, 190, 30)**



Entonces, lo que hace la función es crear una **tabla** de 20 colores desde los 190°, hacia atrás, hasta los 30°. La función hace el recorrido de las tonalidades en el Círculo Cromático en sentido horario.

Si alguno de los límites de la función excede a los 360°, dicho valor dará la vuelta en el Círculo, literalmente. Por cada giro se restan 360°. Ejemplo:

mi_tabla = **table.make(“color”, 10, 190, 390)**



La función tomará a 10 colores desde los 190° hasta los 30° en sentido anti horario: 30° = 390° - 360°

Para agregar tags a los colores de la tabla, hacemos como en los primeros ejemplos:

mi_tabla = **table.make("color", 7, 60, 150, "\\1c")**

```
mi_tabla = {
                [1] = \1c&H00FFFF&,
                [2] = \1c&H00FFBF&,
                [3] = \1c&H00FF7F&,
                [4] = \1c&H00FF3F&,
                [5] = \1c&H00FF00&,
                [6] = \1c&H3FFF00&,
                [7] = \1c&H7FFF00&
        }
```

- **Ejemplo 6.** Cuando **limit_i** y **limit_f** son dos **strings** de colores:

tbl = **table.make("color",6, "&HFFFFFF&", "&H000000&")**

```
tbl = {
                [1] = "&HFFFFFF&",
                [2] = "&HCCCCCC&",
                [3] = "&H999999&",
                [4] = "&H666666&",
                [5] = "&H333333&",
                [6] = "&H000000&"
        }
```

Hagamos un ejemplo práctico en el **Kara Effector**, para el cuál usaremos un **Template Type: Char** y la plantilla **ABC Template leadin**:

Variables:

```
colores = table.make( "color", char.n,
          "&H00F0FF&", "&H000DFF&", "\\1c" )
```

Como podemos ver, el tamaño de la **tabla** es **char.n**, que es el número total de caracteres (letras, números, signos) de la línea karaoke.

Para asignar los colores a cada carácter, hacemos:

Add Tags: Add Tags Language: Lua

```
colores[char.i]
```

De esta manera, asignamos al primer carácter, el primer color; al segundo le asignamos el segundo color y así con todos los caracteres de la línea karaoke:

- **Ejemplo 7.** Cuando **limit_i** y **limit_f** se "fusionan" es un solo parámetro para ingresar más de dos colores. Este nuevo parámetro será ingresado en forma de **tabla**:

Variables:

```
Hues = {"&H00F0FF&", "&H000DFF&", "&H18E419&"} ;
colores = table.make( "color", char.n, Hues, "\\1c" )
```

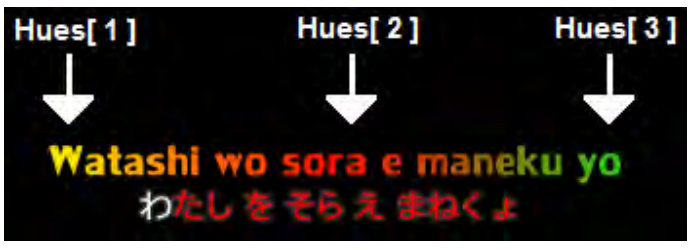
En "**Variables**" declaramos una **tabla** (**Hues**, en la imagen anterior) con las tonalidades de referencia (amarillo, rojo y verde, en este caso) para que la función **table.make** genere los colores equidistantes entre ellas. Nótese que resalto el punto y coma (;) con el cual se deben separa las variables.

Obtenemos los colores de la **tabla** "**colores**" de la misma forma que en el ejemplo anterior:

Add Tags: Add Tags Language: Lua

```
colores[char.i]
```

Y vemos los resultados:



Con el método usado en el anterior ejemplo, se pueden hacer **Gradientes** (degradaciones) de tres o más colores. Todo depende de los resultados deseados y del efecto que queremos hacer. Les recomiendo que practiquen con otros tres colores distintos y luego usando más colores.

- **Ejemplo 8. Objet** = "**alpha**" y límites numéricos. Los límites numéricos ya no están entre 0 y 360, como en el caso de **Objet** = "**color**"; en este caso los valores de los límites van desde **0** a **255**:

mi_tabla = **table.make("alpha", 10, 45, 86)**

```
mi_tabla = {
            [1]  = "&H2D&",  ← 45 en Hexadecimal
            [2]  = "&H31&",
            [3]  = "&H36&",
            [4]  = "&H3A&",
            [5]  = "&H3F&",
            [6]  = "&H43&",
            [7]  = "&H48&",
            [8]  = "&H4C&",
            [9]  = "&H51&",
            [10] = "&H56&"  ← 86 en Hexadecimal
    }
```

Para asignarle los tags, hacemos lo mismo que en los ejemplos de colores. Ejemplo:

mi_tabla = **table.make("alpha",8, 50, 200, "\\1a", "\\3a")**

```
mi_tabla = {
            [1] = "\\1a&H32& \\3a&H32&",
            [2] = "\\1a&H47& \\3a&H47&",
            [3] = "\\1a&H5C& \\3a&H5C&",
            [4] = "\\1a&H72& \\3a&H72&",
            [5] = "\\1a&H87& \\3a&H87&",
            [6] = "\\1a&H9D& \\3a&H9D&",
            [7] = "\\1a&HB2& \\3a&HB2&",
            [8] = "\\1a&HC8& \\3a&HC8&"
    }
```

- **Ejemplo 9**. Usando los límites como **strings alpha**:

mi_tabla = **table.make("alpha", 6, "&H4E&", "&HAD&")**

```
mi_tabla = {
            [1] = "&H4E&",
            [2] = "&H61&",
            [3] = "&H74&",
            [4] = "&H87&",
            [5] = "&H9A&",
            [6] = "&HAD&"
    }
```

- **Ejemplo 10**. "fusionar" los parámetros **limit_i** y **limit_f** para poder ingresar una tabla con tres o más **strings alpha**:

Alphas = {"&HFF", "&HA2&", "&H16&", "&HDE&"};

mi_tabla = **table.make("alpha", 12, Alphas)**

- **Ejemplo 11. Objet** = **Tags strings:**

mi_tabla = **table.make("\\blur", 5)**

```
mi_tabla = {
            [1] = "\\blur5",
            [2] = "\\blur5",
            [3] = "\\blur5",
            [4] = "\\blur5",
            [5] = "\\blur5"
    }
```

Es decir que el valor de Size (5 para el anterior ejemplo), no solo indica el tamaño de la **tabla**, sino que también se concatena (se une) con el tag ingresado ("**\\blur**").

- **Ejemplo 12:**

mi_tabla = **table.make("\\fscx", 6, 80, 120)**

```
mi_tabla = {
            [1] = "\\fscx80",
            [2] = "\\fscx88",
            [3] = "\\fscx96",
            [4] = "\\fscx104",
            [5] = "\\fscx112",
            [6] = "\\fscx120"
    }
```

Los valores numéricos equidistantes entre **limit_i** y **limit_f** (80 y 120) se concatenan al tag "**\\fscx**".

---

table.rmake(Objet, Size, limit_i, limit_f, ...): esta función es similar a **table.make**, pero con la diferencia que los elementos de la **tabla** ya no están ni organizados ni equidistantes entre sí, sino que crea los elementos de forma totalmente aleatoria, teniendo en cuenta los limites inferior y superior de los parámetros **limit_i** y **limit_f**.

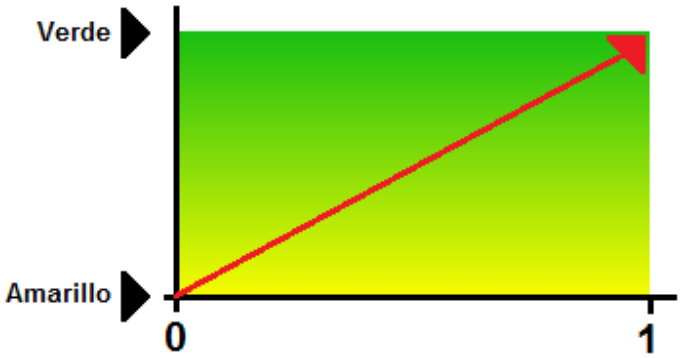
table.gradient(color1, color2, algorithm): crea una tabla de colores (o de alphas), correspondientes a un Gradiente entre los parámetros **color1** y **color2**. El tamaño de la **tabla** generada dependerá del **Template Type**, de tal manera que a cada objeto karaoke le corresponda un único elemento. Si por ejemplo el **Template Type** es **Translation Word**, entonces el tamaño de la **tabla** que se generará será **word.n**, de modo que por cada palabra de la línea karaoke, haya un elemento en la **tabla** que le corresponde.

El parámetro **algorithm** es opcional y determina el modo de transición desde **color1** hasta **color2**.

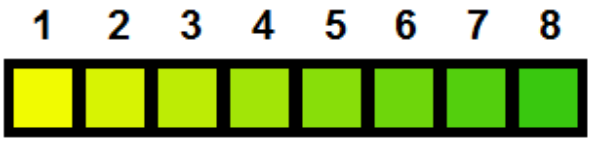
- **Ejemplo 1. Template Type: Word**
  word.n = 8 (suponiendo que una línea tiene 8 palabras. Este número es solo para saber cómo se determina el tamaño de la **tabla**)

mi_tabla = **table.gradient("&H00FFFF&", "&H12BE12&")**

Como hemos omitido al tercer parámetro de la función (**algorithm**), entonces la transición entre el amarillo y el verde se hará de forma lineal:



Los 8 elementos de la tabla generada serán:

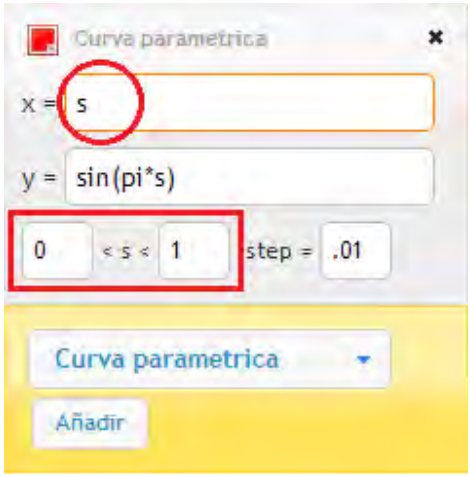


De este modo, la degradación se hace normalmente, ya que omitimos un algoritmo que modifique la transición entre los dos colores ingresados.

- **Ejemplo 2. Template Type: Syl**
  syl.n = 12

Uno de los métodos simples para crear un **algoritmo** para esta función, es usar un graficador de funciones online; el que generalmente usamos es "**fooplot**" en la opción de **Curva Paramétrica**. Los pasos son:

- En [**x =**] ponemos la letra (s)
- El dominio se pone desde 0 a 1

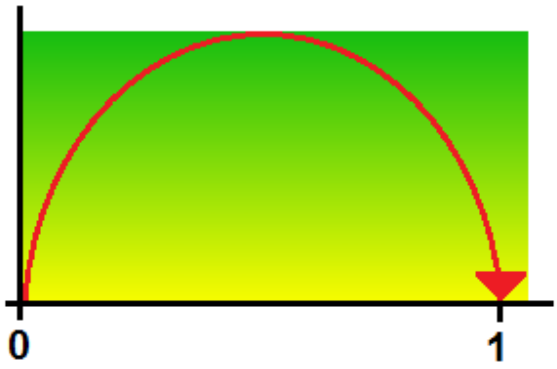


- En [**y =**] declaramos el algoritmo en función de la letra (s), como en la imagen anterior, y en la gráfica veremos esto:

Hecho esto, copiamos el algoritmo hecho en [y] y lo pegamos en el tercer parámetro de la función, entre comillas simples o dobles, y añadiendo el signo de porcentaje (%) antes de cada letra (s) que haya en el algoritmo:

sin(pi*s) => "sin(pi*%s)"

mi_tabla = **table.gradient(“&H00FFFF&”, “&H12BE12&”, “sin(pi*%s)”)**



Entonces se hará un Gradiente desde el amarillo hasta el verde y luego regresará al amarillo:

- **Ejemplo 3:**

```
Variables:
mi_tabla =
tabla.gradient( "&H00F0FF&", "&H000DFF&", "%s^3" )
```

Vemos la gráfica de (s^3) desde 0 a 1:

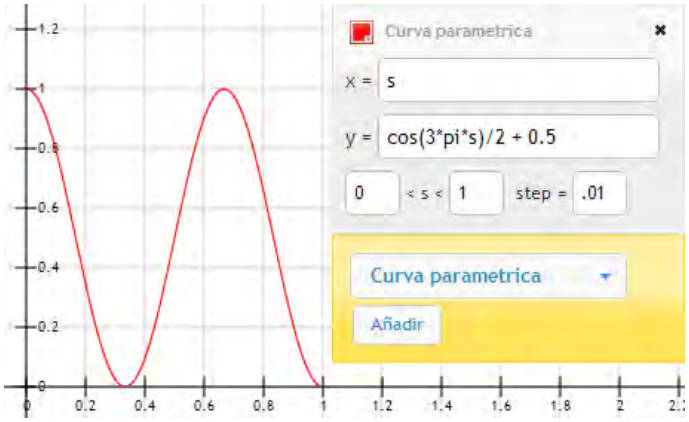
Con un **Template Type: Char**, llamamos a los colores que la **tabla** creó de la siguiente manera:

```
Add Tags:  Add Tags Language: Lua
"\\1c" .. mi_tabla[ char.i ]
```

El Gradiente creado por el algoritmo “**%s^3**” se verá en el vídeo más o menos como en la siguiente gráfica:

Notamos que el rojo empieza casi al final de la línea y ya no es simétrico como en el **Gradiente lineal**.

- **Ejemplo 3:**
  Algoritmo = “**cos(3*pi*%s)/2 + 0.5**”



Del algoritmo dependerá la transición entre el color 1 y el color2, así que las posibilidades son infinitas, hay tantas transiciones como algoritmos puedan crear.

Para hacer Gradientes de alphas con esta función, solo se deben colocar los **strings alphas** en ambos parámetros de la misma, ejemplo:

mi_tabla = **table.gradient(“&HFF&”, “&HD8&”)**

---

table.gradient2(...): esta función crea una **tabla** con los colores (o alphas) de un **Gradiente Lineal** entre todos los elementos ingresados (dos o más) en la función.

Al igual que la anterior función, el tamaño de la **tabla** es dependiente del **Template Type**.

- **Ejemplo 1:**

mi_tabla = **table.gradient2(“&H00FFFF&”, “&H12BE12&”)**

Entonces la función crea un **Gradiente Lineal** entre los dos colores ingresados.

- **Ejemplo 2:**

alphas = { “&H00&”, “&HAA”, “&H5D&”, “&HFF&” };

mi_tabla = **table.gradient2(alphas)**

En resumen, los tres puntos seguidos (…) en la función, hacen referencia a los colores o alphas a ingresar, o a una **tabla** de colores o de alphas, de la cual necesitemos crear una **tabla** del Gradiente Lineal generados por ellos.

Si le ingresamos 3, 4, 7, 10 o la cantidad de colores que deseemos, la función hará una **tabla**, en donde el tamaño dependerá del **Template Type**, con el **Gradiente Lineal** de todos los colores ingresados. Aplica de la misma manera para los alphas.

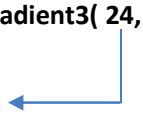
---

table.gradient3(Size_table, ...): esta función hace exactamente los mismo que la función anterior, pero el tamaño de la **tabla** generada ya no dependerá del tipo de plantilla (**Template Type**), sino que dependerá del parámetro **Size_table**.

- **Ejemplo 1:**

colores = {“&H00FF00&”, “&HFFAA00&”, “&H00DDFF&”};

mi_tabla = **table.gradient3(24, colores)**



Tamaño de la **tabla**

Recordemos que este **Tomo 5.1** es una extensión del **Tomo V**, en el que vienen explicadas cinco funciones de la librería **table [KE]**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoDCE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:

En el **Tomo VI** veremos un poco más de los lenguajes **LUA** y **Automation Auto-4**, ya que el saber más de ellos nos ayudará a comprender la estructura de cualquier Efecto hecho en el **Kara Effector** o incluso, cualquier otro hecho en **Aegisub**.

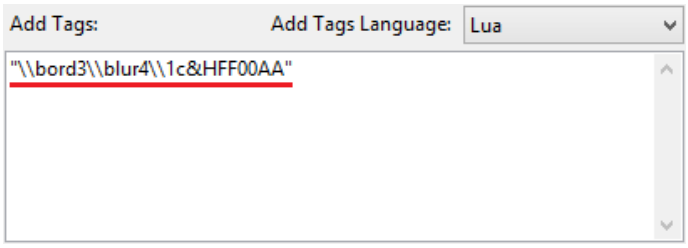
Entender un Efecto es el primer paso para desarrollar uno propio, tanto en el **Aegisub** o en **Kara Effector**. Espero que lo visto en este Tomo sea de fácil comprensión para todos.

OBJETOS

Para hacer un Efecto Karaoke o una simple Plantilla con tags para un archivo de Subtítulos en lenguaje **LUA**, en el **Kara Effector** o en **Automation Auto-4**, solo existen dos **HERRAMIENTAS**, dichas Herramientas reciben el nombre de "**strings**" y "**values**".

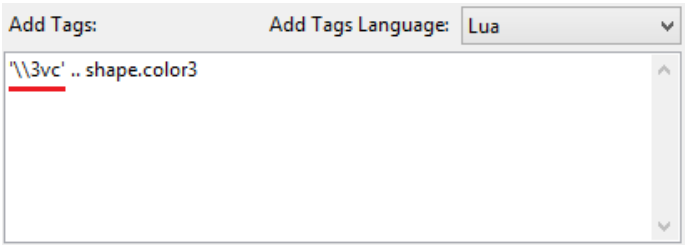
OBJETOS: STRING

Un **string** (cadena) es un caracter o suseción de ellos que carecen de algún tipo de valor. En lenguaje **LUA** un **string** es todo aquello que esté entre comillas, ya sean dobles o simples. Veamos algunos ejemplos de strings en **LUA**:



En este caso, todo aquello que está entre las comillas dobles es un string, ya que para el lenguaje **LUA** esto sería lo mismo que una palabra cualquiera, sin valor alguno.

En el siguiente ejemplo veremos los dos tipos de **Objetos**:



Lo que está entre las comillas simples es un **string**, el resto es **value**.

En **LUA**, si algo está entre comillas es un **string**, incluso si es un número o si hay operaciones matemáticas, ejemplo:

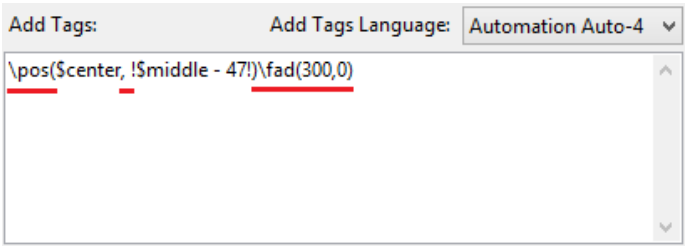
"12 + 3"

Al estar entre comillas no se realiza la operación (suma).

12 + 3

Ya sin las comillas, LUA ejecuta la operación y devuelve el resultado (15).

Por otro lado, en **Automation Auto-4**, un **string** también es fácil de reconocer. Un **string** es todo aquello que no está precedido del signo dólar (variables dólar) ni tampoco está dentro de los signos de admiración. Ejemplos:



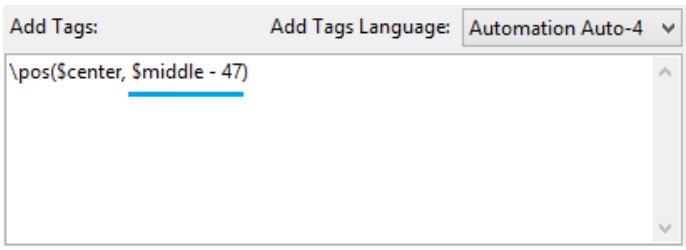
De esta expresión, los únicos **Objetos** que no son un **string** en lenguaje **Automation Auto-4** son:

$center

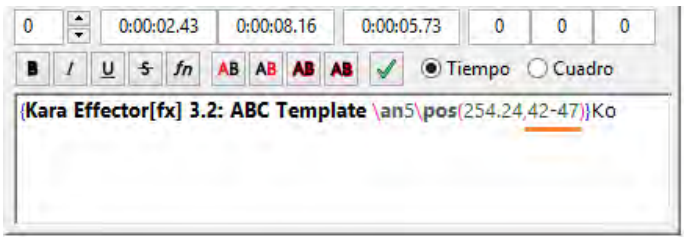
!$middle – 47!

El resto, incluídos los paréntisis y las comas, son un **string** y por ende carecen de valor operacional. Podemos afirmar que los tags con que hacemos los Efectos también son un **string**, aunque para el **Aegisub** sí tengan valor operacional o funcional, ya que ni en **LUA** ni en **Automation Auto-4** se pueden hacer operaciones con ellos.

Un **error** común que se comete a veces en el lenguaje **Automotion Auto-4** es intentar hacer una operación sin poner los signos de admiración:



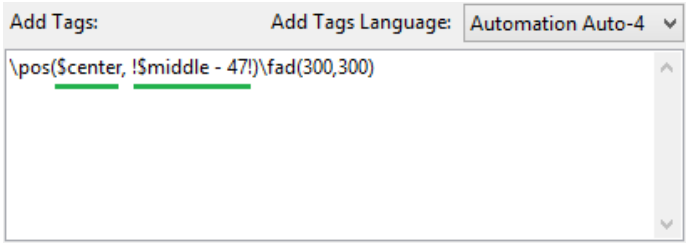
Al no poner los signos de admiración, el **Kara Effector** da por sentado que estamos escribiendo un string y por ello no ejecuta la operación que habíamos pensado:



Se ve claramente cómo queda expresada la sustracción (resta), pero no devolvió el resultado de la misma. **Aegisub** toma en cuenta solo al 42 y omite el resto.

OBJETOS: VALUE

El **value** es todo lo que tiene algún tipo de valor, ya sea número, operacional o funcional. Un **value** es todo lo que no es un **string.** En **LUA** un **value** es todo aquello que no está dentro de las comillas simple o dobles. De forma similar, en **Automation Auto-4**, un **value** es todo aquello que no está precedido del signo dólar (variable dólar) ni tampoco está dentro de los signos de admiración.



En verde podemos ver dos ejemplos de **values** (valores) en lenguaje **Automation Auto-4**.

En **LUA**, como ya lo sabemos, sino está dentro de las comillas, en este caso dobles, es un **value**. Los dos puntos seguidos (..) tienen valor operacional (concatenación) y la variable **l.outline** tiene un valor numérico (espesor del borde medido en pixeles).

Ahora veamos un cuadro con los diferentes tipos de **value** y algunos ejemplos:

TIPO DE VALUE	EJEMPLOS
Numérico	Pi, e, 1276, syl.center, line.middle
Operación	+, -, /, *, ^, .. , %, :
Función	math.random, string.format
Tabla	A = {1, 2, 'F'} B = {"&HFF&", "&H00&"} C = {[1] = 43; [2] = 86} D = { }
Operador Lógico	for, if, while, repeat, or, and
Variable	Color1 = "&HFF0000&" Radius = 6*pi Linevc = ""

Una vez claro los conceptos de **string** y **value** ya podemos seguir viendo las Librerías que aun nos faltan. La que veremos a continuación es referente a los **strings** y con las funciones que hay en ellas notarán que los **string** son más que simples objetos que carecen de valor.

Librería "string" [LUA]

string.byte(s)	Retorna el **Valor Numérico** del caracter **s** según su equivalente decimal en la Tabla ASCII Veamos un ejemplo de la Tabla: 64 @ 96 ` 65 A 97 a 66 B 98 b 67 C 99 c 68 D 100 d Vemos que el valor decimal de "@" es 64, así como el de "b" es 98.
	Ejemplo 1: **string.byte**("A") = 65 Ejemplo 2: Str = "C" **string.byte**(Str) = 67 Los valores decimales de la **Tabla ASCII** van desde el 0 hasta el 255. El modo abreviado de usar esta función es así: Ejemplo 3: s = "B" s**:byte**() = 66 Ejemplo 4: s = "d" s**:byte**() = 100 Esta función tiene más modo de uso, pero al menos por el momento no serán de gran utilidad y de ahí que las haya omitido.
string.char(…)	Retorna el caracter asignado por la **Tabla ASCII** de un número entre 0 y 255. Ejemplo: **string.char**(65) = "A" **string.char**(66) = "B" **string.char**(65, 66, 100) = "ABd"
string.find(s, ptr)	Retorna las posiciones del inicio y final del string **s** dentro de un string mayor **ptr**. Ejemplo 1: L = "Demo lua string" s = "lua" **string.find**(s, L) = 6, 8 6 es el inicio y 8 es el final: D e m o ␣ l u a ␣ s 1 2 3 4 5 6 7 8 La manera abreviada es: s:**find**(L) = 6, 8 Si el string **s** no pertenece al string **ptr**, entonces retorna **nil**
string.format (s, …)	Inserta valores al string **s**. El modo general de asignar los valores el por medio del string "%s". Ejemplo 1: **string.format**("\\be%s", **l.shadow**) Es decir que inserta el valor de **l.shadow** al string "\\be%s". Hay dos formas abreviadas de usar esta función en el Kara Effector: • **format(s, …)** • **F(s, …)** Ejemplo 2: **F**("\\pos(%s,%s)", syl.center, 20*pi)

	Por cada vez que aparezca “%s” en el string, se debe asignar el valor que se insertará en esa posición. Este modo es el más usado en el Kara Effector y es la función string que más se usa en él. Los demás modos no son tan relevantes para hacer Efectos Karaokes.
string.gsub (s, ptr, replace, n)	Remplaza al string **s** que pertenece al string **ptr**, por un valor u otro string llamado **replace**, **n** cantidad de veces. El modo abreviado de uso es: **ptr:gsub(s, replace, n)** **n** es un entero positivo, es opcional en la función. Ejemplo 1: ptr = “el sol en la mañana” ptr:**gsub**(“a”, “X”) = “el sol en lX mXñXnX” La función remplazó al carácter “a” por “X” todas las veces. Ejemplo 2: ptr = “el sol en la mañana” ptr:**gsub**(“a”, “X”, 2) = “el sol en lX mXñana” El remplazo se hizo solo dos veces, dado que **n** = 2 Esta función también es mucho más extensa de lo que aparenta, pero de eso nos ocuparemos más adelante.
string.len(s)	Retorna la cantidad de caracteres que tiene el string **s**. El modo abreviado es: **s:len()** Ejemplo 1: s = “&HFFFFFF&” s:**len**() = 9 Ejemplo 2: s = “El sol” s:**len**() = 6 No olvidemos que el espacio entre palabra y palabra también es un caracter.
string.lower(s)	Retorna al string **s** con todas las letras mayúsculas en él convertidas en minúsculas. El modo abreviado es: **s:lower()** Ejemplo 1: s = “La Noche” s:**lower**() = “la noche”

	Ejemplo 2: s = “EL SOL” s:**lower**() = “el sol”
string.rep(s, n)	Repite al string **s** una **n** cantidad de veces. El modo abreviado es: **s:rep(n)** Ejemplo 1: s = “Demo” s:**rep**(3) = “DemoDemoDemo” Ejemplo 2: s = “lua ” s:**rep**(5) = “lua lua lua lua lua”
string.reverse(s)	Invierte al string **s**. El modo abreviado es: **s:reverse()** Ejemplo 1: s = “Demo” s:**reverse**() = “omeD” Ejemplo 2: s = “La Luna y el Sol” s:**reverse**() = “loS le y anuL aL”
string.sub(s, i, j)	Recorta al string **s** desde la posición **i** hasta la posición **j**. Tanto **i** como **j** son números enteros y pueden ser positivos o negativos. Al ser positivos la posición se empieza a contar de izquierda a derecha, si son negativos, se cuenta al revés. El modo abreviado es: **s:sub(i, j)** Ejemplos: s = “El Sol” s:**sub**(1, -1) = “El sol” s:**sub**(1, 1) = “E” s:**sub**(-3, -1) = “Sol” s:**sub**(2, -1) = “l Sol” El parámetro **j** es opcional s:**sub**(-2) = “o” s:**sub**(4) = “S”
string.upper(s)	Retorna al string **s** con todas las letras minúsculas en él convertidas en mayúsculas. El modo abreviado es: **s:upper()** Ejemplo 1: s = “La Noche” s:**upper**() = “LA NOCHE” Ejemplo 2: s = “¡¿Qué?, no puedes!” s:**upper**() = “¡¿QUÉ?, NO PUEDES!”

De esta Librería también omití un par de funciones que quizás, al necesitarlas más adelante, las explique a cada una de ellas, aunque al ser una biblioteca de **LUA** se hace un poco más simple hallar información en la web sobre ellas.

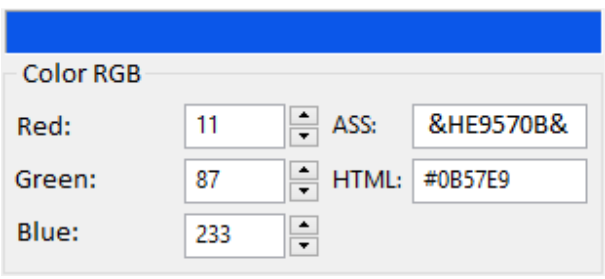
Terminada esta librería, es momento para ver las formas abreviadas de las funciones comúnmente usadas en el **Kara Effector**:

ABREVIATURA	FUNCIÓN
pi	math.pi
sin	math.sin
cos	math.cos
tan	math.tan
asin	math.asin
acos	math.acos
atan	math.atan
sinh	math.sinh
cosh	math.cosh
tanh	math.tanh
log	math.log10
ln	math.log
abs	math.abs
floor	math.floor
ceil	math.ceil
deg	math.deg
rad	math.rad
r	math.random
R	math.R
Rf	math.Rfake
rand	math.random
format	string.format
F	string.format

Todas las anteriores abreviaturas de las funciones aplican tanto en lenguaje **LUA** como en **Automation Auto-4**.

A continuación veremos un poco de dos de las **teorías de color** que maneja el formato .ass que nos servirán para entender las siguientes Librerías.

TEORÍA DEL COLOR RGB (Red, Green, Blue):



De la anterior imagen vemos cómo el tono de azul que está en la parte superior de la misma, está formado por tres valores de rojo, verde y azul:

Tono	Valor Decimal	Valor Hexadecimal
Rojo	11	0B
Verde	87	57
Azul	233	E9

En formato .ass el tono de azul del ejemplo anterior sería:

&HE9570B&

De forma general, todo color en formato .ass tiene la siguiente estructura:

&H [Azul] [Verde] [Rojo] &

La estructura de un color en formato .ass es similar a la del formato **HTML**, pero los tonos invertidos, como se ve en la imagen anterior:

#0B57E9

El **Kara Effector** tiene una función que convierte un color de formato **HTML** a .ass:

color.ass("#0B57E9") = &HE9570B&

Veamos un listado de colores conocidos en formato **HTML** que con la anterior función ya sabemos cómo pasarlos a formato .ass. En la Tabla aparece el color, el nombre en inglés y el código del mismo en formato **HTML**:

AliceBlue	#F0F8FF
AntiqueWhite	#FAEBD7
Aqua	#00FFFF
Aquamarine	#7FFFD4
Azure	#F0FFFF
Beige	#F5F5DC
Bisque	#FFE4C4
Black	#000000
BlanchedAlmond	#FFEBCD
Blue	#0000FF
BlueViolet	#8A2BE2
Brown	#A52A2A
BurlyWood	#DEB887
CadetBlue	#5F9EA0
Chartreuse	#7FFF00
Chocolate	#D2691E
Coral	#FF7F50
CornflowerBlue	#6495ED

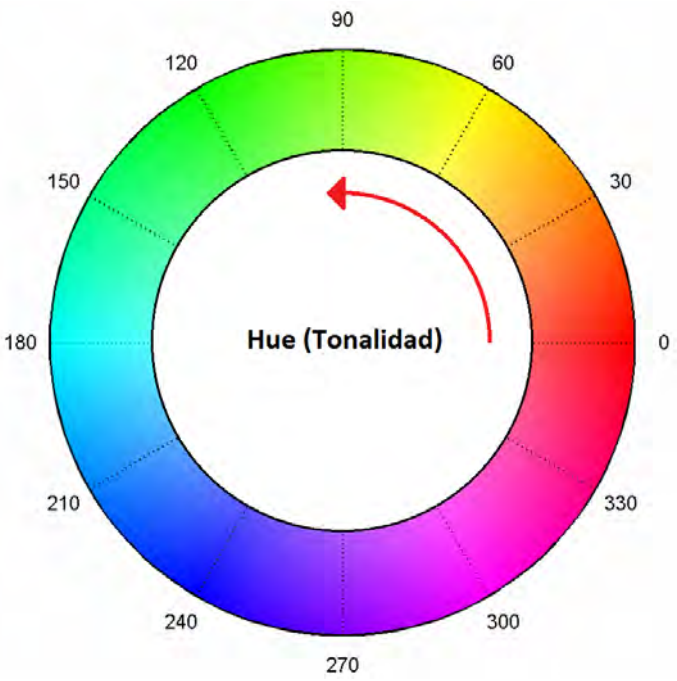
Cornsilk	#FFF8DC
Crimson	#DC143C
Cyan	#00FFFF
DarkBlue	#00008B
DarkCyan	#008B8B
DarkGoldenrod	#B8860B
DarkGray	#A9A9A9
DarkGreen	#006400
DarkKhaki	#BDB76B
DarkMagenta	#8B008B
DarkOliveGreen	#556B2F
DarkOrange	#FF8C00
DarkOrchid	#9932CC
DarkRed	#8B0000
DarkSalmon	#E9967A
DarkSeaGreen	#8FBC8F
DarkSlateBlue	#483D8B
DarkSlateGray	#2F4F4F
DarkTurquoise	#00CED1
DarkViolet	#9400D3
DeepPink	#FF1493
DeepSkyBlue	#00BFFF
DimGray	#696969
DodgerBlue	#1E90FF
Firebrick	#B22222
FloralWhite	#FFFAF0
ForestGreen	#228B22
Fuchsia	#FF00FF
Gainsboro	#DCDCDC
GhostWhite	#F8F8FF
Gold	#FFD700
Goldenrod	#DAA520
Gray	#808080
Green	#008000
GreenYellow	#ADFF2F
Honeydew	#F0FFF0
HotPink	#FF69B4
IndianRed	#CD5C5C
Indigo	#4B0082
Ivory	#FFFFF0
Khaki	#F0E68C
Lavender	#E6E6FA
LavenderBlush	#FFF0F5
LawnGreen	#7CFC00
LemonChiffon	#FFFACD
LightBlue	#ADD8E6
LightCoral	#F08080
LightCyan	#E0FFFF
LightGoldenrodYellow	#FAFAD2

LightGray	#D3D3D3
LightGreen	#90EE90
LightPink	#FFB6C1
LightSalmon	#FFA07A
LightSeaGreen	#20B2AA
LightSkyBlue	#87CEFA
LightSlateGray	#778899
LightSteelBlue	#B0C4DE
LightYellow	#FFFFE0
Lime	#00FF00
LimeGreen	#32CD32
Linen	#FAF0E6
Magenta	#FF00FF
Maroon	#800000
MediumAquamarine	#66CDAA
MediumBlue	#0000CD
MediumOrchid	#BA55D3
MediumPurple	#9370DB
MediumSeaGreen	#3CB371
MediumSlateBlue	#7B68EE
MediumSpringGreen	#00FA9A
MediumTurquoise	#48D1CC
MediumVioletRed	#C71585
MidnightBlue	#191970
MintCream	#F5FFFA
MistyRose	#FFE4E1
Moccasin	#FFE4B5
NavajoWhite	#FFDEAD
Navy	#000080
OldLace	#FDF5E6
Olive	#808000
OliveDrab	#6B8E23
Orange	#FFA500
OrangeRed	#FF4500
Orchid	#DA70D6
PaleGoldenrod	#EEE8AA
PaleGreen	#98FB98
PaleTurquoise	#AFEEEE
PaleVioletRed	#DB7093
PapayaWhip	#FFEFD5
PeachPuff	#FFDAB9
Peru	#CD853F
Pink	#FFC0CB
Plum	#DDA0DD
PowderBlue	#B0E0E6
Purple	#800080
Red	#FF0000
RosyBrown	#BC8F8F
RoyalBlue	#4169E1

SaddleBrown	#8B4513
Salmon	#FA8072
SandyBrown	#F4A460
SeaGreen	#2E8B57
SeaShell	#FFF5EE
Sienna	#A0522D
Silver	#C0C0C0
SkyBlue	#87CEEB
SlateBlue	#6A5ACD
SlateGray	#708090
Snow	#FFFAFA
SpringGreen	#00FF7F
SteelBlue	#4682B4
Tan	#D2B48C
Teal	#008080
Thistle	#D8BFD8
Tomato	#FF6347
Transparent	#FFFFFF
Turquoise	#40E0D0
Violet	#EE82EE
Wheat	#F5DEB3
White	#FFFFFF
WhiteSmoke	#F5F5F5
Yellow	#FFFF00
YellowGreen	#9ACD32

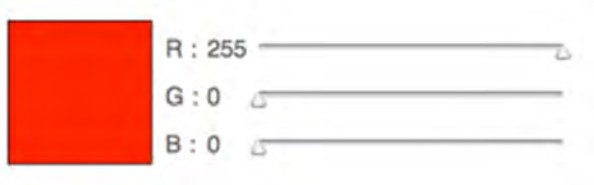
TEORÍA DEL COLOR HSV (Hue, Saturation, Value):

Hue: es la tonalidad de un color, cuyo valor es un real desde 0 hasta 360, como vemos en la imagen:



En la anterior imagen vemos cómo **Hue** = 0 equivale al rojo, **Hue** = 60 equivale al amarillo, **Hue** = 240 equivale al azul.

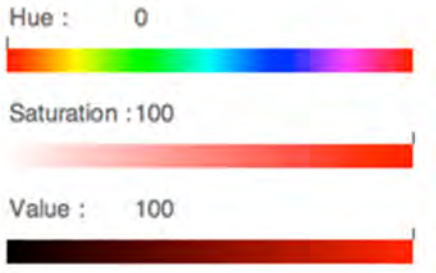
Veamos un ejemplo con el color Rojo. Con la teoría del color **RGB** sería (los valores de R, G y B son reales entre 0 como mínimo y 255 como máximo):



En formato HTML: **#FF0000**

En formato .ass: **&H0000FF&**

Pero en la teoría HSV sería:



- Hue = 0
- Saturation = 100
- Value = 100

Lo que da pie a las siguientes dos definiciones que hacen parte a la teoría HSV.

Saturation: es la cantidad de blanco que tendrá la tonalidad **Hue**. Su valor es un número real entre 0 y 100, donde 0 completamente blanco y 100 equivale a que la tonalidad no tendría nada de blanco.

Value: es la cantidad de negro que tendrá la tonalidad **Hue**. Su valor es un número real entre 0 y 100, donde 0 completamente negro y 100 equivale a que la tonalidad no tendría nada de negro.

Si **Saturation** es 0, no importa el valor de **Hue**, el color siempre será **Blanco**. De manera similar pasaría con **Value**, ya que si es 0, no importaría el valor de **Hue**, el color que resultaría siempre sería **Negro**.

Para poner en práctica la teoría del color **HSV**, **Aegisub** ya trae por default un par de funciones que hacen posible convertir los tres valores (H, S y V) en un color en formato .ass, y la primera de ellas es:

HSV_to_RGB(H, S, V): convierte los valores de **H**, **S** y **V** en valores de 0 a 255 de Rojo, Verde y Azul. Es decir que esta función retorna tres resultados al mismo tiempo.

Para esta función de Aegisub, **Hue** sigue siendo un real entre 0 y 360, pero **Saturation** y **Value** son un número real entre 0 y 1.

Ejemplo 1:

R, G, B = **HSV_to_RGB(0, 1, 1)**

- R = 255
- G = 0
- B = 0

Ejemplo 2:

R, G, B = **HSV_to_RGB(264, 0.75, 0.2)**

- R = 28.05
- G = 12.75
- B = 51

Y la siguiente función del **Aegisub** es la que convierte estos tres valores en un color en formato .ass para que pueda ser usado en los tags de colores:

ass_color(R, G, B): transforma los valores de R, G y B en un color en formato .ass: **&H[Azul][Verde][Rojo]&**

Ejemplos:

ass_color(255, 0, 0) = &H0000FF&

ass_color(47, 82, 242) = &HF2522F&

ass_color(158, 80, 153) = &H99509E&

lo que quiere decir que debemos combinar estas dos funciones para convertir los valores de H, S y V en un color en formato .ass:

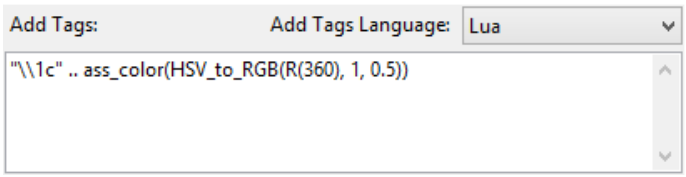
ass_color(HSV_to_RGB(H, S, V))

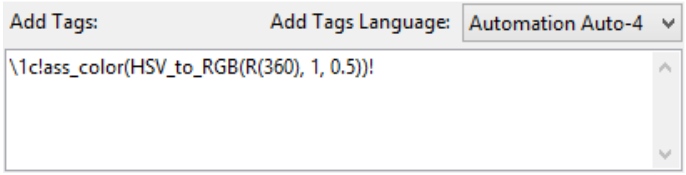
veamos algunos ejemplos de esta combinación:

Estilo	Efecto	Texto
Default	template	!_G.ass_color(_G.HSV_to_RGB(45, 1, 0.5))!
Default	karaoke	
Default	fx	&H005F7F&
Default	fx	&H005F7F&

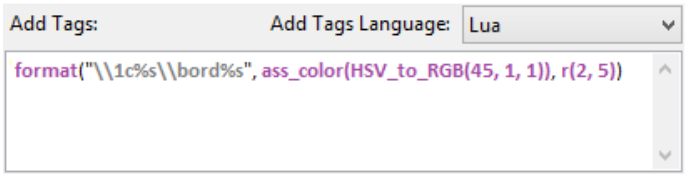
Recordemos que desde el **Aegisub**, para usar cualquiera de las funciones que por default que vienen en él, éstas deben iniciar por "**_G.**" como se ve en la imagen anterior. En el **Kara Effector** da igual si se coloca este prefijo o no, las funciones son reconocidas de las dos formas.

Un ejemplo práctico en el **Kara Effector** sería:





Uno más usando la función **string.format**:



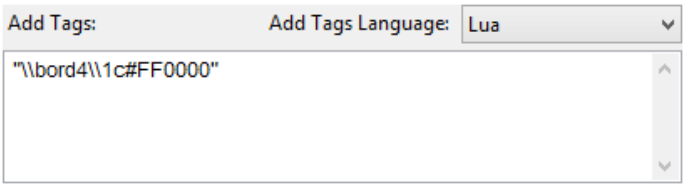
Es decir que con esta combinación de funciones podemos obtener un color con valores constantes, como en el ejemplo de la imagen anterior, o con valores aleatorios (random), como en las otras dos anteriores imágenes.

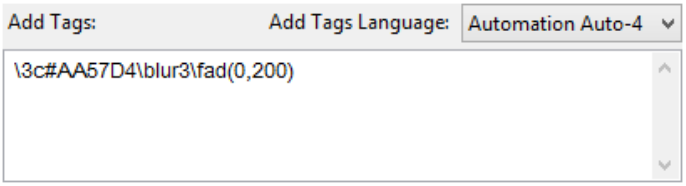
En este punto, el nivel de complejidad ha ido en aumento y cada vez tenemos un mayor conocimiento y contamos con más herramientas para desarrollar nuestros propios Efectos. La mayoría de las próximas funciones de las siguientes librerías tienen aplicación directamente para hacer nuevos Efectos y vendrán acompañadas con varios ejemplos y ejercicios prácticos. No olviden visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia.

Kara Effector 3.2:

Para la fecha de publicación de este **Tomo VII** ya pueden descargar la más reciente actualización del **Kara Effector** que pone como fecha el 1 de Enero de 2014. La nueva actualización consiste en poder usar en nuestros Efectos los colores en formato **HTML** sin tener la necesidad de convertirlos a formato .ass y tener mayor diversidad de opciones al poder usar ambos formatos de manera libre.

La actualización fue inspirada al desarrollar el tema de las teorías de color que se usan en el **Aegisub** y les mostraré unos cuantos ejemplos de usarla:





Con esta actualización podemos copiar el código de un color en formato **HTML** y usarlo en el **Kara Effector**. En la web hay muchas páginas en donde podemos encontrar estos colores en dicho formato. Recomiendo la siguiente:

http://www.computerhope.com/htmcolor.htm

En esta web encontraremos un extenso listado de colores en formato **HTML** y así poder elegir el que más se ajuste al Efecto que aplicaremos:

#357EC7	**Slate Blue**
#488AC7	**Silk Blue**
#3090C7	**Blue Ivy**
#659EC7	**Blue Koi**
#87AFC7	**Columbia Blue**
#95B9C7	**Baby Blue**

Librería "random" [KE]:

Es una librería exclusiva del **Kara Effector** que consiste en una serie de funciones con resultados aleatorios que podemos aplicar en el desarrollo de los Efectos. La Librería "**random**" contiene las siguientes funciones:

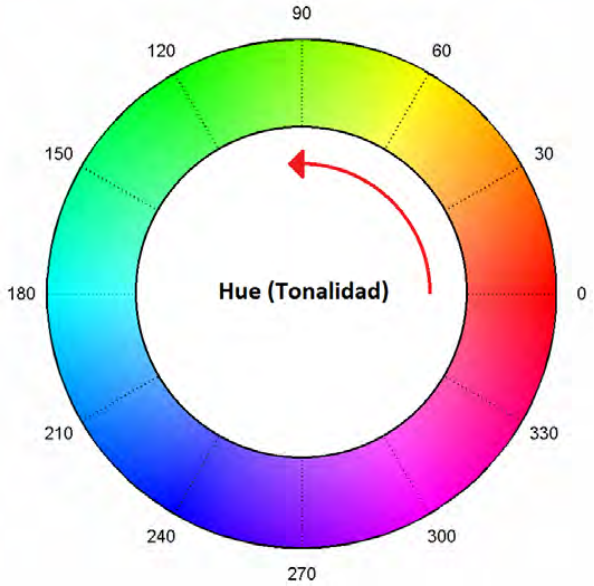
- **random.color**
- **random.colorvc**
- **random.alpha**
- **random.alphava**
- **random.e**
- **random.unique**

Y a continuación veremos en detalle en qué consiste casa una de ellas.

random.color(H, S, V): retorna un color en formato .ass según la teoría HSV. **H**, **S** y **V** son parámetros opcionales y pueden ser valores numéricos o tablas.

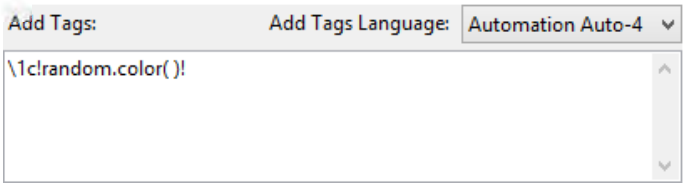
Modo 1: sin ningún parámetro

random.color(): retorna un color en formato .ass correspondiente a un tono (**Hue**) al azar entre 0 y 360 de la teoría **HSV**.



En este modo (Modo 1), la función retorna un tono al azar, pero tanto **S** (**Saturation**) como **V** (**Value**) son constantes y el valor de cada uno de ellos es 100, o sea que el color será solo alguno de los de la imagen anterior.

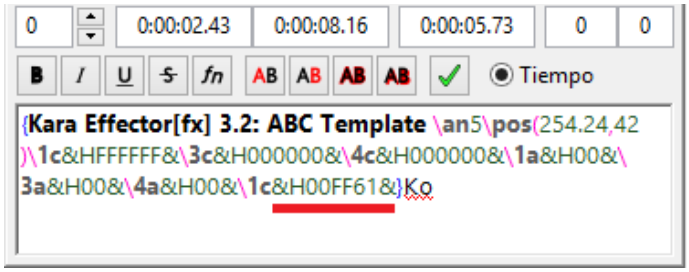
En el **Kara Effector**, en un Efecto tipo "**Syl**", podemos poner a prueba el Modo 1 de esta función:



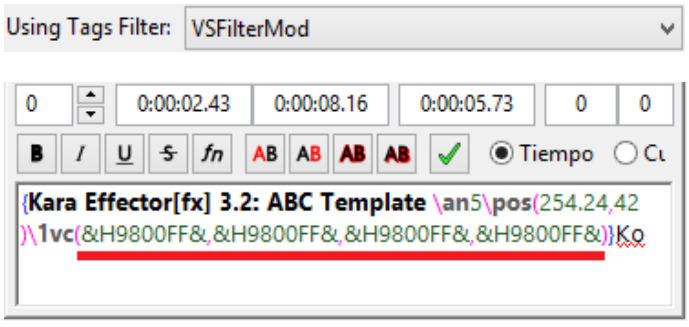
Entonces en el vídeo veremos algo como esto:

Para cada Sílaba la función generó un color aleatorio con un valor entre 0 y 360 del anterior círculo cromático.

Este es un ejemplo del color que puede retornar la función en este modo, siempre y cuando hayamos seleccionado la opción "**VSFilter 2.39**" o la opción "**No Tags Color and Alpha**" en **Using Tag Filter:**



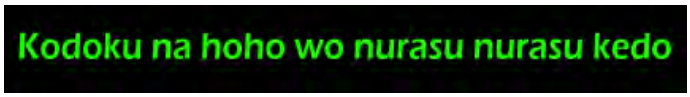
Si por el contrario, elegimos la opción "**VSFilterMod**", el color se multiplicará cuatro veces y saldrán separados por comas dentro de un paréntesis (en formato **VSFilterMod**):



Modo 2: con un parámetro constante

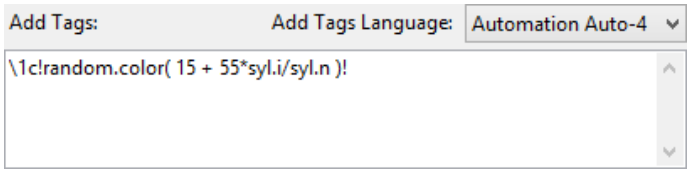
random.color(H): retorna un color en formato .ass correspondiente al tono (**Hue**) entre 0 y 360 que le hayamos asignado a la función. Ejemplo:

random.color(112) = "&H00FF21&"

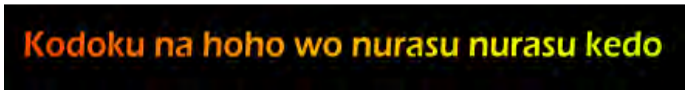


En el círculo cromático vemos que el valor 112 le corresponde a un todo de verde claro. En este modo, la función retorna un color constante correspondiente al **Hue** del valor que le ingresemos.

Usar la función **random.color** para que retorne un color constante es ideal para hacer degradaciones con los colores **HSV**. Ejemplo: (**Template Type**: Syl)



Es un degradado **HSV** con los valores desde 15 para la primera Sílaba, hasta 70 (15 + 55 = 70) para la última:



Modo 3: con una Tabla como parámetro

random.color({H_i, H_f}): retorna un color en formato .ass correspondiente a un valor aleatorio entre **H_i** y **H_f**. Ejemplo:

random.color({270, 320}) = "&HFF00AA&"

Vemos que el tono (**Hue**) que resulta corresponde a un valor aleatorio entre 270 y 320 del círculo cromático **HSV**. O sea que cuando algún parámetro es una Tabla, entonces la función hace un Random entre los dos valores de la Tabla, cada vez que se ejecute la función.

Si combinamos los tres Modos de la función **random.color** (ausencia de uno o más de sus parámetros, uno o más parámetros constantes y por último, que uno o más de los parámetros sean una Tabla), obtenemos todo un mundo nuevo de posibilidades (15 en total):

- **random.color()**
- **random.color(H)**
- **random.color({H_i, H_f})**
- **random.color(H, S)**
- **random.color(H, {S_i, S_f})**
- **random.color({H_i, H_f}, S)**
- **random.color({H_i, H_f}, {S_i, S_f})**
- **random.color(H, S, V)**
- **random.color(H, S, {V_i, V_f})**
- **random.color(H, {S_i, S_f}, V)**
- **random.color({H_i, H_f}, S, V)**
- **random.color(H, {S_i, S_f}, {V_i, V_f})**
- **random.color({H_i, H_f}, S, {V_i, V_f})**
- **random.color({H_i, H_f}, {S_i, S_f}, V)**
- **random.color({H_i, H_f}, {S_i, S_f}, {V_i, V_f})**

En resumen:

- el parámetro **H** es un valor entre 0 y 360 o una tabla con dos elementos numéricos entre 0 y 360. Su valor por default es aleatorio entre 0 y 360.
- **S** y **V** son, o un valor entre 0 y 100 o una tabla con dos elementos numéricos entre 0 y 100. Sus valores por default son siempre de 100 para cada uno.

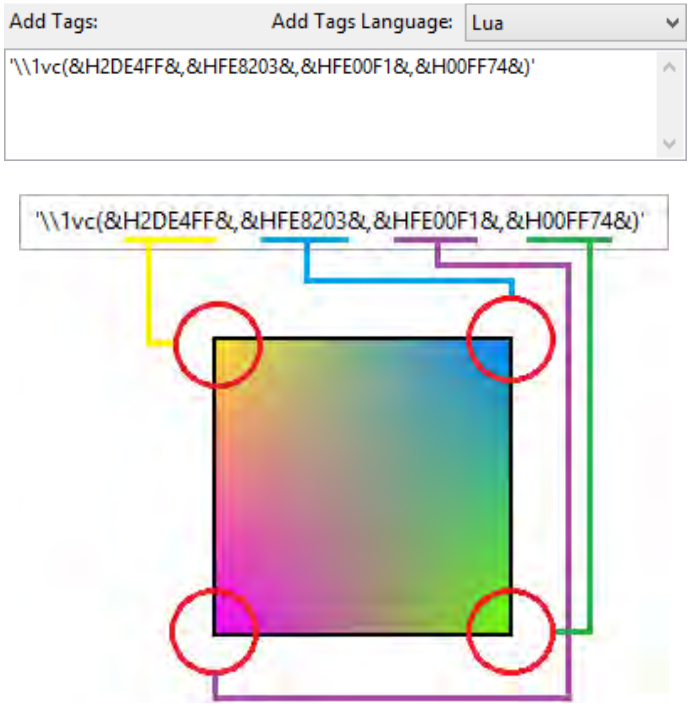
Ejemplos:

- **random.color(115)**
- **random.color({30, 220})**
- **random.color(304, 70)**
- **random.color(216, {60, 80})**
- **random.color({320, 340}, 90)**
- **random.color({125, 150}, {50, 100})**
- **random.color(0, 55, 92)**
- **random.color(86, 10, {0, 72})**
- **random.color(328, {60, 80}, 64)**
- **random.color({80, 120}, 77, 81)**
- **random.color(143, {0, 100}, {50, 80})**
- **random.color({45, 77}, 44, {66, 100})**
- **random.color({0, 105}, {70, 90}, 16)**
- **random.color({170, 204}, {55, 76}, {47, 88})**

Recomiendo poner a prueba en el **Kara Effector** cada uno de los anteriores ejemplos con el fin de practicar con esta función hasta que se familiaricen con cada uno de sus modos y opciones de uso. Esta es la primera función de la Librería "**random**" y restan cinco más por ver.

random.colorvc(H, S, V): es similar a **random.color** con la única diferencia que retorna un color en formato del filtro **VSFiltermod**.

Un color en formato **VSFilterMod** está compuesto por cuatro colores para cada uno de sus cuadrantes. Ejemplo:



O sea que todo color en formato **VSFilterMod** tiene la siguiente estructura:

(color SI, color SD, color II, color ID)

- **color SI:** Superior Izquierdo
- **color SD:** Superior Derecho
- **color II:** Inferior Izquierdo
- **color ID:** Inferior Derecho

Y lo que hace la función **random.colorvc** es usar la función **random.color** para generar a cada uno de esos cuatro anteriores colores. Los modos y todas las combinaciones posibles de la función **random.colorvc** son exactamente los mismos que en **random.color**, vista anteriormente:

- **random.colorvc()**
- **random.colorvc(H)**
- **random.colorvc({H_i, H_f})**
- **random.colorvc(H, S)**
- **random.colorvc(H, {S_i, S_f})**
- **random.colorvc({H_i, H_f}, S)**
- **random.colorvc({H_i, H_f}, {S_i, S_f})**
- **random.colorvc(H, S, V)**
- **random.colorvc(H, S, {V_i, V_f})**
- **random.colorvc(H, {S_i, S_f}, V)**
- **random.colorvc({H_i, H_f}, S, V)**
- **random.colorvc(H, {S_i, S_f}, {V_i, V_f})**
- **random.colorvc({H_i, H_f}, S, {V_i, V_f})**
- **random.colorvc({H_i, H_f}, {S_i, S_f}, V)**
- **random.colorvc({H_i, H_f}, {S_i, S_f}, {V_i, V_f})**

No sobra recordarles que para usar colores en formato **VSFilterMod** debemos seleccionar esa opción en **Using Tags Filter**, de otro modo el **Kara Effector** convertirá el color a formato **VSFilter 2.39**.

---

random.alpha(A1, A2): retorna una transparencia (**alpha**) aleatoria en formato .ass (**VSFilter 2.39**) desde el valor de **A1** hasta **A2**.

Los parámetros **A1** y **A2** pueden ser números reales entre 0 y 255, o valores **alpha** entre "&H00&" y "&HFF&".

Modo 1: sin parámetros

random.alpha(): retorna una transparencia al azar entre totalmente visible (0 en decimal o "&H00" en formato .ass), hasta totalmente invisible (255 en decimal o "&HFF&" en formato .ass).

Modo 2: parámetros numéricos

random.alpha(A1, A2): retorna una transparencia aleatoria entre los valores numéricos de **A1** y **A2**. Ejemplos:

- **random.alpha(55, 142)**
- **random.alpha(0, 200)**
- **random.alpha(180, 255)**

Modo 3: parámetros **alpha** en formato .ass

random.alpha(A1, A2): retorna una transparencia aleatoria entre los valores **alpha** (.ass) de **A1** y **A2**. Ejemplos:

- **random.alpha("&HA3&", "&HED&")**
- **random.alpha("&H0C&", "&H7F&")**

Una especie de Modo 4 sería combinar los modos 2 y 3. Ejemplos:

- **random.alpha(120, “&HED&”)**
- **random.alpha(“&H0C&”, 215)**

---

random.alphava(A1, A2): es muy similar a la anterior función y retorna una transparencia (**alpha**) aleatoria en formato **VSFilterMod** desde el valor de **A1** hasta **A2**. Los Modos y formas de uso son los mismos que usamos en **random.alpha**

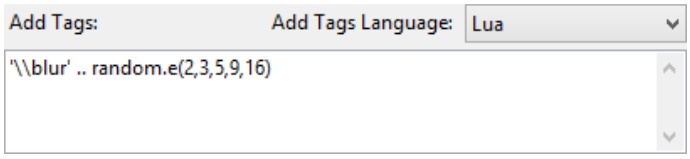
Para cada una de las cuatro transparencias (alpha), que conforman a una transparencia en formato **VSFilterMod**, se ejecuta el random que hace que sea prácticamente imposible que las cuatro sean iguales. Ejemplos:

- **random.alphava(90, 180)**
- **random.alphava(0, 100)**
- **random.alphava(“&HC4&”, “&HFF&”)**
- **random.alphava(“&H00&”, “&H86&”)**
- **random.alphava(12, “&HAA&”)**
- **random.alphava(“&HDF&”, 125)**

---

random.e(…): retorna un parámetro al azar de entre todos los que se le hayan ingresado o un elemento al azar, entre los elementos de una tabla que se le haya ingresado como parámetro.

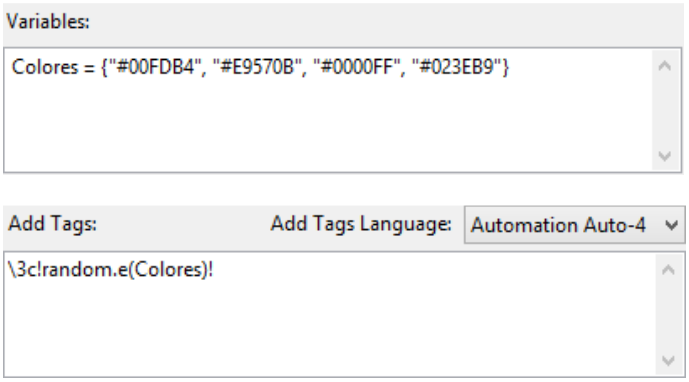
Ejemplo 1:



La función retornaría alguno de esos cinco valores que se le ingresaron, seleccionado de forma aleatoria, o sea que los posibles resultados serán:

- \blur2
- \blur3
- \blur5
- \blur9
- \blur16

Ejemplo 2:



Declaré una tabla llamada “Colores”, entonces la función **random.e** selecciona un elemento al azar de entre los cuatro que tiene la tabla.

---

random.unique(T, i): primero desordena la posición de los elementos de la tabla **T** para posteriormente retornar al elemento **T[i]**. **T** puede ser una tabla propiamente dicha o un número entero positivo. En el caso de que **T** sea un entero, entonces se crea una tabla con los números desde 1 hasta **T**.

Ejemplo 1:

random.unique(syl.n, syl.i): primero crea una tabla con los números consecutivos desde 1 hasta syl.n:

T = {1, 2, 3, 4, 5, 6, 7, 8, … , syl.n}

Luego desordena la posición de los elementos de la tabla de manera aleatoria:

T = {5, 8, 4, syl.n, 2, 6, 3, … , 1, 7 }

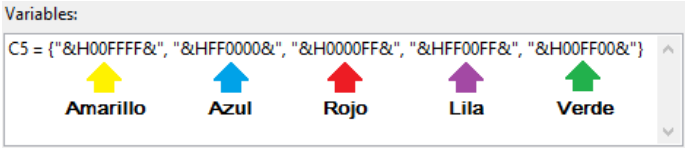
Por último, retorna **T**[syl.i]:

- **T**[1] = 5
- **T**[2] = 8
- **T**[3] = 4
- **T**[4] = syl.n
- **T**[syl.n - 1] = 1
- **T**[syl.n] = 7

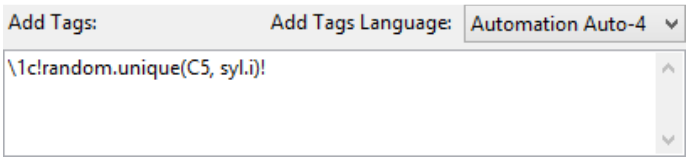
O sea que la función **random.unique** nunca repite un resultado, a menos que sea usada una cantidad mayor de veces que el tamaño de la tabla.

Ejemplo 2:

Declaramos una tabla con cinco colores dispuestos de la siguiente forma:



Si intentamos usar un color para cada Sílaba, y si una Línea de Texto tiene más de cinco Sílabas, entonces esto hace imposible que la función **random.unique** le asigne un único color de esta tabla a cada una de las Sílabas. Lo que hace la función es agotar las opciones y luego repite el mismo patrón una y otra vez:



Como la tabla "**C5**" de nuestro ejemplo solo tiene cinco elementos, el resultado **C5**[6] o cualquier otro mayor no existiría, entonces la función repite los resultados:

Para esta línea de texto la función reorganizó los colores de la tabla en forma aleatoria así:

1. Rojo
2. Amarillo
3. Lila
4. Verde
5. Azul

Y vemos que el patrón, al menos en esta línea, se repitió tres veces:

Esta se podría considerar como la función más compleja de la Librería "**random**", pero con un poco de práctica se irán acostumbrando y encontrarán la forma de darle una aplicación en algunos de sus Efectos y proyectos.

Con la conclusión de la Librería "**random**" se da por terminado el **Tomo VII**, con la recomendación de poner en práctica los ejemplos vistos en él. No olviden descargar la más reciente actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia.

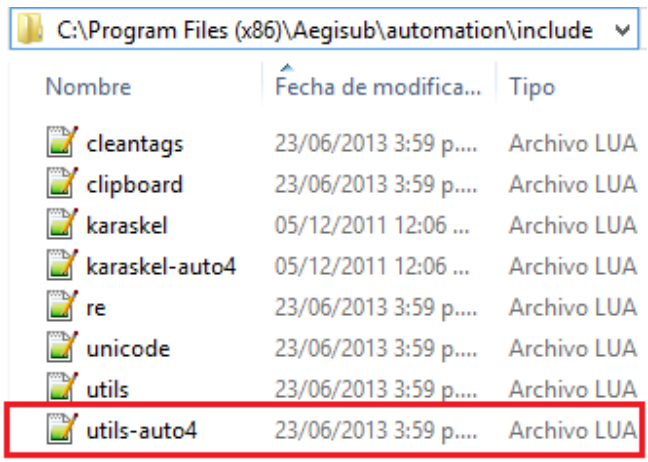
Kara Effector 3.2:

Esta es la entrega del **Tomo VIII** y cada vez estamos más cerca de comprender a cada una de las herramientas con las que podemos contar en el **Kara Effector** a la hora de hacer un Efecto para nuestros Subtítulos. Siguiendo con este orden de ideas, continuaremos viendo las funciones y variables que aún nos restan por ver.

Las Funciones Compartidas de Automation Auto-4 y del Kara Effector:

En el **Aegisub** se pueden usar una serie de funciones que hacen parte de la Librería del **Automation Auto-4**, que lleva por nombre: **utils auto-4**

La Librería **utils auto-4** es un archivo .lua que esta en la carpeta "**include**" del **Aegisub**, y en ella están todas las funciones que se pueden usar para hacer Efectos en los **Templates**:



Todas la funciones de esta Librería se pueden usar en el **Kara Effector**. Esas funciones son:

- **ass_color(r, g, b)**
- **ass_alpha(a)**
- **ass_style_color(r, g, b, a)**
- **alpha_from_style(scolor)**
- **color_from_style(scolor)**
- **HSV_to_RGB(H, S, V)**
- **HSL_to_RGB(H, S, L)**
- **interpolate(pct, min, max)**
- **interpolate_color(pct, first, last)**
- **interpolate_alpha(pct, first, last)**

La explicación de estas y otras funciones del **Aegisub** están en su **Web Oficial**:

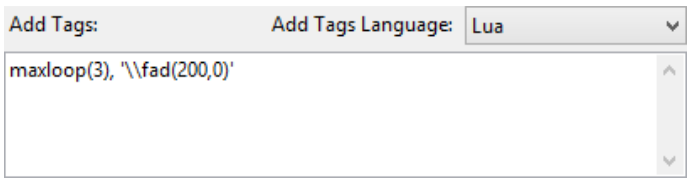
http://docs.aegisub.org/manual/Automation_4_utils.lua

Algunas de las anterires funciones, me imagino que las han usado o las han visto en algún Efecto, es por eso que no profundizaré en detalles con ellas, o no al menos por ahora.

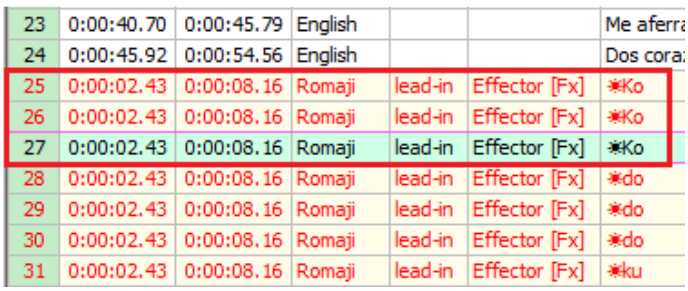
Hay otras funciones que también pertenecen al **Aegisub** y se pueden usar en el **Kara Effector**:

- **maxloop(new_loop)**
- **relayer(new_layer)**
- **retime(modo, add_start, add_end)**

maxloop(n): esta función determina el cantidad de veces que se repetirá un Línea fx. Ejemplo:

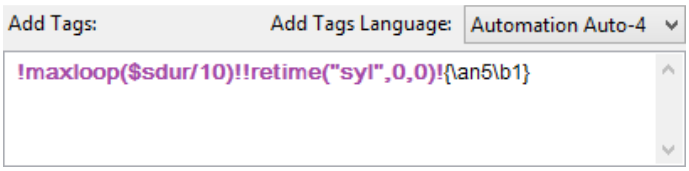


Entonces cada Línea fx se triplica:



23	0:00:40.70	0:00:45.79	English			Me aferra
24	0:00:45.92	0:00:54.56	English			Dos cora
25	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱Ko
26	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱Ko
27	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱Ko
28	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱do
29	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱do
30	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱do
31	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱ku

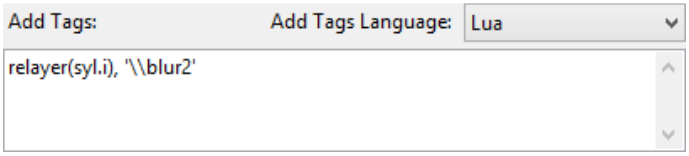
Otro Ejemplo, en Lenguaje **Automation Auto-4:**



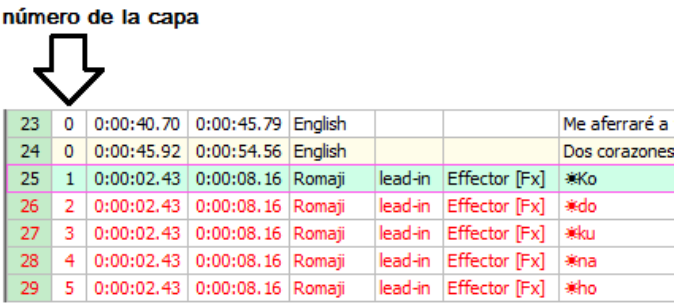
Es decir que maxloop cumple una función muy similar a la celda de texto "**loop**", pero con la diferencia de que la función **maxloop** se puede usar dentro de las nuevas funciones que se pueden hacer en la celda de texto "**Variables**".

relayer(n): esta función determina el número de la capa de cada una de las líneas generadas por un efecto. El entero "**n**" determina el número de la capa.

Un ejemplo sencillo en lenguaje **LUA** podría ser:



Y este sería el resultado:

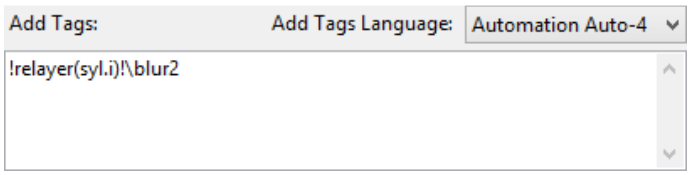


23	0	0:00:40.70	0:00:45.79	English			Me aferraré a
24	0	0:00:45.92	0:00:54.56	English			Dos corazones
25	1	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱Ko
26	2	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱do
27	3	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱ku
28	4	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱na
29	5	0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱ho

O sea que el número de la capa de cada línea de efecto generada equivale a la posición de la sílaba dentro de la línea karaoke, esto es **syl.i**

Recordemos que el número de la capa (**layer**) determina la visibilidad de dos o más objetos que coinciden parcial o totalmente en su posición en el vídeo. Un objeto en el vídeo (ya sea una letra, una sílaba, una línea, una shape o lo que sea que pongamos en nuestras líneas en el **Aegisub**), se verá en frente del otro, sí o solo sí su número de capa es mayor a la del otro ebjeto con el que coincide en su posición o, si tienen el mismo número de capa y su número de línea en el script del **Aegisub** es mayor.

Y para usar esta función en lenguaje **Automation Auto-4** lo único que debemos hacer es escribirla dentro de los ya conocidos signos de admiración:



Es decir que relayer cumple una función muy similar a la celda de texto "layer", pero con la diferencia de que la función relayer se puede usar dentro de las nuevas funciones que se pueden hacer en la celda de texto "variables".

---

retime(modo, add_start, add_end): es una de las funciones más importantes y más usadas del lenguaje **Automation Auto-4**, ya que "**re-timea**" los tiempos de las líneas de efecto generadas y es la función que hace posible que categorizemos nuestros efectos en entradas, efectos se sílaba activa, efectos de salida y demás tipos de efectos dependiendo del **modo** que usemos en la función.

Esta función se puede usar también en el **Kara Effector** exactamente de la misma manera que se usa en el **Aegisub**, con la significativa difierencia de que en el **Kara Effector**, esta función tiene muchos más **modos** que la versión original en **Automation Auto-4**.

Los **modos** de la función **retime** que se pueden usar en **Automation Auto-4** son 10:

1. preline
2. line
3. postline
4. presyl
5. start2syl
6. syl
7. syl2end
8. postsyl
9. sylpct
10. set or abs

Estos 10 **modos** se pueden usar en el **Kara Effector** de la misma manera que se hace en el **Aegisub**, ya sea en lenguaje **LUA** o en **Automation Auto-4**.

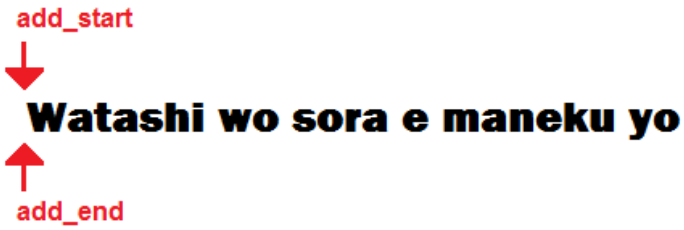
Los modos de la función **retime** que se pueden aplicar en el **Kara Effector** son 29:

1. preline
2. line
3. postline
4. preword
5. start2word
6. word
7. word2end
8. postword
9. presyl
10. start2syl
11. syl
12. syl2end
13. postsyl
14. prefuri
15. start2furi
16. furi
17. furi2end
18. postfuri
19. prechar
20. start2char
21. char
22. char2end
23. postchar
24. linepct
25. wordpct
26. sylpct
27. furipct
28. charpct
29. set o abs

retime("preline", add_start, add_end): el prefijo "pre" es la abreviatura de "previous" (previo, anterior, antes de) y "line" es "línea" en español. O sea que "preline" significa: antes de la línea.

El inicio y final del modo "**preline**" equivalen al tiempo de inicio de la línea karaoke (line.start_time). En este mismo orden de ideas, **add_start** es el tiempo en milisegundos que se agregará o sustraerá al inicio, y **add_end** es el tiempo en milisegundos que se agregará o sustraerá al tiempo final, que como ya lo había mencionado, es el mismo que el tiempo de inicio para este modo. Para sustraer tiempo, ya sea al inicio, al final o a ambos, ponemos valores negativos dentro de la función **retime**.

La siguiente gráfica muestra en dónde se encuentran por default el **add_start** y el **add_end** en el modo "**preline**":



Lo que equivale a: **retime("preline", 0, 0)**

El modo "**preline**" puede ser usado en todo tipo de efecto o **Template Type**. Los recordamos:

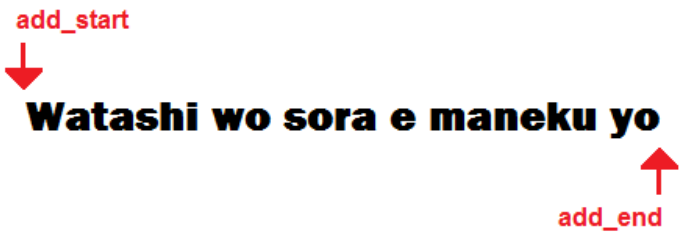
- **Line**
- **Syl**
- **Furi**
- **Char**
- **Translation Line**
- **Translation Word**
- **Translation Char**

Algunos ejemplos de la función **retime** en modo "**preline**":

- **retime("preline", -200, 0)**
- **retime("preline", -300, 200)**
- **retime("preline", 400, 1000)**
- **retime("preline", 0, 2000)**
- **retime("preline", -800, -100)**

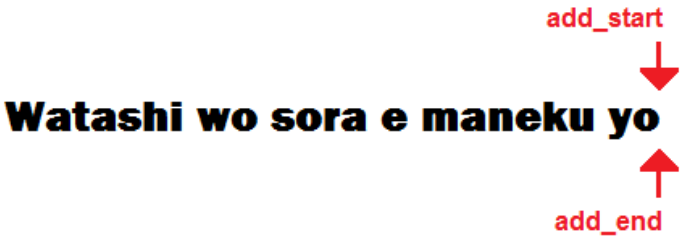
---

retime("line", add_start, add_end): el tiempo de inicio equivale al tiempo de inicio de la línea karaoke (line.start_time) y el tiempo final, al tiempo final de cada una de las líneas de karaoke (line.end_time). Este **modo** está habilitado para todos los **Template Type**.



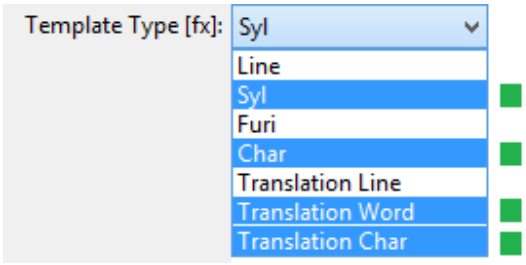
---

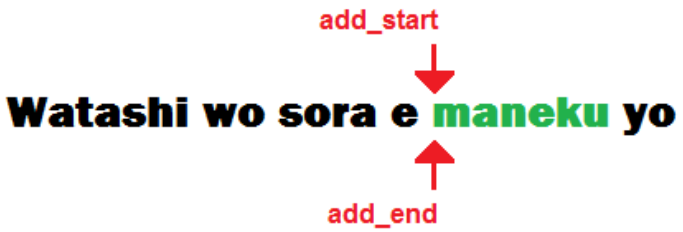
retime("postline", add_start, add_end): "post" de posterior, o sea, el tiempo después de la línea. El tiempo de inicio equivale al tiempo final de la línea karaoke (line.end_time) y el tiempo final, también al tiempo final de cada una de las líneas de karaoke (line.end_time). Este **modo** está habilitado para todos los **Template Type**.



---

retime("preword", add_start, add_end): literalmente "antes de la **palabra**". El tiempo de inicio equivale al tiempo de inicio de cada una de las palabras en cada línea karaoke (line.start_time + word.start_time) y el tiempo final, también al tiempo de inicio de cada una de las palabras en cada línea. Este **modo** está habilitado para los siguientes **Template Type**: Syl, Char, Translation Word y Translation Char.

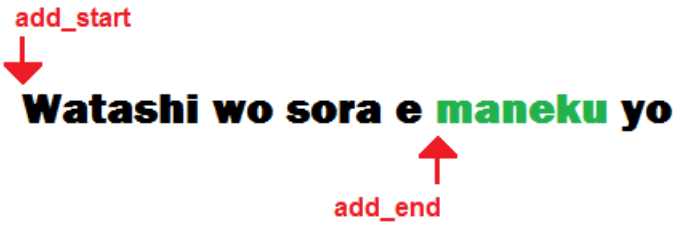




---

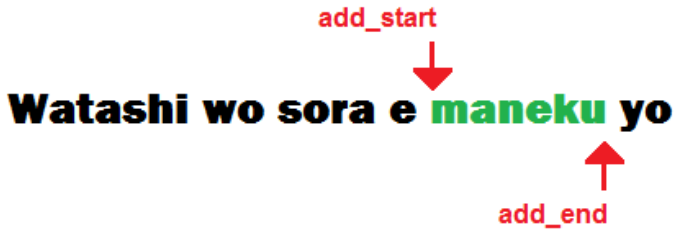
retime("start2word", add_start, add_end): literalmente "desde el inicio hasta la **palabra**". El tiempo de inicio equivale al tiempo de inicio de cada una de las líneas (line.start_time) y el tiempo final, al tiempo de inicio de cada **palabra** (line.start_time + word.start_time).

Este **modo** está habilitado para los siguientes **Template Type**: Syl, Char, Translation Word y Translation Char.



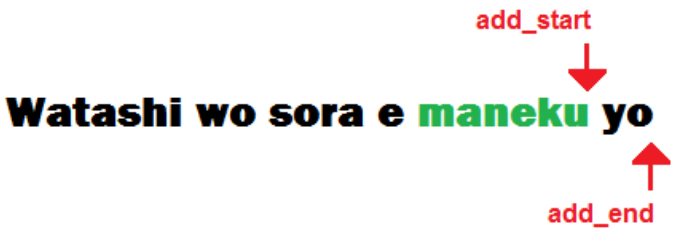
---

retime("word", add_start, add_end): literalmente "**palabra**". El tiempo de inicio equivale al tiempo de inicio de cada una de las palabras en cada línea karaoke (line.start_time + word.start_time) y el tiempo final, al tiempo final de cada una de las palabras (line.start_time + word.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Char, Translation Word y Translation Char.



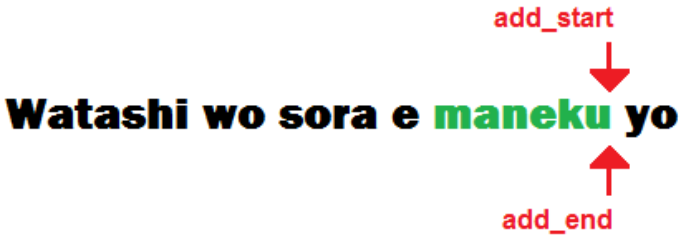
---

retime("word2end", add_start, add_end): literalmente "desde la **palabra** hasta el final". El tiempo de inicio equivale al tiempo final de cada una de las palabras de cada línea karaoke (line.start_time + word.end_time) y el tiempo final, al tiempo final de cada una de las líneas karaoke (line.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Char, Translation Word y Translation Char.



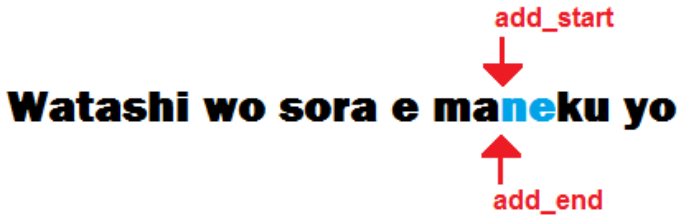
---

retime("postword", add_start, add_end): literalmente "después de la **palabra**". El tiempo de inicio equivale al tiempo final de cada una de las palabras en cada línea karaoke (line.start_time + word.end_time) y el tiempo final, también al tiempo final de cada una de las palabras (line.start_time + word.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Char, Translation Word y Translation Char.



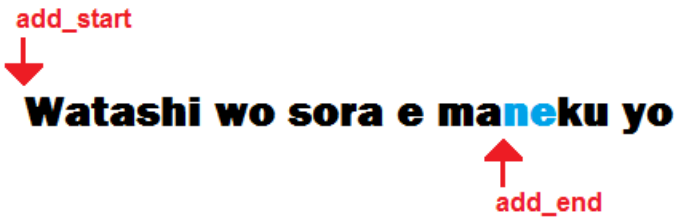
---

retime("presyl", add_start, add_end): literalmente "antes de la **sílaba**". El tiempo de inicio equivale al tiempo de inicio de cada una de las sílabas en cada línea karaoke (line.start_time + syl.start_time) y el tiempo final, también al tiempo de inicio de cada una de las sílabas en cada línea. Este **modo** está habilitado para los siguientes **Template Type**: Syl, Furi, Char y Translation Char.

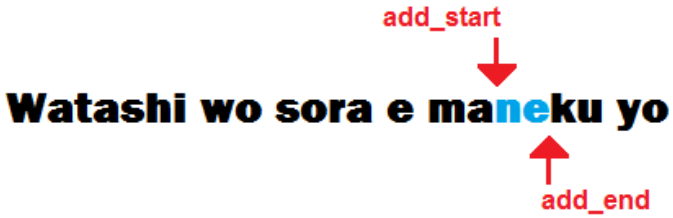


---

retime("start2syl", add_start, add_end): literalmente "desde el inicio hasta la **sílaba**". El tiempo de inicio equivale al tiempo de inicio de cada una de las líneas (line.start_time) y el tiempo final, al tiempo de inicio de cada **sílaba** (line.start_time + syl.start_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Furi, Char y Translation Char.

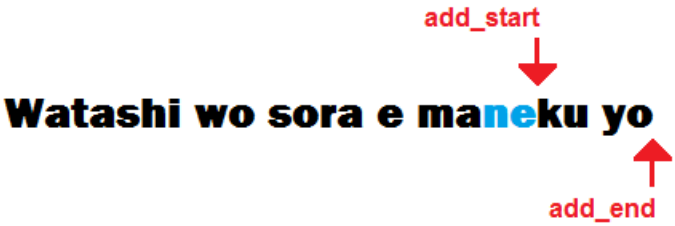


retime("syl", add_start, add_end): literalmente "**sílaba**". El tiempo de inicio equivale al tiempo de inicio de cada una de las sílabas en cada línea karaoke (line.start_time + syl.start_time) y el tiempo final, al tiempo final de cada una de las silabas (line.start_time + syl.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Furi, Char y Translation Char.



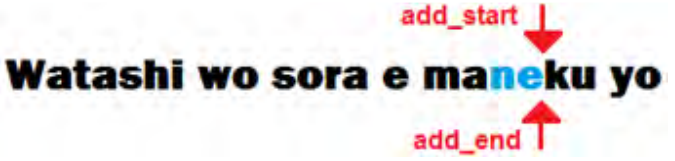
---

retime("syl2end", add_start, add_end): literalmente "desde la **sílaba** hasta el final". El tiempo de inicio equivale al tiempo final de cada una de las sílabas de cada línea karaoke (line.start_time + syl.end_time) y el tiempo final, al tiempo final de cada una de las líneas karaoke (line.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Furi, Char y Translation Char.

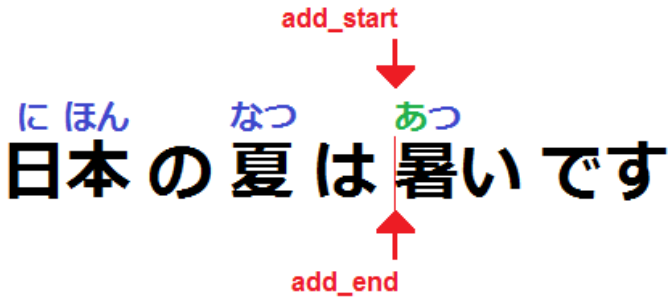


---

retime("postsyl", add_start, add_end): literalmente "después de la **sílaba**". El tiempo de inicio equivale al tiempo final de cada una de las sílabas en cada línea karaoke (line.start_time + syl.end_time) y el tiempo final, también al tiempo final de cada una de las sílabas (line.start_time +syl.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Syl, Furi, Char y Translation Char.

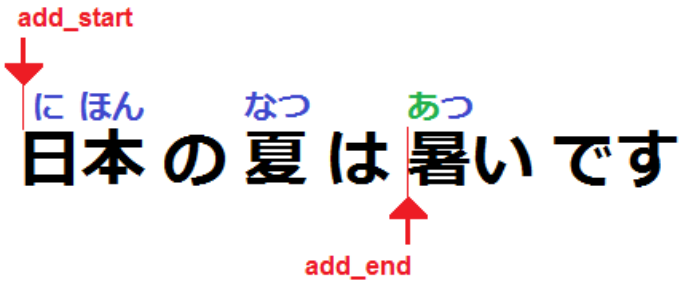


retime("prefuri", add_start, add_end): literalmente "antes del **furigana**". El tiempo de inicio equivale al tiempo de inicio de cada uno de los furiganas en cada línea karaoke (line.start_time + furi.start_time) y el tiempo final, también al tiempo de inicio de cada una de los furiganas en cada línea.



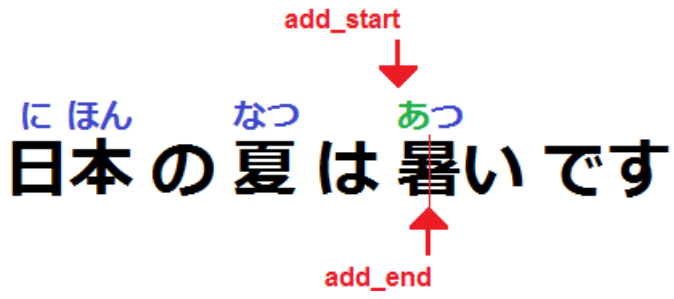
---

retime("start2furi", add_start, add_end): literalmente "desde el inicio hasta el **furigana**". El tiempo de inicio equivale al tiempo de inicio de cada una de las líneas (line.start_time) y el tiempo final, al tiempo de inicio de cada **furigana** (line.start_time + furi.start_time).

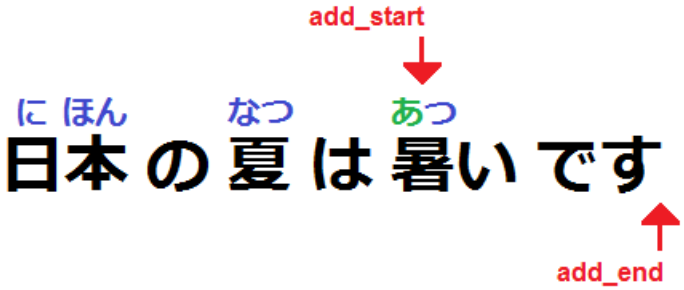


---

retime("furi", add_start, add_end): literalmente "**furigana**". El tiempo de inicio equivale al tiempo de inicio de cada uno de los furiganas en cada línea karaoke (line.start_time + furi.start_time) y el tiempo final, al tiempo final de cada uno de los furigana (line.start_time + furi.end_time).

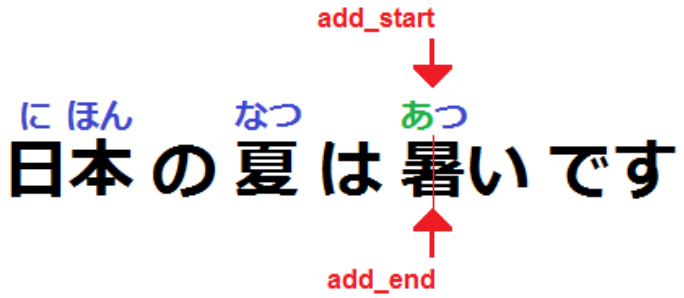


retime(“furi2end”, add_start, add_end): literalmente “desde el **firigana** hasta el final”. El tiempo de inicio equivale al tiempo final de cada uno de los furiganas de cada línea karaoke (line.start_time + furi.end_time) y el tiempo final, al tiempo final de cada una de las líneas karaoke (line.end_time).



---

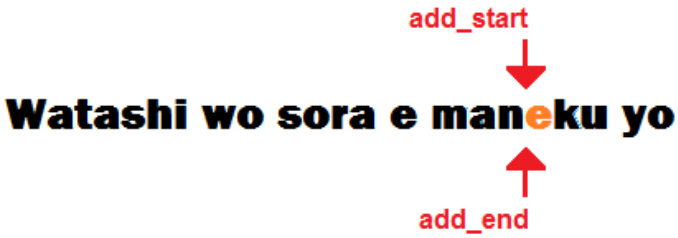
retime(“postfuri”, add_start, add_end): literalmente “después del **furigana**”. El tiempo de inicio equivale al tiempo final de cada uno de los furiganas en cada línea karaoke (line.start_time + furi.end_time) y el tiempo final, también al tiempo final de cada uno de los furiganas (line.start_time + furi.end_time).



Olvidaba mencionar que los 5 modos furiganas solo se pueden aplicar al **Template Type**: Furi, ya que no tiene mucho sentido aplicarle este tipo de modos a los demás **Template Type**.

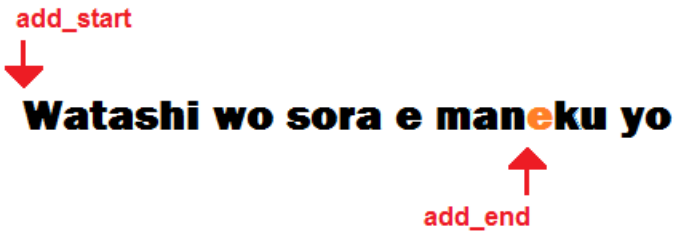
---

retime(“prechar”, add_start, add_end): literalmente “antes del **caracter**”. El tiempo de inicio equivale al tiempo de inicio de cada uno de los caracteres en cada línea karaoke (line.start_time + char.start_time) y el tiempo final, también al tiempo de inicio de cada uno de los caracteres en cada línea. Este **modo** está habilitado para los siguientes **Template Type**: Char y Translation Char.



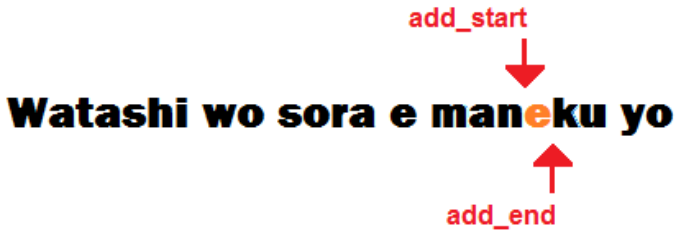
---

retime(“start2char”, add_start, add_end): literalmente “desde el inicio hasta el **caracter**”. El tiempo de inicio equivale al tiempo de inicio de cada una de las líneas (line.start_time) y el tiempo final, al tiempo de inicio de cada **caracter** (line.start_time + char.start_time). Este **modo** está habilitado para los siguientes **Template Type**: Char y Translation Char.



---

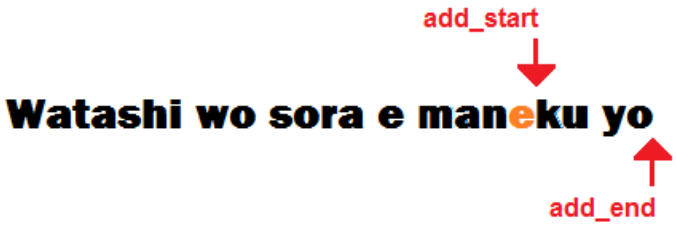
retime(“char”, add_start, add_end): literalmente “**caracter**”. El tiempo de inicio equivale al tiempo de inicio de cada uno de los caracteres en cada línea karaoke (line.start_time + char.start_time) y el tiempo final, al tiempo final de cada uno de los caracteres (line.start_time + furi.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Char y Translation Char.



---

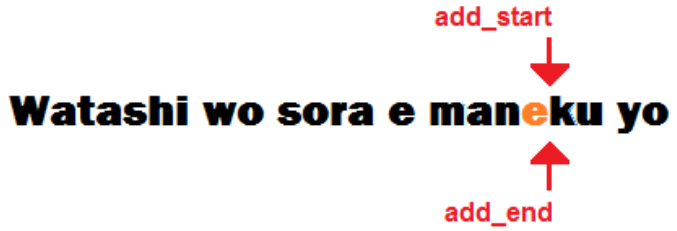
retime(“furi2end”, add_start, add_end): literalmente “desde el **furigana** hasta el final”. El tiempo de inicio equivale al tiempo final de cada uno de los furiganas de cada línea karaoke (line.start_time + furi.end_time) y el tiempo final, al tiempo final de cada una de las líneas karaoke (line.end_time). Este **modo** está

habilitado para los siguientes **Template Type**: Char y Translation Char.



---

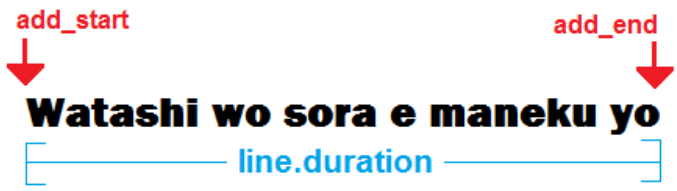
retime("postchar", add_start, add_end):
literalmente "después del **furigana**". El tiempo de inicio equivale al tiempo final de cada uno de los furiganas en cada línea karaoke (line.start_time + furi.end_time) y el tiempo final, también al tiempo final de cada uno de los furiganas (line.start_time + furi.end_time). Este **modo** está habilitado para los siguientes **Template Type**: Char y Translation Char.



A continuación veremos los 5 modos "**pct**" de la función **retime**, que son: "**linepct**", "**wordpct**", "**sylpct**", "**furipct**" y "**charpct**".

"**pct**" es la sigla de "**percent**" (porcentaje en español). El 0% equivale al inicio del modo y el 100% al tiempo final, o sea, al total de la duración.

retime("linepct", 0, 100):
usada con estos dos valores (0 y 100) es lo mismo que el modo "line".



La diferencia real es cuando se usan otros valores para que se adicionen o resten del tiempo inicial de la línea.

Ejemplo:

- line.start_time = 1200
- line.duration = 3600 ms
- line.end_time = 4800
- **retime("linepct", 0, 50)**

0% de 3600 = 0

50% de 3000 = 1800, entonces:

1200 + 0 = 1200 ← sería el tiempo de inicio del re-timeo

1200 + 1800 = 3000 ← sería el tiempo final del re-timeo

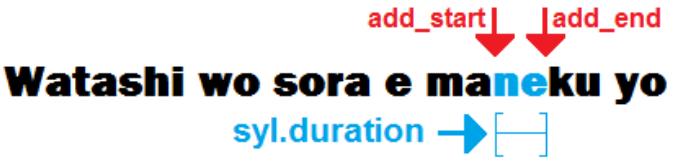
De manera muy similar funcionan los otros 4 modos "pct".

---

retime("wordpct", 0, 100):
usada con estos dos valores (0 y 100) es lo mismo que el modo "word".

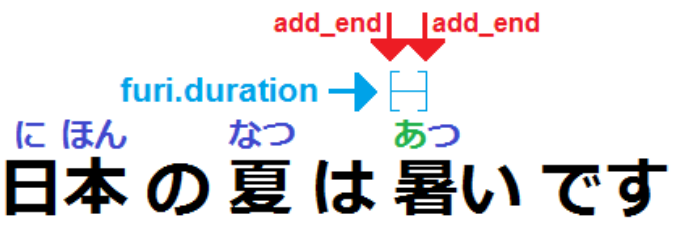
---

retime("sylpct", 0, 100):
usada con estos dos valores (0 y 100) es lo mismo que el modo "syl".



---

retime("furipct", 0, 100):
usada con estos dos valores (0 y 100) es lo mismo que el modo "furi".



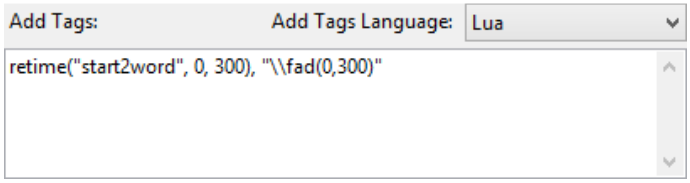
retime("charpct", 0, 100):
usada con estos dos valores (0 y 100) es lo mismo que el modo "char".

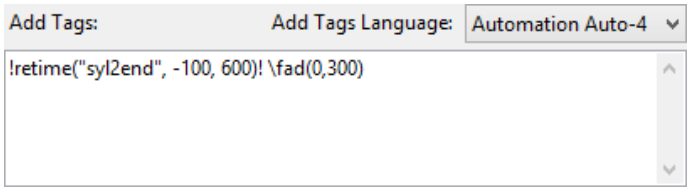
Y por último, está el modo "**set**" o "**abs**", que es lo mismo. Y hace referencia al tiempo absoluto con respecto al cero del vídeo, ejemplo:

retime("set", 1200, 23100): entonces el tiempo de inicio del re-timeo será 1200 ms y el tiempo final será 23100 ms.

No pareciera ser sencillo memorizar los **29 modos** de la función **retime**, pero es solo cuestión de prática y de un poco de sentido común dado el nombre de cada uno de los modos. Y como ya lo había mencionado, esta función es de gran utilidad y más para aquellos que ya la han usado en **Automation Auto-4** y están familiarizados con los 10 modos por default del Aegisub.

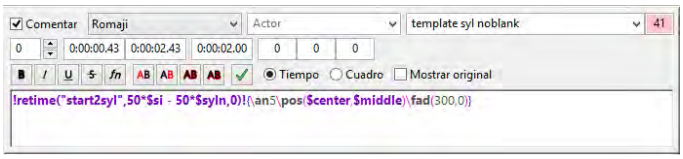
Estos son dos cortos ejemplos de cómo usar la función **retime** dependiendo del lenguaje que más nos guste, **LUA** o **Automation Auto-4**:

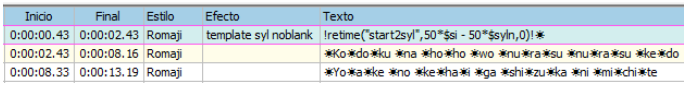




Y a manera de práctica, pueden usar el resto de los modos para que sepan qué uso darle acada uno de ellos y crear efectos de entrada (**lead-in**), de sílaba activa (**hi-light**), de salida (**lead-out**) o la combinación que deseen. Las posibilidades son muchas y los efectos también.

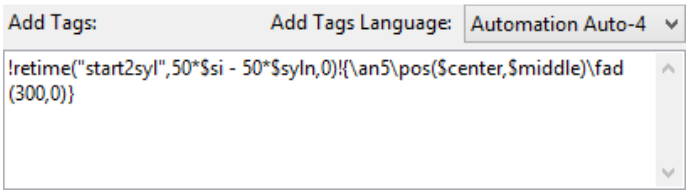
El **Kara Effector** también nos da la posibilidad de pegar en él algún **Template** copiado desde el **Aegisub.** Ejemplo:





Inicio	Final	Estilo	Efecto	Texto
0:00:00.43	0:00:02.43	Romaji	template syl noblank	!retime("start2syl",50*$si - 50*$syln,0)!*
0:00:02.43	0:00:08.16	Romaji		*Ko*do*ku *na *ho*ho *wo *nu*ra*su *nu*ra*su *ke*do
0:00:08.33	0:00:13.19	Romaji		*Yo*a*ke *no *ke*ha*i *ga *shi*zu*ka *ni *mi*chi*te

En el caso de que nos guste este Template en particular, que de casualidad es de un Efecto que nos gusta, entonces lo copiamos tal cual y lo pegamos en el **Kara Effector** usando el lenguaje **Automation Auto-4**, así:



Y al aplicar el Efecto, el resultado será el mismo que en el **Aegisub**, lo que implica una gran ventaja desde donde se lo mire.

El **Tomo VIII** se despide con la recomendación de poner en práctica los ejemplos vistos en él. No olviden descargar la más reciente actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia.

Kara Effector 3.2:

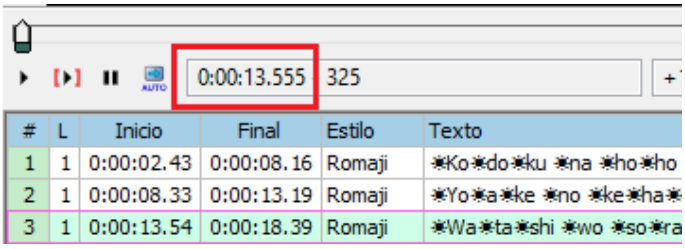
En el **Tomo IX** seguiremos profundizando en las librerías del **Kara Effector**, ya que eso nos dará las herramientas necesarias para aumentar nuestro nivel al momento de hacer un Efecto.

El **Kara Effector** tiene muchas funciones, pero no todas ellas son para hacer Efectos. Hay algunas funciones que nos sirven de apoyo para crea otras nuevas, también hay funciones que hacen que el **Kara Effector** pueda llevar a cabo su tarea y hay otras que nos ayudan en la generación de los Efectos.

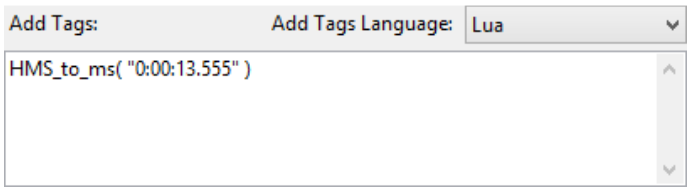
En las librerías que veremos a continuación hay de todo tipo de funciones y para todos los gustos. Como siempre digo, a la final todo depende de qué queremos hacer para tener claro por cuál función no decidiremos o cuál es la que mejor se ajusta a nuestro proyecto.

Librería:
Funciones de Tiempo [KE]

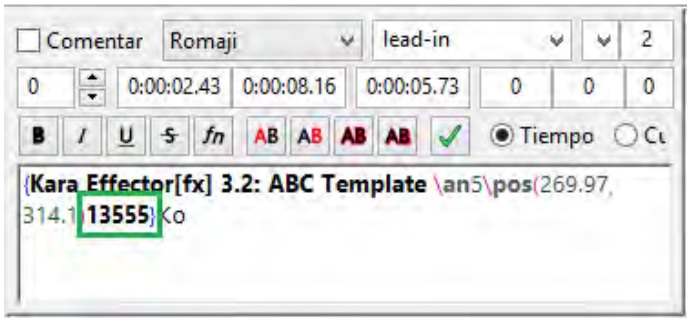
HMS_to_ms(Time_HMS): esta función convierte un tiempo dado, de formato **HMS** (horas, minutos y segundos) a formato ms (milisegundos). El tiempo **HMS** debe ser ingresado entre comillas, ya sean comillas sencillas o dobles. Ejemplo:



Copiamos el tiempo que está dentro del recuadro rojo en la anterior imagen, y lo pegamos dentro de la función, sin olvidar colocarlo entre comillas:



Y la función convertirá ese tiempo que estaba en formato **HMS** a formato ms, cuyo resultado cuenta como un valor numérico:

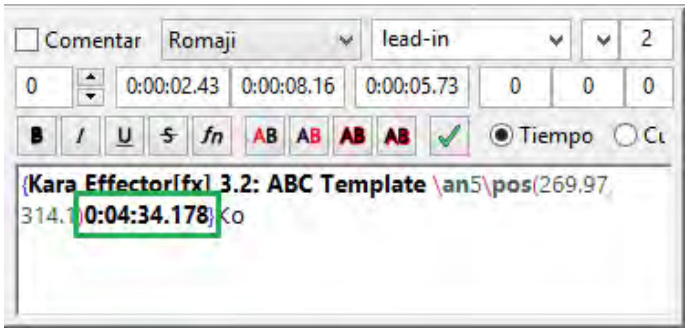


O sea que: **HMS_to_ms("0:00:13.55") = 13555 ms**

---

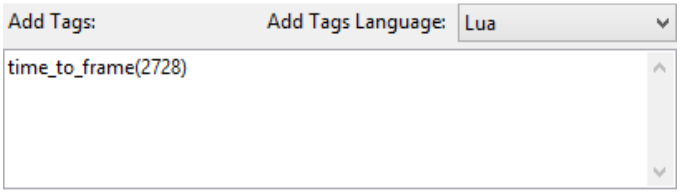
ms_to_HMS(Time_ms): esta función convierte un tiempo dado, de formato ms (milisegundos) a formato **HMS** (horas, minutos y segundos). El tiempo ms debe ser ingresado como un valor numérico, sin comillas. Ejemplo:

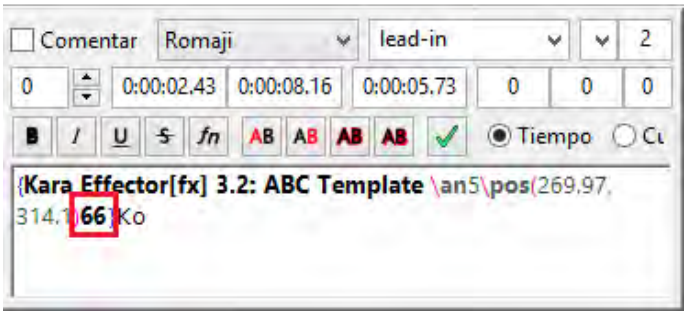
Obtenemos:



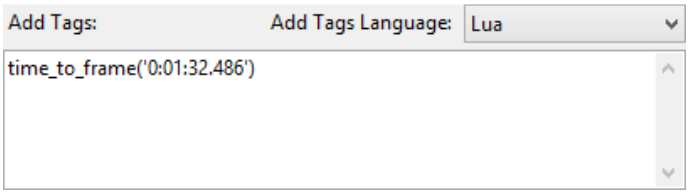
time_to_frame(time): convierte un tiempo dado, ya sea en formato **HMS** o ms, en la cantidad de "frames" (cuadros) que ocuparía ese tiempo en el vídeo que se está usando para aplicar un Efecto.

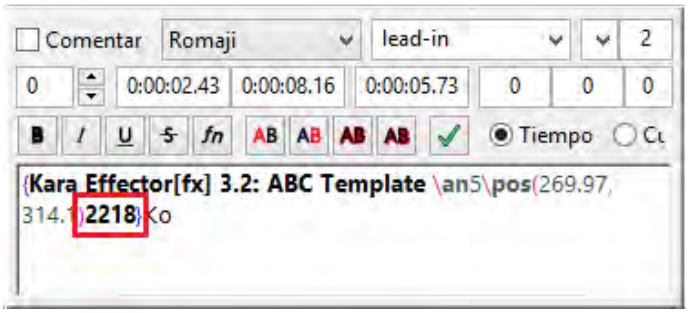
Ejemplo con tiempo en formato ms (milisegundos):





Ejemplo con tiempo en formato **HMS**:

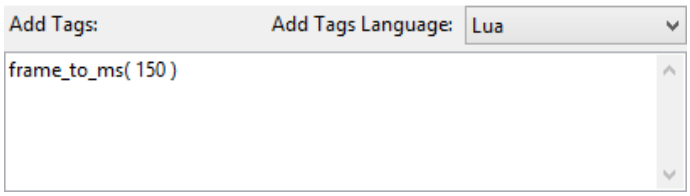


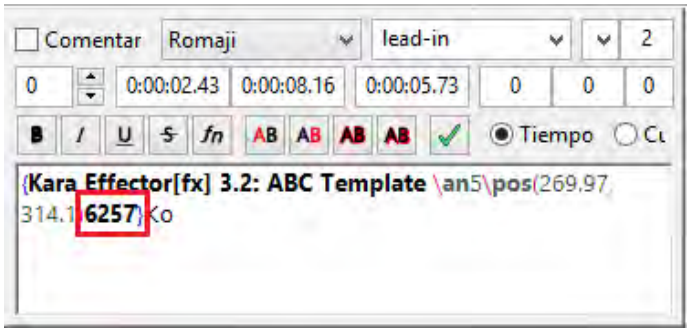


De lo anterior se concluye que 2728 ms equivalen a 66 frames (cuadros), y que en "0:01:32:486" (1 minuto, 32 segundos y 486 ms) hay 2218 frames.

---

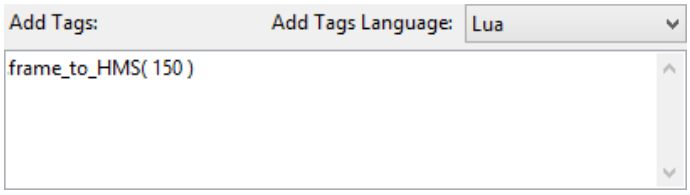
frame_to_ms(frames): esta función convierte el número de la cantidad de frames a ms (milisegundos). La cantidad de frames debe ser ingresada como un valor numérico, sin las comillas, ejemplo:

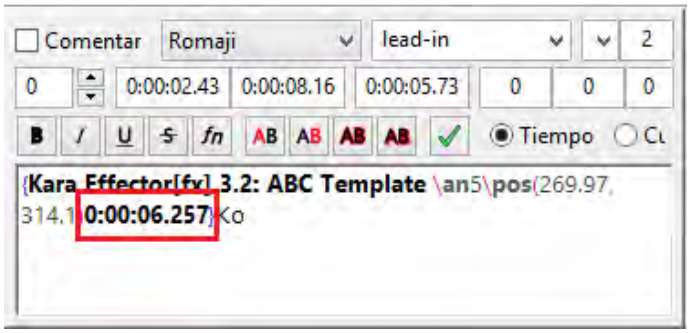




---

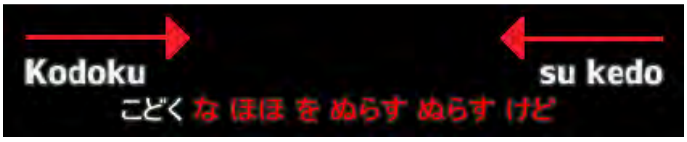
frame_to_HMS(frames): esta función convierte el número de la cantidad de frames a un tiempo en formato **HMS** (horas, minutos y segundos). La cantidad de frames debe ser ingresada como un valor numérico, sin las comillas, ejemplo:





---

time_mid1(delay): esta función, dependiendo el tipo de Efecto (este y los siguientes ejemplos están hechos con **Template Type: Syl**, pero se puede usar para todos los tipos menos el Line y Template Line), si se usa en el tiempo de inicio de la Línea, hace que las sílabas aparezcan de forma progresiva, desde los extremos hacia en centro de la Línea, como se puede apreciar en el siguiente ejemplo:



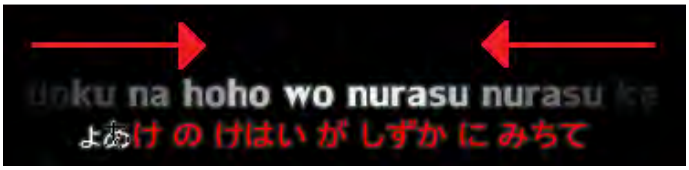
Esta función adicionada al tiempo de inicio de la Línea, nos sirve para hacer efectos **lead-in** (efectos de entrada). El tiempo que separa la aparición de una sílaba con respecto de la otra, es el **delay** (retraso), que para este ejemplo fue de 50 ms:

25	0	0:00:02.89	0:00:08.16	Romaji	lead-in	Effector [Fx]	*Ko
26	0	0:00:02.94	0:00:08.16	Romaji	lead-in	Effector [Fx]	*do
27	0	0:00:02.99	0:00:08.16	Romaji	lead-in	Effector [Fx]	*ku
28	0	0:00:03.04	0:00:08.16	Romaji	lead-in	Effector [Fx]	*na
29	0	0:00:03.09	0:00:08.16	Romaji	lead-in	Effector [Fx]	*ho
30	0	0:00:03.14	0:00:08.16	Romaji	lead-in	Effector [Fx]	*ho

Lo que muestra la imagen anterior es que la diferencia entre 0:00:02.890 y 0:00:02.940 es de 50 ms, que fue el **delay** usado para este ejemplo. Y como ya se había dicho antes, esta función también puede ser usada con los **Template Type**: Furi, Char, Translation Char y Translation Word; con resultados similares.

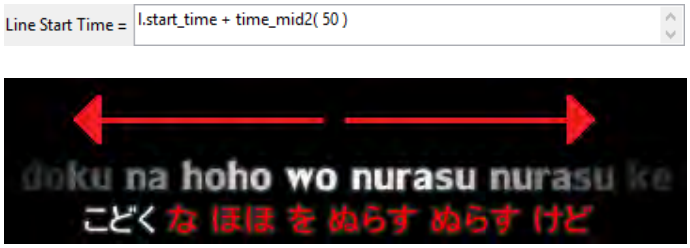
A mayor **delay**, mayor será el tiempo en que aparezca una sílaba (caracter o palabra, dependiendo del Template Type) con respecto de la inmediatamente anterior. O sea que uno decide el retraso de la función dependiendo del resultado que queremos obtener.

La función **time_mid1(delay)** usada en el tiempo final de la Línea, nos ayuda a hacer efectos **lead-out** (de salida) y hace que las sílabas desaparezcan desde los extremos hacia el centro de la línea, ejemplo:



O sea que la función **time_mid1(delay)** nos sirve para hacer efectos, tanto **lead-in** como **lead-out**, dependiendo en qué tiempo de la Línea la usemos: **Line Start Time** o **Line End Time**, tiempo de inicio y final, respectivamente.

time_mid2(delay): esta función, dependiendo el tipo de Efecto, si se usa en el tiempo de inicio de la Línea, hace que las sílabas aparezcan de forma progresiva, desde el centro hacia los extremos de la Línea. Es la función opuesta a **time_mid1**, como se puede apreciar en el siguiente ejemplo:



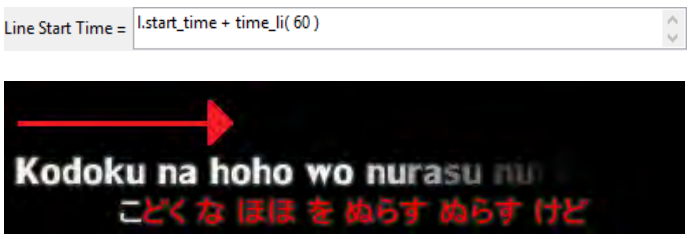
También se puede usar esta función en los Template Type: Furi, Char, Tranlation Char y Translation Word

La función **time_mid2(delay)** usada en el tiempo final de la Línea, nos ayuda a hacer efectos **lead-out** (de salida) y hace que las sílabas desaparezcan desde el centro hacia los extremos de la Línea. Ejemplo:

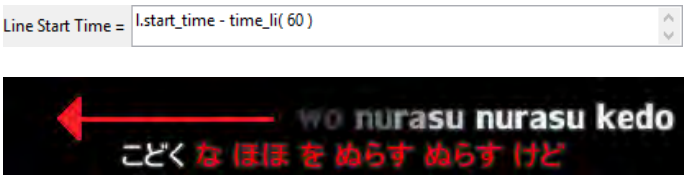
---

time_li(delay): esta función es similar a las dos funciones anteriormente vistas. Esta función se puede sumar o restar al tiempo de inicio de la Línea y por eso su nombre: **time_li** (**time lead-in**). Y al igual que las dos anteriores funciones, se puede usar en los **Template Type**: Syl, Furi, Char, Template Char y Translation Word. Para este ejemplo usaré **Template Type**: Syl

Modo suma: hace que las sílabas aparezcan de forma progresiva de izquierda a derecha en la Línea.



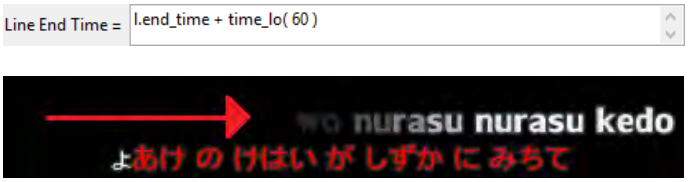
Modo resta: hace que las sílabas aparezcan de forma progresiva de derecha a izquierda en la Línea.



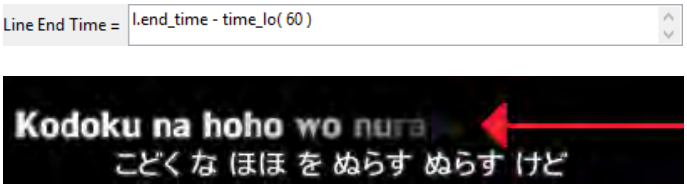
---

time_lo(delay): es similar a **time_li**. Esta función se puede sumar o restar al tiempo final de la Línea y por eso su nombre: **time_lo** (**time lead-out**). Y al igual que las anteriores funciones, se puede usar en los Template Type: Syl, Furi, Char, Template Char y Translation Word.

Modo suma: hace que las sílabas desaparezcan de forma progresiva de izquierda a derecha en la Línea.



Modo resta: hace que las sílabas desaparezcan de forma progresiva de derecha a izquierda en la Línea.



Con esta función culmina la librería de las funciones de tiempo del Kara Effector y da paso a la próxima librería.

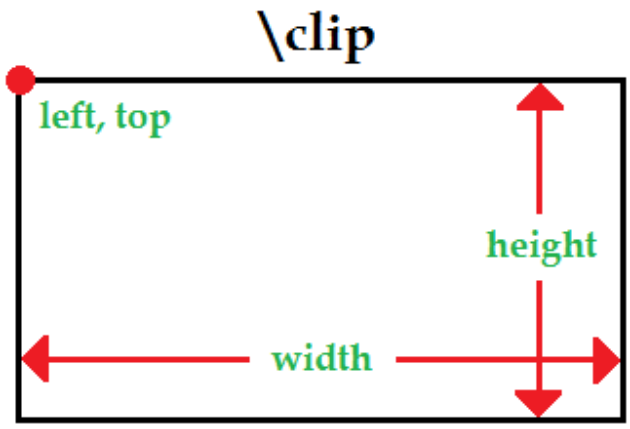
Librería: tag [KE]

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 supeior izquierdo del rectángulo que lo generará.

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



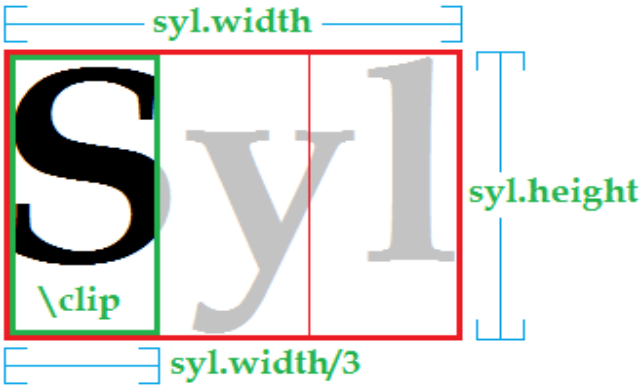
Veamos algunos ejemplos para empezar a aclarar los conceptos hasta acá vistos:

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

```
tag.clip( syl.left, syl.top, syl.width, syl.height )
```

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

```
tag.clip( )
```

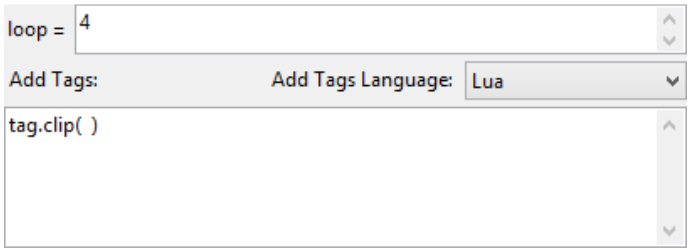
Es decir, que los valores por default que usará la función dependen del Template Type.

> Para Template Type: **Line** y **Translation Line**, los valores por deafult 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 deafult 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 usemos en la celda de texto "**loop**". Ejemplo:

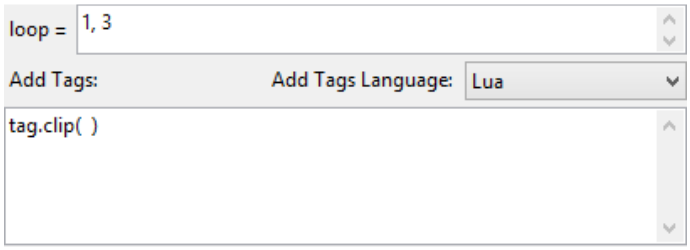
- **loop**: 4
- Template Type: **Translation Word**



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 dimenciones 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**

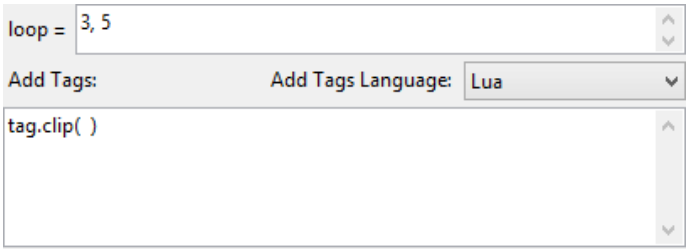
- **loop**: 1, 3
- Template Type: **Char**

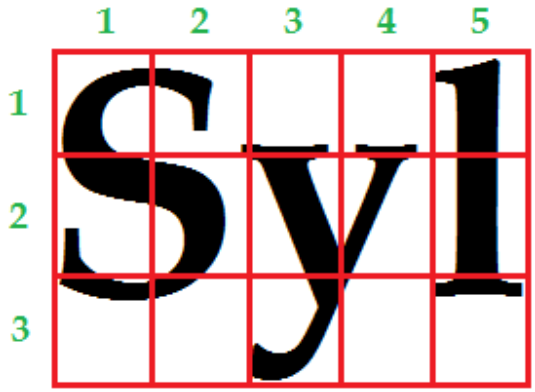


Lo que hace el loop usado de ese modo es que la función **tag.clip** cree **clip's verticales** dentro del rectángulo hecho por las medidas ingresadas y como es un **Template Type**: Char y usamos los valores por default de la función, hará que cada carácter quede dentro de los tres clip's creados.

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

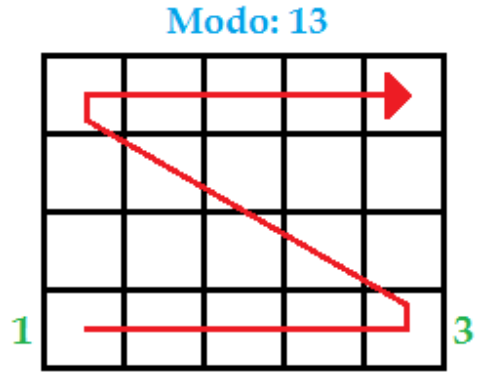
- **loop:** 3, 5
- Template Type: **Syl**



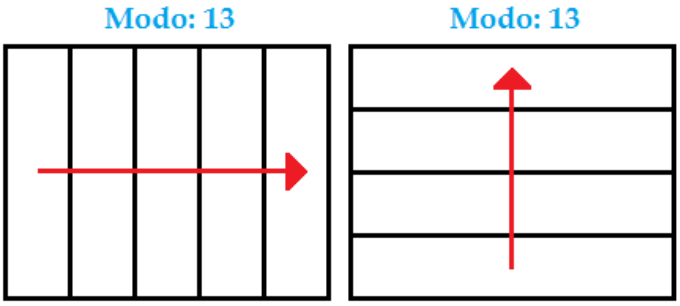


La retícula creada por la función **tag.clip** será de **3** clip´s horizontales por **5** verticales y la cantidad total del **loop** será: 3 x 5 = 15

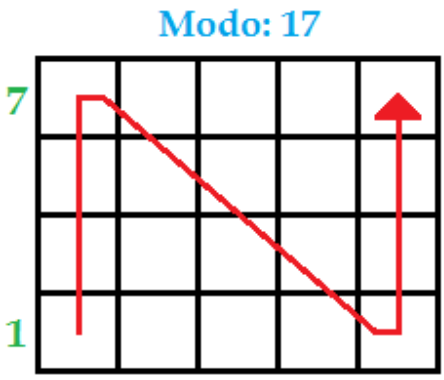
El parámetro **mode** en la función **tag.clip** es un número que asiganmos con el fin de determinar el orden de los clip's. Son **8 modes** y los veremos a cada uno de ellos:



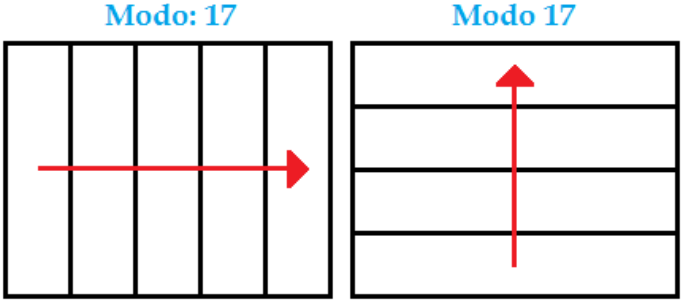
- **mode 13:** la imagen anterior muestra el orden de los clip´s en el **mode 13** de un **tag.clip** reticulado. En el caso de los clip´s verticales, el **mode 13** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería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



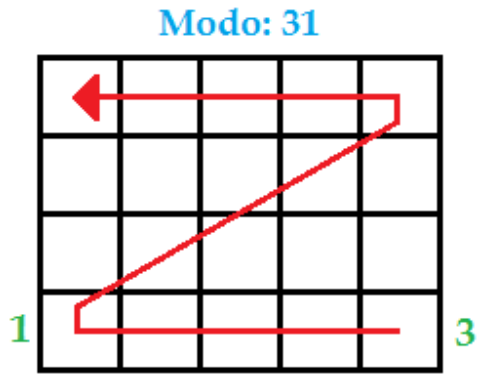
- **mode 17:** para un **tag.clip** reticulado, el orden de los clip's es como el de la siguiente imagen, es decir, el primre clip es el de la esquina inferior izquirda y el último es el de la esquina supeior derecha, siguiendo la trayectoria de la gráfica:



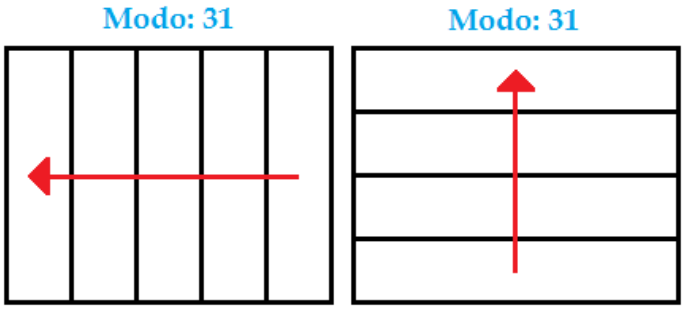
En el caso de los clip´s verticales, el **mode 17** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:



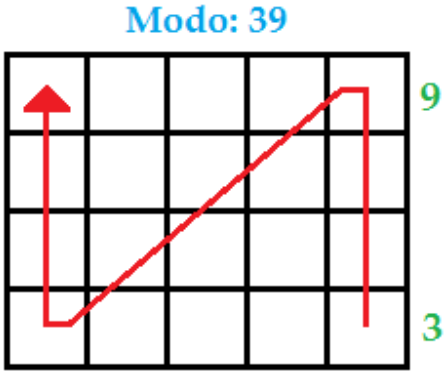
- **mode 31:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina inferior derecha y el último es el de la esquina supeior izquierda, siguiendo la trayectoria de la gráfica:



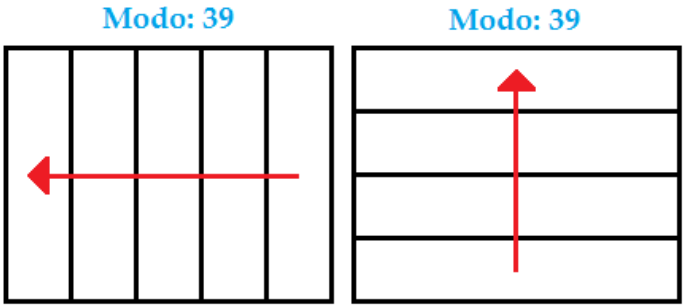
En el caso de los clip´s verticales, el **mode 31** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:



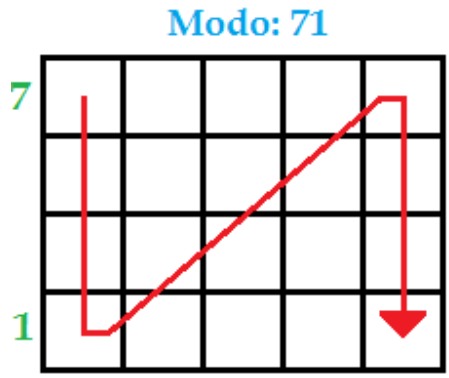
- **mode 39:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina inferior derecha y el último es el de la esquina supeior izquierda, siguiendo la trayectoria de la gráfica:



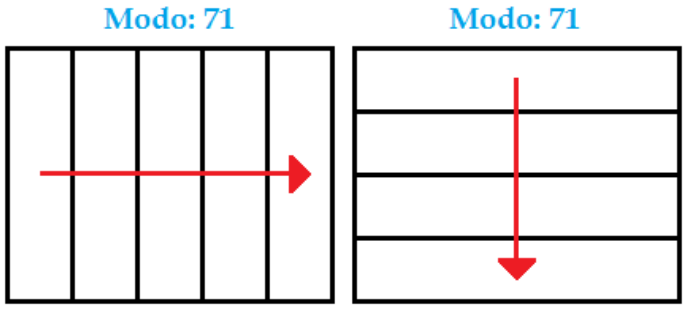
En el caso de los clip´s verticales, el **mode 39** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo inferior y el último el del extremo superior:



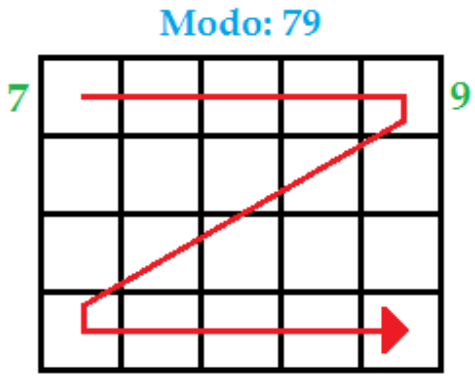
- **mode 71:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina superior izquierda y el último es el de la esquina inferior derecha, siguiendo la trayectoria de la gráfica:



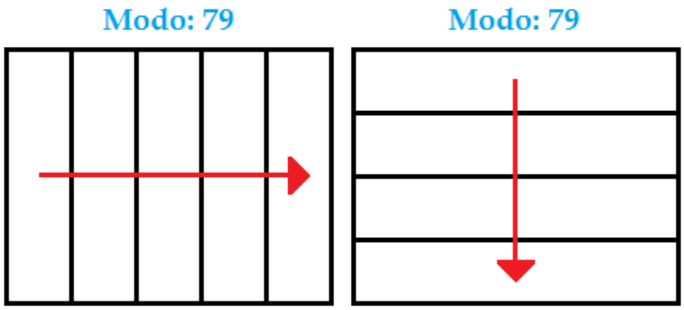
En el caso de los clip´s verticales, el **mode 71** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:



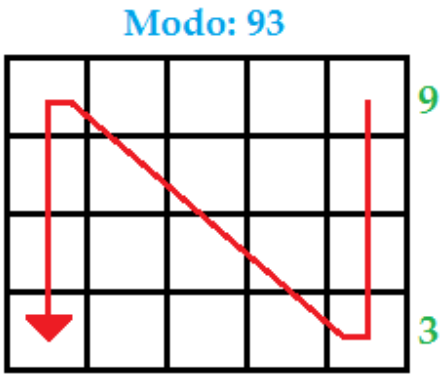
- **mode 79:** es el modo por default de la función. para un **tag.clip** reticulado, el orden de los clip's sería el siguiente; el primer clip es el de la esquina superior izquierda y el último es el de la esquina inferior derecha, siguiendo la trayectoria de la gráfica:



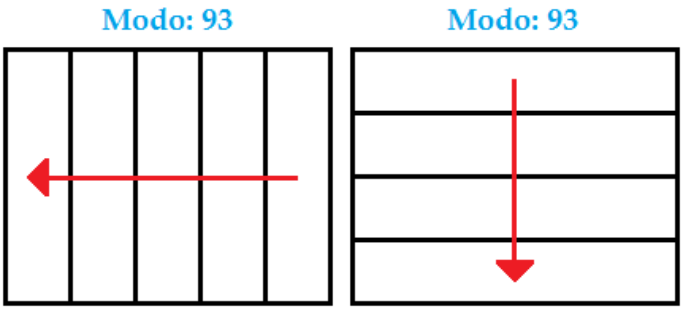
En el caso de los clip´s verticales, el **mode 79** hace que el primer clip sea el del extremo izquierdo y el último el del extremo derecho. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:



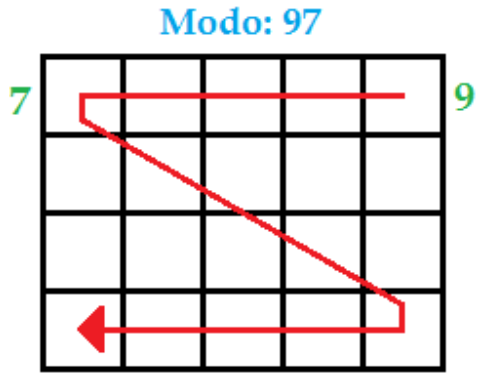
- **mode 93:** para un **tag.clip** reticulado, el orden de los clip's sería; el primer clip es el de la esquina superior derecha y el último es el de la esquina inferior izquierda, siguiendo la trayectoria de la gráfica:



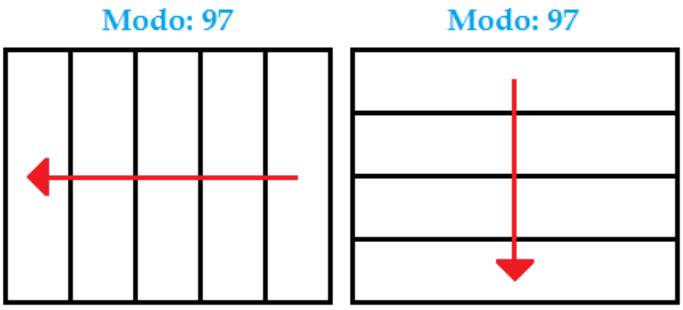
En el caso de los clip´s verticales, el **mode 93** hace que el primer clip sea el del extremo derecho y el último el del extremo izquierdo. Para los clip's horizontales el primero sería el del extremo superior y el último el del extremo inferior:



- **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:



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 eligir según sea nuestra necesidad en un Efecto.

Recordemos que el número **mode** lo escribirimos 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**)

Es todo por el momento y damos por terminado el **Tomo IX**. En el **Tomo X** del **Kara Effector** continuaremos no solo con la función **tag.clip** sino también con el resto de la Librería "**tag**", ya que vale la pena dedicarle más de tiempo y espacio al estudio de estas funciones y todos los efectos que podemos hacer con ellas.

Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia.

Kara Effector 3.2:

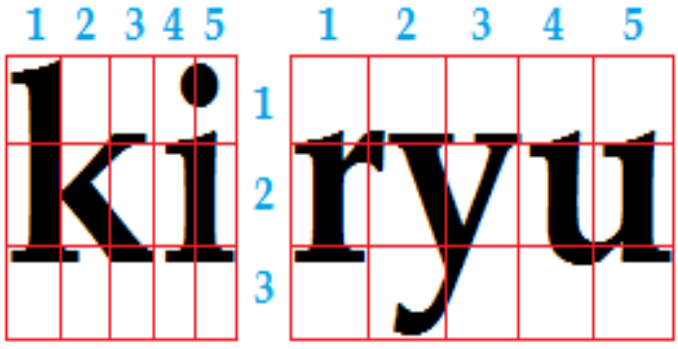
Como lo había mencionado al final de la anterior entrega, este **Tomo X** es la continuación de la librería "**tag**", que como las demás librerías del **Kara Effector**, es importante que sepamos en qué consiste cada una de sus funciones para poder sacarle el máximo provecho posible.

Sin más, retomaremos la librería "**tag**" en la función en donde nos habíamos quedado; la función **tag.clip**

Librería tag [KE]:

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 cuadricula con la función **tag.clip** y es de la siguiente forma, ejemplo:

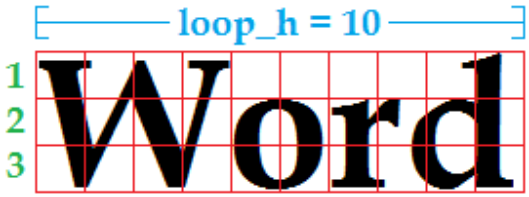
Pero este método no garantiza que los clip´s tengan las mismas dimensiones, es decir que sean cuadrados:

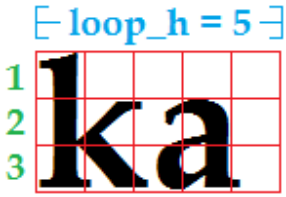


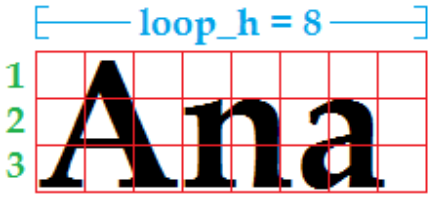
En la imagen anterior vemos cómo para algunas sílabas las porporciones de los clip's no son para nada cuadradas y eso es porque la cantidad de clip´s verticales es constante, lo que no es ideal dados los distintos anchos de la Líneas, Palabras, Sílabas, Caracteres y demás.

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

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:



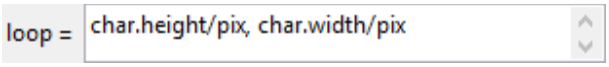




Entonces el **loop_h** varía en todos los casos con el fin de que los clip's den la función **tag.clip** sean cuadrados. La segunda forma de hacerlo es:

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

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



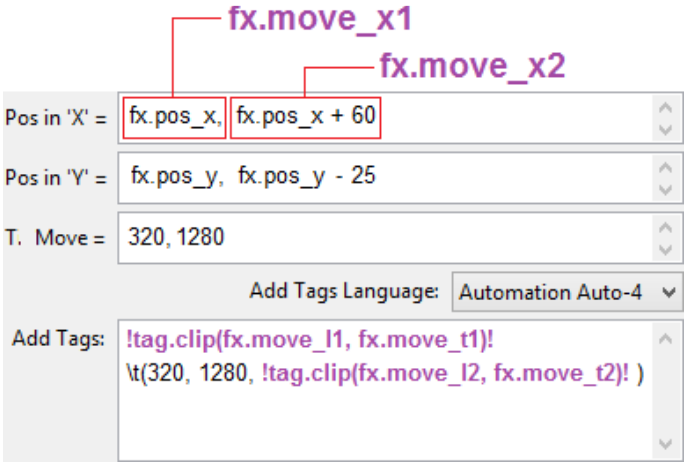
Si por ejemplo, un caracter mide 70 pixeles de ancho por 40 de alto, tendríamos:

- 40/pix = 40/10 = 4
- 70/pix = 70/10 = 7
- 4 x 7 = 28

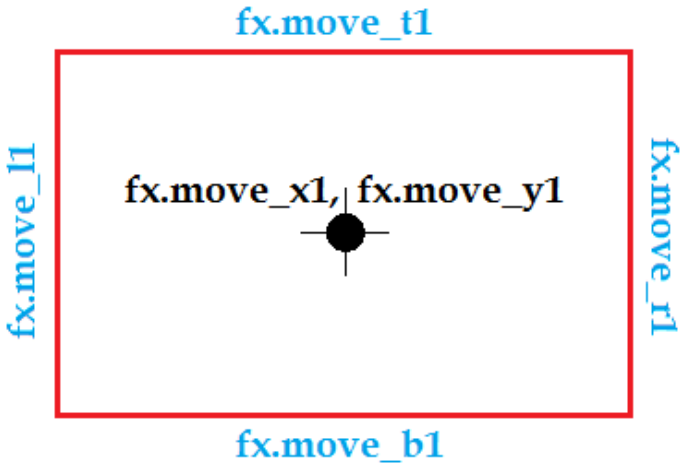
O sea que obtendíamos 28 clip's de 10 x 10 pixeles de la función **tag.clip**, que a su vez sería la cantidad del **loop**, es decir que **maxj** = 28

A menor tamaño de los clip´s, mayor será la cantidad del loop, y a moyor 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.



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 dimenciones 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 las función **tag.clip**, pero con la gran diferencia que siempre genera el mismo clip sin impotar 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 funcion de la libería "**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 la 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:

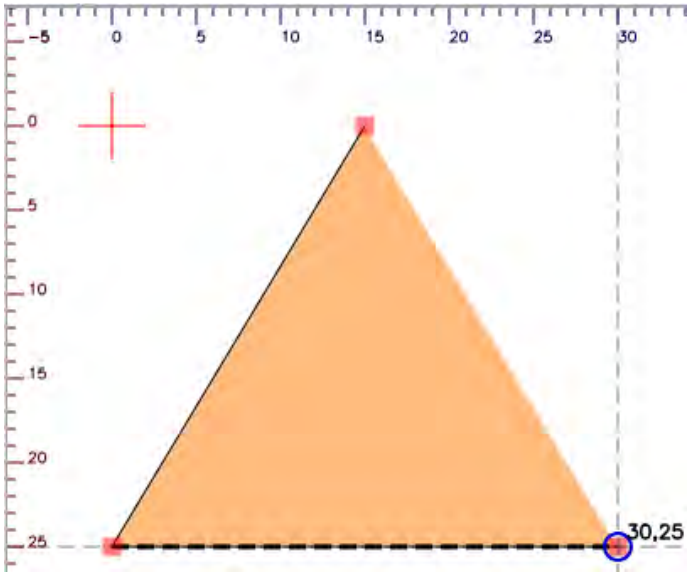
- **x:** coordenada con respecto al eje "x" que hace referencia a la posición en donde estará el centro de la **shape**, su valor por default es **fx.move_x1**
- **y:** coordenada con respecto al eje "y" que hace referencia a la posición en donde estará el centro de la **shape**, su valor por default es **fx.move_y1**
- **Dx:** cantidad de desplazamiento en pixeles con respecto al centro de la **shape**, con referencia al eje "x". Su valor por deafault 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 pixeles con respecto al centro de la **shape**, con referencia al eje "y". Su valor por deafault es **fx.move_y2 – fx.move_y1**

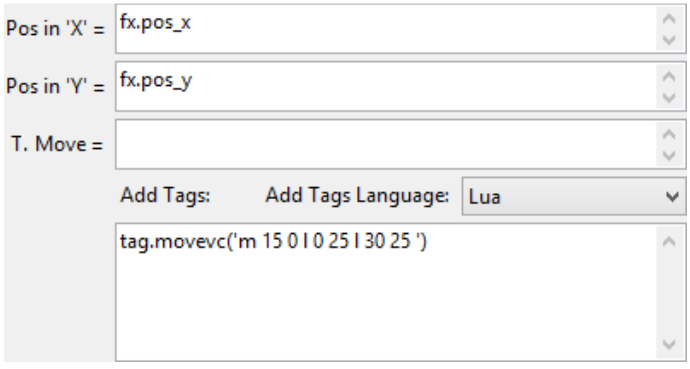
  De no existir un **fx.move_y2**, recordemos que su valor por default es **fx.move_y1**, por lo que el valor por default del parámetro **Dy**, en este caso, sería: **fx.move_y1 – fx.move_y1 = 0**

- **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.movet_i**
  De no existir un **fx.movet_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**

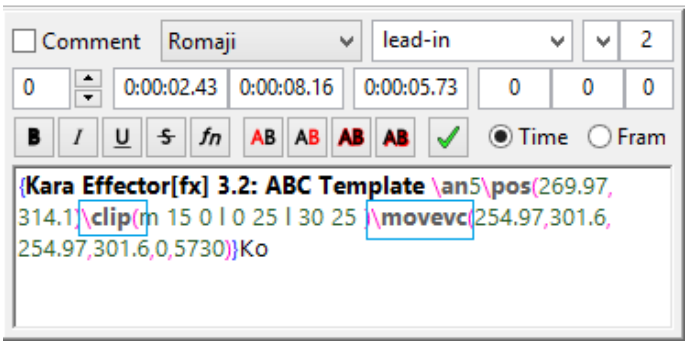
Para el siguiente ejemplo usaré un simple triángulo hecho en el **AssDraw3**, como lo podemos ver en esta imagen:



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



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:

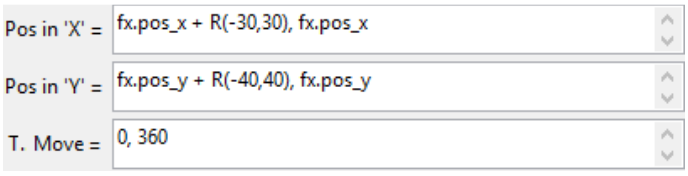


Como el tag de posición es un **\pos** entonces el **\movevc** solo posiciona al clip. Ahora veremos cómo queda el clip del triángulo en cada una de las Sílabas de la Línea:

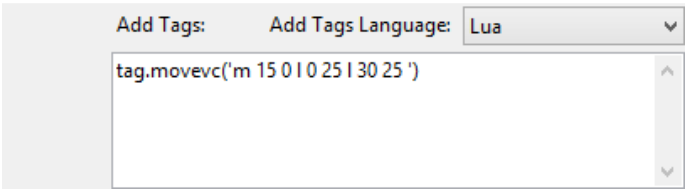
Como el triángulo es mucho más pequeño que el tamaño de cada una de las sílabas, entonces éstas no alcanzan a ser totalmente visibles:

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:



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



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:

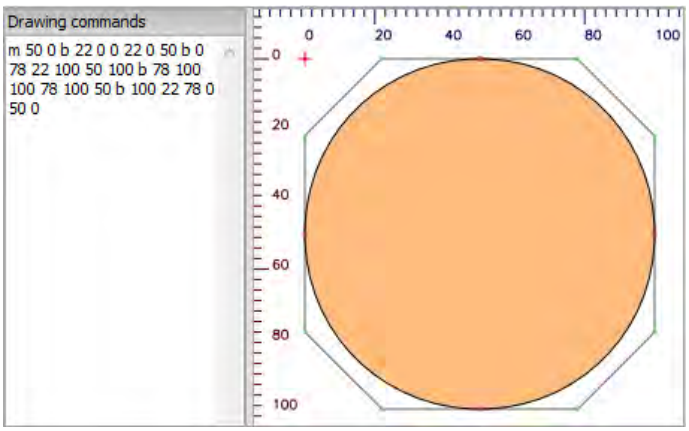
Que luego de 360 ms las sílabas y los clip´s quedaran en su posición narutal:

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.

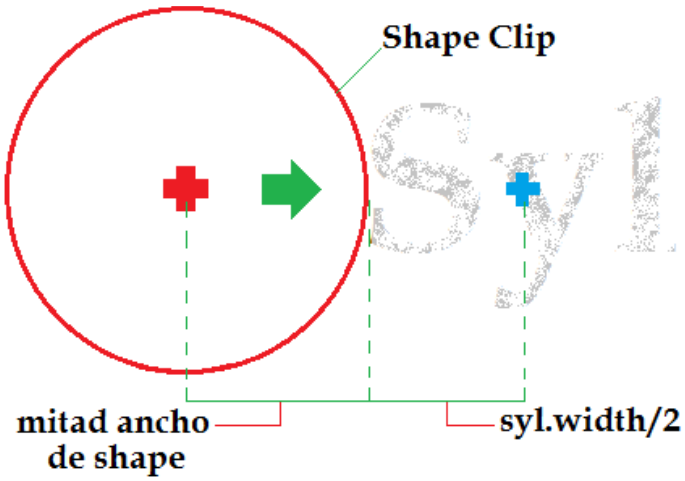
Ahora veremos un ejemplo de objeto estático y clip móvil, es decir, en donde tengamos que usar el resto de los parámetros de la función **tag.movevc**:

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

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:

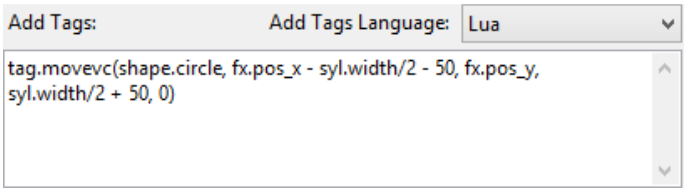


Dicha posición inicial de clip sería:

- **fx.pos_x – syl.width/2 – 50, fx.pos_y**

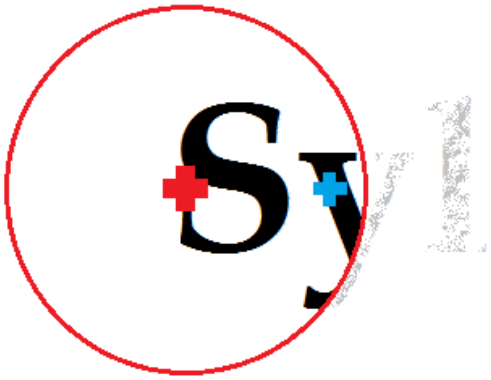
Y la posición final del clip será:

- **fx.pos_x, fx.pos_y**



- **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.

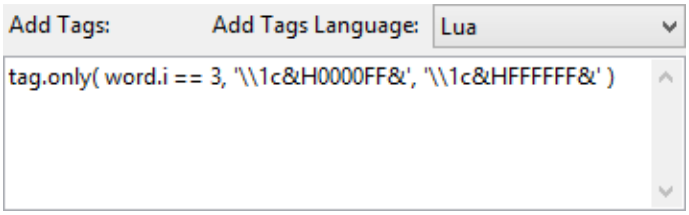
---

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**.

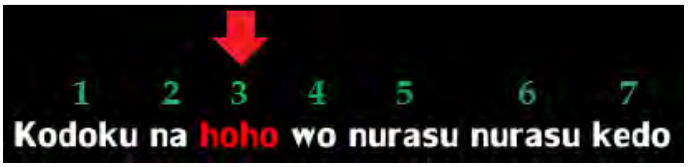
Veamos los tipos de condiciones que podemos usar en la función **tag.only**:

- == 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 decidirnos 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.

Con el final de la función **tag.only** se da por terminado el **Tomo X**, pero no la librería “tag”, ya que aún nos resta por ver unas cuantas funciones más de ella.

En el **Tomo XI** del **Kara Effector** continuaremos con el resto de la Librería “**tag**”. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial** lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. También pueden visitarnos en nuestra página de **Facebook**: www.facebook.com/karaeffector

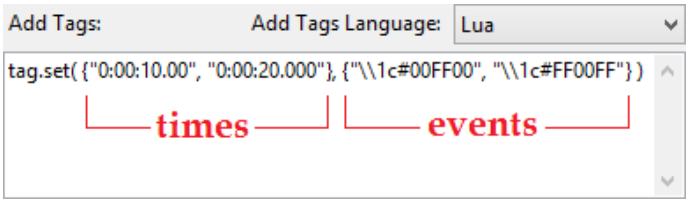
Kara Effector 3.2:

Este **Tomo XI** es la continuación de las funciones de la librería "**tag**". No hay mucho más que se pueda decir acerca de esta librería, sino que como todas las vistas en los Tomos del **Kara Effector**, es muy importante conocerla y saber qué tipo de ayuda nos puede ofrecer cada una de las funciones que contiene.

Librería "tag" [KE]

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 primerio ("**\\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✱i ✱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✱i ✱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:

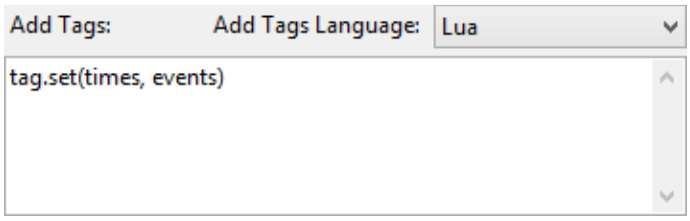
Como la línea 3 inicia en 13.54 s y finaliza en 18.39 s, su color primario será siempre el Verde:

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 practicamente 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 tranformaciones 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.

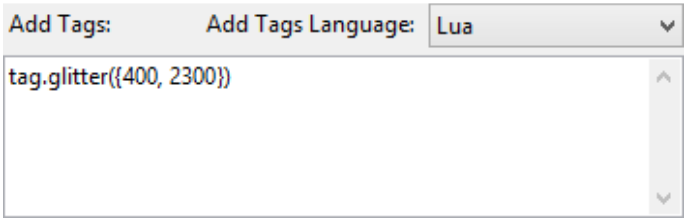
El parámetro "**time**" puede ser un tiempo en ms, lo que hará que las transformaciones al azar se hagan en el lapso de tiempo entre 0 y dicho valor. Ejemplo:

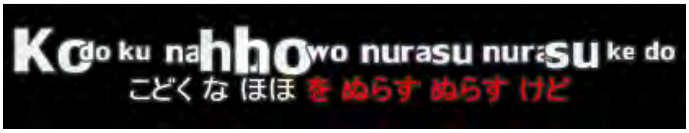
tag.glitter(1200)

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:

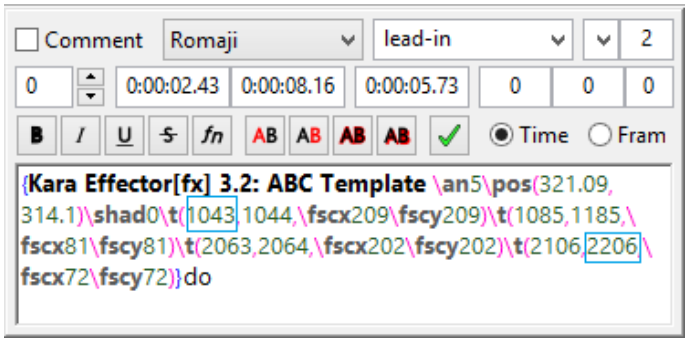
tag.glitter({400, 2300})

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:

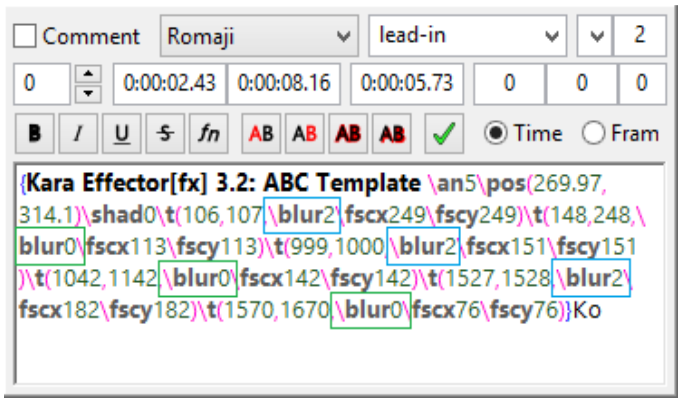


El valor por default de "**time**" es **fx.dur**, o sea, la duración total de cada una de las líneas fx:

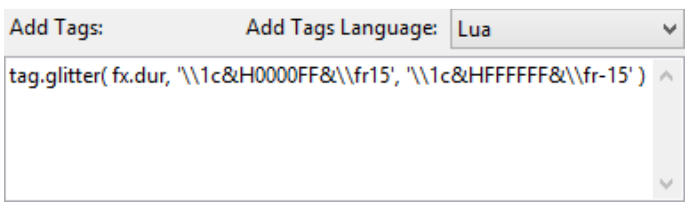
tag.glitter( ) = tag.glitter(fx.dur)

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 \1c, entonces debemos usar en "**add_f**" otro tag \1c, 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 transformaciondes 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 tranformaciones 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")

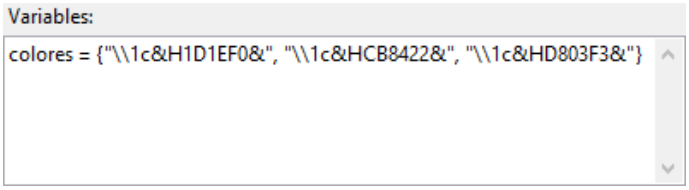
Obtendríamos las siguientes transformaciones:

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

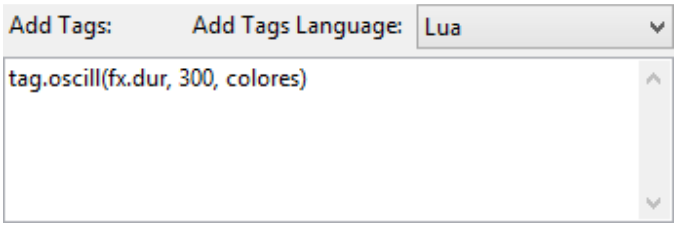
Los tres tags ingresados en la función se alternaron en las transformaciones, a manera de ciclo. Este procedimiento será el mismo sin importar la cantidad de tags, lo que hace que esta función tenga muchas utilidades, como alternar cambios de tamaños, de colores, de blur, de ángulos o lo que nos podamos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



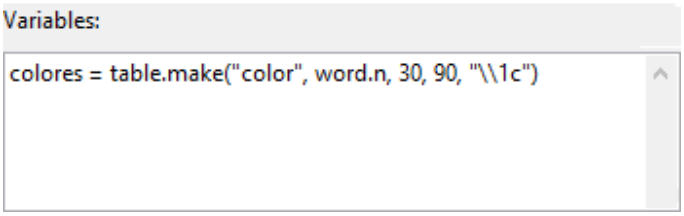
Y en **Add Tags** ponemos:



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

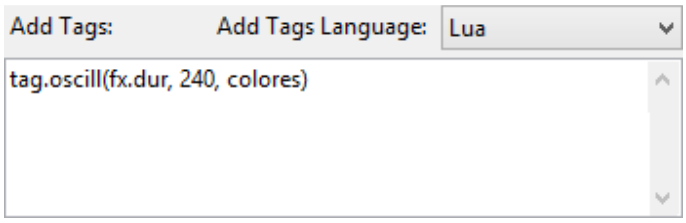
La última opción que nos da la función **tag.oscill** es poder decidir cuál será el primer elemento de la **tabla** de tags, con el que empezará la primera transformación entre todas las retornadas. Ejemplo:

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



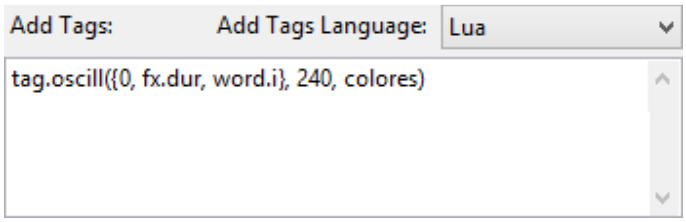
Recordemos que la función **table.make** crea una **tabla**, en este caso de colores, de word.n de tamaño, con colores equidistantes entre los ángulos 30° y 90° con un tag \1c.

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



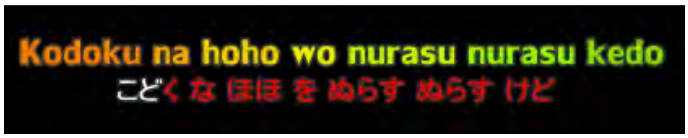
Así la primera transformación de todas las generadas en cada palabra (**word**) empezarán con el mismo color:

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



- **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**



Hecho esto, el tag de la primera transformación para la primera palabra, será el pimer 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.

---

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&")

Y con la función **tag.ipol** se da por terminada la librería "**tag**" y seguimos avanzando en el estudio de recursos del **Kara Effector** y profundizando cada vez más en el mundo de los Karaokes.

Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. También pueden visitarnos y dejar su comentario en nuestra página de **Facebook**: www.facebook.com/karaeffector

Kara Effector 3.2:

El **Tomo XII** arrancará con las librerías "**color**" y "**alpha**" del **Kara Effector**, ya que sus funciones son muy similares entre sí.

Librerías Color y Alpha [KE]:

Son dos librerías con funciones dedicadas a los colores y a las transparencias (alpha). Algunas de ellas nos servirán para hacer Efectos directamente y otras como apoyo para los mismos.

color.to_RGB(color_or_table): retorna los valores de la cantidad de Rojo, Verde y Azul (Red, Green and Blue) en base decimal, de un color asignado. El formato del color ingresado puede ser .ass o **HTML**

La función retorna los valores en una **tabla**. Ejemplo:

color.to_RGB("&HF4E67A") = { 122, 230, 244 }

- **122** (Rojo "7A")
- **230** (Verde "E6")
- **244** (Azul "F4")

Y para el caso que le ingresemos una **tabla** de colores, la función retornará una **tabla**, en donde cada uno de sus elementos son a su vez otra **tabla**, que contienen los valores de Rojo, Verde y Azul de cada uno de los colores:

colores = { "&HF108B4&", "&H30F304&" }

color.to_RGB(colores)

= { {180, 8, 241}, {4, 243, 48} }

---

color.to_HVS(color_or_table): esta función es similar a **color.to_RGB**, pero con la diferencia que retorna los valores en base decimal del Tono, la Saturación y del Valor (Hue, Saturation and Value). La forma de usar esta función es la misma que la de **color.to_RGB**

color.vc(color_or_table): convierte a un color o a cada uno de los colores de una **tabla**, en formato "**vc**", que es el formato de los colores en el **VSFilterMod**.

Ejemplo 1:

- **color.vc("&HFF00A3&")**
- **= (FF00A3, FF00A3, FF00A3, FF00A3)**

Ejemplo 2:

- **colores = { "&HFFFFFF&", "H000000&" }**
- **color.vc(colores)**
- **={(FFFFFF,FFFFFF,FFFFFF,FFFFFF), (000000,000000,000000,000000) }**

Es decir, que si ingresamos en la función un solo color, se retornará ese mismo color en formato "**vc**". Si ingresamos una **tabla** de colores, retornará una **tabla** con cada uno de los colores en formato "**vc**".

---

alpha.va(alpha_or_table): hace exactamente lo mismo que la función **color.vc**, pero con las transparencias (alpha). Ejemplo:

- **alpha.va("&HFF&") = (FF, FF, FF, FF)**

---

color.r( ): retorna un color al azar en formato .ass entre todos los del espectro, incluidos el Blanco, el Negro y todos los matices entres ellos y los demás colores.

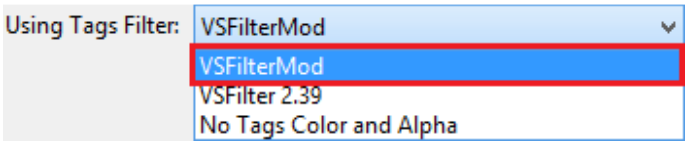
---

alpha.r( ): retorna una transparencia (alpha) al azar en formato .ass, entre 0 (totalmente visible) y 255 (invisible).

---

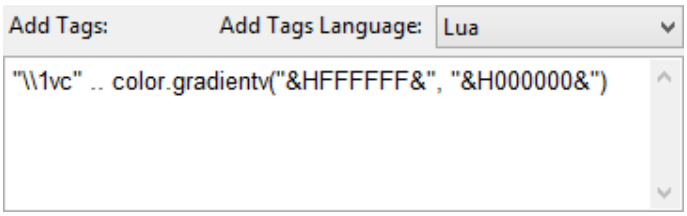
color.rc( ): retorna un color al azar en formato "**vc**" entre todos los del espectro. El random se ejecuta de manera diferente para cada uno de los cuatro colores que conformarán al color final. Ejemplo:

color.gradientv(color_top, color_bottom): esta función es de uso exclusivo del filtro **VSFilterMod**, ya que retorna un tag "**vc**" (vector color), y para ello es necesario que la opción de dicho filtro esté seleccionada en la primera ventana del **Kara Effector**:

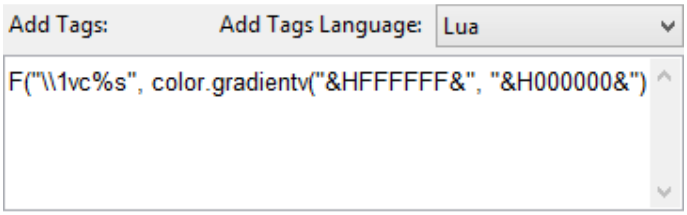


Esta función hace un **gradiente** (degradado) vertical entre dos colores asignados, **color_top** y **color_bottom**.

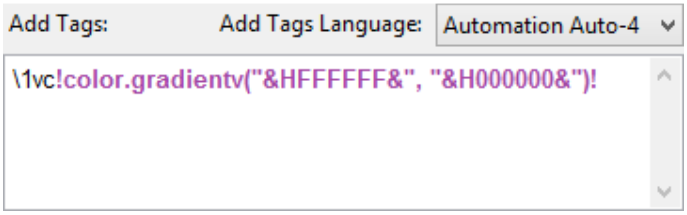
Un ejemplo simple en lenguaje **LUA** sería:



La anterior imagen muestra el método de concatenado en lenguaje **LUA**, pero recordemos que también lo podemos hacer usando la función **string.format** vista muchas veces anteriormente:

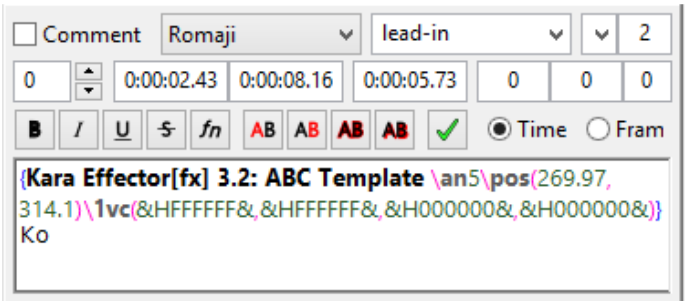


Y este mismo ejemplo en lenguaje **Automation-auto4** es:



Cualquiera de los tres anteriores métodos es válido para hacer Efectos en el **Kara Effector**. Del anterior ejemplo, en pantalla veremos el **gradiente** entre el Blanco y el Negro:

En la siguiente imagen vemos el tag "**vc**" que hace posible el **gradiente** (\1vc):



En el ejemplo anterior vimos cómo usar la función con ambos parámetros "**string**" (en este caso, colores), pero ambos parámetros también pueden ser **tablas** o en caso de ser necesario, también combinaciones entre un **string** y una **tabla**:

color.gradientv(color_top, color_bottom)			
Caso	**color_top**	**color_bottom**	**return**
1	string	string	string
2	string	tabla	tabla
3	tabla	string	tabla
4	tabla	tabla	tabla

Caso 1: al usar la función con string y string, entonces se retorna un **string** correspondiente al gradiente vertical de los dos strings ingresados.

Caso 2: retorna una **tabla** de gradientes entre el string del primer parámetro y cada uno de los elementos de la tabla ingresada.

Caso 3: retorna una **tabla** de gradientes entre cada uno de los elementos de la tabla ingresada y el string del segundo parámetro.

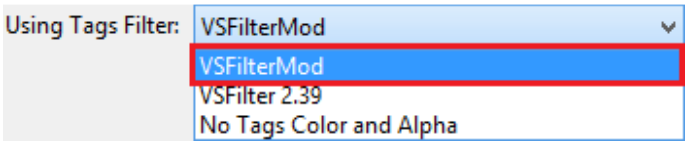
Caso 4: retorna una **tabla** de gradientes entre las posibles combinaciones de los productos cartesianos entre los elementos de ambas tablas ingresadas.

---

alpha.gradientv(alpha_top, alpha_bottom): hace exactamente lo mismo que la función **color.gradientv**, pero con las transparencias (alpha). Ejemplo:

- **alpha.gradientv("&HFF&", "&H44&")**
  = (&HFF&, &HFF&, &H44&, &H44&)

color.gradienth(color_left, color_right): esta función es de uso exclusivo del filtro **VSFilterMod**, ya que retorna un tag "**vc**" (vector color), y para ello es necesario que la opción de dicho filtro esté seleccionada en la ventana de inicio del **Kara Effector**:

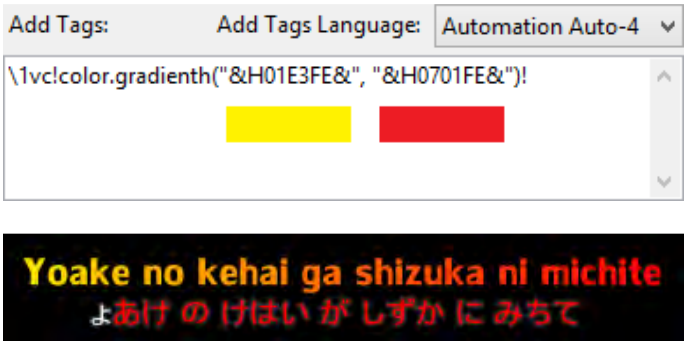


Esta función hace un **gradiente** (degradado) horizontal entre dos colores asignados, **color_left** y **color_right** (color izquierdo y color derecho).

Las opciones de uso de esta función son los mismos que los de la función **color.gradientv**:

color.gradienth(color_left, color_right)			
Caso	**color_left**	**color_right**	**return**
1	string	string	string
2	string	tabla	tabla
3	tabla	string	tabla
4	tabla	tabla	tabla

Veamos un ejemplo de cómo usar esta función:



---

alpha.gradienth(alpha_left, alpha_right): hace exactamente lo mismo que la función **color.gradienth**, pero con las transparencias (alpha). Ejemplo:

- **alpha.gradienth("&HAA&", "&H00&")**
  = (&HAA&, &H00&, &HAA&, &H00&)

Las anteriores cuatro funciones guardan relación cercana con los gradientes, las dos primeras con los verticales y las dos siguiente con los horizontales.

color.vc_to_c(color_vc): esta función convierte al color ingresado, de formato "**vc**" (**VSFilterMod**) a formato "**c**" (**VSFilter 2.39**). El parámetro **color_vc** puede ser un **string** o una **tabla** de **string**, de manera que si es un **string**, la función retornará a ese color cambiado de un formato al otro, y si es una **tabla**, retornará una **tabla** con cada uno de los colores de la misma, convertidos en formato "**c**".

color.vc_to_c(color_vc)		
Caso	**color_vc**	**return**
1	string	string
2	tabla	tabla

---

alpha.va_to_a(alpha_va): hace lo mismo que la función **color.vc_to_c**, pero con las transparencias (alpha).

---

color.c_to_vc(color_c): es la función opuesta a **color_c_to_vc**. Convierte al color ingresado, de formato "**c**" (**VSFilter 2.39**) a formato "v**c**" (**VSFilterMod**).

Los modos son los mismos que los de su función opuesta:

color.c_to_vc(color_c)		
Caso	**color_c**	**return**
1	string	string
2	tabla	tabla

---

alpha.a_to_va(alpha_a): hace lo mismo que la función **color.c_to_vc**, pero con las transparencias (alpha).

---

color.interpolate(color1, color2, index_ipol): es similar a la función **_G.interpolate_color** del **Automation Auto-4**, pero con la diferencia que tanto **color1** como **color2** pueden ser o un **string** o una **tabla**.

El parámetro **index_ipol** es opcional y es un número real entre 0 y 1 inclusive. Su valor por default es 0.5 con el fin de que la función retorne el color intermedio entre **color1** y **color2**. Si **index_ipol** es cero o cercano a él, la función retorna un color cercano a **color1**, y si es 1 o cercano a 1, retornará un color cercano al de parámetro **color2**, o sea que el color que se retornará dependerá únicamente de este valor (**index_ipol**).

Los modos de usar esta función son los siguientes:

color.interpolate(color1, color2, index_ipol)			
Caso	**color1**	**color2**	**return**
1	string	string	string
2	string	tabla	tabla
3	tabla	string	tabla
4	tabla	tabla	tabla

Caso 1: al usar la función con string y string, entonces se retorna un **string** correspondiente a la interpolación entre los dos colores, dependiendo del valor de **index_ipol**, o justo el color intermedio, en el caso de que no exista dicho parámetro.

Caso 2: retorna una **tabla** con las interpolaciones entre el color del primer parámetro y cada uno de los colores de la tabla ingresada en el segundo parámetro.

Caso 3: retorna una **tabla** con las interpolaciones entre cada uno de los colores de la tabla ingresada en el primer parámetro y el color del segundo parámetro.

Caso 4: retorna una **tabla** con las interpolaciones entre las posibles combinaciones de los productos cartesianos entre los colores de ambas tablas ingresadas.

---

alpha.interpolate(alpha1, alpha2, index_ipol): similar a la función **_G.interpolate_alpha** del **Automation Auto-4**. Hace lo mismo que la función **color.interpolate**, pero con las transparencias (alpha).

---

color.delay(time_i, delay, color_i, color_f, ...): es una función que convierte progresivamente al parámetro **color_i** (color inicial) al parámetro **color_f** (color final). Es exclusiva del filtro **VSFilterMod** ya que retorna un color en formato "**vc**" seguido de cuatro transformaciones que contienen colores también en formato "**vc**". Las tres primeras transformaciones son colores al azar entre **color_i** y **color_f** y la última contiene a **color_f**.

El parámetro **time_i** es el tiempo relativo al tiempo de inicio de la línea fx y corresponde al momento en que iniciarán las transformaciones.

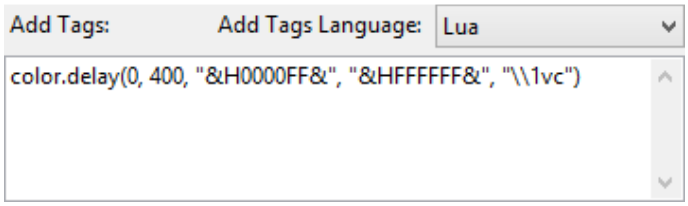
El parámetro **delay** es la duración total de las cuatro transformaciones que retorna la función.

El parámetro (**…**) hace referencia a uno o más tags de colores "vc" que eventualmente retornará la función. Las posibilidades serían:

- "\\1vc"
- "\\3vc"
- "\\4vc"
- "\\1vc", "\\3vc"
- "\\1vc", "\\4vc"
- "\\3vc", "\\4vc"
- "\\1vc", "\\3vc", "\\4vc"

Ejemplos:

- **color.delay**(0, 600, "&HFFFFFFF&", "&H000000&", "\\1vc")
- **color.delay**(1200, 2000, "&HFF00FFF&", "&H00FF00&", "\\1vc", "\\3vc")
- **color.delay**(500, 1000, "&HFF0000&", "&H00FFFF&", "\\3vc")



---

alpha.delay(time_i, delay, alpha_i, alpha_f, ...): hace lo mismo que la función **color.delay**, pero con las transparencias (alpha).

---

c**olor.movedelay(dur, delay, mode, …):** es similar a la función **color.delay**, pero con la diferencia que la cantidad de transformaciones que retorna depende del parámetro **dur**, que es la duración total de la función. La otra diferencia es que en esta función se pueden ingresar la cantidad de colores que deseemos o también podemos poner como cuarto parámetro a una tabla de colores.

El parámetro **dur** es la duración total de todas las transformaciones que retorna la función.

El parámetro **delay** es la duración que tendrá cada una de las transformaciones retornadas por la función.

El parámetro **mode** es un número entero que indica a cuál o a cuáles tags de colores "vc" se aplicará la función. Pueden ser:

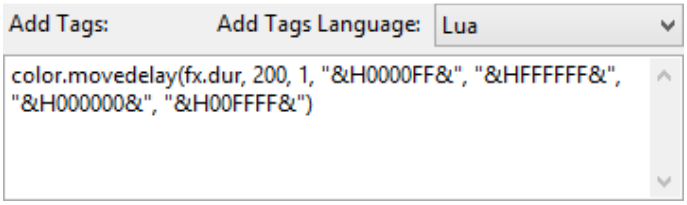
- 1 = \\1vc
- 3 = \\3vc
- 4 = \\4vc
- 13 = \\1vc\\3vc
- 14 = \\1vc\\4vc
- 34 = \\3vc\\4vc
- 134 = \\1vc\\3vc\\4vc

Si se quiere, el parámetro **mode** también se puede usar como un **string** que indique el o los tags a los que queremos que se aplique la función, ejemplo:

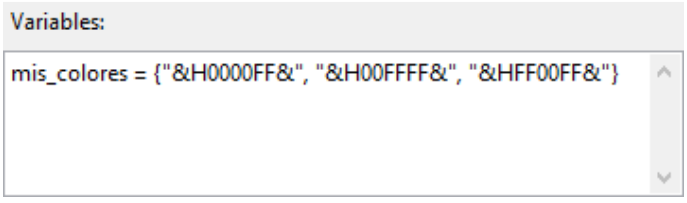
mode = "\\1vc\\3vc"

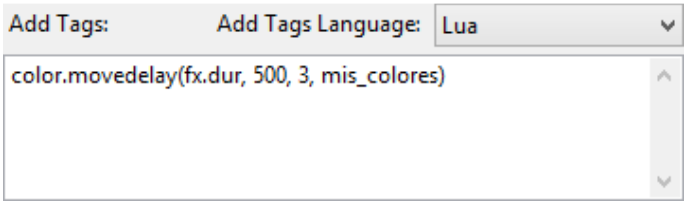
El parámetro (**…**) hace referencia a los colores ingresados. Puede ser:

- Un solo color
- Una serie de colores (separados por comas):



- Una tabla de colores:





color.set(times, colors, …): esta función retorna una o más transformaciones de colores de la tabla **colors** con respecto a los tiempos de la tabla **times**.

Todos los tiempos de la tabla **times** están medidos con referencia al cero absoluto del vídeo y a cada uno de los tiempos de esta tabla le debe corresponder un color de la tabla **colors**.

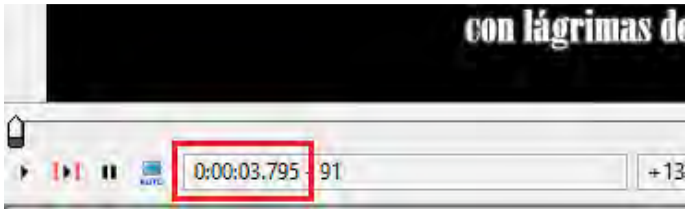
El parámetro (**…**) hace referencia al tag o tags de colores a los que afectarán las transformaciones que retornará la función. Las posibilidades serían:

- "\\1vc"
- "\\3vc"
- "\\4vc"
- "\\1vc", "\\3vc"
- "\\1vc", "\\4vc"
- "\\3vc", "\\4vc"
- "\\1vc", "\\3vc", "\\4vc"

Dado que esta función no es exclusiva de ninguno de los filtros que se pueden usar en el **Aegisub**, entonces las posibilidades de extenderían a:

- "\\1c"
- "\\3c"
- "\\4c"
- "\\1c", "\\3c"
- "\\1c", "\\4c"
- "\\3c", "\\4c"
- "\\1c", "\\3c", "\\4c"

Cada uno de los tiempos de la tabla del parámetro **times** deben ser copiados de la parte inferior izquierda del vídeo en el **Aegisub**. Dichos tiempos están en formato **HMSms** (Horas, Minutos, Segundo y milisegundos). Ejemplo:



Los tiempos deben ser pegados en la tabla **times** usando comillas simples o dobles para que el **Kara Effector** pueda reconocerlos a cada uno de ellos como un **string**.

Esta sería una opción para definir los dos parámetros en la celda de texto "**Variables**":

Variables:

```
times  = {"0:00:03.795", "0:00:10.552"};
colors = {"&H00FFFF&", "&HFF00FF"}
```

Notamos cómo ambas tablas tienen la misma cantidad de elementos. La cantidad total de transformaciones que la función retornará depende de cada uno de nosotros, puede ser mínimo un elemento y máximo hasta donde lleguemos a necesitar.

Al definir las tablas **times** y **colors** en "**Variables**" (el nombre con que se definen las tablas es decisión de cada uno), usaremos la función en **Add Tags**, de alguna de las siguientes dos maneras:

1. Indicando el tipo de variable que será cada una de las dos tablas:

Add Tags: Add Tags Language: Lua

```
color.set( var.line.times, var.line.colors, "\\1c" )
```

2. Como usamos punto y coma (**;**) para separar las tablas al definirlas en "**Variables**", entonces las podemos llamar con tan solo escribir el nombre con que las definimos:

Add Tags: Add Tags Language: Lua

```
color.set( times, colors, "\\1c" )
```

Otra forma de usar la función es ingresando directamente las tablas dentro de ella, sin necesidad de definirla:

Add Tags: Add Tags Language: Lua

```
color.set( {"0:00:03.795", "0:00:10.552"}, {"&H00FFFF&",
"&HFF00FF"}, "\\3c")
```

Cuando algún tiempo de la tabla **times** está definido como un **string** (como en los ejemplos anteriores), entonces la transformación correspondiente a ese tiempo tendrá su duración por default que es de 1 ms. Es decir que el cambio de un color a otro será instantáneo.

Si queremos que alguna transformación tenga una duración superior a la que tiene por default (1 ms), hay dos opciones posibles para poderlo hacer.

1. Convertir a dicho tiempo en otra **tabla** con dos elementos, y dentro de ella poner los tiempos de inicio y final de la transformación, ejemplo:

```
Variables:
times  = {"0:00:03.795", {"0:00:10.552" , "0:00:11.052"} };
colors = {"&H00FFFF&", "&HFF00FF"}
```

En este caso, la transformación tendrá una duración de 500 ms que es la diferencia entre los dos tiempos de la tabla del segundo elemento de la tabla **times**. Esta opción tiene la ventaja de poner los dos tiempos sin la necesidad de calcular previamente la duración de la transformación.

2. Convertir a dicho tiempo en otra **tabla** con dos elementos, y dentro de ella poner el tiempo de inicio de la transformación y la duración en ms de la misma, ejemplo:

```
Variables:
times  = {"0:00:03.795", {"0:00:10.552" , 360} };
colors = {"&H00FFFF&", "&HFF00FF"}
```

Haciéndolo de este modo, la transformación tendrá una duración de 360 ms, que es el valor que pone el segundo elemento de la tabla correspondiente al segundo tiempo de la tabla **times**. La ventaja de esta opción es poner la duración de la transformación sin poner el tiempo final.

Los anteriores dos métodos son válidos para cualquiera o para todos los elementos de la tabla **times**, y como ya lo había mencionado, son usados para modificar la duración por default de las transformaciones que se retornarán al usar la función.

Los colores de la tabla **colors** pueden estar todos en formato del filtro **VSFilter 2.39**, del **VSFilterMod** o una combinación de ambos, si así se quiere. Ejemplo:

```
Variables:
times  = {"0:00:03.795", "0:00:10.552"};
colors = { "(&H00FFFF&,&H00FFFF&,&HAA00D8&,
&HAA00D8&)", "&HFF0FF&" };
```

Del ejemplo anterior vemos cómo el primer color está en formato "**vc**" y el segundo está en formato normal. Es obvio que para que los colores salgan tal cual como los pusimos en formato "**vc**", la opción del filtro **VSFilterMod** debe estar seleccionada en la ventana de inicio del **Kara Effector**.

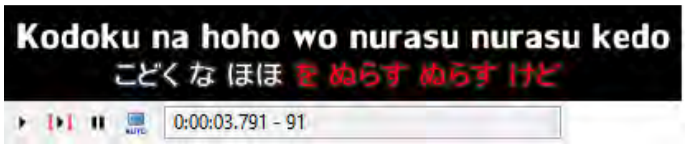
Un tercer formato que puede tener uno o más colores de la tabla **colors** es el de una **tabla** de colores. Ejemplo:

```
Variables:
times  = { "0:00:03.795", "0:00:10.552" };
colors = { {"&H00FFFF&", "&HF5900F&", "&HA62090&"},
"&HFF00FF&"}
```

En el anterior ejemplo, el primer elemento de la tabla **colors** es una tabla de 3 colores (amarillo, azul y morado), pero la cantidad puede ser la que nosotros convengamos. La función la usaríamos en **Add Tags** del mismo modo ya aprendido hasta este momento:

```
Add Tags:   Add Tags Language: Lua
color.set( times, colors, "\\1c" )
```

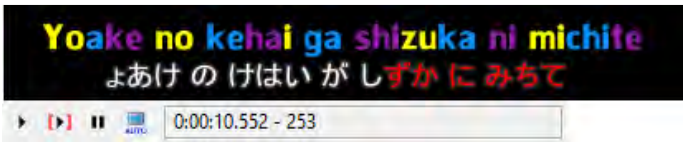
En la siguiente imagen vemos el texto justo antes de que ocurra la primera transformación:



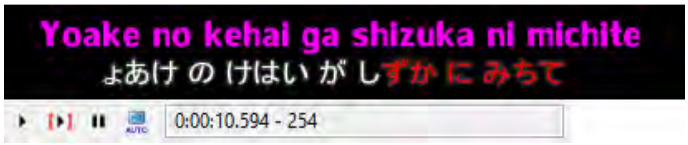
Y lo que sucederá en la primera transformación, dado que el primer elemento de la tabla **colors** es una tabla de colores y que el ejemplo se está haciendo en el modo

Template Type: Syl, es que a cada sílaba le corresponderá un color de esa tabla; pero como dicha tabla solo tiene 3 colores, entonces a la cuarta sílaba se le volverá a asignar el color 1 de la tabla, lo mismo que a las séptima sílaba y así sucesivamente hasta completar todas las sílabas:

En la próxima imagen veremos el texto justo antes de la tercera transformación:



Y así se verá el texto justo después del cambio de color, ya que el segundo elemento de la tabla **colors** era el color magenta (&HFF00FF&):



Esta tercera habilidad que tienen los elementos de la tabla de **colors** de poder ser una tabla de colores nos da muchas posibilidades, como hacer "**máscaras**" o **degradaciones** en una o más de las transformaciones que retorna la función. Ejemplo:

Variables:

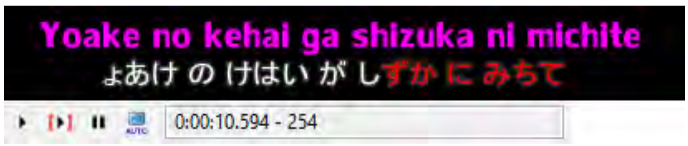
```
times  = { "0:00:03.795", "0:00:10.552" };
colors = { table.make('color', syl.n, 15, 100),
"&HFF00FF&" }
```

El primer elemento de la tabla **colors** está determinado por la función **table.make** así:

- El modo "**color**" creará una tabla de colores.
- **syl.n** determinará el tamaño de dicha tabla.
- 15 y 100 serán los límites inferior y superior de la degradación con respecto al círculo cromático de la teoría del color **HSV**. Los 15° corresponden un tono naranja rojizo de dicho círculo, y los 100° a un tono verde limón.

De este modo, la función hará que el texto cambie a la degradación anteriormente descrita:

Y para la segunda transformación el texto pasará al color que se haya asignado como segundo elemento en la tabla **colors** (magenta = &HFF00FF&):



Todo es cuestión de poner a volar un poco la imaginación y empezar a experimentar con las distintas opciones que nos ofrece esta función, y las posibilidades son casi que infinitas al igual que los resultados.

---

alpha.set(times, alphas, …): esta función hace lo exactamente lo mismo que la función **color.set**, pero usa las transparencias (alpha) en vez de colores.

Es todo por ahora, pero las librerías **color** y **alpha** aún continuarán en el **Tomo XIII**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. También pueden visitarnos y dejar su comentario en nuestra página de **Facebook**: www.facebook.com/karaeffector

Kara Effector 3.2:

El **Tomo XIII** es la continuación de las librerías "**color**" y "**alpha**" del **Kara Effector** que se iniciaron en el **Tomo** anterior, en el cual ya vimos numerosas funciones y aún restan muchas más por ver.

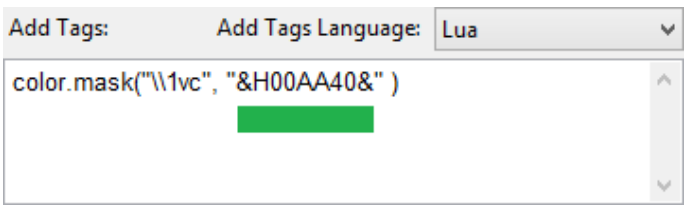
Librerías Color y Alpha [KE]:

color.mask(mode, color, color_mask): esta función retorna un color para cada uno de los elementos de una línea de texto (word, syl, char y demás) a manera de "**máscara**".

El parámetro **mode** hace referencia al único tag de color en formato "**vc**" en el que se realizará la "**máscara**", es decir:

- "\\1vc"
- "\\3vc"
- "\\4vc"

El parámetro **color** puede ser, tanto un **string** de color como una **tabla** de colores. Ejemplo:

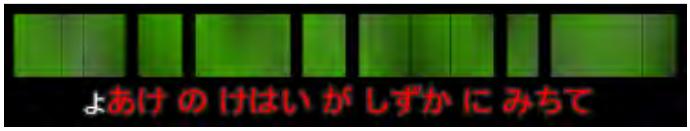


En este ejemplo, el parámetro **color_mask** no está ya que es opcional, y el parámetro **color** es un **string**, un color en tono de verde.

El texto se vería de la siguiente forma:

Este ejemplo está hecho con un **Template Type: Syl**, y podemos notar cómo cada una de las sílabas tiene el mismo todo de verde, pero mezclado en ciertas partes con tonos entre el blanco y el negro.

Si remplazamos a cada sílaba por un rectángulo con sus mismas dimensiones, se podrá apreciar más claramente el efecto de la "**máscara**":

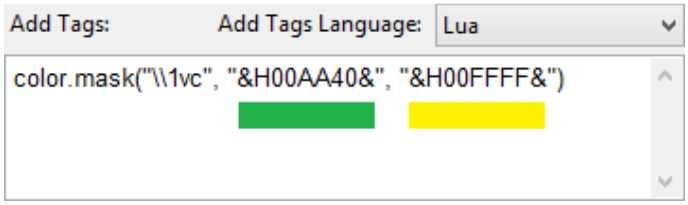


La función **color.mask** hace por default la "**mascara**" con el color ingresado en el parámetro **color** y los tonos al azar entre el blanco y el negro.

En el caso en que el parámetro **color** sea una **tabla** de colores, la función retornará una **tabla** con la "**máscara**" de cada uno de los colores de dicha tabla.

El parámetro **color_mask**, al igual que el parámetro **color**, también puede ser tanto un color **string** como una **tabla** de colores. Ejemplos:

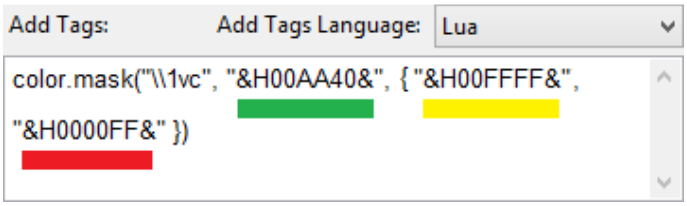
- **color_mask: string**



De este modo, la función ya no hace la "**máscara**" con los tonos por default entre el blanco y el negro, sino que la hace entre el parámetro **color** y el color ingresado en el parámetro **color_mask** (verde y amarillo):

Así que de esta forma uno decide un solo color con el que el parámetro **color** hará la "**máscara**".

- **color_mask: table**



Si empleamos al parámetro **mask_color** como una **tabla** (en este ejemplo, una de dos colores: amarillo y rojo), la función hará la "**máscara**" entre el color principal que es el parámetro **color**, y tonos al azar entre todos los colores de la tabla de colores **mask_color**:

Resumiendo los modos, tenemos:

color.mask (mode, color, color_mask)			
Caso	**color**	**color_mask**	**return**
1	string	string	string
2	string	tabla	string
3	tabla	string	tabla
4	tabla	tabla	tabla

---

alpha.mask(mode, alpha): es similar a la función **color.mask**, con la diferencia que esta función carece del tercer parámetro, y que trabaja con las transparencias (alpha) en lugar de colores.

---

color.movemask(dur, delay, mode, color): esta función es similar a **color.mask**, ya que también genera una "**máscara**", pero ésta da la sensación de movimiento gracias a una serie de transformaciones. Dicho efecto de movimiento es realizado de derecha a izquierda.

El parámetro **dur** es la duración total de la función, o sea la duración total de todas las transformaciones. Puede ser un **número** o una **tabla** con dos números, el primero de ellos indicaría el inicio de la función y el segundo el tiempo final, ambos tiempos son relativos al tiempo de inicio de cada una de las líneas fx generadas.

Las opciones del parámetro **dur** son:

1. dur
2. {time1, time2}

El parámetro **delay** cumple con la misma tarea que su parámetro homónimo en la función **tag.oscill**, y estas son sus opciones:

1. delay
2. { { delay, dur_delay } }
3. { delay, accel }
4. { {delay, dur_delay }, accel }
5. { delay, accel, dilat }
6. { { delay, dur_delay }, accel, dilat }

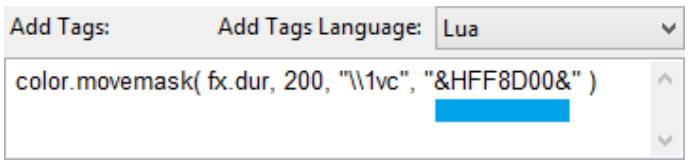
Todas las anteriores variables mencionadas para este parámetro hacen referencia a valores numéricos, es decir, a números.

- **delay**: valor numérico que indica cada cuánto suceden las transformaciones generadas.
- **dur_delay:** valor numérico que indica la duración de cada una de las transformaciones generadas. Su valor por default es **delay**.
- **accel:** valor numérico que indica la aceleración de las transformaciones generadas. Su valor por default es 1.
- **dilat:** valor numérico que indica el tiempo que se va extendiendo cada una de las transformaciones con respecto a la anterior, o sea que extiende la duración. Su valor por default es 0.

El parámetro **mode** hace referencia al tag de color en formato "**vc**" que se retornará en las transformaciones generadas por la función:

1. "\\1vc"
2. "\\3vc"
3. "\\4vc"

El parámetro **color** hace referencia a un **string** de color. A diferencia de la función **color.mask**, acá este parámetro ya no puede ser una **tabla**. Ejemplo:



Del anterior ejemplo tenemos:

- **dur** = fx.dur
- **delay** = 200
- **mode** = "\\1vc"
- **color** = "&HFF8D00"

Las "**máscara**" se genera igual que con **color.mask**, pero ahora se mueve de derecha a izquierda con la duración de cada transformación en 200 ms.

---

alpha.movemask(dur, delay, mode, alpha): es similar a la función **color.movemask**, con la diferencia que trabaja con las transparencias (alpha) en lugar de colores.

---

color.setmovemask(delay, mode, times, colors): esta función es la combinación de la función **color.set** y la función **color.movemask**, sus parámetros son:

- **delay**: es un valor numérico que hace referencia a la duración de cada una de las transformaciones, al igual, también indica cada cuánto suceden las mismas.
- **mode**: hace referencia al tag de color en formato "**vc**" al que afectará la función:
  1. "\\1vc"
  2. "\\3vc"
  3. "\\4vc"
- **times:** hace referencia a la **tabla** con los tiempos copiados de la parte inferior izquierda del vídeo en el **Aegisub**, pero todos los elementos de la tabla deben ser un string, ya no pueden ser una tabla para modificar el tiempo del cambio de un color a otro.
- **colors:** hace referencia a la **tabla** de colores que hacen pareja con cada uno de los tiempos de la tabla **times**, pero esta vez solo pueden un **string** de color, ya no pueden ser tablas.

Entonces es fácil deducir lo que esta función hace.

Ejemplo:

Variables:

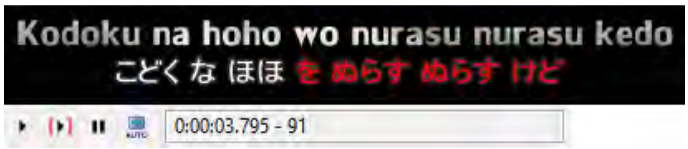
```
times  = { "0:00:03.795", "0:00:10.552" };
colors = { "&H00FFFF&", "&HA62090&" }
```

Y en **Add Tags**:

Add Tags: Add Tags Language: Lua

```
color.setmovemask( 200, "\\1vc", times, colors )
```

Entonces, justo antes de la primera transformación se verá algo como esto:



Luego la primera transformación:

---

color.movemaskv(dur, delay, mode, color, color_mask): similar a la función **color.movemask**, pero ahora la "**máscara**" no se moverá de derecha a izquierda, sino de forma vertical, de abajo a arriba. La otra diferencia está en que el parámetro **color** ya no puede ser una **tabla**, solo está habilitado para ser un **string** de color.

El resto de las cualidades y características son todas las mismas, y el uso entre una función y otra, solo depende de los resultados que queremos obtener.

---

alpha.movemaskv(dur, delay, mode, alpha): similar a la función **color.movemaskv**, pero carece del quinto parámetro y que trabaja con las transparencias (alpha) en vez de colores.

color.masktable(color): esta función crea una **tabla** con los colores necesarios para hacer una "**máscara**" dependiendo del **Template Type**, ya que éste determina el tamaño de la **tabla**.

El parámetro **color** puede ser tanto un **string** de color, como una **tabla** de colores. Para el primer caso, retorna una **tabla** con la "**máscara**" de dicho color y para el caso en que el parámetro **color** sea una **tabla**, la función retorna una tabla de tablas, es decir, una **tabla** en donde cada uno de sus elementos es otra tabla, y cada una de ellas contiene la "**máscara**" correspondiente a cada color de la **tabla** ingresada.

---

alpha.masktable(alpha): similar a la función **color.masktable**, pero con la diferencia que trabaja con las transparencias (alpha) en vez de colores.

---

color.module(color1, color2): esta función retorna la interpolación completa entre los colores de los parámetros **color1** y **color2**, con respecto al **loop**.

Los parámetros color1 y color2 pueden ser tanto strings de colores, como tablas de colores:

color.moduke (color1, color2)			
Caso	**color1**	**color2**	**return**
1	string	string	string
2	string	tabla	tabla
3	tabla	string	tabla
4	tabla	tabla	tabla

Caso 1: al usar la función con ambos parámetros **strings**, entonces se retorna un **string** de color correspondiente a la interpolación entre los dos colores. Se puede decir que es la forma más convencional de usar esta función.

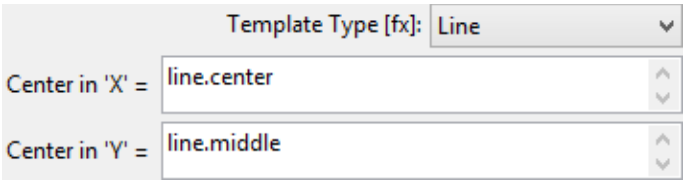
Caso 2: retorna una **tabla** con las interpolaciones entre el color del primer parámetro y cada uno de los colores de la tabla ingresada en el segundo parámetro.

Caso 3: retorna una **tabla** con las interpolaciones entre cada uno de los colores de la tabla ingresada en el primer parámetro y el color del segundo parámetro.

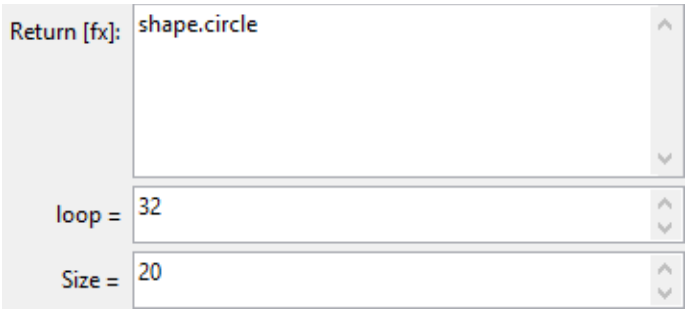
Caso 4: retorna una **tabla** con las interpolaciones entre las posibles combinaciones de los productos cartesianos entre los colores de ambas tablas ingresadas.

Recordemos que todas las interpolaciones en esta función dependen únicamente del **Loop** (**maxj**). Ejemplo:

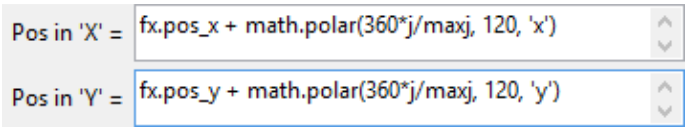
- **Template Type: Line**



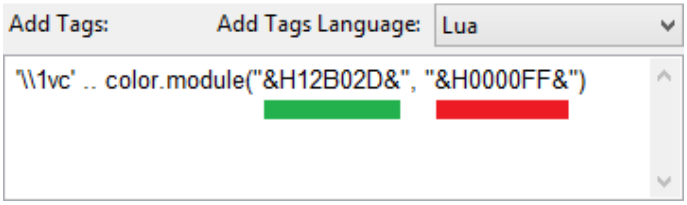
- Configuramos a **Return [fx], loop** y **Size**



- En **Pos in "X"** y **Pos in "Y"** usaremos la función **math.polar** para que los 32 círculos se ubiquen equidistantemente respecto al centro de cada línea karaoke, con un radio de 120 pixeles:

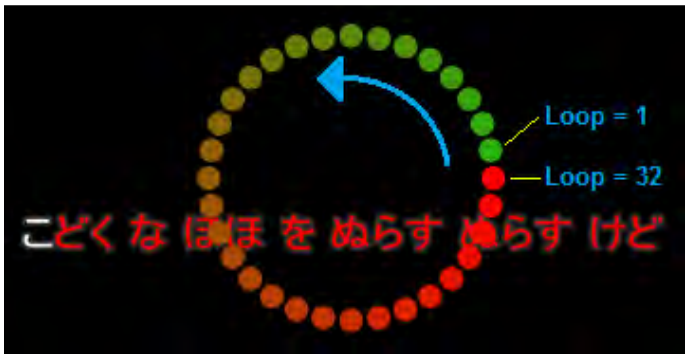


- Por último, en **Add Tags** llamamos a la función de la siguiente forma (a esta altura ya deben estar familiarizados con los métodos de concatenación entre dos **string**):



Entonces la función asignará a cada uno de los **loops** un único color correspondiente a la interpolación entre los dos colores asignados en los parámetros **color1** y **color2**.

Se pueden notar los círculos y el color primario ("\\1vc") asignado a cada uno de ellos:



La función **color.module** se puede usar con los tags de colores de ambos filtros, o sea:

1. \\1c
2. \\3c
3. \\4c
4. \\1vc
5. \\3vc
6. \\4vc

La función **color.module** recibe su nombre gracias a la variable **module** del **Kara Effector**, que recordemos hace referencia a la interpolación de todos los números entre cero y uno, con respecto al **Loop**.

---

alpha.module(alpha1, alpha2): similar a la función **color.module**, pero con la diferencia que trabaja con las transparencias (alpha) en vez de colores.

---

color.module1(color1, color2): esta función es similar a **color.module**, pero retorna la interpolación completa entre los colores de los parámetros **color1** y **color2**, con respecto al **Temaplate Type**.

Como su nombre lo indica, genera la interpolación por medio de la variable **module1**, que hace referencia a la interpolación entre cero y uno sin importar el Loop y teniendo en cuenta al **Template Type** (ejemplo: desde 1 hasta **syl.n**, o desde 1 hasta **word.n**).

Se podría decir que **color.module** interpola a todos los elementos de cada una de las partes de línea karaoke, por

otra parte, la función **color.module1** interpola a la línea de karaoke de principio a fin sin importar el **Loop** que tenga cada una de las parte de la misma.

---

alpha.module1(alpha1, alpha2): similar a la función **color.module1**, pero con la diferencia que trabaja con las transparencias (alpha) en vez de colores.

---

color.module2(color1, color2): esta función es similar a **color.module** y a **color.module1**, pero retorna la interpolación completa entre los colores **color1** y **color2**, con respecto a todas las líneas a las que le se aplica un efecto. Es decir que genera la interpolación desde el primer elemento (char, syl, word y demás) de la primera línea a la que se le aplique un efecto, hasta el último elemento de la última línea a la que se le haya aplicado un efecto.

Esta función hereda su nombre de la variable **module2** del **Kara Effector**, que hace una interpolación de los números entre cero y uno con respecto a la cantidad de líneas a las que se les aplique un efecto.

---

alpha.module2(alpha1, alpha2): similar a la función **color.module2**, pero con la diferencia que trabaja con las transparencias (alpha) en vez de colores.

Con el final de este **Tomo XIII** se dan por concluidas las librerías **color** y **alpha**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

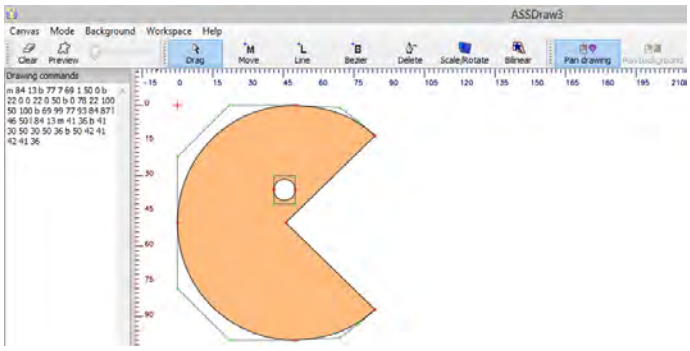
www.facebook.com/karaeffector

Kara Effector 3.2:

El **Tomo XIV** es el comienzo de una nueva librería y cada vez estamos más cerca de terminarlas todas y así ampliar aún más las herramientas a nuestra disposición a la hora de desarrollar un **Efecto Karaoke**, un **Logo**, un **Cartel** y todo aquello que nos dispongamos a hacer para nuestros proyectos en el **Aegisub** y el **Kara Effector**.

Librería Shape [KE]:

Se conoce como **shape** a un dibujo hecho por vectores en el **AssDraw3** que viene por default en el **Aegisub**, y que nos sirven de apoyo como una herramienta más a la hora de desarrollar un Efecto.



La Librería **shape** hace referencia a dichos dibujos y a las funciones relativas a ellos. Aparte de las funciones en esta Librería, el **Kara Effector** consta de un amplio listado de **shapes** (dibujos) prediseñadas para hacer uso de ellas y a continuación veremos el nombre, su tamaño en pixeles y una pre-visualización de las mismas.

Shapes Prediseñadas del Kara Effector

shape.circle	shape.triangle
100 x 100 px	100 x 106 px
	

Ejemplo:

En dicho caso se retornaría un **círculo** en el lugar en donde antes estaban las sílabas de la línea karaoke:

Shapes Prediseñadas del Kara Effector

shape.rectangle	shape.pentagon
100 x 100 px	100 x 95 px
	
shape.hexagon	**shape.octagon**
100 x 116 px	100 x 100 px
	
shape.heart	**shape.heart2t**
100 x 106 px	100 x 106 px
	

Shapes Prediseñadas del Kara Effector

shape.heart_b	shape.shine1t
100 x 106 px	100 x 100 px
	
shape.shine2t	**shape.shine3t**
100 x 100 px	100 x 100 px
	
shape.shine4t	**shape.trebol**
100 x 100 px	100 x 106 px
	
shape.feather	**shape.diamond**
100 x 100 px	100 x 100 px
	
shape.gear	**shape.bubble**
100 x 100 px	100 x 100 px
	
shape.note1t	**shape.note2t**
100 x 56 px	100 x 100 px
	

Shapes Prediseñadas del Kara Effector

shape.note3t	shape.note4t
100 x 100 px	100 x 100 px
	
shape.star	**shape.star1t**
100 x 95 px	100 x 95 px
	
shape.star2t	**shape.star3t**
100 x 116 px	100 x 116 px
	
shape.star4t	**shape.star5t**
100 x 116 px	100 x 116 px
	
shape.star6t	**shape.star7t**
100 x 95 px	100 x 95 px
	
shape.star8t	**shape.star9t**
100 x 95 px	100 x 116 px
	

Shapes Prediseñadas del Kara Effector

shape.star10t	shape.sakura
100 x 116 px	100 x 130 px
	
shape.sakura1t	**shape.sakura2t**
100 x 116 px	100 x 116 px
	
shape.sakura3t	**shape.sakura4t**
100 x 116 px	100 x 106 px
	
shape.sakura5t	**shape.sakura6t**
100 x 100 px	100 x 116 px
	
shape.sakura7t	**shape.snow1t**
100 x 116 px	100 x 108 px
	
shape.snow2t	**shape.snow3t**
100 x 96 px	100 x 94 px
	

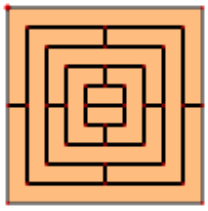
Shapes Prediseñadas del Kara Effector

shape.flower1t	shape.flower2t
100 x 95 px	100 x 95 px
	
shape.flower3t	**shape.flower4t**
100 x 95 px	100 x 95 px
	
shape.flower5t	**shape.flower6t**
100 x 95 px	100 x 95 px
	
shape.flower7t	**shape.flower8t**
100 x 95 px	100 x 95 px
	
shape.flower9t	**shape.flower10t**
100 x 95 px	100 x 95 px
	
shape.flower11t	**shape.flower12t**
100 x 116 px	100 x 116 px
	

Shapes Prediseñadas del Kara Effector

shape.flower13t	shape.flower14t
100 x 116 px	100 x 130 px
	
shape.flower15t	**shape.flower16t**
100 x 116 px	100 x 116 px
	
shape.flower17t	**shape.flower18t**
100 x 116 px	100 x 116 px
	
shape.flower19t	**shape.flower20t**
100 x 116 px	100 x 116 px
	
shape.flower21t	**shape.flower22t**
100 x 116 px	100 x 116 px
	
shape.flower23t	**shape.flower24t**
100 x 116 px	100 x 116 px
	

Shapes Prediseñadas del Kara Effector

shape.flower25t	shape.flower26t
100 x 116 px 	100 x 130 px 
shape.flower27t	shape.flower28t
100 x 116 px 	100 x 116 px 
shape.flower29t	shape.cristal17
100 x 116 px 	100 x 100 px 
shape.geometric10	shape.diagonal13r
100 x 100 px 	100 x 100 px 
shape.diagonal13l	
100 x 100 px 	

Como pueden ver, son muchas las opciones de Shapes a escoger entre todas las Shapes que vienen por default en el **Kara Effector**, todo dependerá del efecto a realizar y de los resultados que queremos obtener.

Con el anterior listado de Shapes prediseñadas del **Kara Effector**, comenzamos con la descripción de las funciones de la Librería **shape**, en donde algunas de ellas nos sirven para modificar y crear nuestras propias Shapes o, generar efectos directamente.

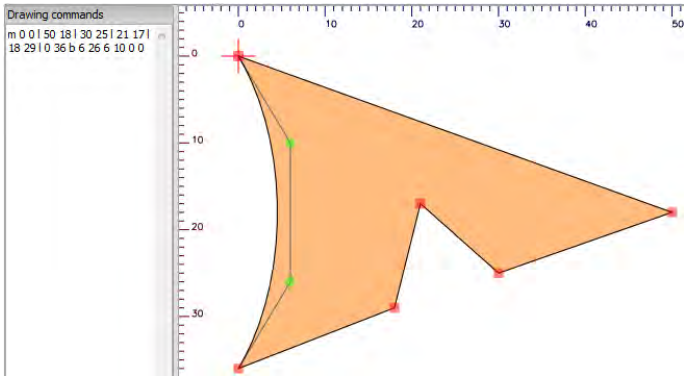
---

shape.rotate(Shape, Angle, x, y): rota a la shape en un ángulo dado respecto al punto **P = (x, y).**

Los parámetros **x** e **y** son opcionales al tiempo, sus valores por default son las coordenadas del punto **P = (0, 0).** El parámetro **Angle** hace referencia a un valor numérico de un ángulo entre 0° y 360°.

Para este y los próximos ejemplos usaremos un **Template Type: Line**, con el fin de generar una única línea por cada línea karaoke y así, sea más sencillo visualizar el resultado.

La **shape** que usaré en estos ejemplos será una muy simple, pero con un diseño en particular que permita ver la rotación de la misma:



Ejemplo:

Por lo general, siempre suelo declarar a las Shapes en la celda de texto "**Variable**", con el fin de hacer un poco más cómodo el uso de la misma, pero también se pueden usar directamente, entre comillas, dentro de las funciones que requieran una **shape** como alguno de los parámetros a usar dentro de ella:

Variables:

```
mi_shape = "m 0 0 l 50 18 l 30 25 l 21 17 l 18 29 l 0 36
           b 6 26 6 10 0 0 "
```

Las dos formas de usar la **shape** son válidos. Con el fin de que se visualice en pantalla, usaremos la función en la celda **Return [fx].**

Opción 1:

Return [fx]:

```
shape.rotate( mi_shape, 45 )
```

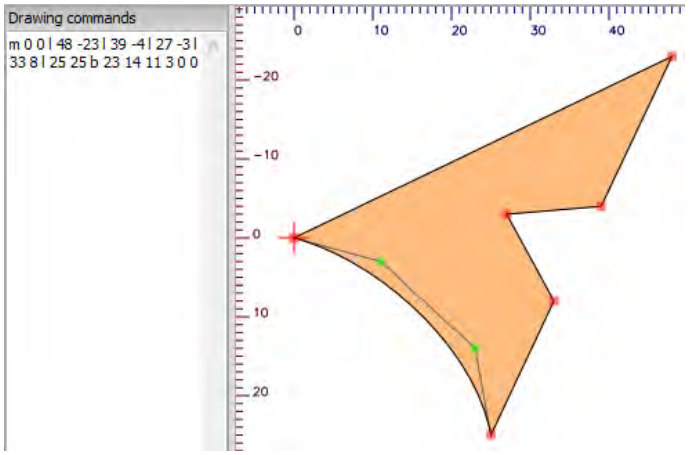
Opción 2:

Return [fx]:

```
shape.rotate( "m 0 0 l 50 18 l 30 25 l 21 17 l 18 29 l 0 36 b 6 26 6 10 0 0 ", 45 )
```

En vídeo:

En el **AssDraw3**:



La **shape** ha rotado 45° respecto al punto **P = (0, 0),** ya que en el ejemplo anterior se omitieron los parámetros **x** e **y**.

Hecho este ejemplo, la pregunta pareciera caer por su propio peso, ¿por qué no simplemente usar el tag \frz para rotar la **shape** en vez de esta función?

Cuando la shape rotada se va a usar como una simple **shape**, entonces sí es lo mismo usar el tag \frz para rotarla y la función **shape.rotate**, pero cuando se va a usar una shape para generar un \clip o un \iclip, entonces ningún tag puede afectar a las Shapes que se encuentren dentro de ellos. Ejemplo:

\clip(m 0 0 l 0 50 l 0 0)

Como la **shape** "m 0 0 l 0 50 l 0 0" está dentro del \clip, no hay forma de modificar las características de la misma, a menos que usemos las funciones de la Librería **shape**. Si quisiéramos rotar la **shape** dentro del \clip, lo haríamos de la siguiente forma. Ejemplo:

Add Tags: Add Tags Language: Automation Auto-4

```
\clip( !shape.rotate( "m 0 0 l 0 50 l 0 0", 30 )! )
```

Entonces se usará la **shape** rotada 30° dentro del \clip.

Nos resta ver ejemplos de cómo usar la función con los parámetros **x** e **y** incluidos. Ejemplos:

- **shape.rotate(mi_shape, 100, 0, 20)**
- **shape.rotate(mi_shape, 150, -30, 0)**
- **shape.rotate(mi_shape, 30, 45, -20)**
- **shape.rotate(mi_shape, 210, -15, -50)**

---

shape.reflect(Shape, Axis): refleja a la shape respecto al eje asignado en el parámetro **Axis**. Ejemplo:

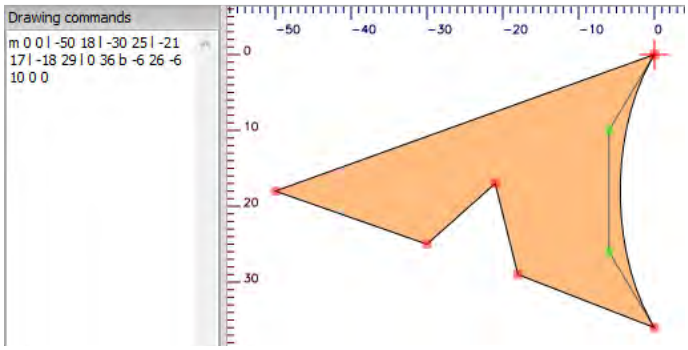
Return [fx]:

```
shape.reflect( mi_shape, 'y' )
```

El parámetro **Axis** puede ser "x" para hacer referencia al eje "x", o "y" para hacer referencia al eje "y", como se acaba de usar en el ejemplo inmediatamente anterior.

Lo que resultaría así:

Y en el **AssDraw3** se verá así:



Entonces decimos que si **Axis** es “x”, la **shape** se reflejará respecto al eje “x”; si es “y” se reflejará en dicho eje, pero hay una tercera opción que hace que la **shape** se refleje respecto a ambos ejes al mismo tiempo, así:

shape.reflect(mi_shape, “xy”)

Cuando **Axis** es “xy” la **shape** se refleja respecto a los ejes “x” e “y” de manera simultánea. Se obtiene el mismo resultado si solo no ponemos el parámetro **Axis**:

shape.reflect(mi_shape)

En ambos casos la **shape** será reflejada respecto a los dos ejes del plano.

Veamos un ejemplo combinando las dos funciones hasta acá vistas de la Librería **shape**:

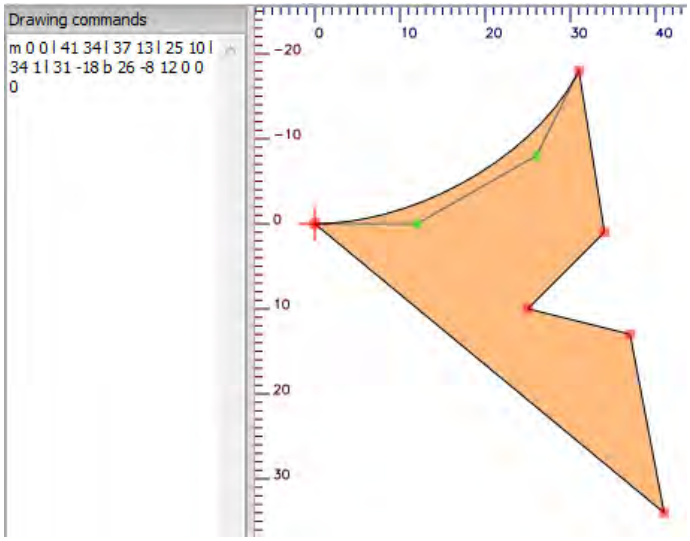
shape.reflect(shape.rotate(mi_shape, 60), “x”)

O sea que la **shape** primero se rota 60° y luego se refleja respecto al eje “x”:

Return [fx]:

```
shape.reflect( shape.rotate( mi_shape, 60 ), "x" )
```

Y en el **AssDraw3** podemos ver el suceso anteriormente descrito:



Son muchas las posibilidades al combinar tan solo estas dos funciones. Espero que puedan inventar sus propios ejemplos y que se puedan sorprender con los diferentes resultados.

Este **Tomo XIV** es solo la introducción de la Librería **shape**, ya que aún nos resta por ver muchas más funciones de la misma y todas las posibilidades que ellas nos ofrecen. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

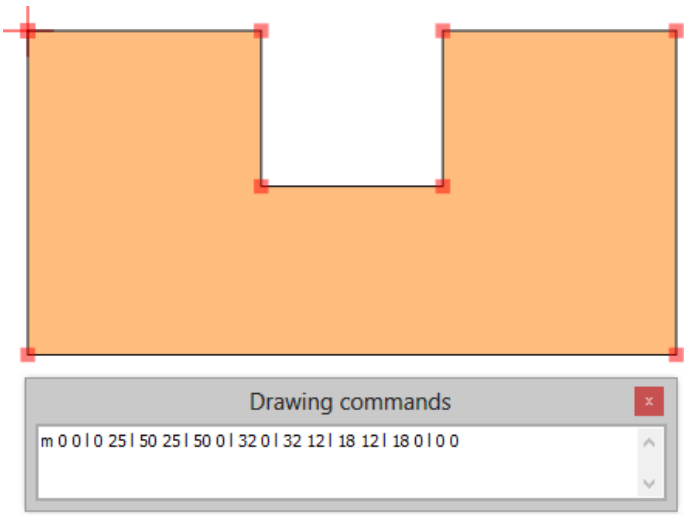
www.facebook.com/karaeffector

Kara Effector 3.2:

El **Tomo XV** es la continuación de la librería **shape**, ya que en el **Tomo** anterior vimos el listado de las Shapes que trae por default el **Kara Effector**, más un par de funciones de la Librería.

Librería Shape [KE]:

Para los siguientes ejemplos en las definiciones de las funciones de esta Librería, usaré esta simple **shape** que mide 50 X 25 px, pero ustedes pueden usar la que quieran y aun así los resultados deben ser visibles:



Y para mayor comodidad, la declaro en "**Variables**":

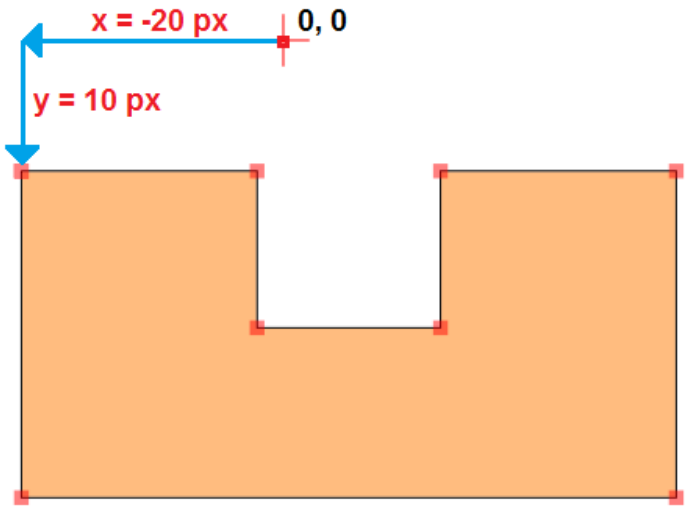
Variables:

```
mi_shape = "m 0 0 l 0 25 l 50 25 l 50 0 l 32 0 l 32 12 l 18
           12 l 18 0 l 0 0 "
```

shape.displace(shape, x, y): desplaza la **shape** tantos pixeles respecto a ambos ejes cartesianos, como le indiquemos en los parámetros **x** e **y**. Ejemplo:

```
Return [fx]:
shape.displace( mi_shape, -20, 10 )
```

Retorna la misma **shape**, pero desplazada 20 pixeles a la izquierda (x = -20) y 10 pixeles hacia abajo (y = 10):



Drawing commands

```
m -20 10 l -20 35 l 30 35 l 30 10 l 12 10 l 12 22 l -2 22 l -2 10 l -20 10
```

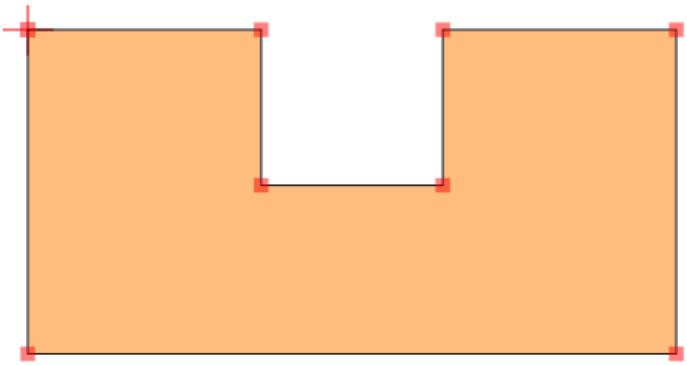
Recordemos que en el **AssDraw3** y en el formato .ass, el eje positivo de “y” es hacia abajo del eje “x” y el negativo, hacia arriba del mismo.

---

shape.origin(shape): desplaza la **shape** tantos pixeles respecto a ambos ejes cartesianos, hasta ubicarla en el **Cuadrante IV** del plano en el **AssDraw3**, respecto al punto de origen **P = (0, 0)**. Para el siguiente ejemplo usaré la **shape** desplazada del ejemplo anterior:

```
Return [fx]:
shape.origin( "m -20 10 l -20 35 l 30 35 l 30 10 l 12 10 l
12 22 l -2 22 l -2 10 l -20 10 " )
```

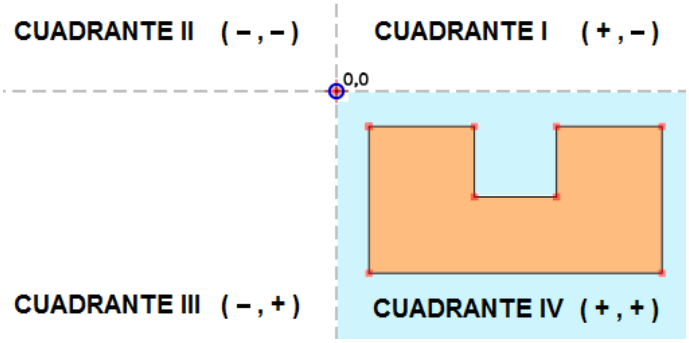
Entonces la función la ubicará en el **Cuadrante IV**:



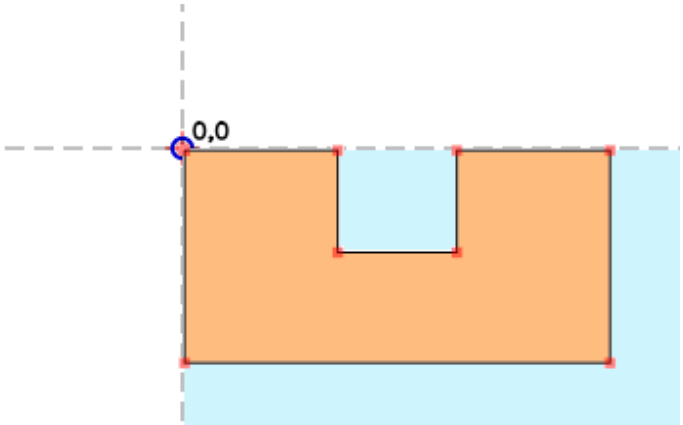
Drawing commands

```
m 0 0 l 0 25 l 50 25 l 50 0 l 32 0 l 32 12 l 18 12 l 18 0 l 0 0
```

En la siguiente imagen se muestran los **Cuadrantes** del **AssDraw3** y los signos que tienen ambas coordenadas en dichos Cuadrantes:



Y la función **shape.origin** desplaza la **shape**, en donde quiera que esté en el plano, al origen del **Cuadrante IV**, que es el Cuadrante en donde ambas coordenadas son positivas. Cuando una **shape** está ubicada en el origen del **Cuadrante IV**, se hace más sencillo aplicarle los tags de modificación y los resultados serán los esperados:

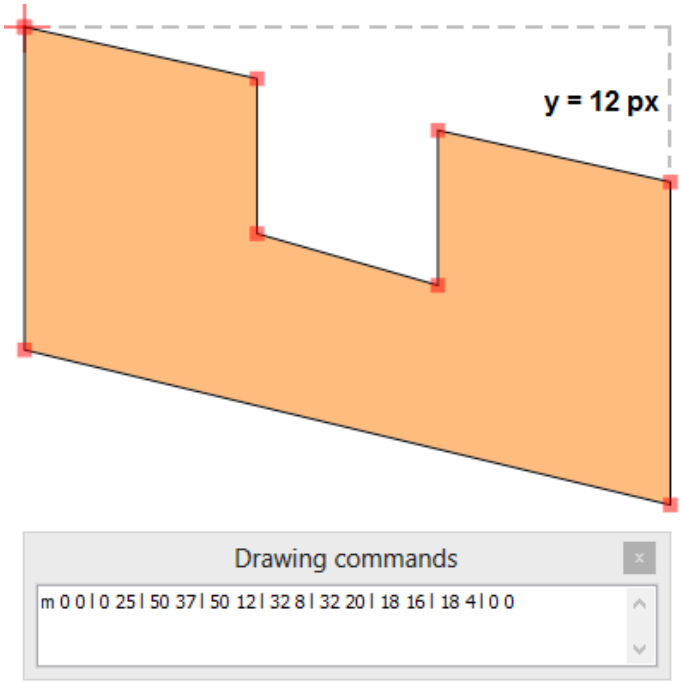


shape.oblique(shape, Pixels, Axis): deforma la **shape** de manera oblicua, en tantos pixeles indicados en el parámetro **Pixels**, respecto al eje asignado **Axis**.

- Ejemplo 1:

```
Return [fx]:
shape.oblique( mi_shape, 12, "y" )
```

Entonces la función deforma la **shape** 12 pixeles positivos respecto al eje "**y**":

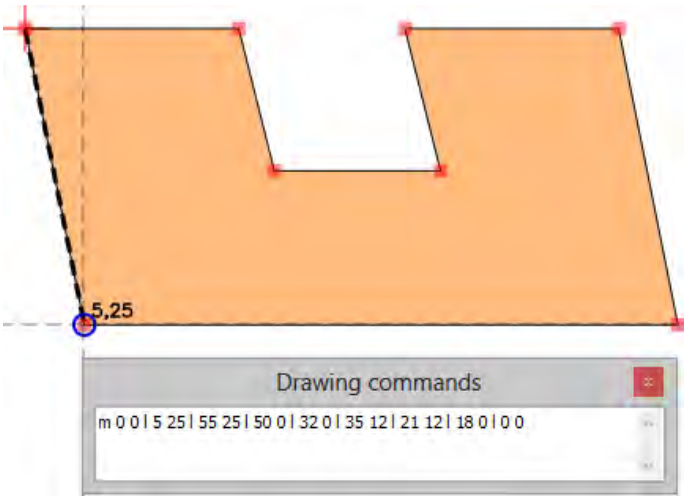


Todas las coordenadas en "x" se conservan, y las de "y" son desplazadas de forma progresiva hasta que aquellas que acompañan a las coordenadas "x" más alejadas del origen, se desplacen la cantidad de pixeles asignados en el parámetro **Pixels** (12 px). **Pixels** también puede ser un valor negativo, lo que deformaría la **shape** hacia arriba.

- Ejemplo 2:

```
Return [fx]:
shape.oblique( mi_shape, 5, "x" )
```

En este caso, la **shape** se deformará 5 pixeles hacia la derecha, dado que **Pixels** es positivo:

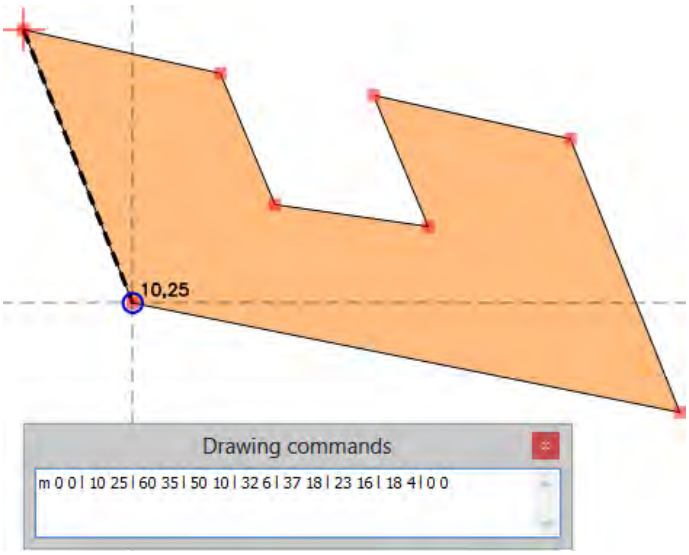


En el anterior ejemplo, las coordenadas que se conservan intactas en las **shape** son las de "y", y las coordenadas de "x" se deforman de forma progresiva.

- Ejemplo 3:

```
Return [fx]:
shape.oblique( mi_shape, 10 )
```

Ahora, al no poner el parámetro **Axis**, entonces la función deforma la **shape** en ambos ejes, en igual cantidad de pixeles (10 px); 10 px hacia la derecha (x = 10px) y 10 px hacia abajo (y = 10 px).



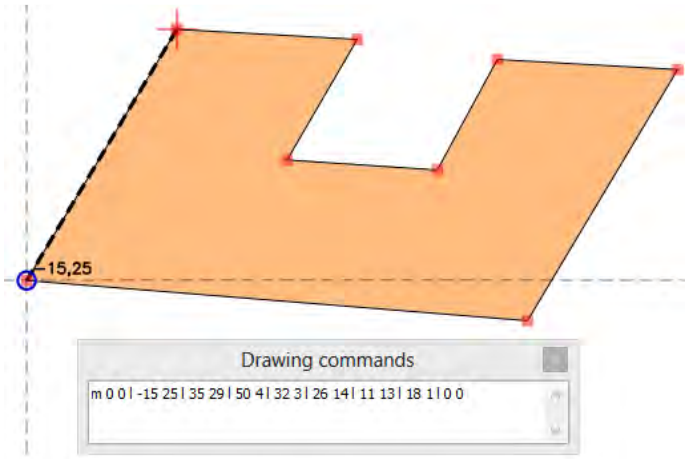
- Ejemplo 4:

Con este método podemos decidir la cantidad de pixeles en que se deformará la **shape** en ambos ejes:

Return [fx]:

```
shape.oblique( mi_shape, { -15, 4 } )
```

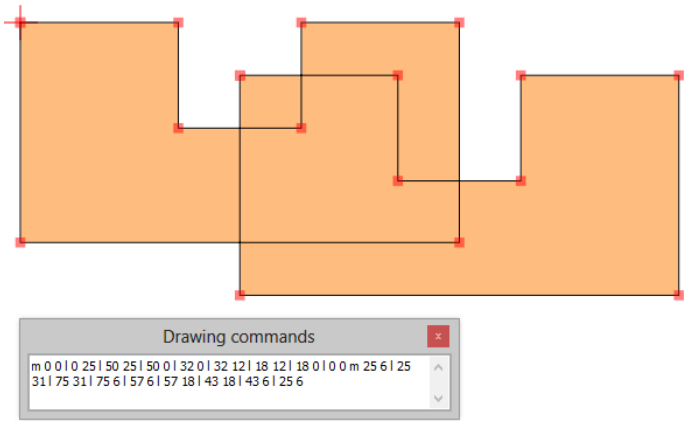
Vemos cómo la **shape** se deformó 15 pixeles a la izquierda (x = -15 px) y 4 pixeles hacia abajo (y = 4 px):



---

shape.reverse(shape): esta función reescribe la **shape** de manera que quede exactamente igual, pero dibujada a la inversa para que pueda ser sustraída de otra.

Para el siguiente ejemplo, dupliqué la misma **shape** y la desplacé varios pixeles respecto a la original, de manera que queden superpuestas como podemos ver en la siguiente imagen:

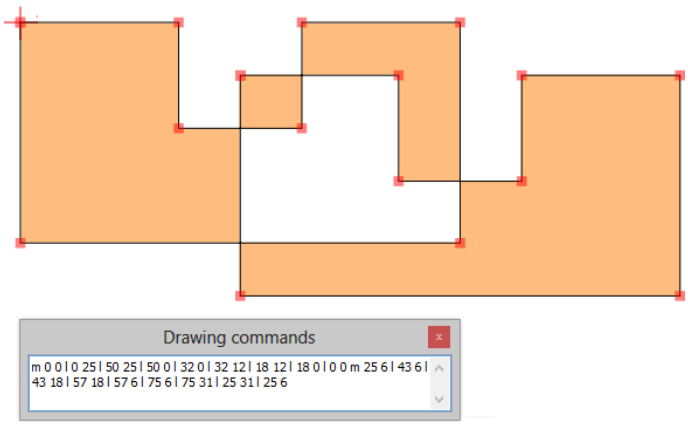


Ahora, aplicaremos la función a esa segunda **shape** para que sea redibujada de manera inversa:

Return [fx]:

```
shape.reverse( "m 25 6 l 25 31 l 75 31 l 75 6 l 57 6 l 57 18 l 43 18 l 43 6 l 25 6 " )
```

Las Shapes siguen siendo las mismas, pero el área que coincide entre ambas es sustraída, ya que una **shape** está dibujada en un sentido (sentido anti horario) y la otra a la inversa (sentido horario):



---

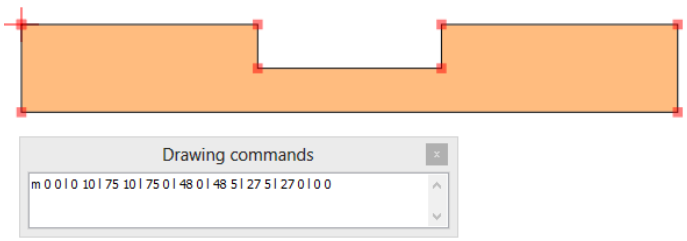
shape.ratio(shape, ratio_x, ratio_y): esta función redimensiona la **shape** en una proporción equivalente a **ratio_x** y **ratio_y**.

- Ejemplo 1:

Return [fx]:

```
shape.ratio( mi_shape, 1.5, 0.4 )
```

ratio_x = 1.5, es decir que la **shape** ahora es 1.5 veces más ancha de lo que era originalmente (150 %):



ratio_y = 0.4, es decir que la altura de la **shape** solo el 40% de la altura original.

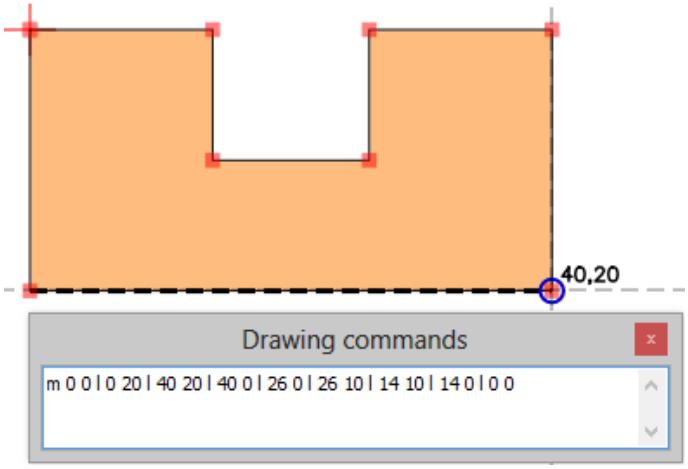
- Ejemplo 2:

En este modo se omite el parámetro **ratio_y**, entonces la función asume que **ratio_x** es la proporción del tamaño final, respecto a ambos ejes:

Return [fx]:

```
shape.ratio( mi_shape, 0.8 )
```

La **shape** que retorna es casi la misma, solo que su tamaño es un 80% (**ratio_x** = 0.8) del tamaño de la **shape** original:



---

shape.size(shape, size_x, size_y): esta función es similar a la función **shape.ratio**, pero con la diferencia que redimensiona la **shape** de tal manera que el ancho de la misma determinado según el parámetro **size_x** en pixeles, y su altura será determinada por el parámetro **size_y**, también en pixeles.

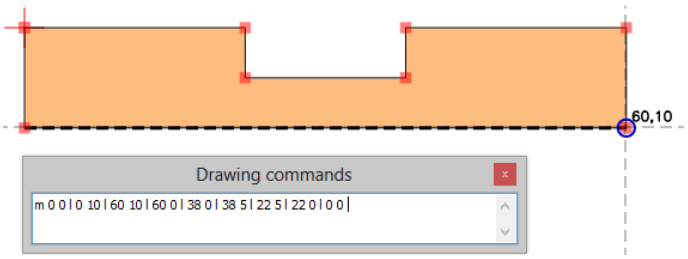
- Ejemplo 1:

De este modo, la **shape** quedará midiendo 60 X 10 px:

Return [fx]:

```
shape.size( mi_shape, 60, 10 )
```

Como se evidencia, tanto en la imagen de la **shape** como en el código de la misma, las dimensiones de la **shape** son las ingresadas en la función (60 X 10 px):



- Ejemplo 2:

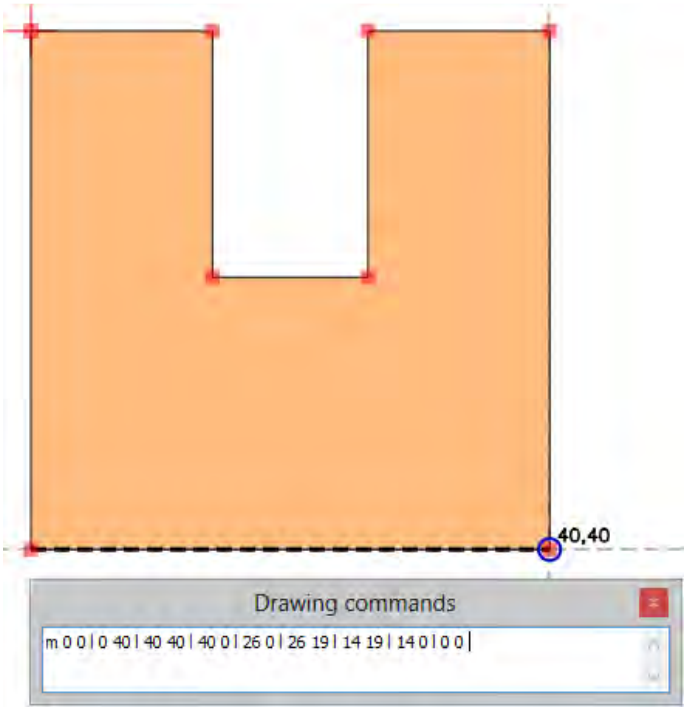
Se omite el parámetro **size_y**, de modo que la función asume que tanto el ancho como el alto de la **shape** serán el mismo, o sea **size_x**:

Return [fx]:

```
shape.size( mi_shape, 40 )
```

Usada la función de este modo, la **shape** queda con las medidas en pixeles que hayamos ingresado en la función, en este caso 40 px.

- Ancho = 40 px
- Alto = 40 px



shape.info(shape): brinda información primaria básica de la **shape**. Dicha información está dada en seis variables:

1. **minx**
2. **maxx**
3. **miny**
4. **maxy**
5. **w_shape**
6. **h_shape**

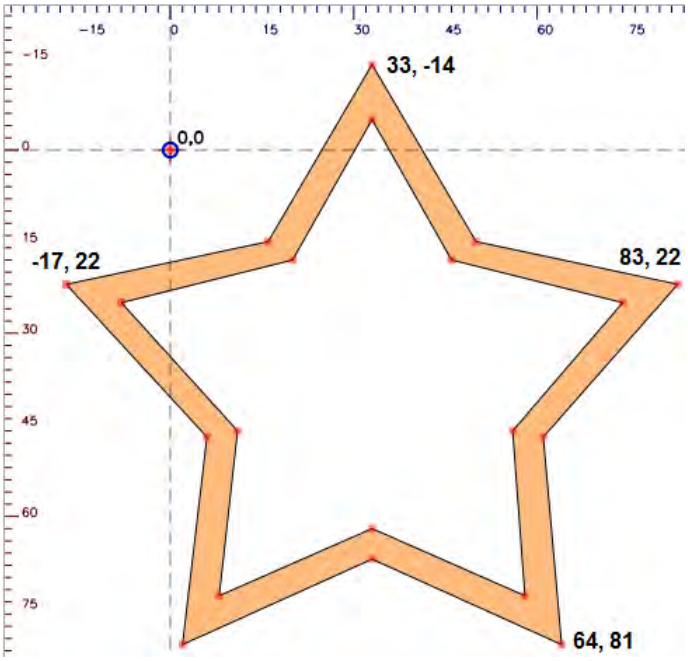
Ejemplo:

Declaramos una variable con el nombre que queramos y la igualamos a la función con una **shape**:

Variables:

```
Shape_info = shap.info( "m -17 22 l 6 47 l 2 81 l 33 67 l 64 81 l 61 47 l 83 22 l 50 15 l 33 -14 l 16 15 l -17 22 m -8 25 l 20 18 l 33 -5 l 46 18 l 74 25 l 56 46 l 58 73 l 33 62 l 8 73 l 11 46 l -8 " )
```

Esta es la shape que corresponde al código anterior:



Al llevar a cabo este procedimiento en la celda de texto "**Variables**", ya podemos usar las anteriores seis variables mencionadas, con los siguientes valores:

- **minx** = -17, que es el mínimo valor respecto a "x"
- **maxx** = 83, máximo valor en "x"
- **miny** = -14, mínimo valor en "y"
- **maxy** = 81, máximo valor en "y"
- **w_shape** = **maxx** – **minx** = 83 – (–17) = 100 px, corresponde al ancho de la shape ingresada.
- **h_shape** = **maxy** – **miny** = 81 – (–14) = 95 px, corresponde al alto de la shape ingresada.

Las anteriores seis variable ya pueden ser usadas como valores numéricos en cualquier otra celda de texto de la ventana de modificación del **Kara Effector**. Ejemplo:

Add Tags: Add Tags Language: Lua

```
"\\fscy" .. h_shape
```

Que a la postre retornará: \fscy85, dado que la altura de la **shape** ingresada era de 85 px.

Es todo por ahora para este **Tomo XV**, pero las funciones de la Librería **shape** aún no llegan a su fin. En el próximo **Tomo** continuaremos profundizando en el mundo de las Shapes y las posibilidades que nos ofrecen. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

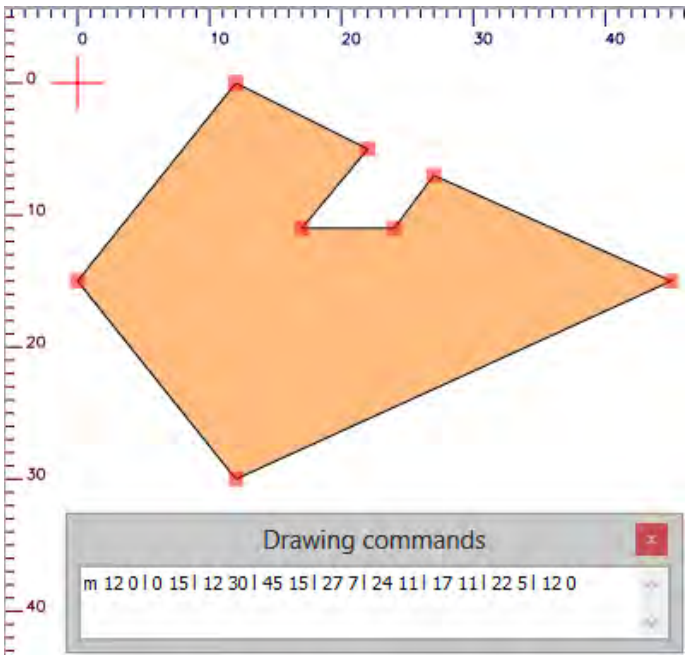
www.facebook.com/karaeffector

Kara Effector 3.2:

En el **Tomo XVI** seguimos con la funciones de la Librería **shape**, pero con la certeza que en este **Tomo** no terminará la misma, ya que es muy extensa y completa con el fin que tengamos a nuestra plena disposición la mayor cantidad de herramientas posibles a la hora de desarrollar nuestros proyectos.

Librería Shape [KE]:

Para los siguientes ejemplos en las definiciones de las funciones, usaremos esta simple **shape** que de 45 X 30 px:



Y como ya es costumbre, declaramos una variable con nuestra **shape** (el nombre es a gusto de cada uno):

Variables:

```
mi_shape = "m 12 0 l 0 15 l 12 30 l 45 15 l 27 7 l 24 11
            l 17 11 l 22 5 l 12 0 "
```

shape.array(shape, loops, A_mode, Dxy): hace un Arreglo o Matriz (duplicaciones) de la **shape**, una cierta cantidad de veces (**loops**), con diversas características y modalidades dependiendo de los otros dos parámetros (**A_mode** y **Dxy**).

Esta función tiene tres modalidades distintas, y cada una tiene una serie de opciones que veremos a continuación en los siguientes ejemplos:

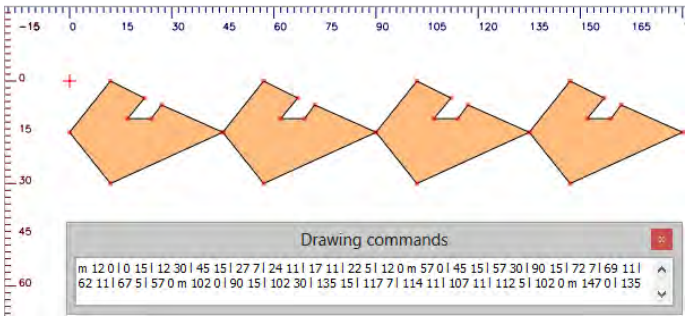
Modo Lineal

- **Ejemplo 1.** Ingresamos la **shape**, el número de repeticiones, que para este ejemplo es 4, y no se pone ni **A_mode** ni **Dxy**:

Return [fx]:

```
shape.array( mi_shape, 4 )
```

Entonces la **shape** se duplicará una a la derecha de la otra, cuatro veces, en forma lineal. El duplicar la **shape** una a la derecha de la otra es equivalente a un Arreglo Lineal con un ángulo de 0°:

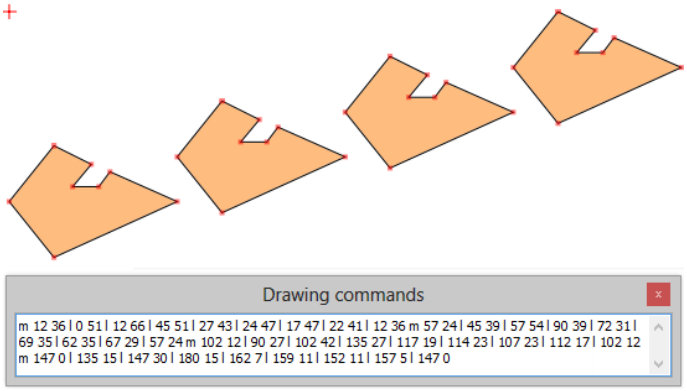


- **Ejemplo 2.** Aparte de la **shape** y la cantidad de repeticiones, el parámetro **A_mode** es 15, que es equivalente a un ángulo de 15°.
  Cuando **A_mode** es un valor numérico, la función asume que dicho valor es el ángulo que tendrá el Arreglo Lineal de la shape resultante:

Return [fx]:

```
shape.array( mi_shape, 4, 15 )
```

El Arreglo es Lineal y con un ángulo de 15°:

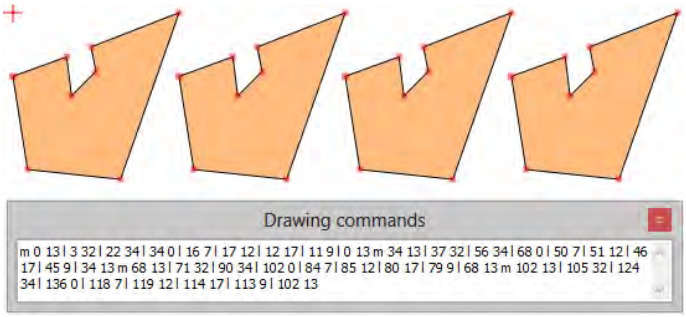


- **Ejemplo 3. A_mode** ahora es una **tabla** con dos valores numéricos, el primero indica el ángulo del arreglo lineal y el segundo, la rotación respecto al centro de la **shape**:

Return [fx]:

```
shape.array( mi_shape, 4, {0, 45} )
```

Entonces el ángulo del arreglo es 0 y la **shape**, antes de ser duplicada, se rotó un ángulo de 45°:

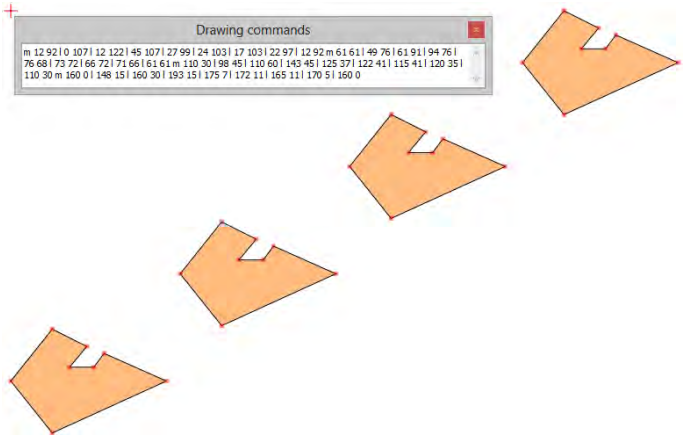


- **Ejemplo 4**. Incluimos al parámetro **Dxy** como valor numérico, que hace referencia a la distancia en px que separará a la **shape** dentro del arreglo:
  - loops : 4
  - Ángulo del Arreglo: 32°
  - Distancia Separadora: 5 px

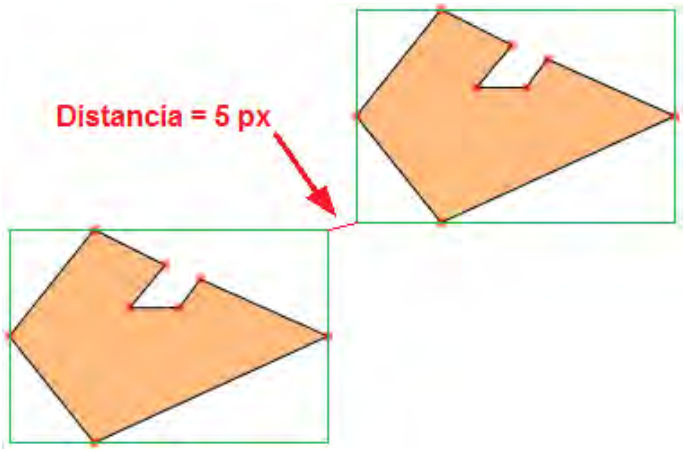
Return [fx]:

```
shape.array( mi_shape, 4, 32 , 5)
```

La distancia entre las Shapes del Arreglo ahora ya no es cero, que es su valor por default, sino 5 px:



En la anterior imagen queda la sensación de que las Shapes están separadas por una distancia mayor a 5 px, pero es porque la forma de la shape no es totalmente rectangular. Si tomamos dos de ellas y las enmarcamos en rectángulos, su pueden ver los 5 px de separación:



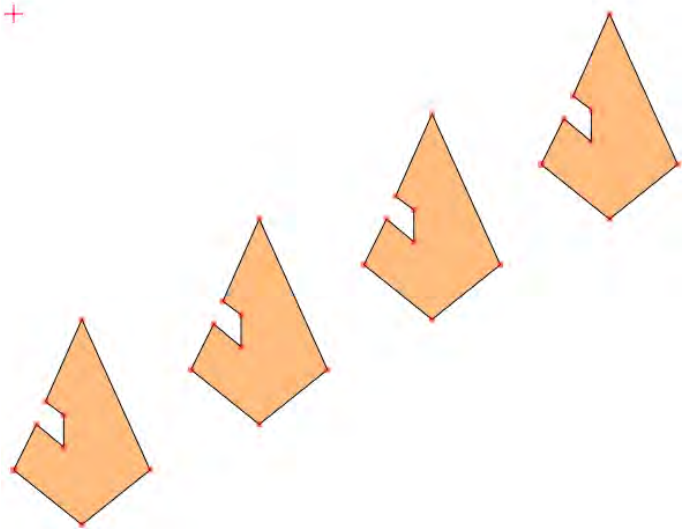
La distancia de separación también puede ser un valor numérico negativo, y lo que hará que el Arreglo Lineal quede más compacto y se acerquen tanto, una shape a la otra, hasta que se superpongan, si eso quisiéramos.

- **Ejemplo 5.** Es una combinación de los ejemplos 3 y 4. Ahora podemos decidir la separación, el ángulo del Arreglo y el ángulo de rotación:

Return [fx]:

```
shape.array( mi_shape, 4, {30, 90} , 10)
```

- loops: 4
- Ángulo del Arreglo: 30°
- Ángulo de Rotación de la **shape**: 90°
- Distancia Separadora: 10 px



Los anteriores ejemplos nos muestran las cinco opciones de Arreglos Lineales que podemos hacer con la función. El **Modo Lineal** es la forma más simple de usar esta función, pero a continuación veremos el próximo modo de uso y sus diversas opciones:

Modo Radial

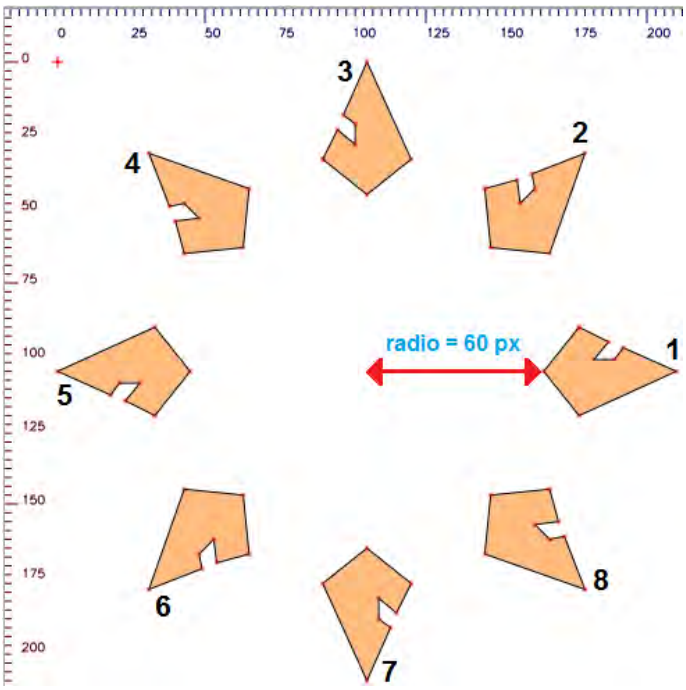
- **Ejemplo 1.** Ingresamos la **shape**, el número de repeticiones del Arreglo, en **A_mode** escribimos entre comillas la palabra "**radial**". **Dxy** equivale al radio del arco:

Return [fx]:

```
shape.array( mi_shape, 8, "radial", 60 )
```

Vemos la **shape** repetida ocho veces. El ángulo del arco es 360° por default, y es por eso que la **shape** se repite en un Arreglo Radial, equidistantemente en esos 360°:

- loops: 8
- Modo: “radial”
- Radio: 60 px



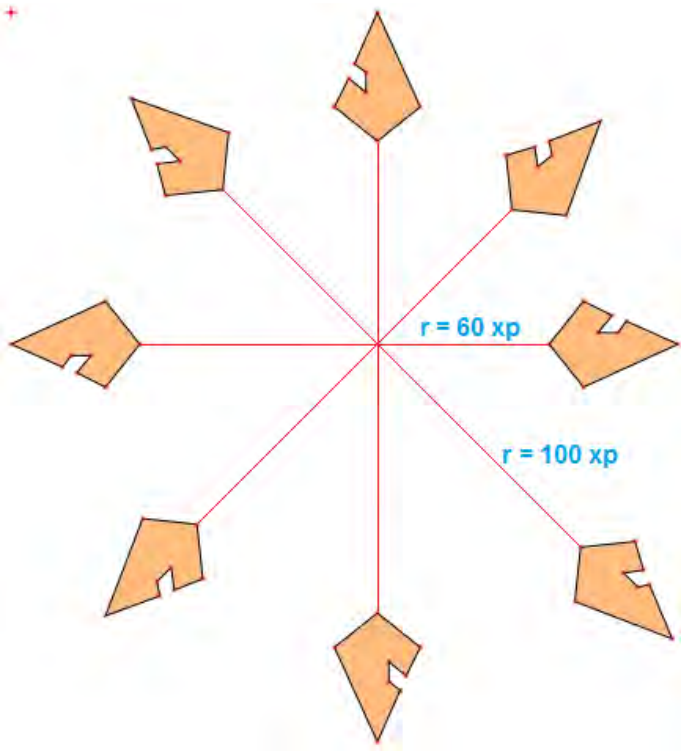
El radio es constante para cada una de las repeticiones del Arreglo (60 px), y cada una de las Shapes está rotada de tal manera que quedan orientadas en dirección del centro imaginario del Arreglo.

- **Ejemplo 2.** Ingresamos la **shape**, el número de repeticiones del Arreglo, en **A_mode** escribimos entre comillas la palabra “**radial**”. Pero ahora **Dxy** es una **tabla** que contiene dos valores numéricos, el primero equivale al radio inicial en pixeles del Arreglo y el segundo al radio final:
  - loops: 8
  - Modo: “radial”
  - Radio Inical: 60 px
  - Radio Final: 100 px

Return [fx]:

```
shape.array( mi_shape, 8, "radial", {60, 100} )
```

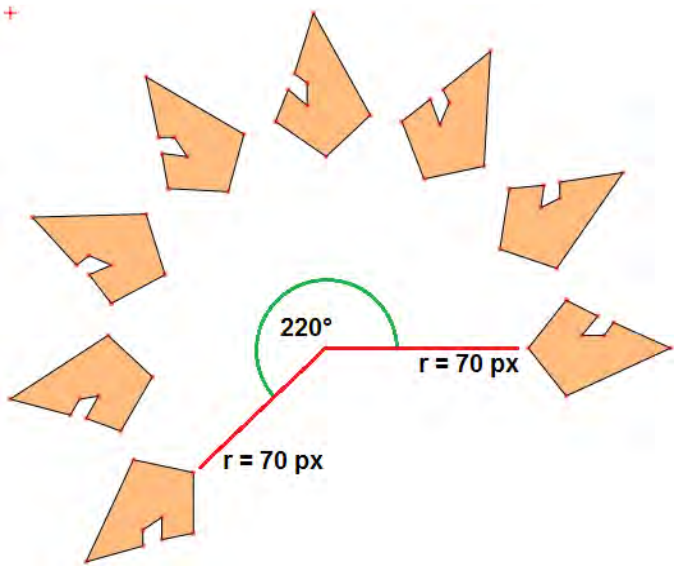
Los radios del Arreglo van aumentando progresivamente desde el Radio Inicial (60 px) hasta el Radio Final (100 px):



- **Ejemplo 3.** Agregamos un tercer valor en la **tabla Dxy**, equivalente al ángulo del arco del Arreglo, que por default era 360°:

Return [fx]:

```
shape.array( mi_shape, 8, "radial", {70, 70, 220} )
```

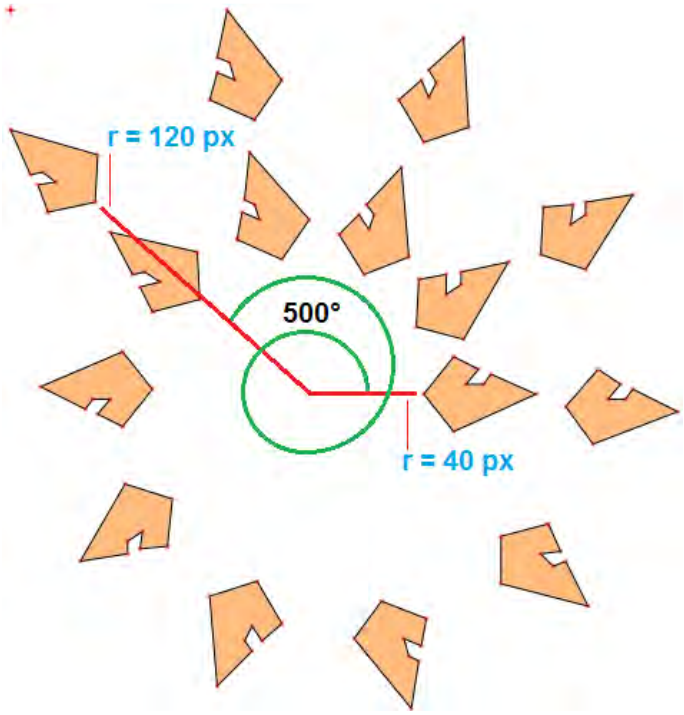


- **Ejemplo 3.1.** Aumentamos el valor del ángulo del Arreglo de tal manera que exceda los 360° de la circunferencia y, ponemos al Radio Inicial y al Radio Final con diferentes valores para evitar que las Shapes se superpongan:

Return [fx]:

```
shape.array( mi_shape, 15, "radial", {40, 120, 500} )
```

  - loops: 15
  - Modo: “radial”
  - Radio Inical: 40 px
  - Radio Final: 120 px
  - Ángulo del Arco: 500°



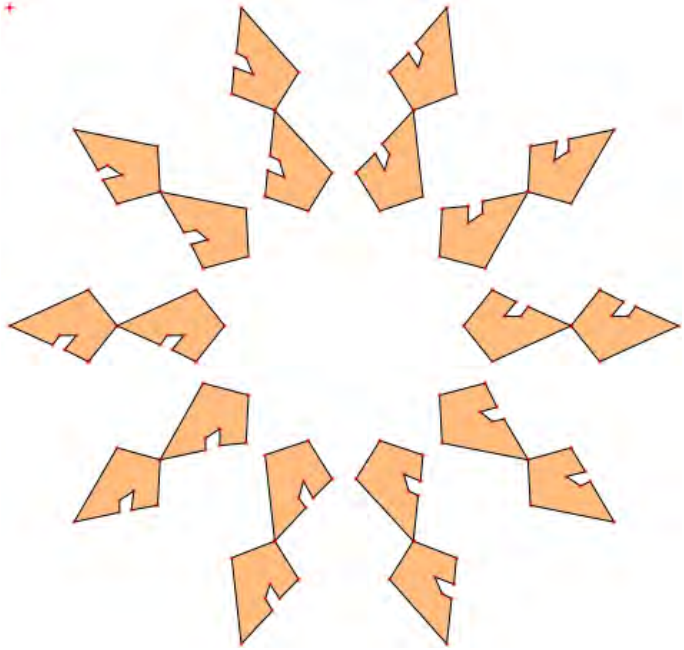
- **Ejemplo 4.** El parámetro **loops** es una **tabla** con dos valores numéricos, el primero indica las veces que se repite la **shape** en el Arreglo Radial, y el segundo indica la veces que se repite el Arreglo de forma y tamaño mayormente progresivo:

Return [fx]:

```
shape.array( mi_shape, {10, 2}, "radial", 50 )
```

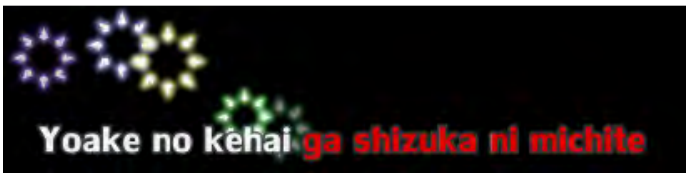
  - loops del Arreglo: 10
  - Repeticiones: 2
  - loops Total: 10 X 2 = 20
  - Modo: “radial”
  - Radio Interior del primer Arreglo: 50 px

Al usar la función de esta forma, la separación entre cada uno de los Arreglos Radiales es por default 0, es decir que el radio interior de cada Arreglo Radial será exactamente del mismo tamaño en pixeles del radio exterior del Arreglo inmediatamente anterior:



La **shape** resultante es cada vez más compleja y resultaría muy laborioso dibujarla manualmente en el **AssDraw3**, además del hecho de no poder hacerlo con tanta precisión y velocidad. Del anterior ejemplo resulta una **shape** muy interesante para hacer un Efecto:

Con la **shape** del **Ejemplo 1**:

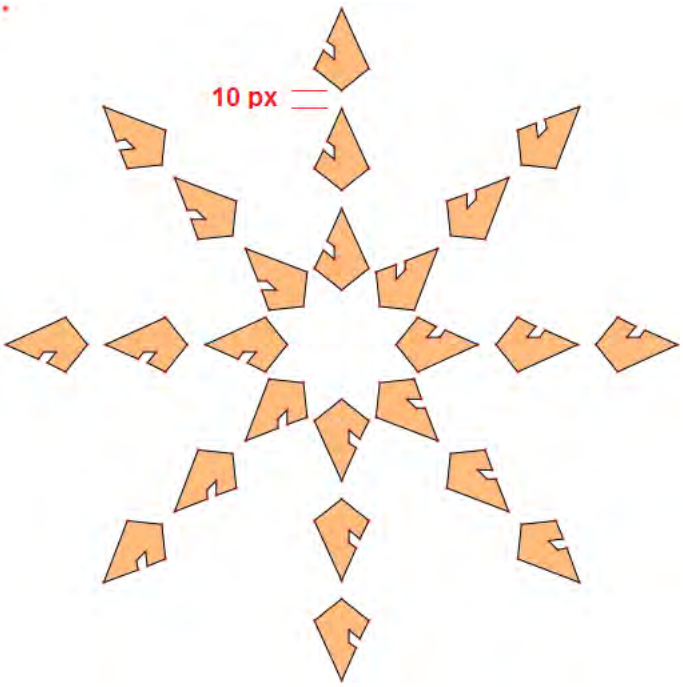


- **Ejemplo 5.** El parámetro **Dxy** es ahora una **tabla** con dos valores numéricos, el primero equivale al radio interior del primer Arreglo y el segundo, a la distancia en pixeles que separará a cada uno de los Arreglos:

Return [fx]:

```
shape.array( mi_shape, {8, 3}, "radial", {30, 10} )
```

  - loops del Arreglo: 8
  - Repeticiones: 3
  - loops Total: 8 X 3 = 24
  - Modo: “radial”
  - Radio Interior del primer Arreglo: 30 px
  - Distancia Separadora entre Arreglos: 10 px



Para los siguientes ejemplos convertiremos a la variable **mi_shape** en una **tabla**, en donde el primer elemento será la shape que inicialmente ya teníamos y el segundo será un círculo de 24 px de diámetro:

Variables:

```
mi_shape = { "m 12 0 l 0 15 l 12 30 l 45 15 l 27 7 l 24 11
l 17 11 l 22 5 l 12 0 ", shape.size(shape.circle, 24) }
```

Círculo de 24 px

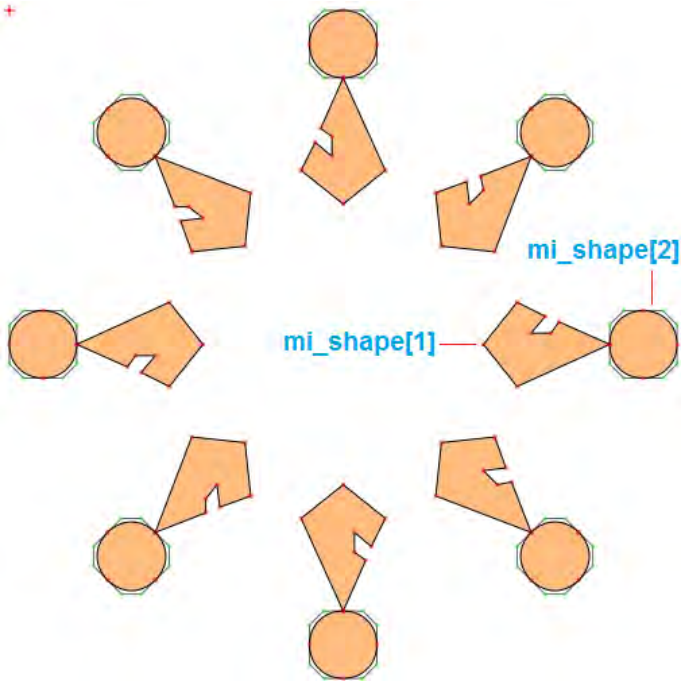
La **tabla mi_shape** de ser completamente de Shapes para poder ser usada dentro la función, así que pueden usar las de estos ejemplos o las que ustedes quieran. La cantidad de Shapes de la **tabla** es ilimitada.

- **Ejemplo 6. mi_shape** es una **tabla** de Shapes:
  - loops del Arreglo: 8
  - Repeticiones: 2
  - loops Total: 8 X 2 = 16
  - Modo: “radial”
  - Radio Interior del primer Arreglo: 50 px

Return [fx]:

```
shape.array( mi_shape, {8, 2}, "radial", 50 )
```

La función toma a la primera **shape** de la **tabla** y la repite en el primer Arreglo, luego toma la segunda para el segundo Arreglo, y así de manera sucesiva para el caso en que la **tabla mi_shape** tenga todavía más elementos.



En la anterior imagen notamos cómo el primer Arreglo está hecho con las repeticiones de **mi_shape[1]** (o sea, el primer elemento de la **tabla mi_shape**) y para el segundo Arreglo se usó **mi_shape[2].**

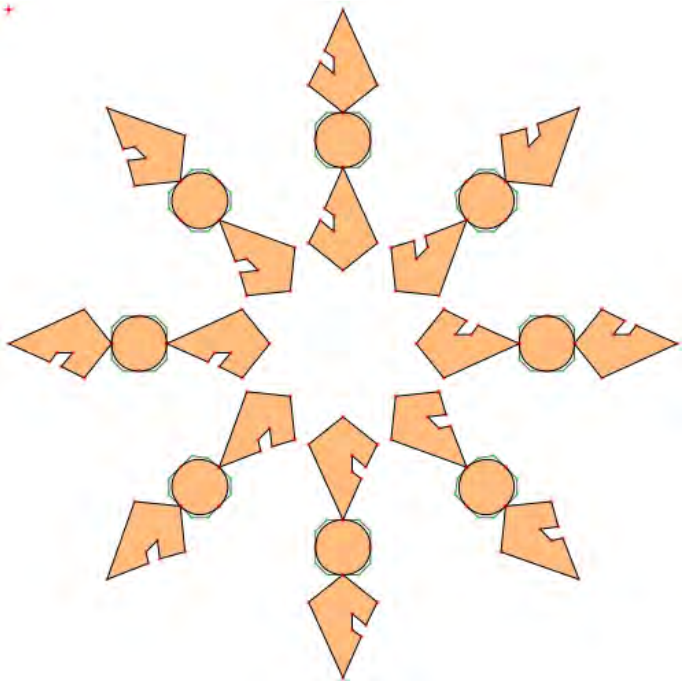
La distancia que separa un Arreglo respecto a otro es 0 px, por default, pero también la podemos modificar.

- **Ejemplo 6.1. mi_shape** es una **tabla** de Shapes:
  - loops del Arreglo: 8
  - Repeticiones: 3
  - loops Total: 8 X 3 = 24
  - Modo: “radial”
  - Radio Interior del primer Arreglo: 32 px

Return [fx]:

```
shape.array( mi_shape, {8, 3}, "radial", 32 )
```

Este ejemplo está hecho para mostrar que no importa que la cantidad de repeticiones de los Arreglos asignada en la tabla **loops** (o sea 3) exceda a la cantidad total de Shapes en la **tabla mi_shape** (**#mi_shape = 2**). La función tomará para el tercer Arreglo nuevamente a la primera **shape** de la **tabla**:

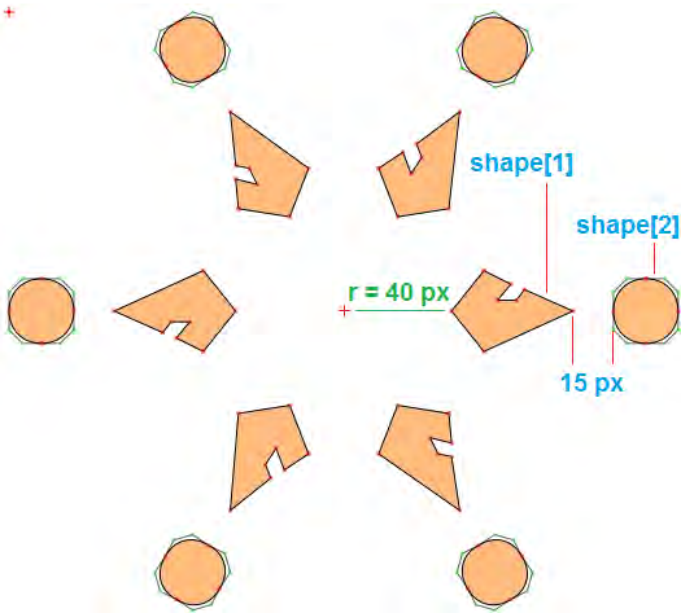


- **Ejemplo 7. mi_shape** es una **tabla** de Shapes y en **Dxy**, en forma de **tabla**, modificamos la distancia que separará a cada uno de los Arreglos:

Return [fx]:

```
shape.array( mi_shape, {6, 2}, "radial", {40,15} )
```

- loops del Arreglo: 6
- Repeticiones: 2
- loops Total: 6 X 2 = 12
- Modo: “radial”
- Radio Interior del primer Arreglo: 40 px
- Distancia Separadora: 15 px



Acá es el final del **Tomo XVI**, pero la función **shape.array** aún tiene muchos más secretos que revelarnos y quedan muchos ejemplos para poner en práctica. Literalmente las opciones son infinitas.

En el **Tomo XVII** continuaremos con el tercer **Modo** de uso de la función **shape.array** y con las demás funciones de la Librería **shape**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

www.facebook.com/karaeffector

Kara Effector 3.2:

En el **Tomo XVII** seguiremos con los Ejemplos del Modo "**radial**" de la función **shape.array** de la Librería shape, y con el inicio del último Modo de dicha función.

Librería Shape [KE]:

El **tomo XVI** había terminado con el Ejemplo 7 del Modo "**radial**", en donde **mi_shape** es una **tabla** en vez de una simple **shape**:

Variables:

```
mi_shape = {"m 12 0 l 0 15 l 12 30 l 45 15 l 27 7 l 24 11 l
17 11 l 22 5 l 12 0 ", shape.size(shape.circle, 24) }
```

shape.array(shape, loops, A_mode, Dxy):

Modo Radial

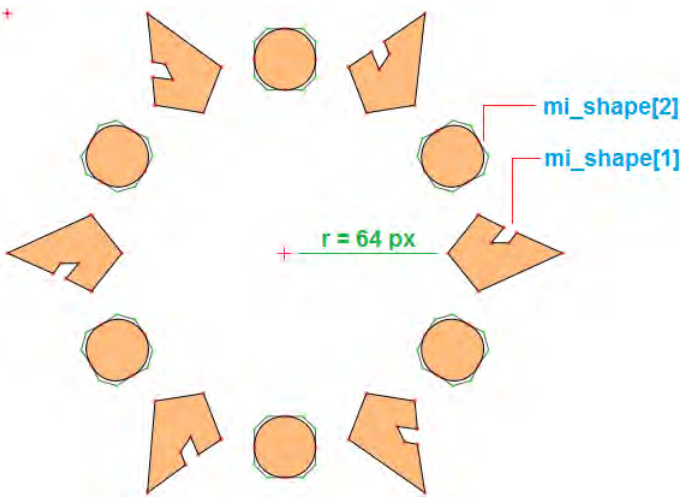
- **Ejemplo 8.** Ingresamos la **tabla mi_shape**, el **loops** vuelve a ser un valor numérico, que para este ejemplo es 6:

Return [fx]:

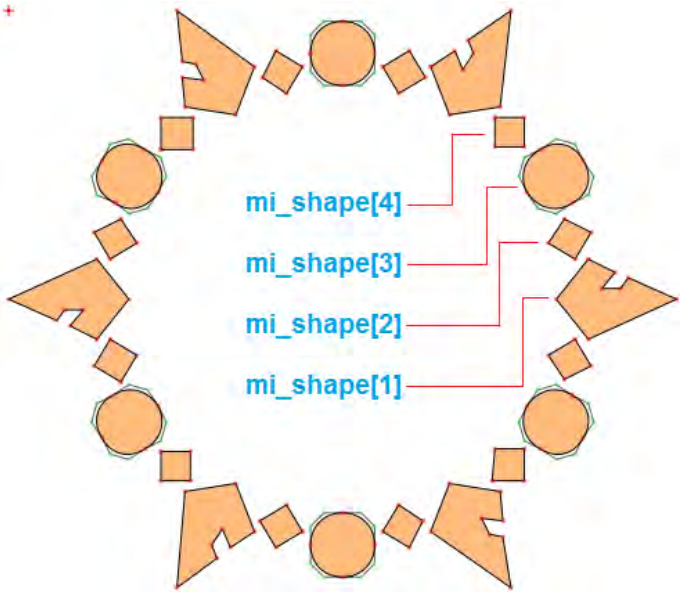
```
shape.array( mi_shape, 6, "radial", 64 )
```

- loops de cada **shape** del Arreglo: 6
- Tamaño de la **tabla**: 2
- loops Total: 6 X 2 = 12
- Modo: "radial"
- Radio Interior del Arreglo: 64 px

Entonces la función hace el **Arreglo Radial** alternando seis veces a cada uno, a los elementos de la **tabla mi_shape**, a un radio de 64 px:



Acá otro ejemplo con una tabla de cuatro Shapes, en la que convenientemente las Shapes 2 y 4 son las mismas:



Modo Matricial

Este modo consiste en hacer los **Arreglos** a modo de una **Matriz** rectangular con una cierta cantidad de repeticiones de la **shape** en ambas direcciones.

Ejemplo 1. Nuevamente iniciamos con **mi_shape** como una **shape**, declarada en forma de variable. El parámetro **loops** siempre debe ser una **tabla** con dos valores numéricos en donde se dan la cantidad de repeticiones, el primer número indica las repeticiones de la **shape** respecto al eje "**x**" (a lo ancho) y el segundo número indica la cantidad de repeticiones respeto al eje "**y**" (a lo alto). Por último, en **A_mode** ponemos la palabra "**array**":

Variables:

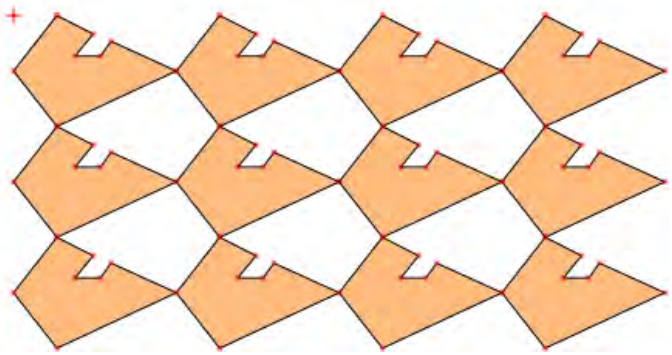
```
mi_shape = "m 12 0 l 0 15 l 12 30 l 45 15 l 27 7 l 24 11 l
            17 11 l 22 5 l 12 0 "
```

Y en **Return [fx]** ponemos así:

Return [fx]:

```
shape.array( mi_shape, {4, 3}, "array" )
```

Se generará el siguiente arreglo:



- loop Horizontal: 4
- loop Vertical: 3
- loops Total: 4 X 3 = 12
- Modo: "array"
- Distancia Separadora: 0 px (por default)

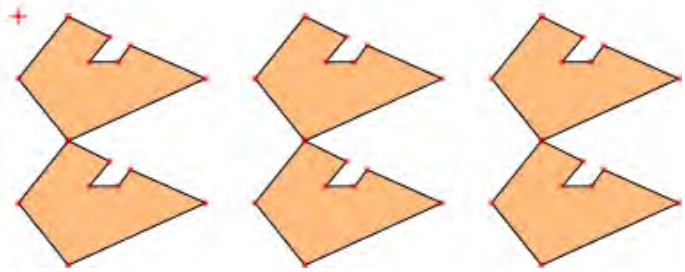
Ejemplo 2. Usamos las mismas configuraciones del ejemplo anterior, pero ahora incluimos a **Dxy** en forma de valor numérico:

Return [fx]:

```
shape.array( mi_shape, {3, 2}, "array", 12 )
```

- loop Horizontal: 3
- loop Vertical: 2

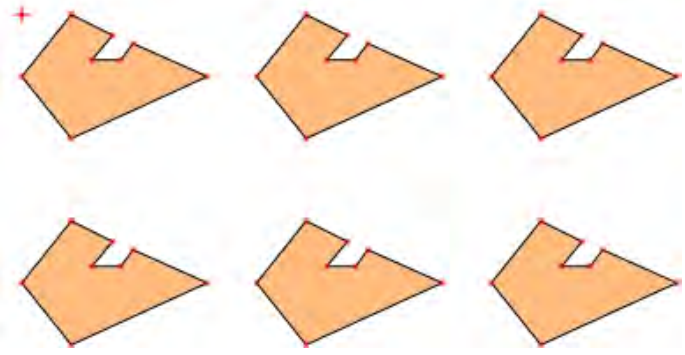
- loops Total: 3 X 2 = 6
- Modo: “array”
- Distancia Separadora Horizontal: 12 px
- Distancia Separadora Vertical: 0 px (default)



Ejemplo 3. Modificamos a **Dxy** a una **tabla** con dos valores numéricos, el primero indicará la distancia separadora horizontal y el segundo la vertical:

```
Return [fx]:
shape.array( mi_shape, {3, 2}, "array", {12, 20} )
```

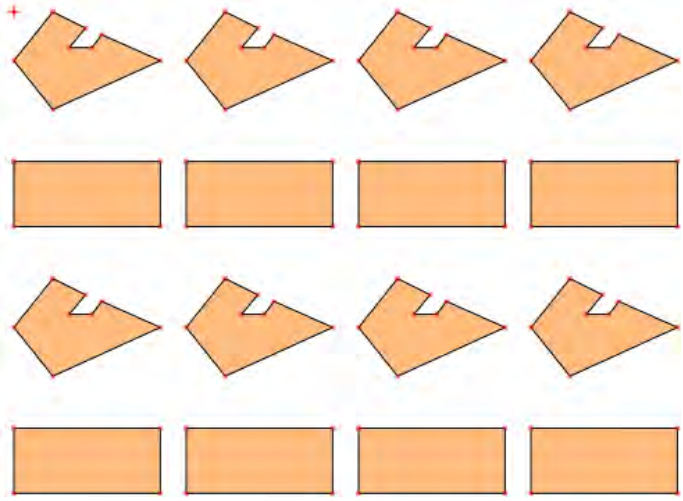
- loop Horizontal: 3
- loop Vertical: 2
- loops Total: 3 X 2 = 6
- Modo: “array”
- Distancia Separadora Horizontal: 12 px
- Distancia Separadora Vertical: 20 px



Ejemplo 4. mi_shape la usamos ahora como una **tabla** de Shapes, la cantidad de elementos de dicha **tabla** es decidida por cada quien:

```
Return [fx]:
shape.array( mi_shape, {4, 4}, "array", {8, 16} )
```

Entonces las Shapes de la **tabla mi_shape** se alternarán en el **Arreglo**, similar como pasaba en el **Arreglo Radial**:



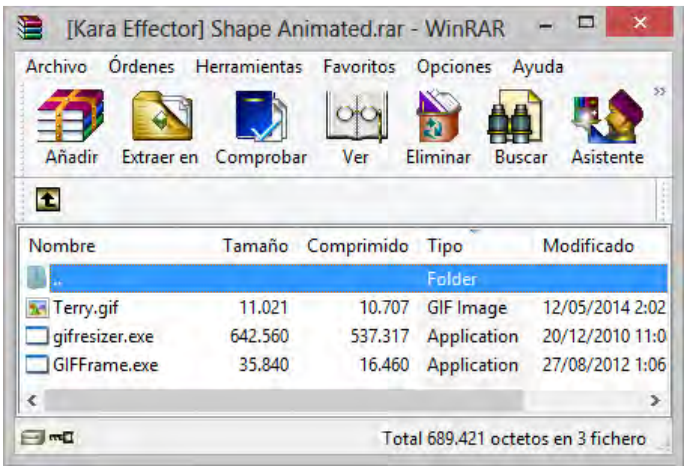
---

shape.animated(dur, frame_dur, frames, Sx, Sy): retorna Shapes rectangulares que contienen imágenes en formato **PNG** para hacer efectos de animaciones.

Esta función es de uso exclusivo para el filtro **VSFilterMod**, ya que está basado en el tag **\1img**, que es el que hace posible el poder insertar imágenes en formato **PNG** en un archivo .ass y por ello no es posible poder visualizar los resultados si solo usamos el **VSFilter 2.39**.

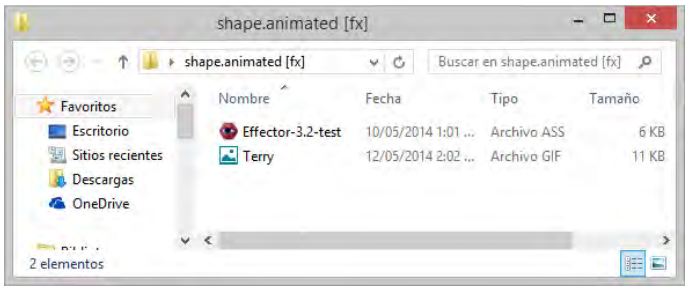
Para el siguiente ejemplo he subido un **.rar** que contiene tres archivos, un archivo **GIF** y dos aplicaciones que no requieren de instalación, que nos ayudarán a manipular y modificar los archivos **GIF**:

[Kara Effector] Shape Animated



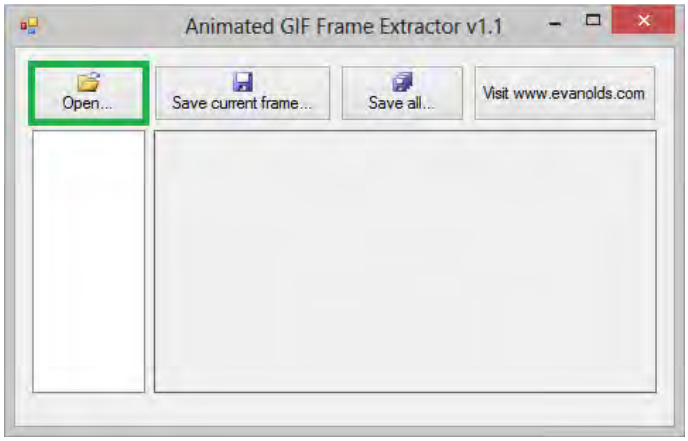
1. **Terry.gif:** es un ejemplo de imágenes animadas que pueden usar para un efecto en esta función.
2. **Gifresizer:** es una aplicación que nos permite dar las dimensiones a las imágenes que necesitemos, para que la animación se ajuste a las proporciones de las líneas karaoke y del vídeo usado.
3. **GIFFrame:** es una aplicación que extrae cada uno de los **frames** de un archivo **GIF**, en formato **PNG**, para poder insertarlas en nuestros karaokes.

El primer paso para usar la función **shape.animated** es tener el archivo **.ass** y el archivo **GIF** en la misma carpeta. No importa el nombre ni la ubicación de la carpeta, lo que importa es que ambos archivos estén en la misma:

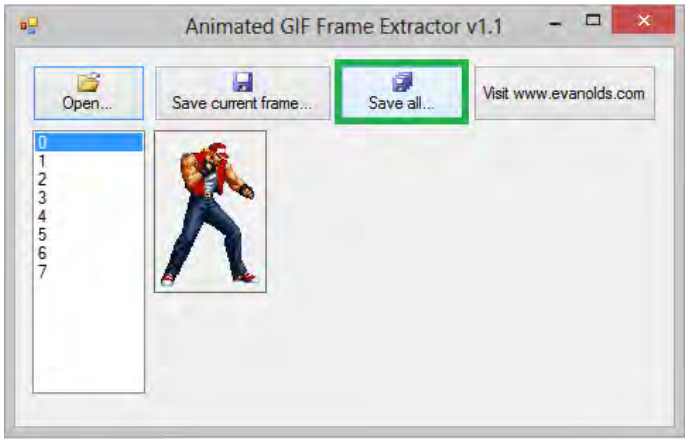


En el caso que nuestra **GIF** tenga un tamaño superior o inferior al que necesitamos (aunque no es recomendable ampliar un **GIF**, ya que la imagen pixela y empieza a perder calidad), abrimos la aplicación **GIFResizer** y en la parte inferior le damos las dimensiones en pixeles que se ajuste a nuestras necesidades. Esta aplicación crea un nuevo archivo, en tal caso se debe guardar en la misma carpeta del archivo **.ass** y del **GIF** original:

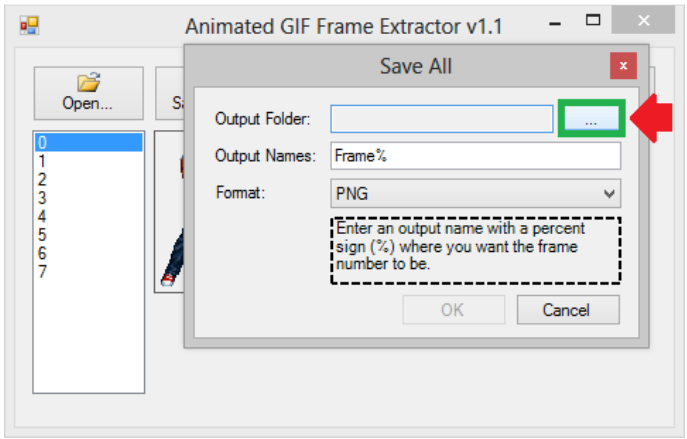
Para este ejemplo, no hubo la necesidad de cambiar el **GIF** de tamaño, pero en cualquiera de los dos casos, el tercer paso es abrir la aplicación **GIFFrame** y le damos al botón que pone "**Open…**":



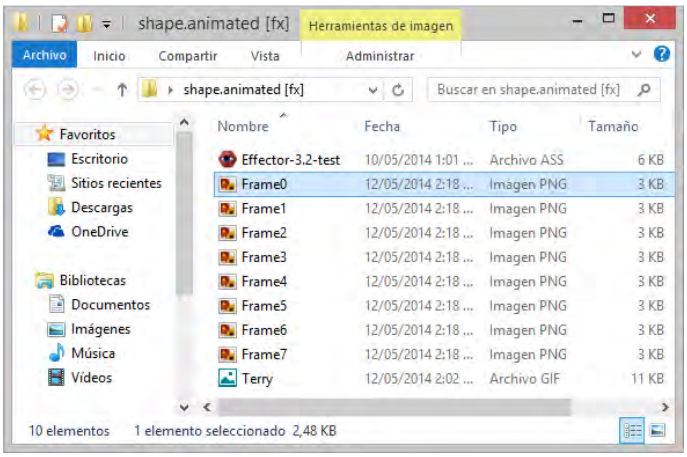
Ubicamos el **GIF** que vamos a utilizar, y pulsamos el botón "**Save All…**":



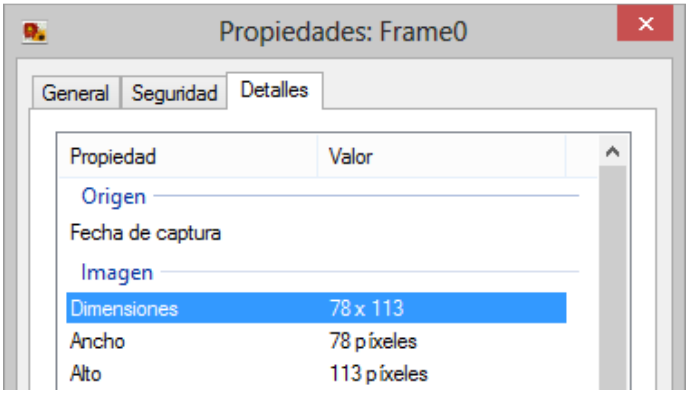
Y en donde pone "**Output Folder**" ubicamos la carpeta en donde están el archivo **.ass** que usaremos y el **GIF**, y en donde pone "**Format**" siempre debe poner "**PNG**":



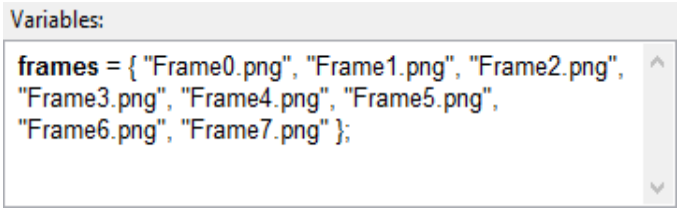
Entonces la aplicación **GIFFrame** extrae las imágenes **PNG** que componen al **GIF**, y éstas quedan en la carpeta en donde estaban el archivo **.ass** y el **GIF**:



Para confirmar las dimensiones de los archivos **PNG**, las podemos ver en sus propiedades (78 X 113 px):



Hasta este punto, podemos decir que hemos hechos todos los pasos preliminares para usar la función y generar un efecto de animación. Los pasos siguientes ya los debemos realizar desde el **Kara Effector** directamente, y uno de ellos es crear una tabla en la celda de texto "**Variables**" que contenga, entre comillas, el nombre de cada una de las imágenes y su extensión (.**png**):



Para este ejemplo, tan solo son ocho imágenes, desde la 0 hasta la 7, y en algunos casos hay archivos **GIF** que están compuestos de mucho más.

Y el segundo paso en la ventana de modificación del **Kara Effector** es llamar a la función **shape.animated** en la celda de texto **Return [fx]:**

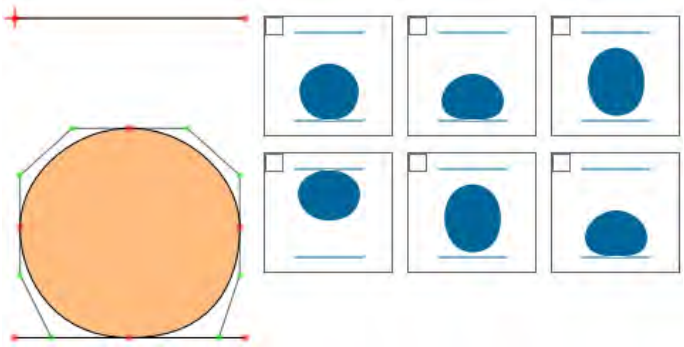
1. En el primer parámetro de la función ponemos el tiempo total de duración de la animación, para este ejemplo he usado la variable **fx.dur**, que ya sabemos que es el tiempo de cada una de las líneas de efecto generadas.
2. En el segundo parámetro debemos poner la duración en milisegundos que tendrá cada uno de los **frames** de la animación de nuestro efecto. Para este ejemplo, **100** ms. Son recomendables valores entre 40 y 300 ms.
3. Para el tercer parámetro ponemos el nombre de la **tabla** declarada en "**Variables**", que para este ejemplo se llama **frames**.
4. Y los parámetros cuatro y cinco son el ancho y el alto en pixeles de cada una de las imágenes **PNG**.

El resto de las configuraciones dependerá del efecto que queremos hacer, configuraciones como el **Template Type**, las posiciones y los tiempos. Para este ejemplo usé un **Template Type: Line**, con posición izquierda de la línea y con los tiempos por default de la misma:

Para el caso en que a la derecha o en la parte superior de las imágenes de la animación se vea alguna parte de otra imagen, debemos reducir las medidas ingresadas, si por ejemplo se ve a la derecha, reducimos un poco al ancho.

shape.animated2(dur, frame_dur, shapes): es similar a **shape.animated**, pero con la diferencia que no retorna imágenes **PNG** sino Shapes. Otra de las diferencias con su función homónima, es que solo son necesarios tres parámetros para que haga la animación: la duración total, la duración de cada **frame** y la tabla de Shapes.

El primer paso es dibujar la secuencia de Shapes que harán el efecto de animación. Para este ejemplo hice seis Shapes que simulan una pelota rebotando:



Las dos líneas horizontales paralelas extras se dibujan para delimitar la animación, es decir que el ancho de las líneas hacen referencia al ancho total y están a una distancia una de la otra, de tal manera que estas dos líneas delimitan el alto de la animación. En este ejemplo las dos líneas ayudan a dar la referencia del suelo, para la línea inferior, y del techo para la línea superior.

El segundo paso es copiar el código de las Shapes y hacer una **tabla** en la celda de texto "**Variables**" con ellas. Recordemos que las Shapes la debemos poner entre comillas, ya sean dobles o sencillas, y el nombre de la **tabla** es a gusto de cada quien:

Variables:

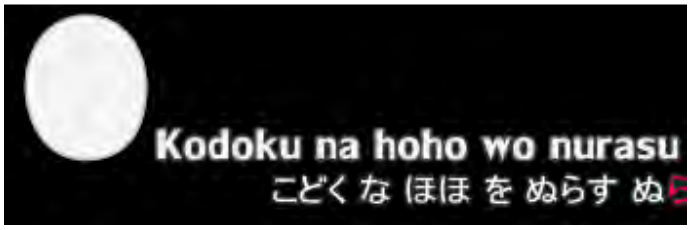
```
shapes = {"m 0 0 l 44 0 m 0 61 l 44 61 m 22 21 b 11 21 1
30 1 40 b 1 49 7 61 22 61 b 39 61 43 49 43 40 b 43 30 33
21 22 21 ", "m 0 0 l 44 0 m 0 61 l 44 61 m 22 28 b 11 28 0
37 0 47 b 0 56 7 61 22 61 b 39 61 44 56 44 47 b 44 37 33
28 22 28 ", "m 0 0 l 44 0 m 0 61 l 44 61 m 22 10 b 11 10 2
19 2 34 b 2 43 7 58 22 58 b 39 58 42 43 42 34 b 42 19 33
10 22 10 ", "m 0 0 l 44 0 m 0 61 l 44 61 m 22 0 b 11 0 0 7
0 17 b 0 25 7 36 22 36 b 39 36 44 25 44 17 b 44 7 33 0 22
0 ", "m 0 0 l 44 0 m 0 61 l 44 61 m 22 10 b 11 10 2 19 2
34 b 2 43 7 58 22 58 b 39 58 42 43 42 34 b 42 19 33 10
22 10 ", "m 0 0 l 44 0 m 0 61 l 44 61 m 22 28 b 11 28 0 37
0 47 b 0 56 7 61 22 61 b 39 61 44 56 44 47 b 44 37 33 28
22 28 " }
```

Llamamos en "**Return [fx]**" a la función y como tercer parámetro ponemos a la tabla declarada en "**Variables**" que contiene a todas las Shapes de la animación:

Return [fx]:

```
shape.animated2(fx.dur, 120, shapes)
```

Y la función generará la animación:



La ventaja de **shape.animation2** es poder hacer modificar fácilmente el tamaño de la misma ya que está hecha con Shapes, y éstas se modifican con los tags **\fscx** y **\fscy**.

Ya hemos avanzado mucho en la Librería **shape** y cada vez resta menos para terminar de ver todas sus funciones. En el **Tomo XVIII** continuaremos con el resto de las funciones hasta que las hayamos visto todas y consideremos que ya dominamos un poco más el mundo de los efectos hechos con Shapes. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

www.facebook.com/karaeffector

Kara Effector 3.2:

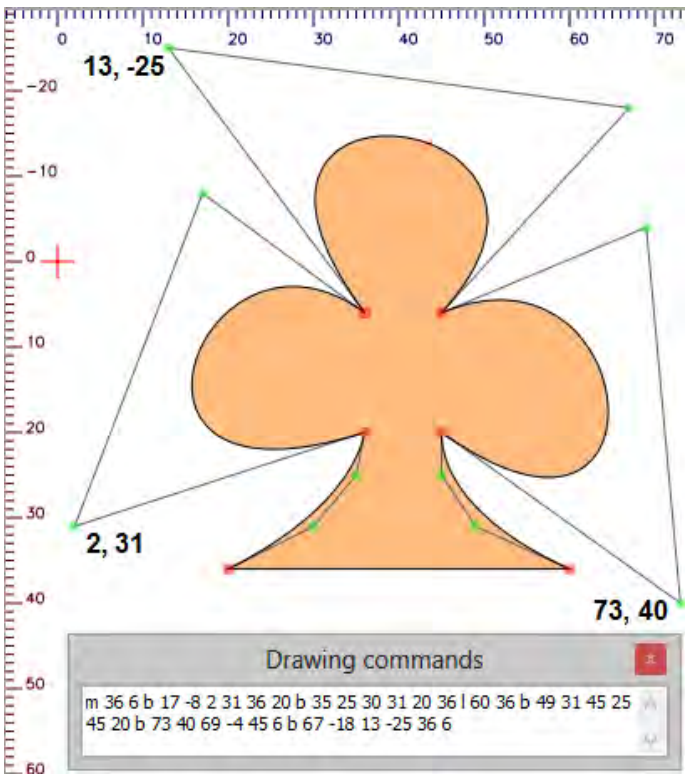
En el **Tomo XVIII** seguimos viendo el resto de las funciones de la Librería **shape** que hemos venido viendo desde hace ya varios Tomos. Habrán notado el gran tamaño de esta Librería y la importancia de la misma, ya que es una parte muy importante de todo lo que se debe saber para hacer efectos de alta calidad.

Librería Shape [KE]:

shape.config(Shape, Return, Ratio): esta función es similar a **shape.info**, pero con la ventaja que retorna mucha más información de la **shape** ingresada.

El parámetro **Ratio** es opcional y hace referencia a un valor por el cual se multiplicarán todos los puntos de la **shape** ingresada.

El parámetro **Return** es el que decide lo que retornará la función y tiene múltiples opciones:

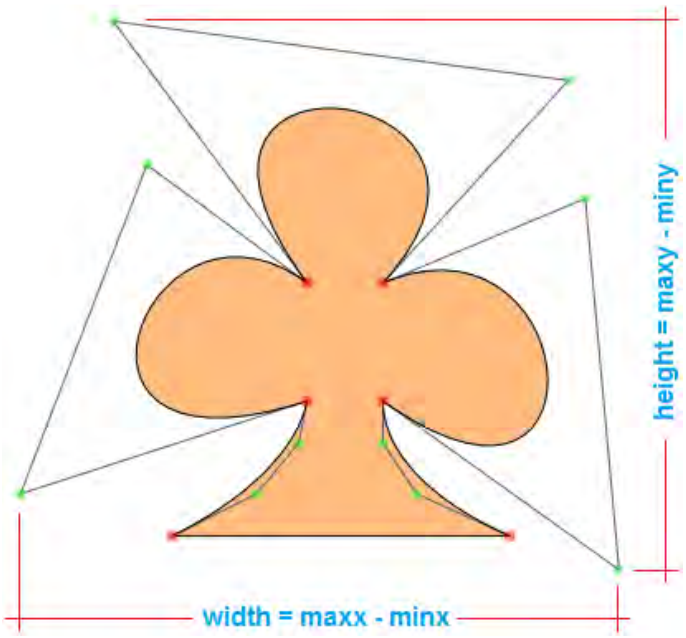


Usaremos la anterior **shape** para los siguientes ejemplos, y como siempre (aunque no es obligatorio), la declararé a modo de variable en la celda de texto "**Variables**" para que sea un poco más simple su uso:

Variables:

```
mi_shape = "m 36 6 b 17 -8 2 31 36 20 b 35 25 30 31
            20 36 l 60 36 b 49 31 45 25 45 20 b 73 40
            69 -4 45 6 b 67 -18 13 -25 36 6 "
```

- **shape.config(mi_shape, "minx"):** este modo retorna el mínimo valor de las coordenadas "x". como se puede ver en la imagen, este valor es 2.
- **shape.config(mi_shape, "maxx"):** este modo retorna el máximo valor de las coordenadas "x", que para esta **shape** es 73.
- **shape.config(mi_shape, "miny"):** este modo retorna el mínimo valor de las coordenadas "y", que para esta **shape** es -25.
- **shape.config(mi_shape, "maxy"):** este modo retorna el máximo valor de las coordenadas "y", que para esta **shape** es 40.
- **shape.config(mi_shape, "width"):** este modo retorna el ancho de la **shape** calculado como la diferencia entre **maxx** y **minx**: 73 – 2 = 71 px



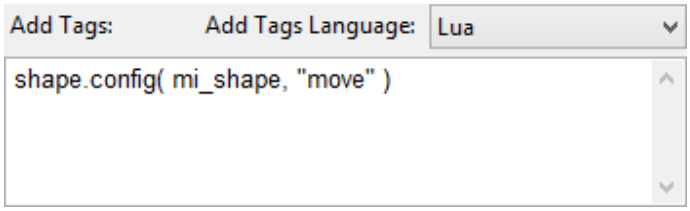
- **shape.config(mi_shape, "height"):** este modo retorna el alto de la **shape** calculado como la diferencia entre **maxy** y **miny**: 40 – (– 25) = 65 px

- **shape.config(mi_shape, "length"):** este modo retorna la medida de la longitud de la shape, es decir la medida de su perímetro:

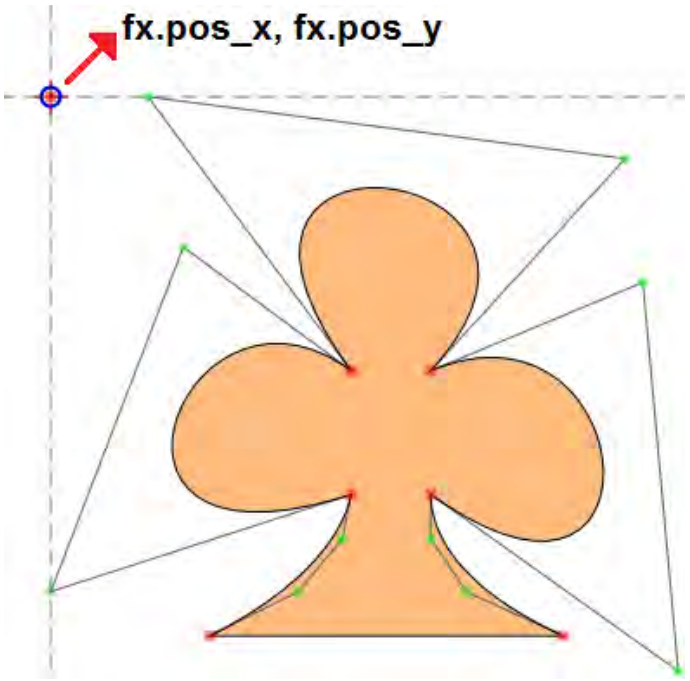
- **shape.config(mi_shape, "segments"):** este modo retorna una **tabla** que contiene las coordenadas de cada uno de los segmentos de la **shape**. Dichas coordenadas están a su vez dentro de una **tabla**:


```
Segmentos = {
   [1] = {36, 6,  17, -8,  2,   31, 36, 20},
   [2] = {36, 20, 35,  25, 30,  31, 20, 36},
   [3] = {20, 36, 60,  36, },
   [4] = {60, 36, 49,  31, 45,  25, 45, 20},
   [5] = {45, 20, 73,  40, 69, -4,  45, 6 },
   [6] = {45, 6,  67, -18, 13, -25, 36, 6 }
}
```

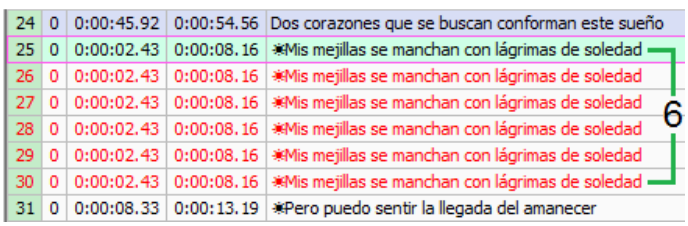
- **shape.config(mi_shape, “move”):** este modo usa toda la información concerniente a la **shape** para generar una secuencia de movimientos a modo de efecto, que hacen que el objeto karaoke se mueva siguiendo la trayectoria de la **shape** ingresada.
  Por ejemplo, en un **Template Type: Line** usamos la función y hacemos algo como esto:



Lo que internamente hace la función es desplazar a la shape a un origen relativo, que ya no es el punto (0, 0) sino el centro en el vídeo del objeto karaoke:

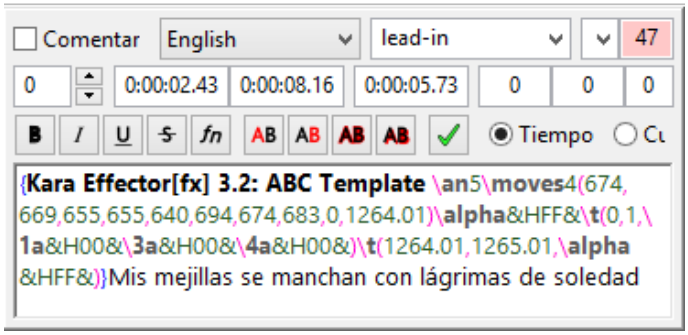


Luego la función creará tantos **loops** de una misma línea, como segmentos tenga la **shape** ingresada en la función, que para este ejemplo son 6, como ya lo habíamos visto en el modo “**segments**”:



24	0	0:00:45.92	0:00:54.56	Dos corazones que se buscan conforman este sueño
25	0	0:00:02.43	0:00:08.16	*Mis mejillas se manchan con lágrimas de soledad
26	0	0:00:02.43	0:00:08.16	*Mis mejillas se manchan con lágrimas de soledad
27	0	0:00:02.43	0:00:08.16	*Mis mejillas se manchan con lágrimas de soledad
28	0	0:00:02.43	0:00:08.16	*Mis mejillas se manchan con lágrimas de soledad
29	0	0:00:02.43	0:00:08.16	*Mis mejillas se manchan con lágrimas de soledad
30	0	0:00:02.43	0:00:08.16	*Mis mejillas se manchan con lágrimas de soledad
31	0	0:00:08.33	0:00:13.19	*Pero puedo sentir la llegada del amanecer

De la anterior imagen se pueden apreciar los seis **loops** por cada línea de fx generada, y que todas ellas tienen los mismos tiempos de inicio y final, pero que gracias a una serie de transformaciones, éstas generan el efecto de movimiento continuo armónico a través del perímetro de la **shape**:



El tag **\alpha&HFF&** genera la invisibilidad, y luego los tags **\1a**, **\3a** y **\4a** dan la transparencia por default del objeto karaoke. Esta característica imposibilita que usemos estos mismos tags para cualquier otra cosa dentro del mismo efecto, porque pueden afectar las transformaciones.

Por cada segmento de la **shape** que esté conformado por una **curva Bezier**, la función retornará un tag **\moves4** para poder moverse a través de ella. Si el segmento de la **shape** es una recta, entonces la función retorna un tag **\move** para moverse de un punto a otro.

Como ya lo había mencionado anteriormente, la función genera el **loop** de forma automática dependiendo de los segmentos que contenga la **shape**, así que para generar un **loop** independiente de ése, debemos hacerlo de la siguiente manera:

Sabemos que el 1 en este caso se pasa por alto, ya que la función generó un **loop** 6, y el 3 hace referencia ahora a la cantidad de repeticiones de esos 6 **loops**. O sea que el **loop** total será de 6 X 3 = 18, pero en pantalla, modificando un poco los tiempos de inicio y final, solo veremos los 3 que hemos asignado previamente en la celda de texto “**loop**”:

Es algo complicado intentar explicar con palabras, incluso con imágenes, lo que esta función hace, ya que el efecto se basa en movimientos y en la sincronía de los mismos. Lo que recomiendo es que pongan la función en práctica con varias Shapes para poder notar las diferencias entre los resultados. Ejemplos

- **shape.config(shape.rectangle, "move")**
- **shape.config(shape.triangle, "move")**
- **shape.config(shape.circle, "move")**

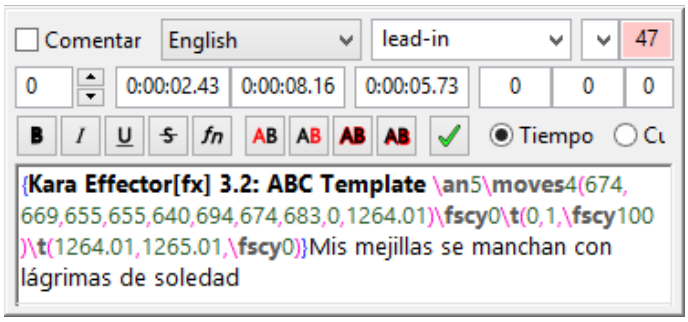
Este último ejemplo haría que el objeto karaoke se mueva siguiendo como trayectoria al perímetro de **shape.circle**, que tiene 100 px de diámetro, que es la medida de esta **shape predeterminada** del **Kara Effector**. Si quisiéramos que un objeto karaoke se moviera siguiendo la trayectoria de un círculo más grande o más pequeño, tenemos las dos siguientes opciones. Ejemplos:

1. **shape.config(shape.circle, "move" , 1.5):** este modo hace que el **Ratio** (1.5) aumente el tamaño de la **shape** ingresada en un 150%, entonces el objeto karaoke se moverá siguiendo la trayectoria de un círculo de 150 px de diámetro.
2. **shape.config(shape.size(shape.circle, 72), "move"):** la función **shape.size** redefine el tamaño del círculo a 72 px, que será el diámetro del círculo por el cual se moverá el objeto karaoke.

Entonces, podemos usar el tercer parámetro de la función **shape.config** (**Ratio**) o usar la función **shape.size**, para modificar las dimensiones de la **shape** ingresada y así redefinir las trayectorias de los desplazamientos.

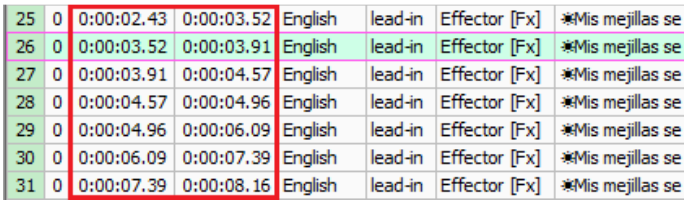
Como les mencioné antes, el modo "**move**" usa los tags de transparencia para generar el efecto de fluidez en el movimiento del objeto karaoke, lo que imposibilitaba el volver usar dichos tags nuevamente en el mismo efecto. Si inevitablemente tuviéramos que usar estos tags, tenemos un modo más que lo hace posible:

- **shape.config(mi_shape, "move2"):** este modo es similar al modo "**move**", ya que hacen lo mismo, pero no usa los tags de transparencia en sus transformaciones. El tag que el modo "**move2**" usa para generar el efecto de movimiento fluido es el tag **\fscy**:



Entonces en el modo "**move2**" el tag que no se puede usar en el resto del efecto es **\fscy**, ya que es el que hace que cada uno de los loops desaparezca en el momento justo y dé la sensación de fluidez en el movimiento.

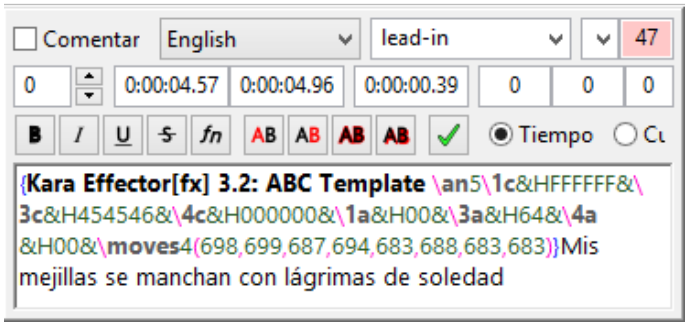
- **shape.config(mi_shape, "move3"):** este modo es similar a los modos "**move**" y "**move2**", genera el mismo efecto de movimiento, pero sin usar los tags de transparencias ni tampoco el tag **\fscy** en sus transformaciones. El modo "**move3**" genera líneas independientes respecto al tiempo, o sea que no necesita de un tag para hacer desaparecer al objeto karaoke que se mueve3 en determinado segmento.



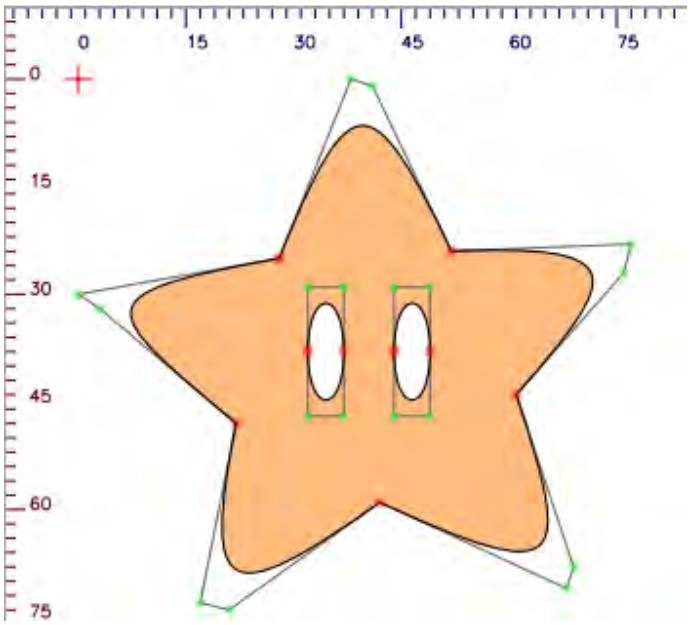
25	0	0:00:02.43	0:00:03.52	English	lead-in	Effector [Fx]	*Mis mejillas se
26	0	0:00:03.52	0:00:03.91	English	lead-in	Effector [Fx]	*Mis mejillas se
27	0	0:00:03.91	0:00:04.57	English	lead-in	Effector [Fx]	*Mis mejillas se
28	0	0:00:04.57	0:00:04.96	English	lead-in	Effector [Fx]	*Mis mejillas se
29	0	0:00:04.96	0:00:06.09	English	lead-in	Effector [Fx]	*Mis mejillas se
30	0	0:00:06.09	0:00:07.39	English	lead-in	Effector [Fx]	*Mis mejillas se
31	0	0:00:07.39	0:00:08.16	English	lead-in	Effector [Fx]	*Mis mejillas se

El modo "**move3**" estará disponible a partir de la **Versión 3.2.7** del **Kara Effector**, para versiones anteriores solo es posible usar los modos "**move**" y "**move2**".

Al ver en detalle una de las líneas de fx generadas en este modo, notamos que el único tag que retorna es el de movimiento, **\move** para las restas de la **shape** y **\moves4** para las curvas **Bezier**, como en este ejemplo:



shape.incenter(Shape): esta función desplaza a la **shape** ingresada en el centro de las coordenadas del **AssDraw3**, con referencia al punto P = (0, 0). Ejemplo:

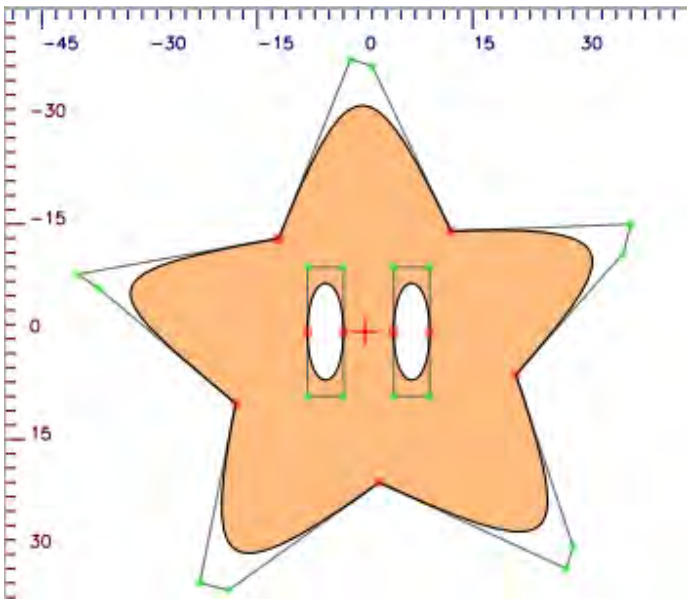


La anterior **shape** está ubicada en el **Cuadrante IV** del **AssDraw3.** Usamos la función con esta **shape**, por ejemplo en la celda de texto **Return [fx]:**

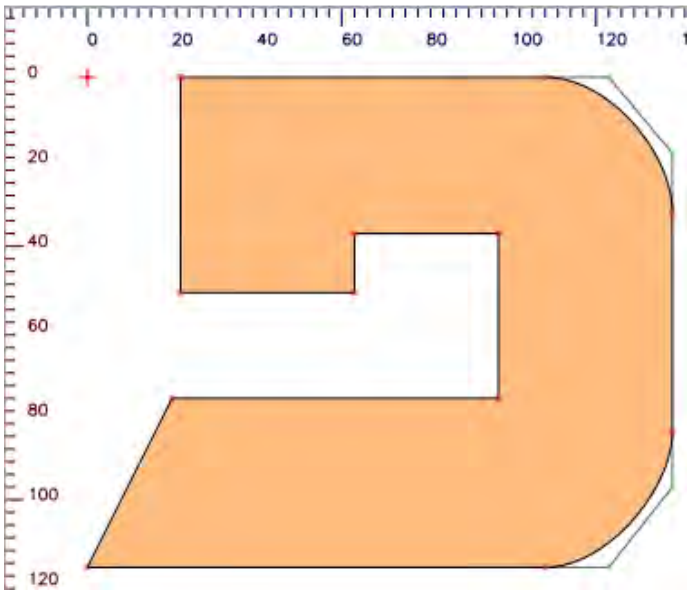
Return [fx]:

```
shape.incenter( mi_shape )
```

Y al copiarla la **shape** generada y pegarla en el **AssDraw3**:



shape.centerpos(Shape, Dx, Dy): desplaza a la **shape** ingresada con referencia a su centro, al punto con coordenadas P = (**Dx**, **Dy**). Ejemplo:



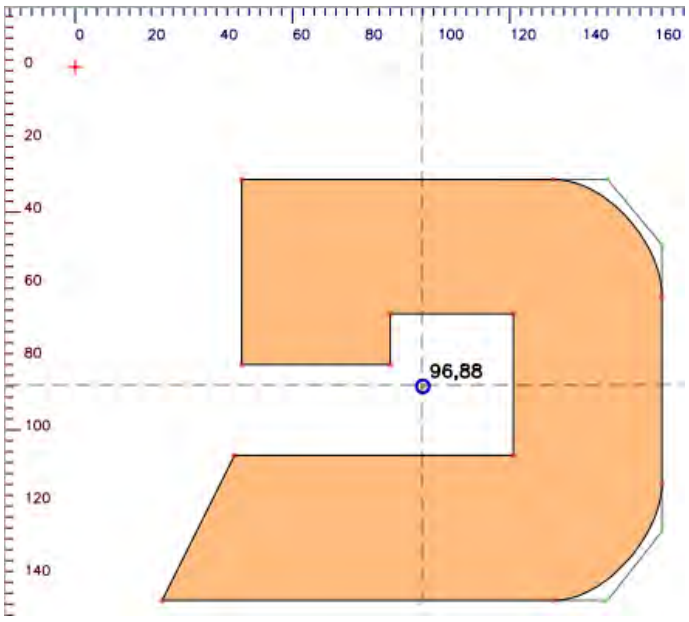
Ingresamos los dos valores que necesitemos desplazar la **shape** respecto a su centro, por ejemplo:

Return [fx]:

```
shape.centerpos( mi_shape, 96, 88 )
```

Dx Dy

Ahora la nueva ubicación del centro de la **shape** es el que se haya especificado en la función (96, 88):



shape.trajectory(loop, D_nim, D_max): crea una **shape** con una definida cantidad de segmentos (**loop**), que distan entre sí dependiendo de los parámetros **D_min** y **D_max** (Distancia mínima y Distancia máxima).

Cada uno de los segmentos es creado de forma aleatoria y dibujados con una **Curva Bezier** de tal manera que, entre todos los segmentos formen una sola trayectoria fluida con cada una de las curvas.

Los parámetros **D_min** y **D_max** son opcionales. En el caso de no ponerlos en la función, éstos tienen sus respectivos valores por default:

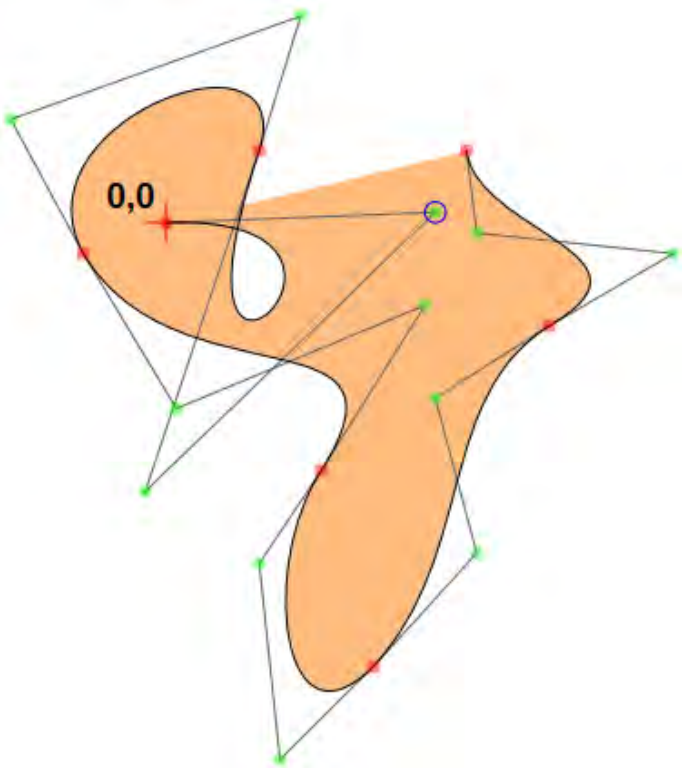
- **D_min** = 10*ratio
- **D_max** = 20*ratio

Ejemplo:

Return [fx]:

```
shape.trajectory( 5 )
```

Se genera una curva fluida con 5 segmentos de **Curvas Bezier**, y cada una de ellas separada de la otra a una distancia aleatoria entre 10 y 20 px:



Las curvas fluidas que genera esta función se pueden usar como una trayectoria de movimiento dentro de la función **shape.config** en los modos "**move**", "**move2**" y "**move3**". Ejemplos:

Add Tags: Add Tags Language: Lua

```
shape.config( shape.trajetory( 6, 25, 50 ), "move" )
```

Add Tags: Add Tags Language: Lua

```
shape.config( shape.trajetory( R(4,8) ), "move2" )
```

Add Tags: Add Tags Language: Lua

```
shape.config( shape.trajetory( 5 ), "move3" )
```

Y para cada uno de los ejemplos anteriores, la función **shape.config** hará que el objeto karaoke de la línea de fx se mueva siguiendo como trayectoria a la curva generada por la función **shape.trajectory**.

Las curvas fluidas que genera la función **shape.trajectory** tienen muchas más aplicaciones y opciones de uso. En los próximos tomos, de a poco iremos viendo cómo sacarle el máximo provecho a esta y otras funciones.

Es todo por ahora. En el **Tomo XIX** continuaremos viendo más de las funciones de la librería **shape**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

www.facebook.com/karaeffector

Kara Effector 3.2:

El **Tomo XIX** es otro más dedicado a la librería **shape**, que como ya habrán notado, es la más extensa hasta ahora vista en el **Kara Effector**. El tamaño de esta librería nos da una idea de la importancia de las Shapes en un efecto karaoke, y es por ello que debemos tomarnos un tiempo en ver y conocer a cada una de las funciones y recursos disponibles para poder dominarlas.

Librería Shape [KE]:

shape.Ltrajectory(length_t, length_c, height_c): es una función similar a **shape.trajectory** con la diferencia de que crea la trayectoria en una sola dirección y con las siguientes especificaciones:

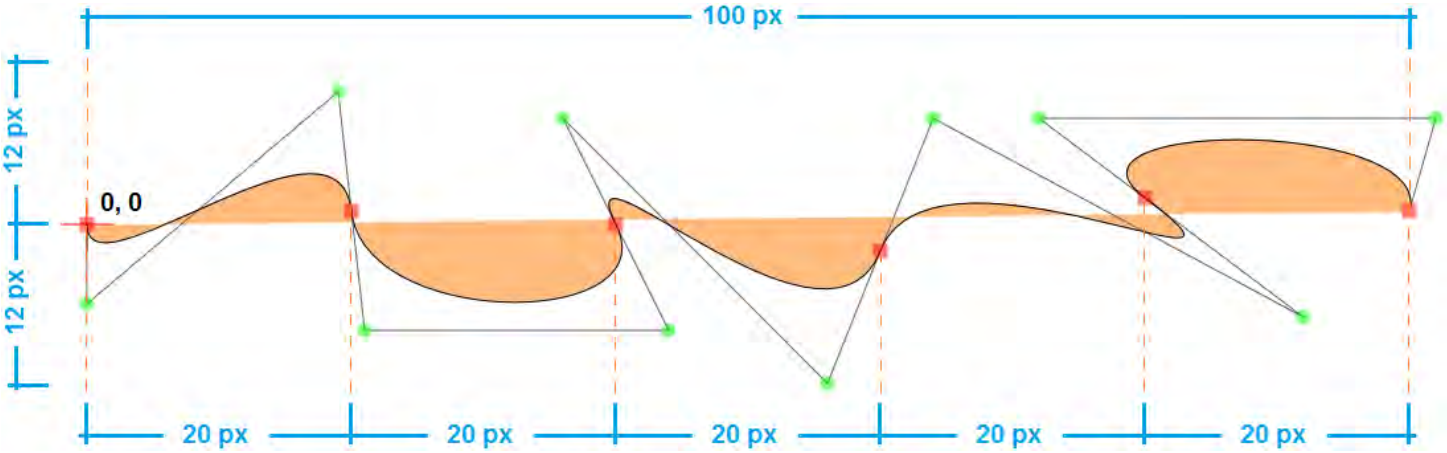
- **length_t:** es la longitud lineal total de la trayectoria medida en pixeles. Su valor por default equivale a la diferencia entre **xres** y **fx.move_x1.**
- **length_c:** es la longitud lineal de cada uno de los segmentos de las **Curvas Bezier** que conforman a la trayectoria. Su valor por default es **xres**/4.
- **height_c:** equivale a la mitad de la altura promedio máxima que tendrá cada una de las curvas de los segmentos. Su valor por default es 40***ratio**.

La función crea la trayectoria a partir del punto P = (0,0) y a 0° de dirección, o sea, hacia la derecha de dicho punto.

Ejemplo:

Return [fx]:

```
shape.Ltrajectory( 100, 20, 12 )
```



En la anterior imagen podemos ver una de las trayectorias creadas aleatoriamente, es fluida y sigue las condiciones de los parámetros ingresados en la función:

- Longitud lineal total: **100 px**
- Longitud lineal de los segmentos: **20 px**
- Máximo ascenso y descenso: **12 px**

Las ventajas que tiene cualquier trayectoria creada por una **shape**, es que las podemos modificar usando las funciones de dicha librería. Ejemplos:

- Modificar el ángulo:
  shape.rotate(shape.Ltrajectory(100, 20, 12)**, 60)**
- Modificar el orden del trazado:
  shape.reverse(shape.Ltrajectory(100, 20, 12) **)**
- Modificar el "**Ratio**" de alguna de las dimensiones:
  shape.ratio(shape.Ltrajectory(100, 20, 12)**, 1, 0.5)**

En fin, las opciones son muchas ya que también podemos combinar dos o más funciones de la librería **shape** para obtener nuevos resultados.

Ejemplo para poner en práctica:

Variables:

```
Trj = shape.reflect( shape.Ltrajectory(100,20,12), "y" )
```

Add Tags: Add Tags Language: Lua

```
shape.config( Trj, "move" )
```

shape.Ctrajectory(Loop, r_min, r_max): crea una trayectoria con centro en el punto P = (0,0) y sin exceder como máximo al radio **r_max**, ni como mínimo al radio **r_min**, en donde ambos radios son ingresados en pixeles.

El parámetro **Loop** indica la cantidad de segmentos que tendrá la trayectoria final retornada y su valor por default es **line.duration**/720.

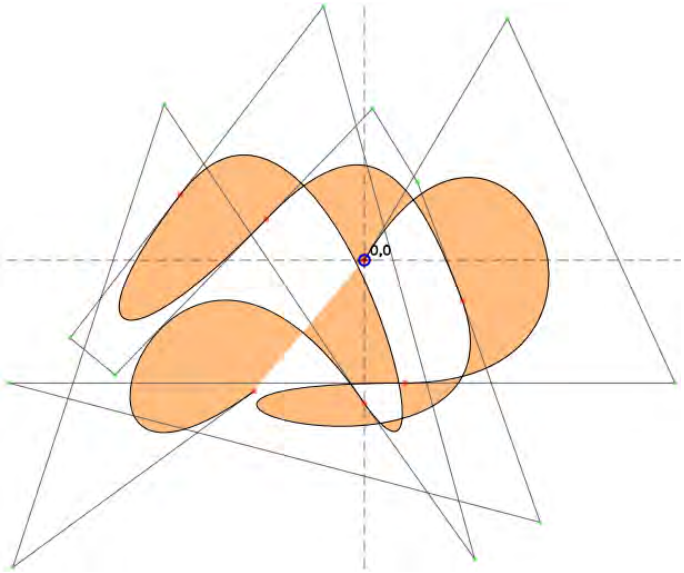
El valor por default de **r_min** es **xres**/40 y el de **r_max** es **xres**/25, o sea que ambos pueden ser opcionales.

Ejemplo:

Return [fx]:

```
shape.Ctrajectory( 5, 20, 50 )
```

Veamos una de las trayectorias fluidas generadas:

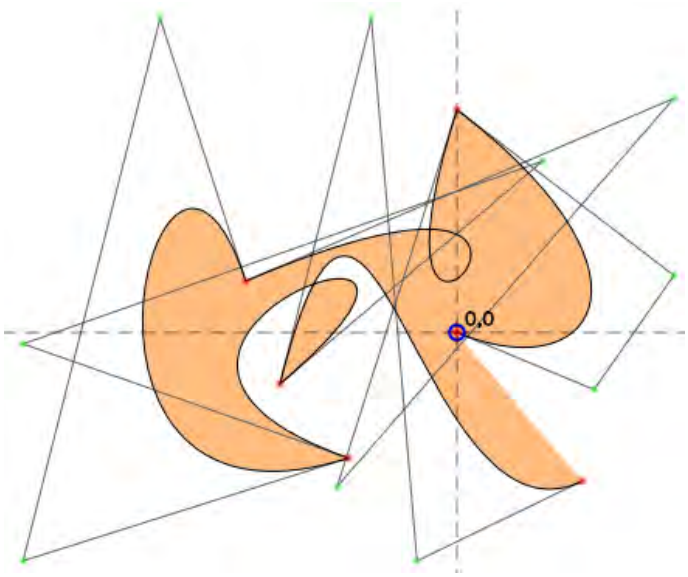


shape.Rtrajectory(Loop, r_min, r_max): es una función similar a **shape.Ctrajectory**, pero la trayectoria que genera ya no es fluida sino totalmente aleatoria (random). Esta función crea una trayectoria con centro en el punto P = (0,0) y sin exceder como máximo al radio **r_max**, ni como mínimo al radio **r_min**, con ambos radios son ingresados en pixeles.

El parámetro **Loop** indica la cantidad de segmentos que tendrá la trayectoria final retornada y su valor por default es **line.duration**/720. El valor por default de **r_min** es **xres**/40 y el de **r_max** es **xres**/25. Ejemplo:

```
Return [fx]:
shape.Rtrajectory( 5, 30, 40 )
```

Y notamos que la trayectoria esta vez ya no es fluida:



---

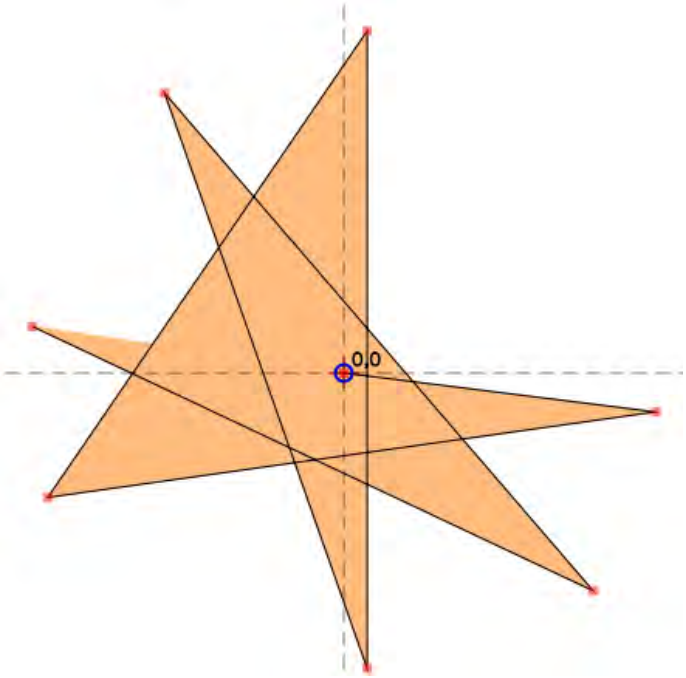
shape.Strajectory(Loop, Radius): esta función es similar a **shape.Rtrajectory**, pero creo la trayectoria con segmentos lineales de forma aleatoria, en vez de usar las **Curvas Bezier** como en las cuatro anteriores funciones.

El parámetro **Loop** indica la cantidad total de segmentos lineales que conformarán la trayectoria y su valor por default es **line.duration**/820. El parámetro **Radius** indica la distancia a partir del punto P = (0,0), de los extremos de los segmentos. Su valor por default es **0.75*line.height**.

Ejemplo:

```
Return [fx]:
shape.Strajectory( 7, 45 )
```

En la siguiente gráfica vemos cómo la trayectoria está conformada por siete segmentos rectos:



---

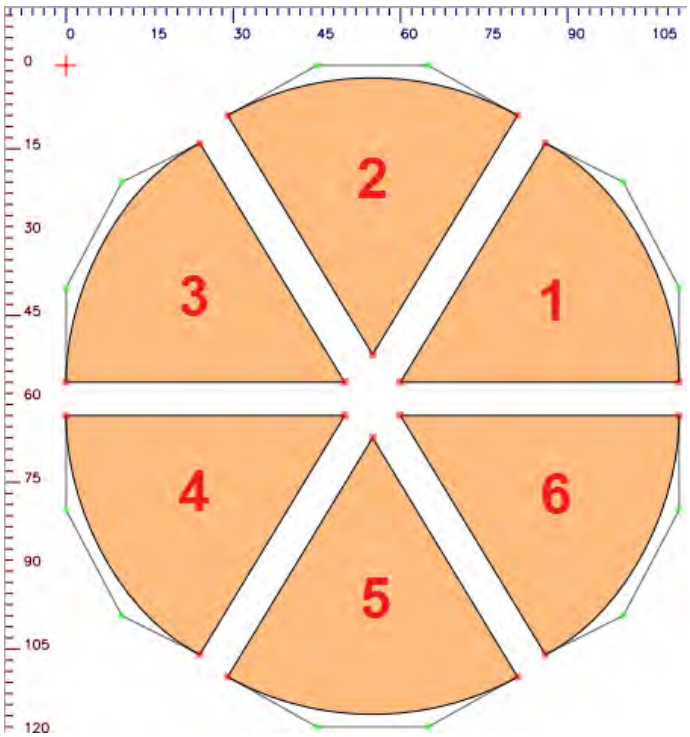
shape.movevc(Shape, Rtn, width, height, x, y, Dx, Dy, t_i, t_f): es similar a la función **tag.movevc**, pero con la diferencia que a esta función se le ingresa una **shape** conformada por dos o más Shapes para ser usadas dentro de un tag **\clip**, que posteriormente serán manipuladas de forma individual por el tag **\movevc**.

- **Shape**: es la shape que está conformada por dos o más Shapes individuales.
- **Rtn**: este parámetro decide qué va a retornar la función y tiene tres opciones:
  - "**shape**": retorna individualmente a cada una de la Shapes que conforma a la shape ingresada en la función.
  - "**loops**": retorna en número equivalente a la cantidad de Shapes que conforman a la shape ingresada.

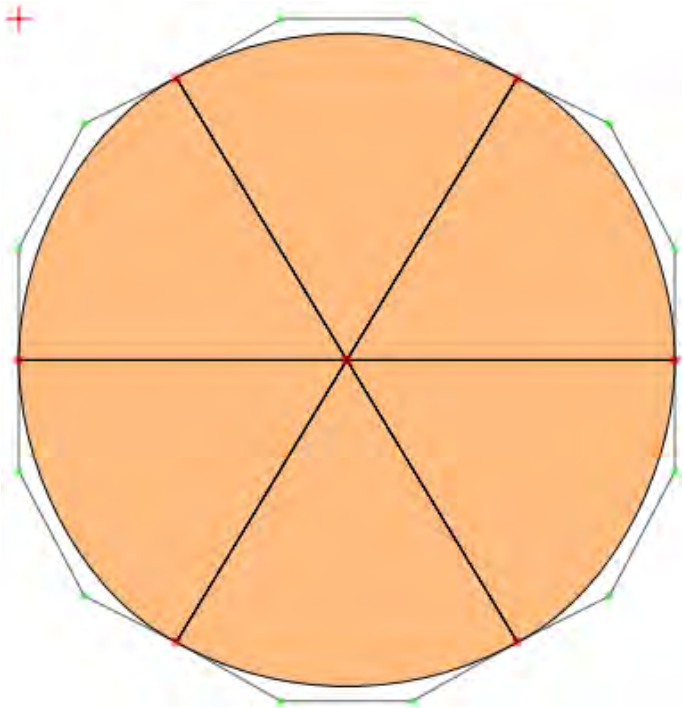
- "**tag**": retorna una serie de tags equivalente al efecto generado por la función.

- **width:** es el ancho total que abarcarán todos los clip's que genere la función y su valor por default es **val_width** (ancho del objeto karaoke: syl, word, char, line y demás. Depende del **Template Type**).
- **height:** es el alto total que abarcarán todos los clip's que genere la función y su valor por default es **val_height** (alto del objeto karaoke: syl, word, char, line y demás. Depende del **Template Type**).
- **x, y:** son las coordenadas que ubicarán el centro de la **shape** ingresada respecto al vídeo. Sus valores por default son:
  - **x = fx.move_x1**
  - **y = fx.move_y1**.
- **Dx, Dy:** son las distancias medida en pixeles en las que se moverán cada uno de los clip's generados, respecto a ambos ejes. Sus valores por default son:
  - **Dx = fx.move_x2 – fx.move_x1**
  - **Dy = fx.move_y2 – fx.move_y1**
- **t_i, t_f:** son los tiempos de inicio y final de los movimientos de cada uno de los clip's. sus valores por default son:
  - **t_i = fx.movet_i**
  - **t_f = fx.movet_f**

Para el ejemplo, usaré el siguiente grupo de Shapes:



Son seis Shapes individuales en total, pero las he juntado de manera que aparenten ser una sola:



A continuación, usaremos el código de la anterior **shape** del **ASSDraw3**, para declarar una variable en la celda de texto "**Variables**":

Variables:

```
Shapes = "m 50 52 l 100 52 b 100 35 90 16 76 9 l 50 52 m 50 52 l 76 9 b 60 0 40 0 24 9 l 50 52 m 50 52 l 24 9 b 10 16 0 35 0 52 l 50 52 m 50 52 l 0 52 b 0 69 10 88 24 95 l 50 52 m 50 52 l 24 95 b 40 104 60 104 76 95 l 50 52 m 50 52 l 76 95 b 90 88 100 69 100 52 l 50 52 "
```

He resaltado las letras "**m**" del código de la **shape** con el fin de poder identificar fácilmente a las seis Shapes individuales que conforman a toda la **shape**.

Para el ejemplo usaré un **Template Type: Syl** y la plantilla de efectos: **[001] ABC Template Hilight Syl**. Y en **Add Tags** llamaremos a la función usando casi todos sus parámetros por default, ya que todos ellos hacen referencia a valores ya ingresados, como las posiciones y los tiempos:

Add Tags: Add Tags Language: Lua

```
shape.movevc( Shapes, "tag" )
```

Entonces la función generará automáticamente un loop equivalente a la cantidad total de Shapes individuales que conforman a la **shape** ingresada (o sea 6) y generará un clip por cada una de esas Shapes con las siguientes posiciones:

Es decir, como es un **Template Type: Syl**, generará seis líneas de fx por cada sílaba de cada línea a la que se le aplique el efecto:

24	0	0:00:45.92	0:00:54.56	English			Dos corazone
25	1	0:00:02.43	0:00:02.60	Romaji	hi-light	Effector [Fx]	✱Ko
26	1	0:00:02.43	0:00:02.60	Romaji	hi-light	Effector [Fx]	✱Ko
27	1	0:00:02.43	0:00:02.60	Romaji	hi-light	Effector [Fx]	✱Ko
28	1	0:00:02.43	0:00:02.60	Romaji	hi-light	Effector [Fx]	✱Ko
29	1	0:00:02.43	0:00:02.60	Romaji	hi-light	Effector [Fx]	✱Ko
30	1	0:00:02.43	0:00:02.60	Romaji	hi-light	Effector [Fx]	✱Ko
31	1	0:00:02.60	0:00:02.76	Romaji	hi-light	Effector [Fx]	✱do

6

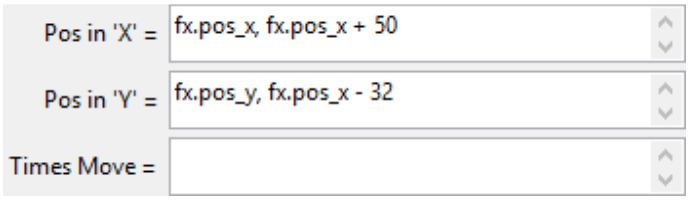
Pero como no le hemos ordenado ningún movimiento ni tampoco le hemos dado ubicaciones distintas a las que ya tiene por default, en el vídeo veremos a cada sílaba de forma normal:

Pero si manualmente eliminamos a una de esas seis líneas, ya se verá la diferencia, ya que la sílaba que se ve en pantalla está formada por seis partes de la misma:

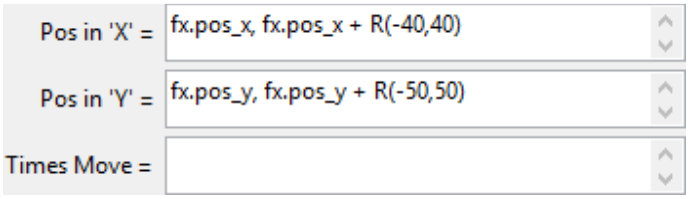
Al ampliar un segmento de los que conforman la sílaba, veremos algo como esto:

O sea que cada clip usa a una única **shape** para hacer visible una sección de la sílaba y el resto quedará invisible.

Ahora, para darle movimiento a los clip's, simplemente le damos movimiento al objeto karaoke, en este caso, a las sílaba:



Pero no tendría mucho sentido hacerlo de esta forma, ya que todos los clip's se moverán exactamente a la misma dirección y en el mismo tiempo. Entonces si queremos que los clip's se muevan a lugares distintos, debemos usar valores aleatorios (random) para dar la ilusión de que la sílaba se fragmenta en pedazos:



Así cada trozo se moverá a lugares distintos respecto a los otros, aunque aún lo seguirán haciendo al mismo tiempo, ya que los tiempos del movimiento se dejaron por default:

Siempre que queramos que los clip's se muevan al mismo lugar a dónde lo hará el objeto karaoke y al mismo tiempo que él, entonces lo que debemos hacer es usar la función solo con los dos primeros parámetros, como lo hicimos en el ejemplo anterior.

Es momento para recordarles la gran cantidad de recursos de la **Memoria RAM** que consumen los tags **\clip**, **\iclip** y **\movec.** Lo recomendable es no exceder un **loop** entre 20 o 25 en el efecto, es decir que la **shape** ingresada en la función esté conformada por, a lo máximo, 25 Shapes individuales.

Exceder las 25 Shapes individuales en la **shape** ingresada en la función hará que la computadora empiece a ponerse lenta a medida que se reproduce el efecto, también hará mucho más lento el proceso en encodeo.

---

shape.movevci(Shape, Rtn, width, height, x, y, Dx, Dy, t_i, t_f): es similar a la función **shape.movevci**, pero con la diferencia que retorna iclip's en lugar de clip's como la anterior función.

---

shape.multi1(Size, Px): crea una **shape** formada de múltiples Shapes cuadradas concéntricas para ser usada en las funciones **shape.movevc** y **shape.movevci**.

El parámetro **Size** indica las dimensiones máximas del cuadrado de mayor tamaño y su valor por default es equivalente a la mayor dimensión entre **val_width** y **val_height** del objeto karaoke, es decir:

- **Size = math.max(val_width, val_height)**

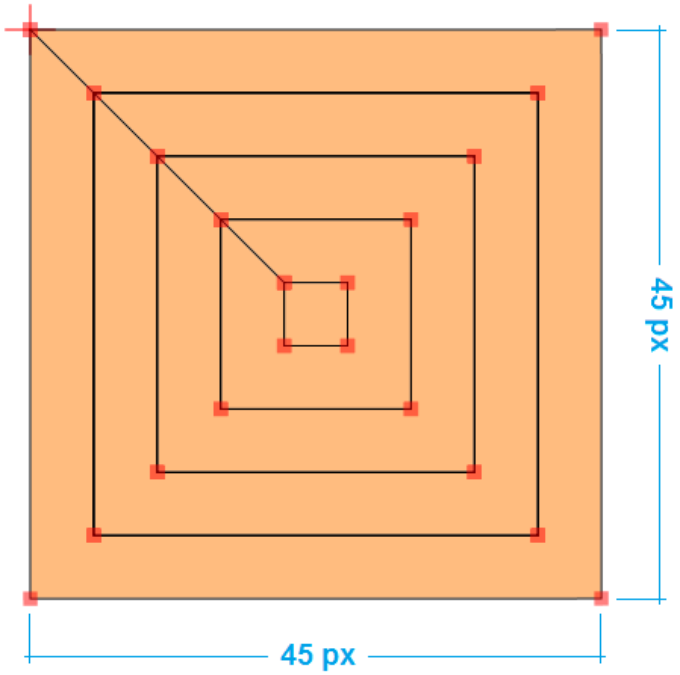
El parámetro **Px** equivale al ancho en pixeles de cada una de las Shapes cuadradas concéntricas que conforman a la **shape** que será retornada. Su valor por default es 4***ratio**.

Ejemplo:

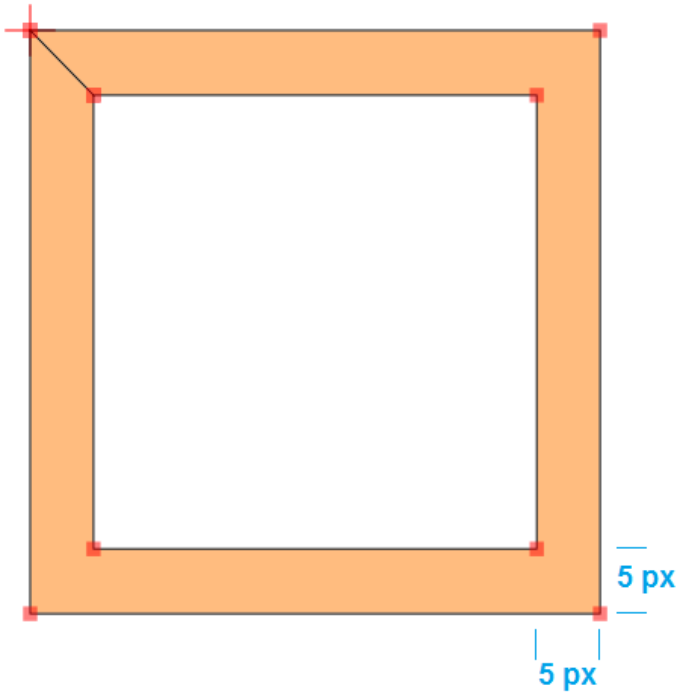
Return [fx]:

```
shape.multi1( 45, 5 )
```

Se generará la siguiente **shape**:



Vemos que las dimensiones de la **shape** son 45 X 45 px, y el ancho de cada una de las Shapes que la conforman es de 5 px:



Otra forma de acceder al valor por default del parámetro **Size** es escribiendo la palabra "**default**" en él, Ejemplos:

- **shape.multi1("default", 8)**
- **shape.multi1("default", 2)**

Entonces, para usar la función **shape.multi1** dentro de la función **shape.movevc** es recomendable usar el valor por default del parámetro **Size**, para que el objeto karaoke sea completamente visible en los clip's generados. Ejemplo:

```
Add Tags:   Add Tags Language: Lua
shape.movevc( shape.multi1( "default", 4 ), "tag" )
```

Lo que generará los siguientes clip's:

La cantidad de clip's generados por la función dependerá de las dimensiones del objeto karaoke; entre más grande sea éste, mayor será la cantidad de clip's generados para poder abarcar toda la dimensión del objeto karaoke.

En algunos casos, el valor por default de **Size** no abarca completamente a las dimensiones del objeto karaoke, ya sea por un borde muy grueso, una sombra muy grande, un **blur** muy marcado u otros factores más; para estos casos, debemos sumar un valor extra que compense el tamaño total de la **shape** generada. Ejemplo:

```
Variables:
new_Size = math.max( val_width, val_height ) + 12
```

O sea que sumamos 12 px para compensar el tamaño de la **shape** generada. Luego usamos esta variable dentro de la función:

```
Add Tags:   Add Tags Language: Lua
shape.movevc( shape.multi1( new_Size, 8 ), "tag" )
```

Usando un poco de imaginación, usamos las funciones de la librería **shape** para lograr nuevos resultados. Ejemplo:

```
Variables:
Shapes = shape.rotate( shape.multi1( ), 45 )
Add Tags:   Add Tags Language: Lua
shape.movevc( Shapes, "tag" )
```

Es todo por ahora. En el **Tomo XX** continuaremos viendo más de las funciones de la librería **shape**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

www.facebook.com/karaeffector

Kara Effector 3.2: Effector Book Vol. I [Tomo XX]

Kara Effector 3.2:

El **Tomo XX** es otro más dedicado a la librería **shape**, que como ya habrán notado, es la más extensa hasta ahora vista en el **Kara Effector**. El tamaño de esta librería nos da una idea de la importancia de las Shapes en un efecto karaoke, y es por ello que debemos tomarnos un tiempo en ver y conocer a cada una de las funciones y recursos disponibles para poder dominarlas.

Librería Shape [KE]:

shape.multi2(width, height, Dxy): es una función similar a **shape.multi1**, ya que también genera una **shape** conformada por múltiples Shapes para ser usada en las funciones **shape.movevc y shape.movevci**.

Esta función genera Shapes diagonales con un ancho de **Dxy**, dentro del rectángulo de medidas **width** X **height**.

- **width:** ancho total de la shape generada.
- **height:** alto total de la shape generada.
- **Dxy:** ancho de las Shapes diagonales

El valor por default del parámetro **width** es **val_width**, el de **height** es **val_height**, y el de **Dxy** es 6***ratio**:

- **width = val_width**
- **height = val_height**
- **Dxy = 6*ratio**

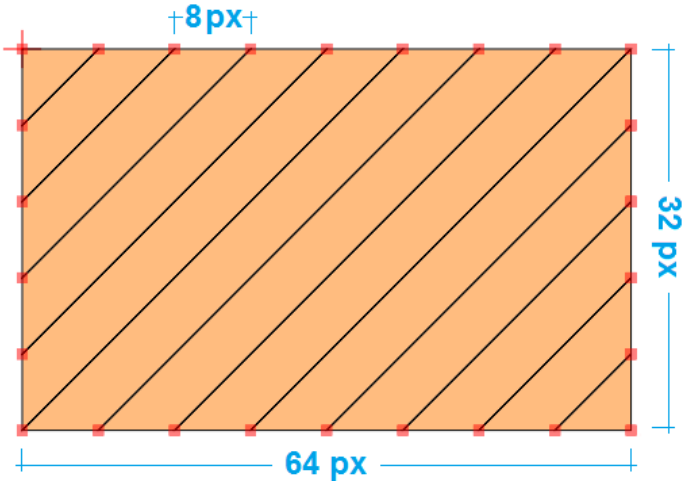
Ejemplo:

Return [fx]:

```
shape.multi2( 64, 32, 8 )
```

Ancho Alto Grosor

Lo que generará siguiente grupo de Shapes:



Para el próximo ejemplo usaré un **Template Type: Word** y los siguientes parámetros en la función:

Variables:

```
Shapes = shape.multi2( word.width, word.height, 5 )
```

Y luego de declarar la anterior variable, la usamos en la función **shape.movevc**:

Add Tags: Add Tags Language: Lua

```
shape.movevc( Shapes, "tag" )
```

Lo que generará la siguiente serie de clip's:

Otra variante que podemos usar es:

Add Tags: Add Tags Language: Lua

```
shape.movevc( shape.reflect( Shapes, "y" ), "tag" )
```

Lo que invertiría el sentido de las diagonales:

El **loop** total de los clip's generados solo dependerá de las dimensiones del rectángulo, así como del ancho que le demos en la función a las diagonales:

Con el uso de más recursos de la librería **shape** se pueden lograr resultados más complejos como este:

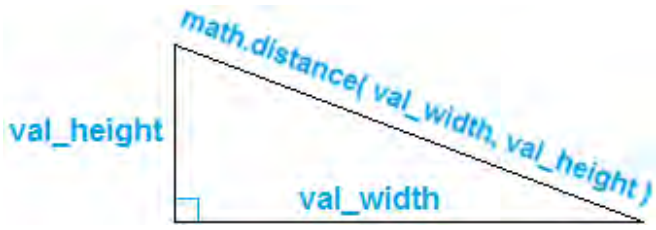
---

shape.multi3(Size, Dxy, Shape): retorna a una **shape** compuesta por Shapes concéntricas respecto a la **shape** ingresada (**Shape**) con un ancho de **Dxy**, y de un tamaño total **Size**.

- **Size:** tamaño total de la **shape** generada.
- **Dxy:** espesor de las Shapes concéntricas.
- **Shape: shape** ingresada.

El valor por default del parámetro **Size** equivale a la medida de la diagonal de un rectángulo de dimensiones **val_width**, **val_height**. El valor por default de **Dxy** es 5***ratio** y el de **Shape** es **shape.circle**:

- **Size = math.distance(val_width, val_height)**



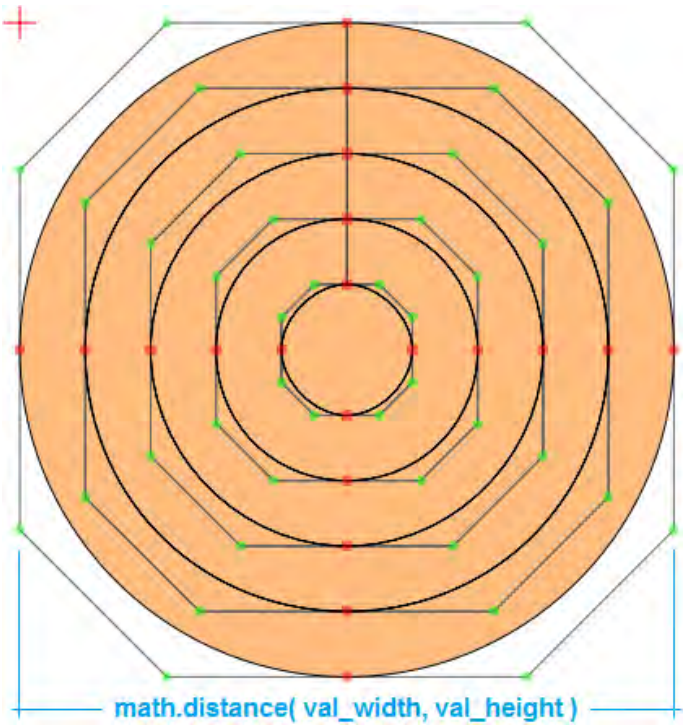
También se puede acceder a este valor si ponemos la palabra "**default**" en este parámetro.

- **Dxy = 5*ratio**
- **Shape = shape.circle**

Ejemplo 1:

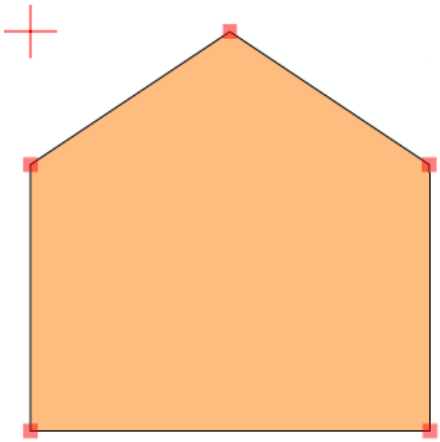
```
Return [fx]:
shape.multi3( "default", 8 )
```

Lo que generará círculos concéntricos de 8 px de espesor cada uno:



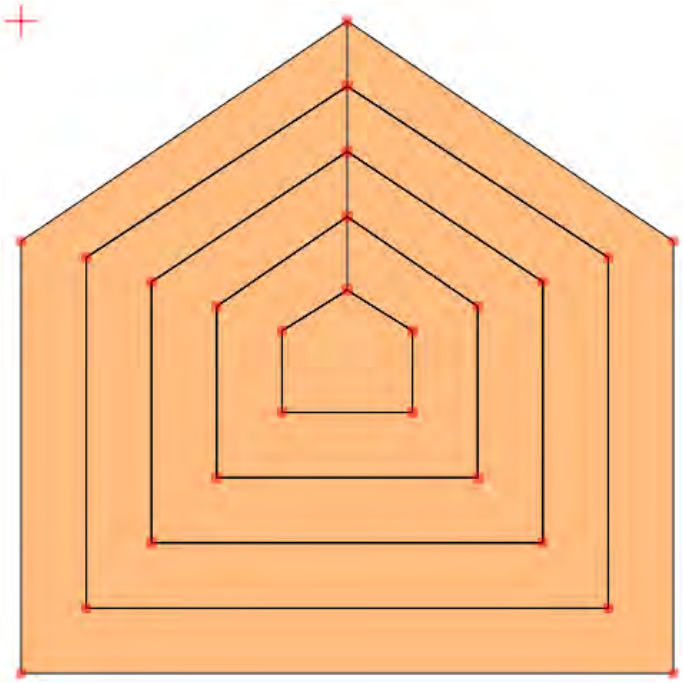
Ejemplo 2:

Para este ejemplo, usaremos la siguiente **shape**:



Con un espesor de 6 px:

```
Return [fx]:
shape.multi3( "default", 6, "m 15 0 l 0 10 l 0 30 l 30 30
                l 30 10 l 15 0 " )
```



O sea que las opciones son infinitas, ya que pueden duplicar de forma concéntrica a cualquier **shape** que se imagines. Como ya sabemos, el **loop** total depende de los valores ingresados en la función al igual que el tamaño del objeto karaoke. Una vez decididos los parámetros en la función, ya podemos usar la **shape** generada dentro de las funciones **shape.movevc** y/o **shape.movevci**.

---

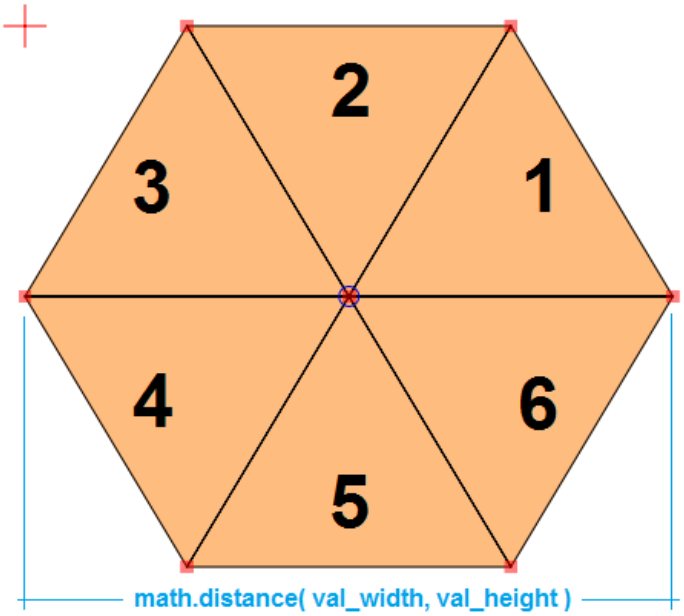
shape.multi4(Size, loop1, loop2): esta función retorna un **Arreglo Radial** de tamaño total **Size** y con una cantidad de repeticiones, en principio determinada por el parámetro **loop1**.
Esta función solo está disponible para la versión **3.2.7** o superior del **Kara Effector**.

Sus valores por default son:

- **Size: math.distance(val_width, val_height)**
- **loop1:** 6
- **loop2:** 1

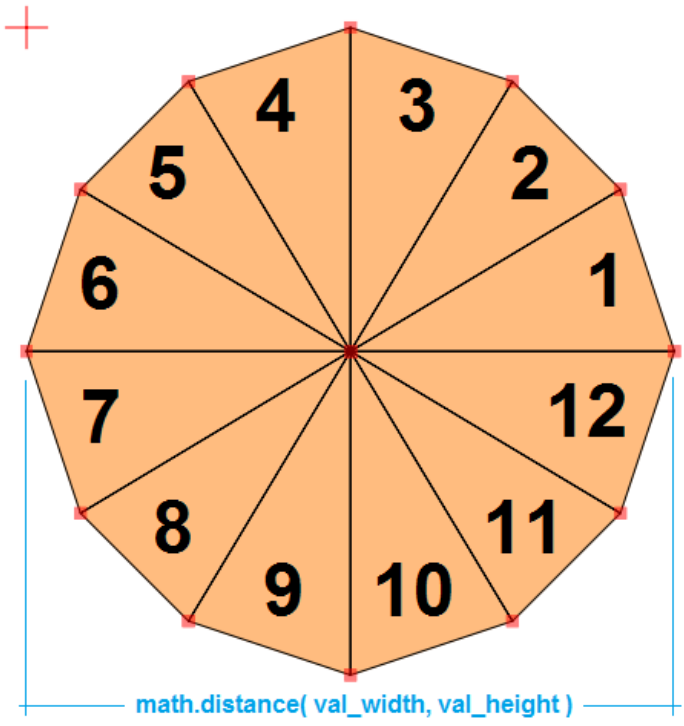
- **Ejemplo 1.** Todos los parámetros por default:

```
Return [fx]:
shape.multi4( )
```



- **Ejemplo 2.** Modificar a **loop1**:

```
Return [fx]:
shape.multi4( "default", 12 )
```



Del Ejemplo 2 notamos cómo el ancho total de la **shape** es el asignado por default:

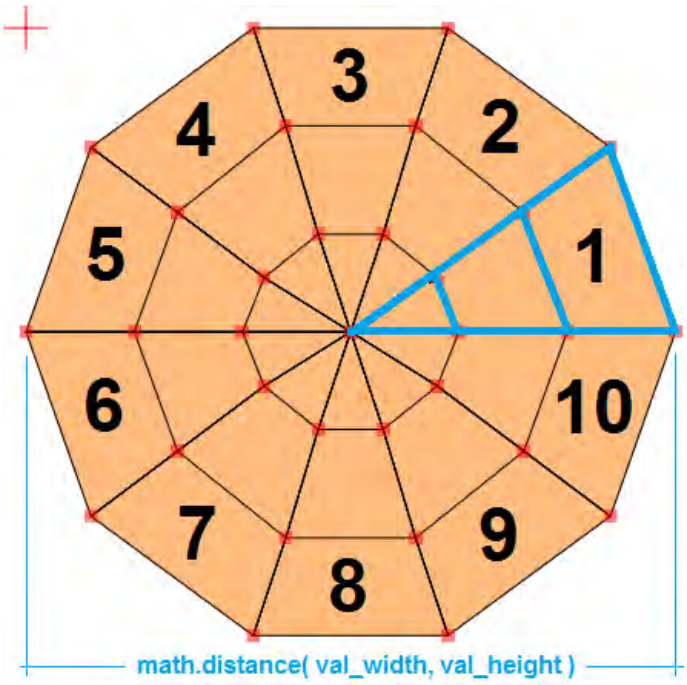
math.distance(val_width, val_height)

Observamos también que el **Arreglo Radial** está compuesto por doce Shapes individuales, y como **loop2** no está, toma su valor por default que es 1, así que el **loop** total sería de 12 X 1 = 12.

- **Ejemplo 3.** Modificar el parámetro **loop2**:

```
Return [fx]:
shape.multi4( "default", 10, 3 )
```

Ahora el parámetro **loop2** es 3, lo que hace que el **Arreglo Radial** se multiplique tres veces:

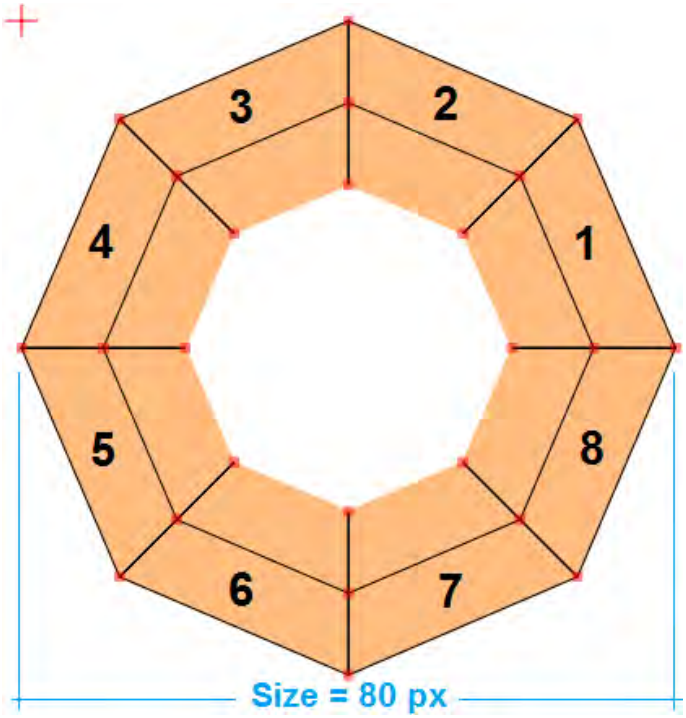


Como **loop1** es 10 en este ejemplo, entonces cada **Arreglo Radial** está compuesto por 10 Shapes individuales, que multiplicados por 3, de **loop2**, hace que el **loop** total sea de 10 X 3 = 30.

- **Ejemplo 4.** Usar un cuarto parámetro en la función que condiciona los anteriores resultados:

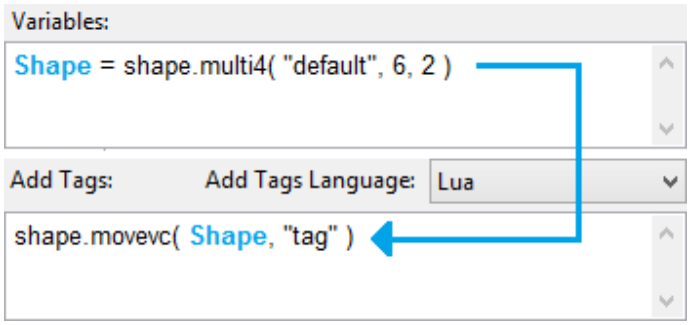
```
Return [fx]:
shape.multi4( 80, 8, 4, 2 )
```

- **Size** = 80 px
- **loop1** = 8
- **loop2** = 4
- Repeticiones a tener en cuanta: 2

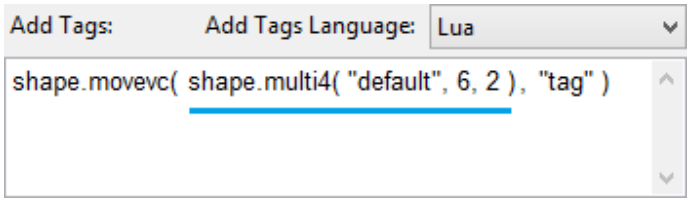


Entonces, de las cuatro repeticiones del **Arreglo Radial**, solo se tendrán en cuenta las dos primeras, gracias al cuarto parámetro de la función **shape.multi4**.

Ya teniendo una mejor idea de cómo usar esta función, se nos hace un poco más simple poderla emplear dentro de la función **shape.movevc**. Ejemplo:



O sino, de forma directa:



Crea un **Arreglo Radial** de 6 Shapes individuales y dicho Arreglo se repite 2 veces, para un total de 12 clip's:

Hecho este ejemplo, ya se podrán imaginar la gran cantidad de opciones que nos ofrece esta función. Todo es cuestión de ensayar y experimentar con las diversas combinaciones hasta que se familiaricen con esta y las demás funciones.

Es todo por ahora. En el **Tomo XXI** continuaremos viendo más de las funciones de la librería **shape**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

www.facebook.com/karaeffector

Kara Effector 3.2: Effector Book Vol. II [Tomo XXI]

Kara Effector 3.2:

El **Tomo XXI** es otro más dedicado a la librería **shape**, que como ya habrán notado, es la más extensa hasta ahora vista en el **Kara Effector**. El tamaño de esta librería nos da una idea de la importancia de las Shapes en un efecto karaoke, y es por ello que debemos tomarnos un tiempo en ver y conocer a cada una de las funciones y recursos disponibles para poder dominarlas.

Librería Shape [KE]:

shape.multi5(Shape, width, height, Dxy): genera una **shape** conformada por múltiples Shapes para ser usada en las funciones **shape.movevc y shape.movevci**.

Esta función está apoyada en la función **shape.array** para generar una **shape múltiple** que se retornará, con las siguientes características:

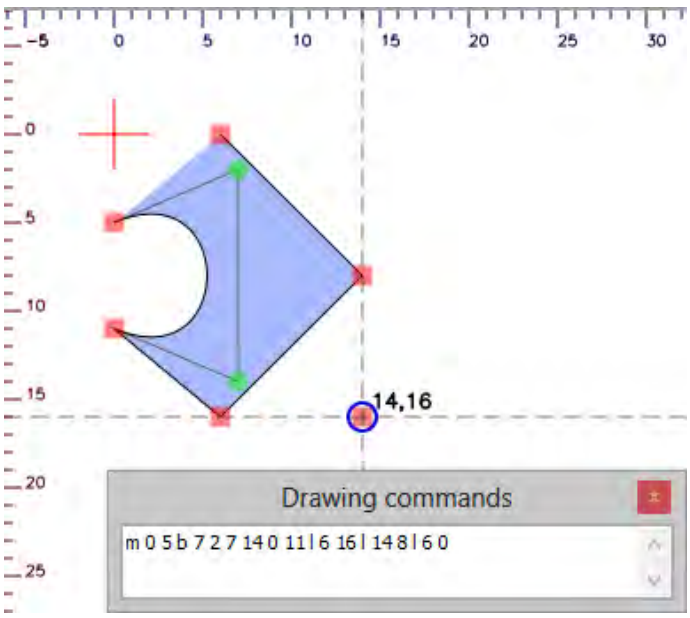
Shape: es la **shape** o la **tabla** de Shapes que se usará como módulo o molde para generar la **shape múltiple**.

width: es el ancho que, a lo mínimo, tendrá la **shape múltiple** generada. Su valor por default es **val_width**, es decir, el ancho del objeto karaoke dependiendo del **Template Type**.

height: es el alto que, a lo mínimo, tendrá la **shape múltiple** generada. El valor que tiene por default es **val_height**, es decir, el alto del objeto karaoke dependiendo del **Template Type**.

Dxy: es la distancia medida en pixeles que separará a una **shape** de la otra. En el caso de ser un número, ésa valor será la distancia horizontal que las separe, y para el caso en que dicho parámetro sea una **tabla** con dos valores numéricos, el primero de ellos indicará la distancia horizontal y el segundo la vertical, ambos en pixeles. Su valor por default es **Dxy** = {0, 0}, o sea, 0 pixeles en cada eje.

Para los próximos ejemplos, usaremos esta simple **shape** que mide 14 px X 16 px:



Y como en los ejemplos de los **Tomos** anteriores, con su código declararemos una variable con el fin de hacer más manejable las funciones en las que la usaremos:

Variables:

```
mi_shape = "m 0 5 b 7 2 7 14 0 11 l 6 16 l 14 8 l 6 0 "
```

Recordemos que el declarar una variable, en este caso, no es obligatorio. Se puede usar la **shape** directamente dentro de la función, siempre y cuando se ponga entre comillas simples o dobles, cualquiera de las dos.

- **Ejemplo 1. Template Type: Syl**

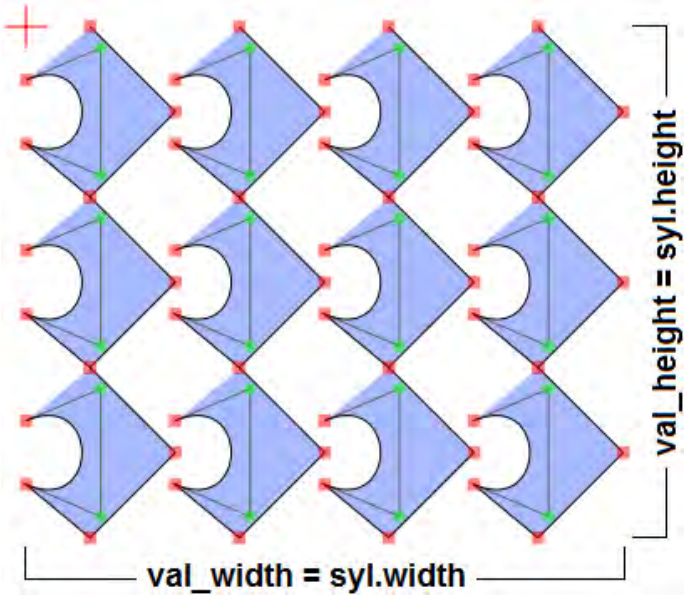
Return [fx]:

```
shape.multi5( mi_shape )
```

- **Shape**: **mi_shape**
- **width**: por default (**syl.width**)
- **height**: por default (**syl.height**)
- **Dxy**: por default (**Dxy** = {0, 0})

Los valores por default de **width** y **height** dependen del **Template Type**, que en este caso es **Syl**.

Vemos en detalle cómo la **shape** ingresada se multiplica en ambos eje hasta alcanzar las medidas **width** y **height**, que en este ejemplo están por default:

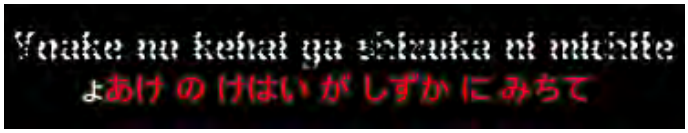


En la imagen anterior la **multi shape** está creada por 12 Shapes individuales (**mi_shape**) y que tanto la distancia horizontal como la vertical que las separa, es de 0 px.

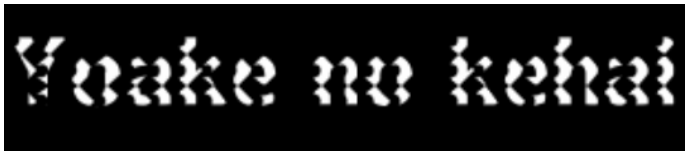
Ahora veamos una pequeña muestra de cómo se vería esta **multi shape** usada en la función **shape.movevc**:

Add Tags: Add Tags Language: Lua

```
shape.movevc( shape.multi5( mi_shape ) )
```



En aumento:

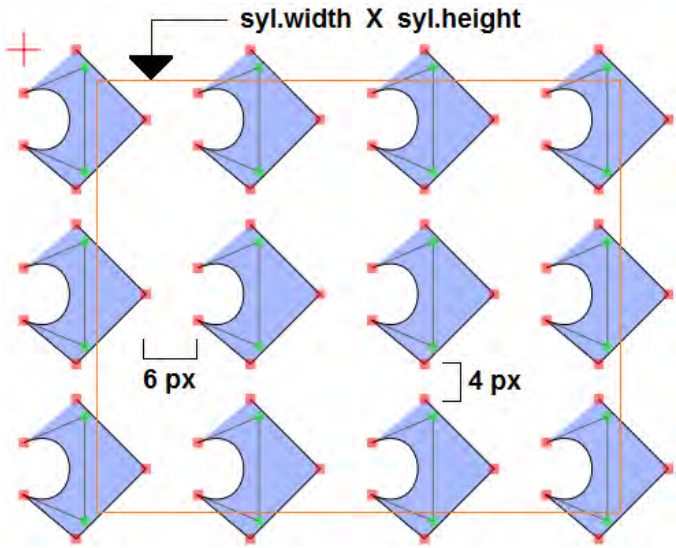


- **Ejemplo 2. Template Type: Syl**
- **width**: **syl.width** + 10
- **height**: **syl.height** + 10
- **Dxy**: {6, 4}

El fin de aumentar las dimensiones de la **multi shape** es para que los clip's generados sean un poco más grande y que dentro de ellos se puedan ver por completo los **objetos karaoke** en el caso en que estén afectados por un **\blur**, de lo contrario, el "brillo" generado por este tag quedaría por fuera de los clip's.

Return [fx]:

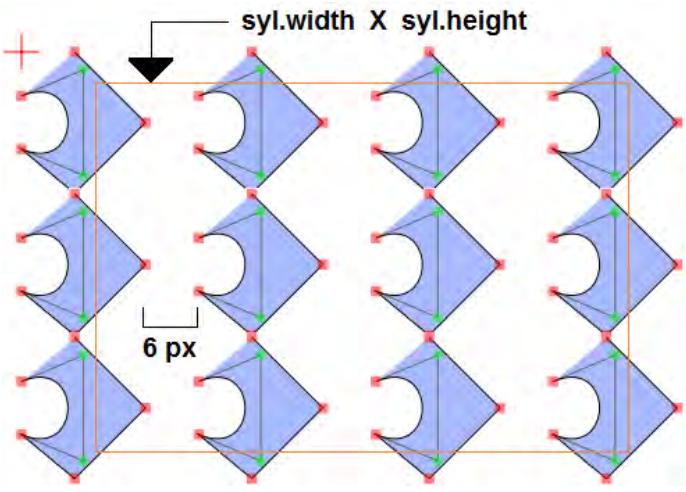
```
shape.multi5( mi_shape, syl.width + 10,
              syl.height + 10, {6, 4} )
```



- **Ejemplo 3. Dxy** = valor numérico:

Return [fx]:

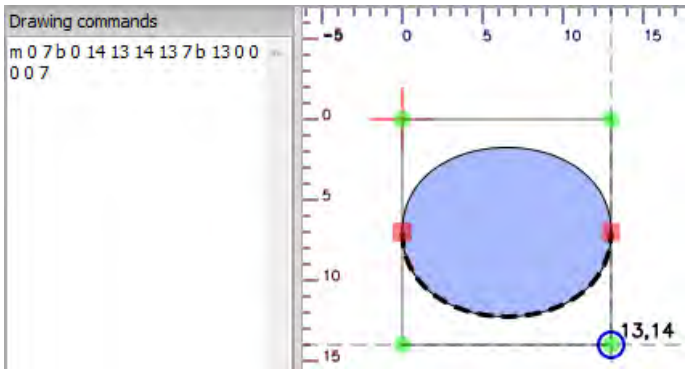
```
shape.multi5( mi_shape, syl.width + 10,
              syl.height + 10, 6 )
```



Como **Dxy** es un valor numérico (6 px), ése será el valor de la distancia que separa las Shapes horizontalmente, y por default, la separación vertical será de 0 px.

- **Ejemplo 4. Shape** = **tabla** de Shapes:

Dibujamos una segunda **shape** que posteriormente estará en la misma **tabla** que la **shape** del ejemplo anterior:

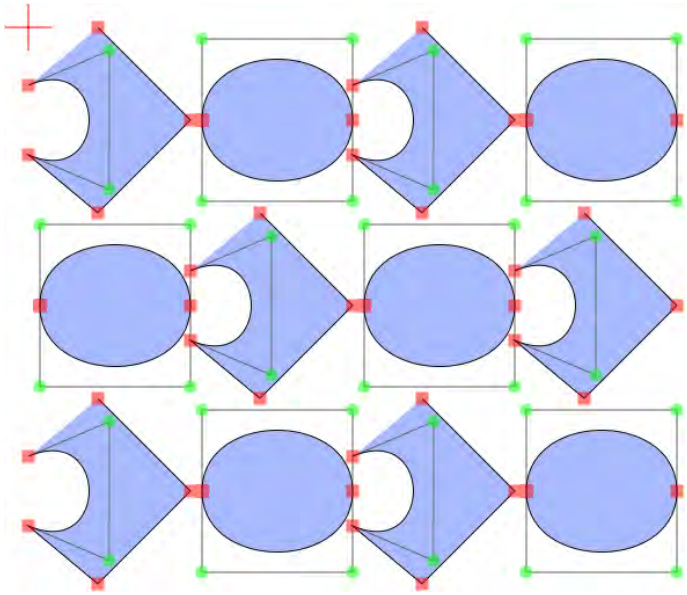


Declaramos la **tabla** con las dos Shapes:

Variables:

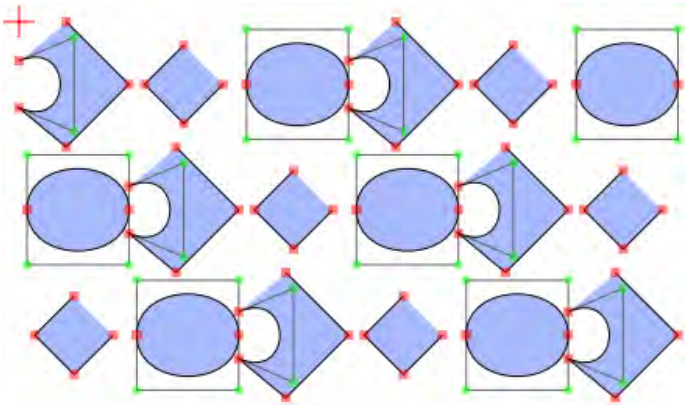
```
mi_shape = {"m 0 5 b 7 2 7 14 0 11 l 6 16 l 14 8 l 6 0 ",
            "m 0 7 b 0 14 13 14 13 7 b 13 0 0 0 0 7 " }
```

En este ejemplo solo hay dos Shapes, pero la cantidad de Shapes en la **tabla** no tiene límite, todo dependerá del efecto que queramos hacer.



Entonces las Shapes de la **tabla mi_shape** se multiplicarán de forma alternada y crearán la **multi shape**.

Un ejemplo de una **multi shape** creada a partir de una **tabla** con tres Shapes:



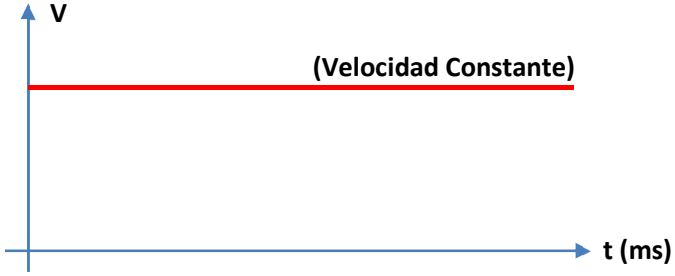
Usada en **shape.movevc**:

Las distancias que separan a las Shapes, tanto vertical como horizontalmente, no necesariamente deben ser valores positivos o cero, también pueden ser valores negativos con el fin de acercar más una **shape** respecto a las otras.

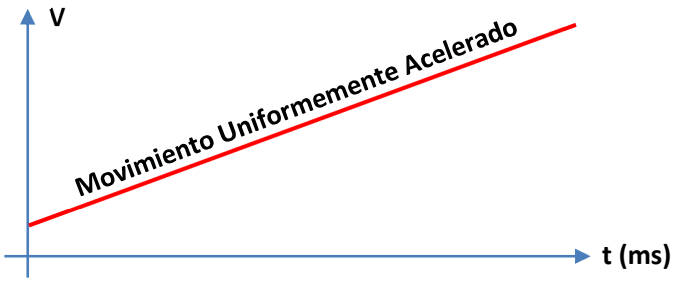
Las siguientes funciones de la **librería shape** están basadas en los conceptos de movimiento y aceleración. Los tags de movimiento en los filtros **VSFilter 2.39** y **VSFilterMod** son:

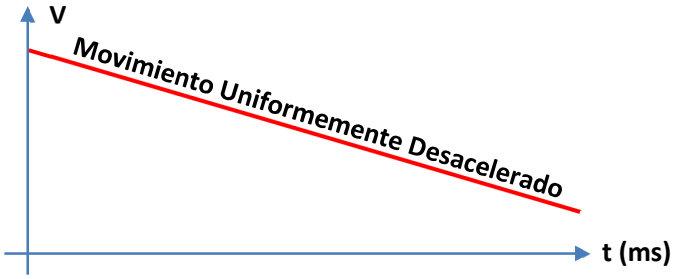
- \move
- \moves3
- \moves4
- \mover
- \movevc

Y ninguno de los anteriores cinco tags puede acelerar o desacelerar al memento de ejecutar el movimiento. Todos ellos generan un movimiento con velocidad constante:



Es decir que la velocidad es la misma siempre, a medida que transcurre el tiempo. Para los dos casos de **Movimientos Uniformemente Acelerados** tenemos:





Para generar este tipo de movimientos en el **Aegisub** hay varios tipos de combinaciones de tags que lo pueden hacer posible. Ejemplos:

- \org(Px, Py) \t(t1, t2, **aceleración**, \frz<**ángulo**>)
- \t(t1, t2, **aceleración**, \fsp<**distancia horizontal**>)
- \t(t1, t2, **aceleración**, \fsvp<**distancia vertical**>)

Cada una de ellas con cierto nivel de complejidad e incluso de imprecisión. Los siguientes tipos de movimientos que veremos están basados en una **shape** "**invisible**" que hará que el **objeto karaoke** se desplace de un punto a otro con la aceleración que nosotros decidamos:

shape.Lmove(x1, y1, x2, y2, t1, t2, accel): genera una **shape** invisible que mueve al **objeto karaoke** en línea recta desde el punto P_1 = (**x1, y1**) hasta P_2 = (**x2, y2**), desde el tiempo **t1** hasta el tiempo **t2** y con una aceleración **accel**. **Lmove** significa **Movimiento Lineal**.

x1 y **x2** son las coordenadas respecto al eje "**x**" y ambas tienen el mismo valor por default en el caso de no usar estos parámetros: **fx.move_x1**

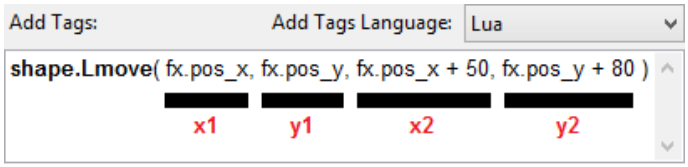
y1 y **y2** son las coordenadas respecto al eje "**y**" y ambas tienen el mismo valor por default en el caso de no usar estos parámetros: **fx.move_y1**

t1 y **t2** son los tiempos de inicio y final del movimiento medidos en milisegundos (**ms**), y sus valores por default son: **fx.movet_i** y **fx.movet_f**

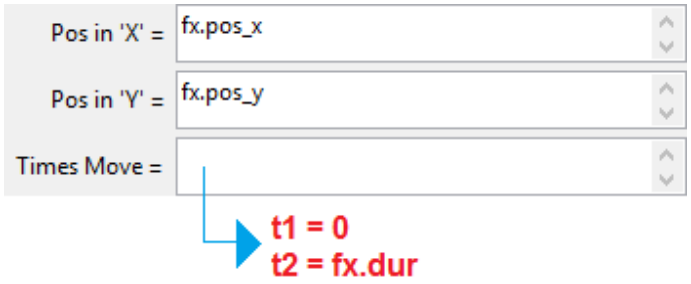
accel es la aceleración del movimiento y su valor por default es 1. Cuando la aceleración es 1, entonces la velocidad es constante. Para valores menores que 1 el movimiento es uniformemente desacelerado y para valores mayores que 1, el movimiento es uniformemente acelerado:

- **accel** < 1: Movimiento Uniforme Desacelerado
- **accel** = 1: Movimiento con Velocidad Constante
- **accel** > 1: Movimiento Uniforme Acelerado

- **Ejemplo 1. Template Type: Word**

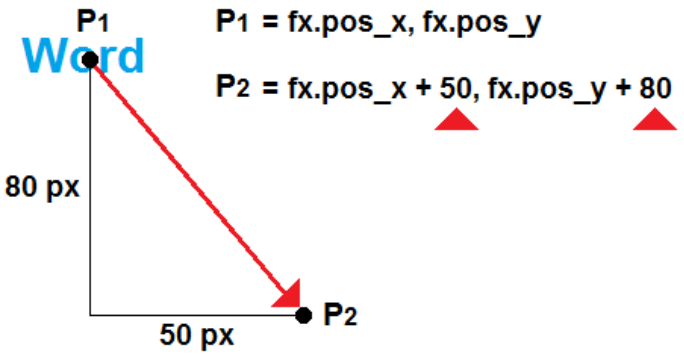


Así, los valores de **t1** y **t2** serán los que tienen por default:



El parámetro **accel** también tendría su valor por default: 1

Entonces la función hará que cada palabra (Word) se mueva desde su posición original hacia un punto ubicado **50 px** a su derecha y **80 px** hacia abajo, en la duración total de la línea de fx y sin ninguna aceleración:

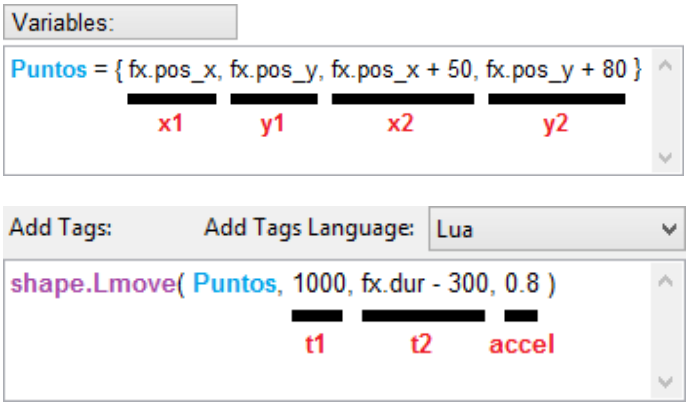


En la siguiente imagen vemos la **shape invisible** que hace que cada palabra (Word) se mueva según los parámetros que hemos ingresado en la función:

0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱Kodoku
0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱na
0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱hoho
0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱wo
0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱nurasu
0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱nurasu
0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 100 100 ✱kedo

Las coordenadas de los dos puntos ingresados en la función también pueden ser primero declaradas en una **tabla**, en la celda de texto "**Variables**".

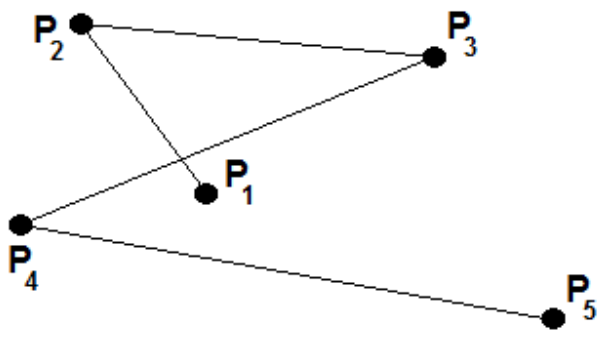
- **Ejemplo 2**:



Cuando se remplazan a los cuatro primeros parámetros de la función, con una **tabla** de puntos, ésta puede tener la cantidad de puntos que queramos:

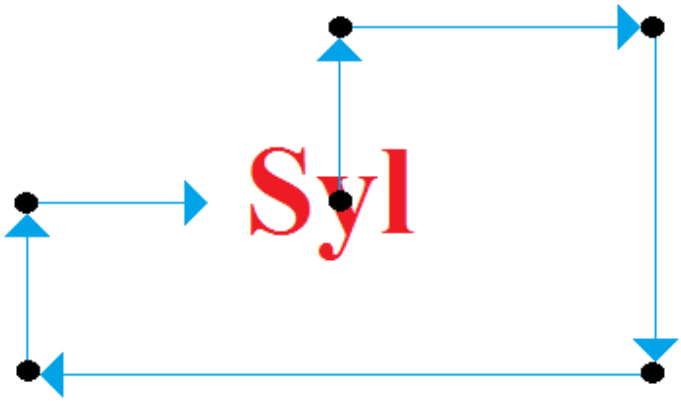
Puntos: { x1, y1, x2, y2, x3, y3,… …xn, yn }

Lo que generará un movimiento lineal entre cada uno de los puntos ingresados en la **tabla**:



En tiempo de desplazamiento entre un punto y otro es proporcional a la distancia entre ellos.

Si por ejemplo quisiéramos que el objeto karaoke se moviera varias veces alrededor de su centro y luego retorne a su posición original, como en la siguiente imagen:



Lo que debemos hacer es declarar a cada una de las coordenadas de esos puntos en una **tabla**, y luego usar dicha **tabla** en la función **shape.Lmove** con los tiempos por default o asignados por nosotros mismos, al igual que la aceleración del movimiento. Los valores de aceleración que recomendamos son entre 0.3 y 2.5

Entonces, a parte de la ventaja que nos da esta función de poder hacer un movimiento acelerado, también podemos mover al **objeto karaoke** en n cantidad de puntos en un tiempo determinado por nosotros mismos. Esta es la primera de ocho funciones que generan movimiento, y cada una de ellas con características distintas que iremos viendo en los próximos **Tomos**.

De estas ocho funciones mencionadas, cuatro de ellas pertenecen a la **librería shape**, y hasta el momento solo hemos visto la referente a movimientos lineales, pero esto es apenas la punta del iceberg, ya que las que vienen son igual o hasta más atractivas como herramienta para crear nuevos efectos.

Es todo por ahora para el **Tomo XXI**, pero la **librería shape** continuará en el próximo **Tomo**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoDCE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXII]

Kara Effector 3.2:

El **Tomo XXII** es otro más dedicado a la librería **shape**, que como ya habrán notado, es la más extensa hasta ahora vista en el **Kara Effector**. El tamaño de esta librería nos da una idea de la importancia de las Shapes en un efecto karaoke, y es por ello que debemos tomarnos un tiempo en ver y conocer a cada una de las funciones y recursos disponibles para poder dominarlas.

Librería Shape [KE]:

shape.Pmove(x(s), y(s), dom, t1, t2, acc, off_t)

Esta función hace que el objeto karaoke se mueva siguiendo la trayectoria de la gráfica de la función definida por los parámetros **x(s)** y **y(s)** en el dominio **dom**.

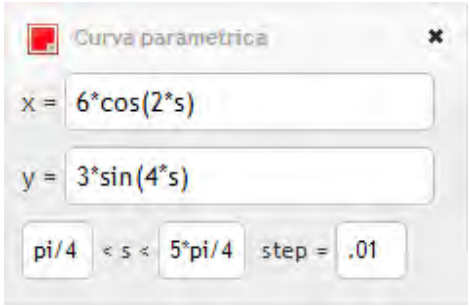
Los parámetros **t1** y **t2** son los tiempos de inicio y final del movimiento del objeto karaoke y sus valores por default son 0 y **fx.dur.**

El parámetro **dom** es el dominio de las funciones paramétricas **x(s)** y **y(s)**. Cuando es un valor numérico, el dominio será el intervalo cerrado entre 0 y dicho valor. Cuando es una **tabla**, el dominio va desde el valor del primer elemento hasta el segundo: {**dom_i**, **dom_f**}

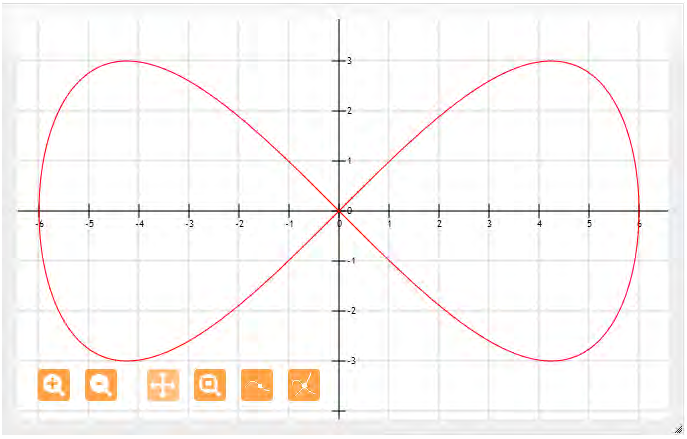
El parámetro **acc** es la aceleración del movimiento en las transformaciones que genera la función, y su valor por default es 1.

El parámetro **off_t** hace referencia al tiempo a añadir o restar de la duración de las transformaciones. Cada una de las transformaciones que genera esta función tiene una duración por default de 2.4***frame_dur**, y con **off_t** podemos añadir o quitar tiempo a esa duración.

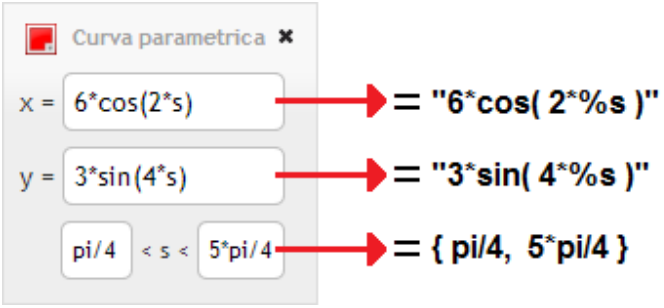
Ejemplo: www.fooplot.com



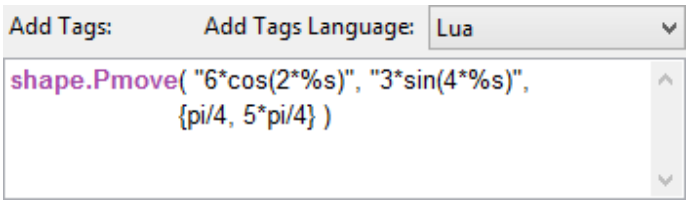
Esta es la gráfica de los anteriores parámetros:



Esto es lo que debemos hacer para pasar la información desde el graficador a la función:



- Poner las ecuaciones paramétricas entre comillas, ya sean simples y dobles.
- No olvidar poner siempre el símbolo del producto (*), o sea: 6*cos en vez de 6cos.
- Añadir el símbolo de porcentaje (%) antes de cada "s" que aparezca en las ecuaciones paramétricas.
- Hacer una **tabla** con los valores de inicio y final del dominio de las funciones paramétricas **x(s)** y **y(s).**



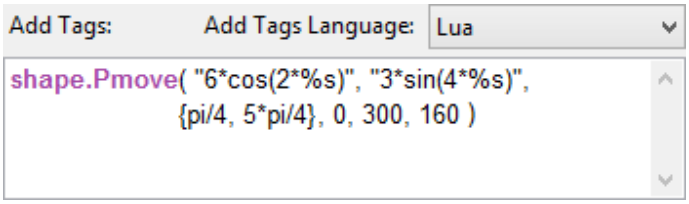
Entonces la función genera una shape invisible y una serie de transformaciones que harán que el objeto karaoke se mueva siguiendo la trayectoria de la gráfica generada por las dos ecuaciones paramétricas y el dominio asignado.

En el caso en que notemos que el movimiento sea poco perceptible, es porque las proporciones de las funciones son muy pequeñas. Las proporciones son los valores por el cual multiplicamos las funciones del anterior ejemplo:

- "cos(2*%s)" → "**6** * cos(2*%s)"
- "sin(4*%s)" → "**3** * sin(4*%s)"

Ejemplo:

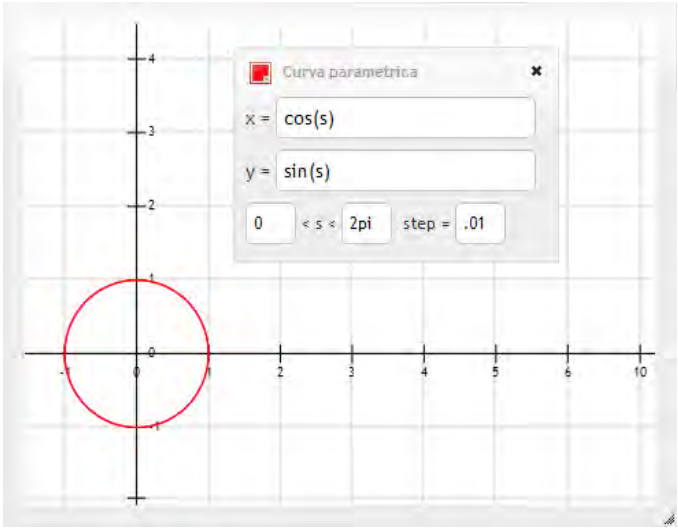
Este sería un ejemplo usando todos los parámetros de la función:



O sea que el movimiento empezará en 0 ms y terminará a los 300 ms. A parte de eso le estamos sumando 160 ms a la duración de las transformaciones generadas.

Ejemplo:

Los siguientes parámetros corresponden a la gráfica de un círculo de 1 px de radio, de modo que si los usamos para la función **shape.Pmove**, entonces no se notaría mucho el movimiento en el vídeo:

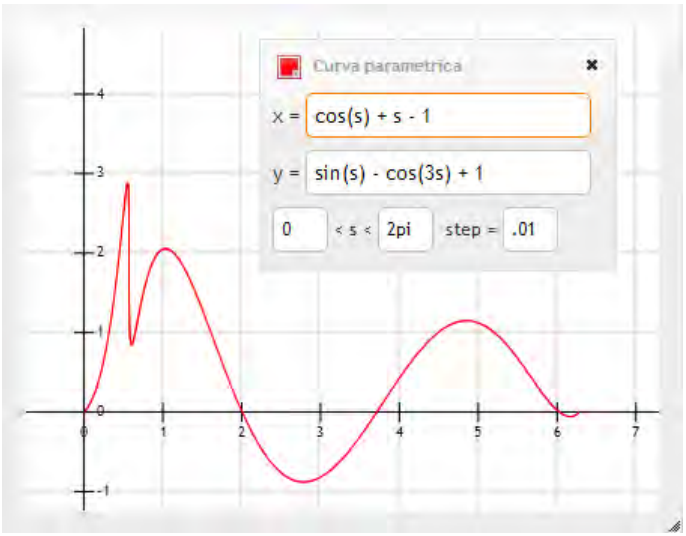


Lo que debemos hacer es multiplicar a cada ecuación paramétrica por un valor tal, que haga que el círculo sea más grande y así sea evidente el movimiento. Ejemplo:

- "100 * cos(%s)"
- "100 * sin (%s)"

Ejemplo:

Podemos definir las ecuaciones paramétricas en el graficador online y una vez que obtengamos una gráfica que sea de nuestro agrado, la podemos usar tal cual y luego ir modificando ambas proporciones con el fin de que el movimiento se ajuste a las necesidades de nuestro efecto:



Las ecuaciones para usar en la función quedarían:

- "cos(%s) + %s – 1"
- "sin(%s) – cos (3*%s) + 1"

Dominio:
- { 0, 2*pi }

Y para modificar las proporciones, debemos encerrar entre paréntesis a las ecuaciones paramétricas y así podremos multiplicar a cada una de ellas por el valor que necesitemos. Ejemplo:

- "45 * (cos(%s) + %s – 1)"
- "72 * (sin(%s) – cos (3*%s) + 1)"

shape.Smove(Shape, t1, t2, off)

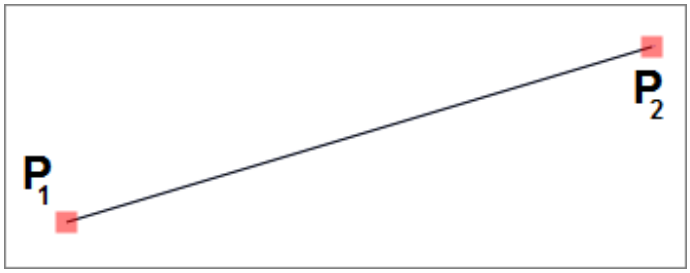
Esta función hace que el objeto karaoke se mueva siguiendo el contorno del perímetro de la shape ingresada "**Shape**", desde **t1** a **t2**.

Los parámetros **t1** y **t2** son los tiempos de inicio y final del movimiento del objeto karaoke y sus valores por default son 0 y **fx.dur.**

El parámetro **off** hace referencia al tiempo a añadir o restar de la duración de las transformaciones. Cada una de las transformaciones que genera esta función tiene una duración por default de 2***frame_dur**, y con **off** podemos añadir o quitar tiempo a esa duración.

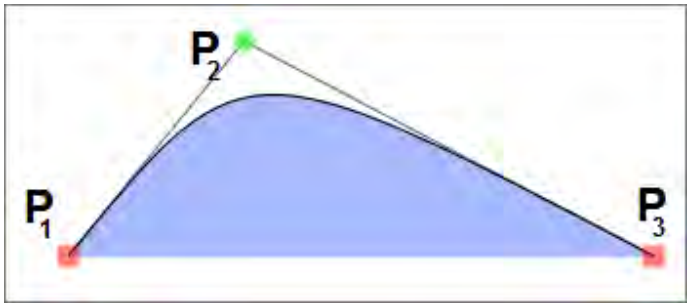
El parámetro **Shape** puede ser o una **tabla** o un **string**. Para el caso del **string**, debe ser el código de la **shape** en formato .**ass**, es decir el código que obtenemos de la **shape** en el **ASSDraw3**; o sea que podemos utilizar cualquiera de las Shapes que por default ya vienen en el **Kara Effector**. Y para el caso en que el parámetro **Shape** sea una **tabla**, ésta debe contener valores numéricos que cumplan con las siguientes características:

- Coordenadas de 2 puntos:
  Shape = { P_{x1}, P_{y1}, P_{x2}, P_{y2} }



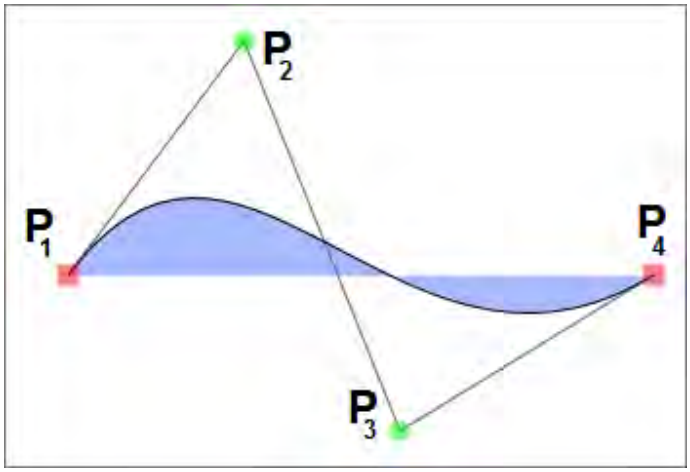
El objeto karaoke se moverá siguiendo la trayectoria de la línea recta que forman los dos puntos ingresados.

- Coordenadas de 3 puntos:
  Shape = { P_{x1}, P_{y1}, P_{x2}, P_{y2}, P_{x3}, P_{y3} }



El objeto karaoke se moverá siguiendo la trayectoria de la **Curva Bezier** que forman los tres puntos ingresados.

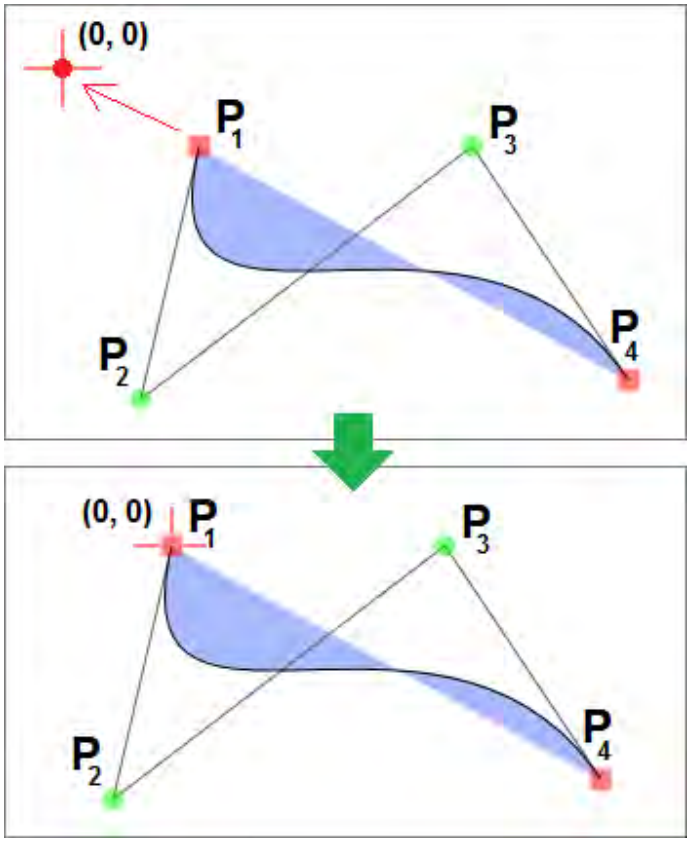
- Coordenadas de 4 puntos:
  Shape = { P_{x1}, P_{y1}, P_{x2}, P_{y2}, P_{x3}, P_{y3}, P_{x4}, P_{y4} }



El objeto karaoke se moverá siguiendo la trayectoria de la **Curva Bezier** que forman los cuatro puntos ingresados.

Es decir, que la función toma las coordenadas de los puntos ingresados y las convierte en una **shape**, para hacer que el objeto karaoke se mueva siguiendo dicha trayectoria.

Explicado esto, podemos decir que la función siempre toma como base a la trayectoria del perímetro de una **shape** para hacer que el objeto karaoke se mueva en el contorno de la misma. Lo siguiente que hace la función es mover la **shape** de tal forma que el primer punto quede en las coordenadas P = (0, 0) del plano en el **ASSDraw3**. Ejemplo:



Una vez se desplaza de la anterior forma a la shape, la función hace coincidir al centro del objeto karaoke con ese punto P = (0, 0), y ahí sí genera las transformaciones que hacen posible el movimiento.

Ejemplo:

Variables:

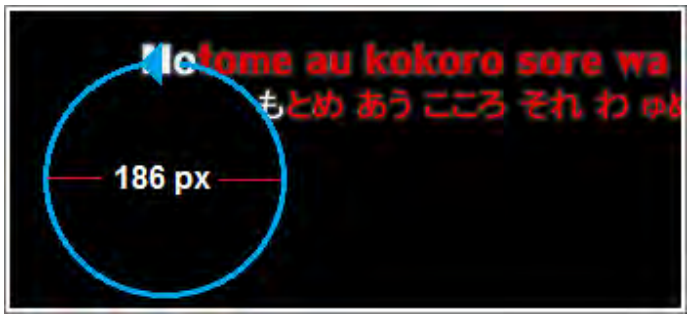
```
mi_shape = shape.size( shape.circle, 186 )
```

Declaramos una **shape** en la celda de texto "**Variable**", aunque sabemos que no es necesario, ya que lo podemos hacer directamente dentro de la función. La **shape** para este ejemplo es un círculo de 186 px de diámetro.

Add Tags: Add Tags Language: Lua

```
shape.Smove( mi_shape, fx.dur/2, fx.dur )
```

- **t1** = **fx.dur**/2
- **t2** = **fx.dur**



El objeto karaoke, en este caso las Sílabas, se moverán siguiendo como trayectoria al perímetro del círculo de 186 px de diámetro a partir de la mitad de la duración total de la línea de fx (**t1** = **fx.dur**/2), hasta el tiempo final de la misma, como se ve en la imagen anterior.

shape.Rmove(Dx, Dy, t1, t2, acc, off)

Esta función hace que el objeto karaoke se mueva aleatoriamente de forma lineal, desde su centro por default (**fx.move_x1**, **fx.move_y1**), sin exceder los límites **Dx** y **Dy**. El movimiento que se genera es perpetuo, dependiendo de los límites de tiempo **t1** y **t2**.

Los parámetros **t1** y **t2** son los tiempos de inicio y final del movimiento del objeto karaoke y sus valores por default son 0 y **fx.dur.**

El parámetro **acc** es la aceleración del movimiento en las transformaciones que genera la función, y su valor por default es 1.

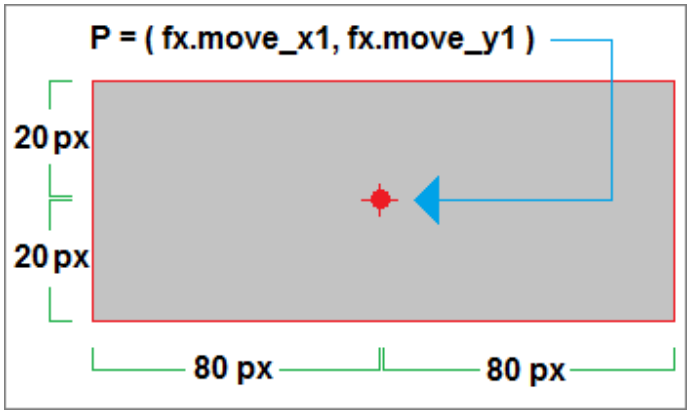
El parámetro **off** hace referencia al tiempo a añadir o restar de la duración de las transformaciones. Cada una de las transformaciones que genera esta función tiene una duración por default de 3.6***frame_dur**, y con **off** podemos añadir o quitar tiempo a esa duración.

Ejemplo:

- Dx = 80
- Dy = 20

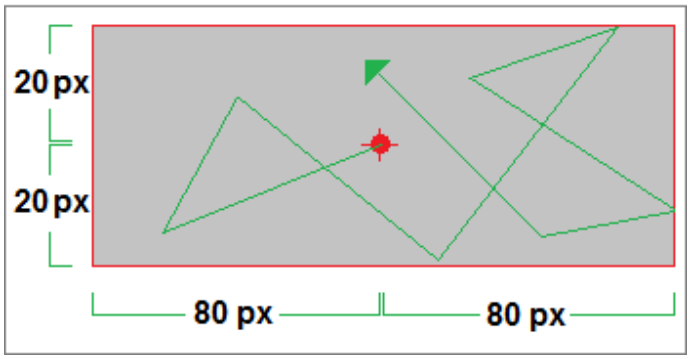
```
Add Tags:  Add Tags Language: Lua
shape.Rmove( 80, 20 )
```

Entonces la función crea un rectángulo imaginario con el doble de los valores ingresados en **Dx** y **Dy**, como ancho y alto de dicho rectángulo:



Y las transformaciones generadas harán que una **shape** invisible cree el **Movimiento Lineal Aleatorio** sin salirse nunca de los límites creados por el rectángulo imaginario:

Movimiento Lineal Aleatorio Delimitado



Esta función tiene una particularidad, que sin importar la cantidad de movimientos generados, la posición final será el mismo punto donde empezó, es decir que se moverá de forma aleatoria para finalmente terminar en donde estaba posicionado inicialmente: P = (**fx.move_x1**, **fx.move_y1**)

Ejemplo:

```
Add Tags:  Add Tags Language: Lua
shape.Rmove( 30, 25, 0, 400, 0.8 )
```

Entonces el objeto karaoke se moverá de forma aleatoria en un rectángulo imaginario de 60 X 50 px (el doble de cada valor ingresado en **Dx** y **Dy**), desde los 0 ms hasta los 400 ms y con una aceleración de 0.8; pero una vez transcurrido ese lapso de tiempo, volverá a su posición inicial de origen.

Las cuatro funciones que generan movimiento por medio de una shape invisible son:

- **shape.Lmove**
- **shape.Pmove**
- **shape.Smove**
- **shape.Rmove**

Aparte de **shape.Smove**, las otras tres funciones pueden generar movimiento con aceleración y cada una de ellas con las característica que hasta acá hemos visto.

El movimiento que generan las transformaciones de estas cuatro funciones es similar al generado por **shape.config**, con la diferencia que esta última genera un **loop** del objeto karaoke para dar la ilusión de movimiento.

Ya para terminar, el uso de estas cuatro funciones puede consumir una cantidad considerable de recursos de nuestra PC y volverla un poco lenta, lo que hará que no podamos apreciar en tiempo real a nuestro efecto. Todo dependerá del tipo de PC que cada uno tenga, pero pensando es este consumo de recursos, más adelante veremos otras cuatro funciones que también nos dan la posibilidad de generar movimiento acelerado con características similares a las que recientemente hemos visto y sin el efecto secundario de la lentitud de nuestra PC. Estas funciones pertenecen a la librería **tag** y en próximos **Tomos** las veremos en detalle.

Es todo por ahora para el **Tomo XXII**, pero la **librería shape** continuará en el próximo **Tomo**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoDCE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXIII]

Kara Effector 3.2:

El **Tomo XXIII** es otro más dedicado a la librería **shape**, que como ya habrán notado, es la más extensa hasta ahora vista en el **Kara Effector**. El tamaño de esta librería nos da una idea de la importancia de las Shapes en un efecto karaoke, y es por ello que debemos tomarnos un tiempo en ver y conocer a cada una de las funciones y recursos disponibles para poder dominarlas.

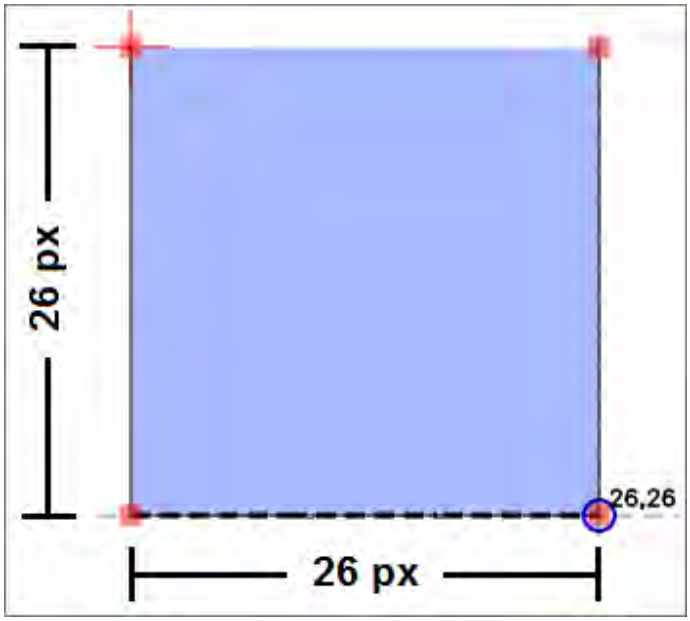
Librería Shape [KE]:

shape.length(Shape)

Esta función retorna la medida en pixeles de la longitud total del perímetro de la **shape** ingresada.

Ejemplo:

Shape.length(“m 0 0 l 0 26 l 26 26 l 26 0 l 0 0 “)

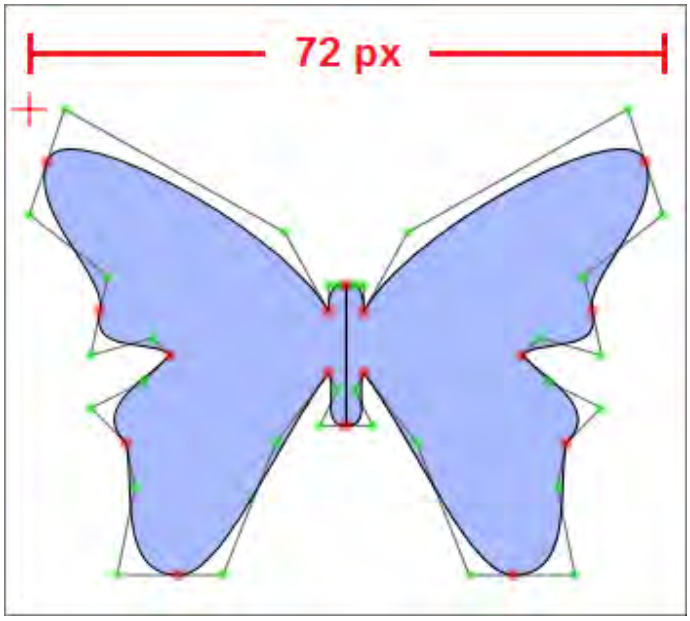


Entonces la función calculará la medida del perímetro de la **shape**, o sea: 26 + 26 + 26 + 26 = 104 px.

shape.width(Shape)

Esta función retorna la medida en pixeles del ancho total de la **shape** ingresada.

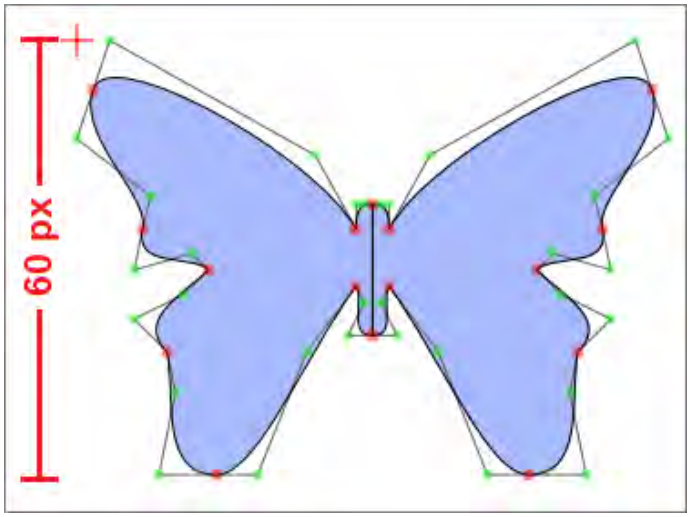
Ejemplo:



shape.height(Shape)

Esta función retorna la medida en pixeles de la altura total de la **shape** ingresada.

Ejemplo:



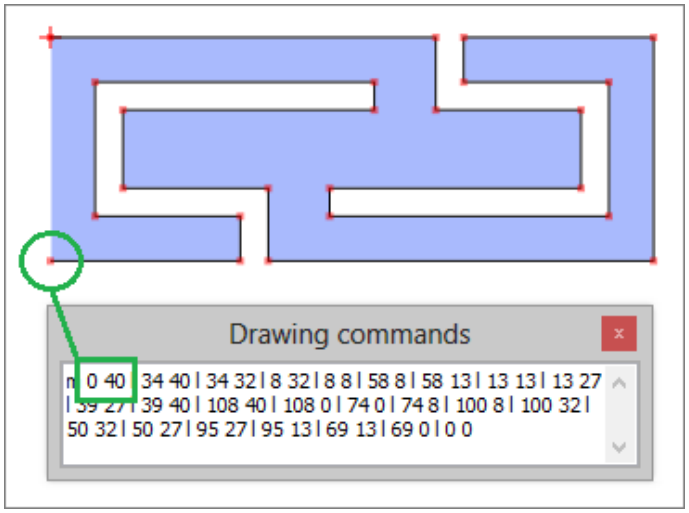
Las tres anteriores funciones están diseñadas para arrojar información básica de una **shape**, información como su longitud, su ancho y su altura. Esta información podrá ser usada en los efectos sin la necesidad de hacer los cálculos.

shape.firstpos(Shape, Px, Py)

Esta función es muy similar a **shape.displace** con la particularidad que mueve a todos los puntos de la **shape** ingresada teniendo en cuenta al primer punto de la misma, como referencia para hacerlo.

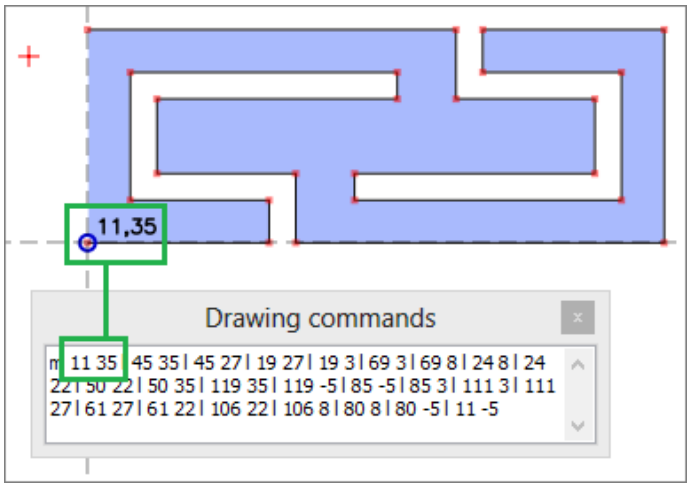
Ejemplo:

En la siguiente imagen vemos el primer punto de una shape, y con referencia a ese punto es que la función la moverá a la nueva posición que le indiquemos con los parámetros **Px** y **Py**:



Px es la coordenada respecto al eje “x” que tendrá el primer punto de la **shape**, y **Py** es la coordenada respecto al eje “y” de dicho punto:

shape.firstpos(mi_shape, 11, 35)



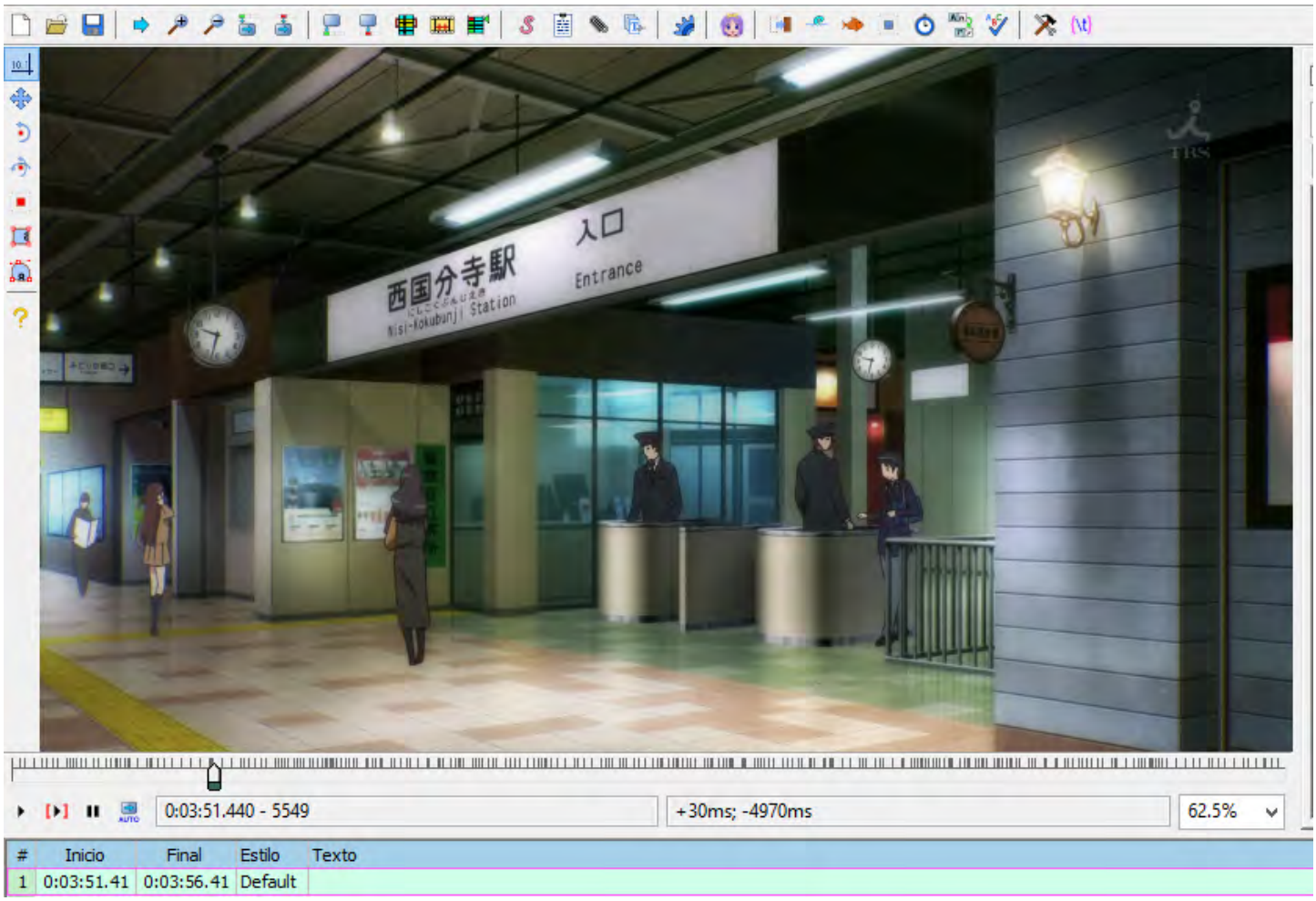
En conclusión, lo que hacemos al darles valores a los parámetros **Px** y **Py**, es asignar el punto exacto en donde quedará el primer punto de nuestra **shape** luego de ser desplazada. Esta función es ideal para reubicar la shape respecto al centro de la **Syl** y luego aplicar una de las funciones de **shape** de movimiento, como **shape.Smove**.

shape.from_clip( )

Esta función está pensada para ser usada en la edición, más que para hacer fx karaokes. Esta función ayuda a crear carteles y lo que hace es convertir un clip dibujado en la pantalla en un shape.

Ejemplo:

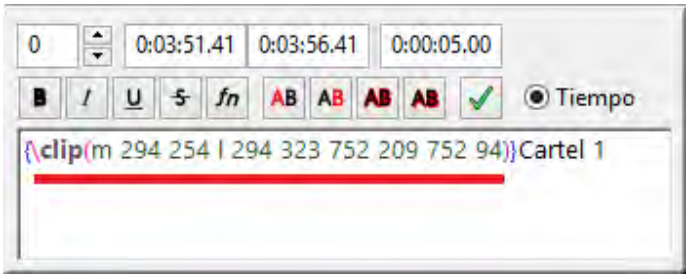
Ubicamos el frame en dónde está el cartel que queremos hacer y le damos los tiempos de inicio y final a una línea en la que posteriormente dibujaremos el clip:



En esa línea creada, escribimos algo, lo que sea, por ejemplo: “cartel 1” y seleccionamos la herramienta para recortar subtítulos en un área vectorial:

Y dibujamos el contorno del cartel:

Hecho esto, en nuestra línea que previamente habíamos creado, se debe ver el código del clip, algo como esto:



Y por último, en **Return [fx]** llamamos a la función:

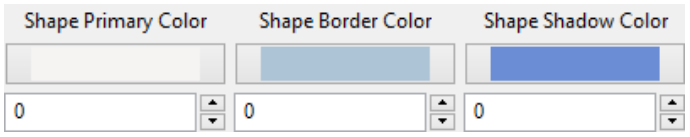
Return [fx]:

```
shape.from_clip( )
```

Y al aplicar ya podemos ver la **shape** justo en el lugar en donde está el cartel:

#	Inicio	Final	Estilo	Texto
1	0:03:51.41	0:03:56.41	Default	✳Cartel 1
2	0:03:51.41	0:03:56.41	Default	✳m 294 254 l 294 323 l 752 209 l 752 94

Recordemos que los colores y transparencias de una shape los podemos modificar con las herramientas de la **Ventana de Modificación** del **KE** destinadas para ello:



shape.redraw(Shape, tract)

Esta función cumple con la tarea de redibujar la shape que ingresemos, pero dividiendo a las partes que la componen en tramos muchos más cortos.

El parámetro **tract** es la medida en pixeles en la que será dividido cada segmento de la **shape** y debe ser un número mayor a cero. Su valor por default es 2.

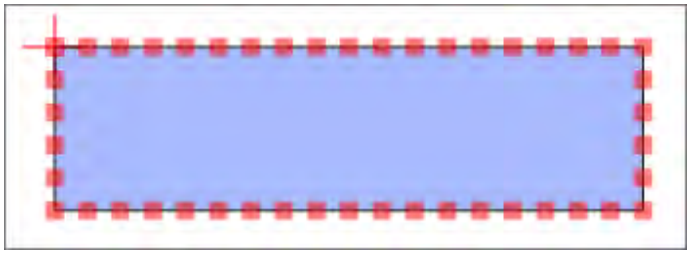
Ejemplo:

Entonces ponemos el código de la anterior **shape** dentro de la función y aplicamos:

Return [fx]:

```
shape.redraw( "m 0 0 l 0 10 l 38 10 l 38 0 l 0 0 " )
```

Y obtendremos esto:

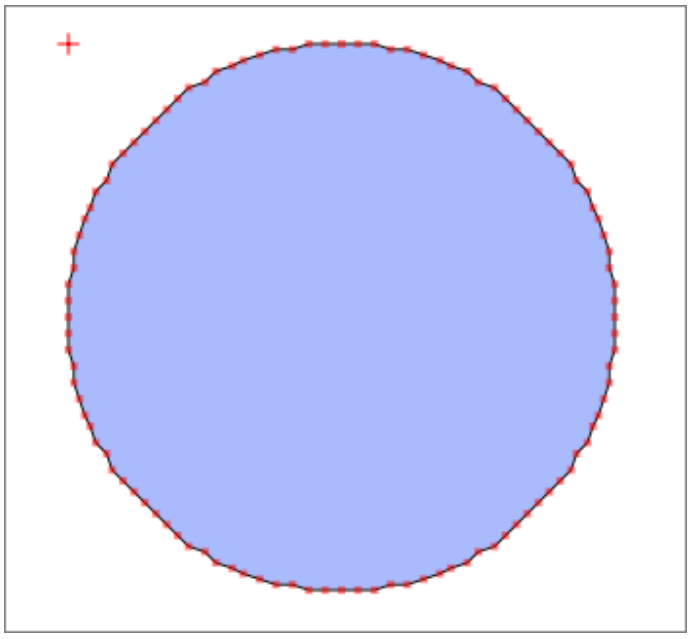


La **shape** que genera la función es la misma, pero dibujada con segmentos cortos de 2 pixeles de largo, ya que no pusimos el parámetro **tract**, entonces la función lo asume por default, es decir: 2 px.

Ejemplo:

Return [fx]:

```
shape.redraw( shape.circle, 3 )
```



Entonces la función redibuja el círculo con tramos rectos de 3 pixeles de longitud.

Puesto en práctica estos dos ejemplos, la pregunta sería: ¿por qué redibujar una **shape** en segmentos rectos más cortos?

Una de las varias respuestas a esa pregunta la puede responder la siguiente función de la **librería shape**:

```
shape.modify( Shape, modify )
```

Esta función modifica a los puntos de una shape por medio de una función ingresada en el parámetro **modify**, que debe tener la siguiente estructura:

modify:

```
function( x, y )
    -- Instrucciones --
    return  format( "%d %d", x, y)
end
```

Ejemplo:

Aplicamos el **shape.redraw** al círculo y la definimos como una variable en la celda de texto "**Variables**":

Variables:

```
mi_shape = shape.redraw( shape.circle, 1 )
```

Seguido de esto, definimos la función que modificará a la anterior **shape**:

Variables:

```
mi_shape = shape.redraw( shape.circle, 1 );
mi_modify = function( x, y )
        x = x + R(-6,6)      Instrucciones
        return format( "%d %d", x, y)
end
```

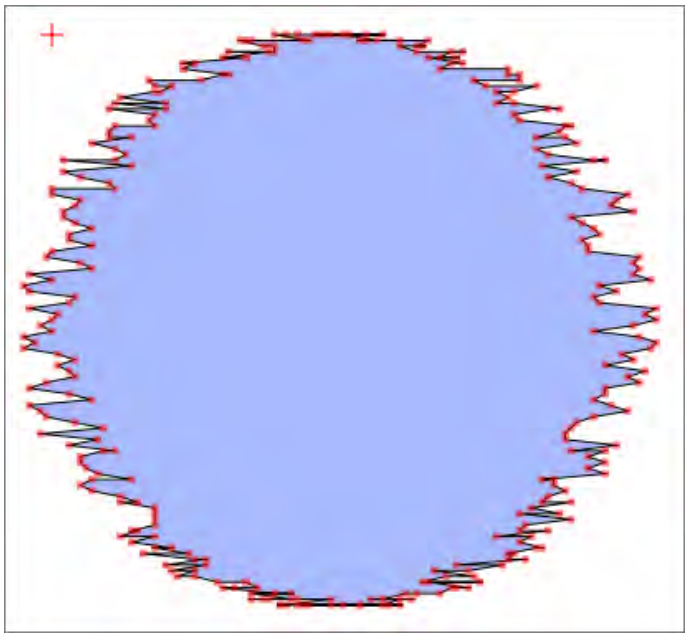
La instrucción de la anterior función declarada es que tome las coordenadas respecto al eje "x" de la **shape** y las modifique en una cantidad aleatoria entre -6 y 6 px.

Ahora llamamos a la función **shape.modify** con las dos variables que acabamos que declarar:

Return [fx]:

```
shape.modify( mi_shape, mi_modify )
```

Lo que dará por resultado algo como esto:



Y lo que antes era un simple círculo, ahora es uno que está distorsionado, como la estática de una TV sin señal.

Otra característica de la función **shape.modify** es que contiene internamente a la función **shape.info**, que como recordaremos, nos da la siguiente información de la **shape**:

- **minx** ← mínima coordenada en "x"
- **maxx** ← máxima coordenada en "x"
- **miny** ← mínima coordenada en "y"
- **maxy** ← máxima coordenada en "y"
- **w_shape** ← ancho de la **shape**
- **h_shape** ← alto de la **shape**

Y esta información la podemos usar dentro de la función que usaremos para modificar a la **shape**. Ejemplo:

```
1  mi_modify = function( x, y )
2      mod = (y - miny)/h_shape
3      x = x + 20*sin( 2*pi*mod )
4      y = 2*y
5      return format( "%d %d", x, y )
6  end
```

Generalmente uso el programa **Notepad++** para hacer las funciones, pero no es obligatorio hacerlo, ya que se pueden hacer directamente en el **Kara Effector** o hasta usar el Bloc de Notas es suficiente para ello.

La anterior función de modificación, declarada en la celda de texto "**Variable**" se vería así:

Variables:

```
mi_shape = shape.redraw( shape.circle, 1 );
mi_modify = function( x, y )
        mod = (y - miny)/h_shape
        x = x + 20*sin( 2*pi*mod )
        y = 2*y
        return format( "%d %d", x, y )
end
```

Lo que quieren decir las instrucciones de la anterior función es que a cada coordenada "x" de la **shape** ingresada, se le sume esta función trigonométrica:

20*math.sin(s) → 0 <= s <= 2*pi

Y que a cada coordenada en "y" se multiplique por 2:

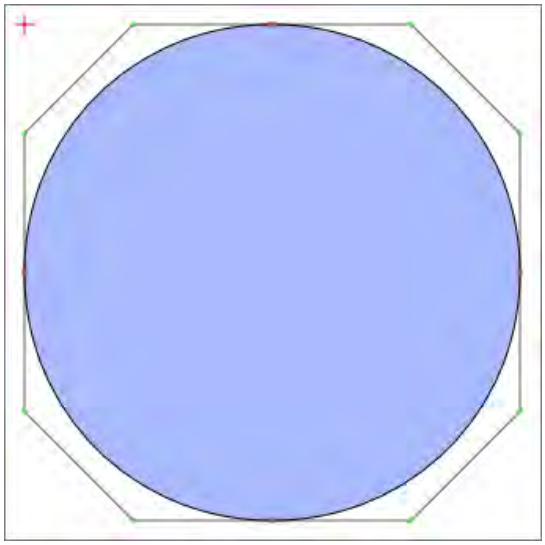
y = 2*y

Y ya sabemos lo que debemos poner en **Return [fx]:**

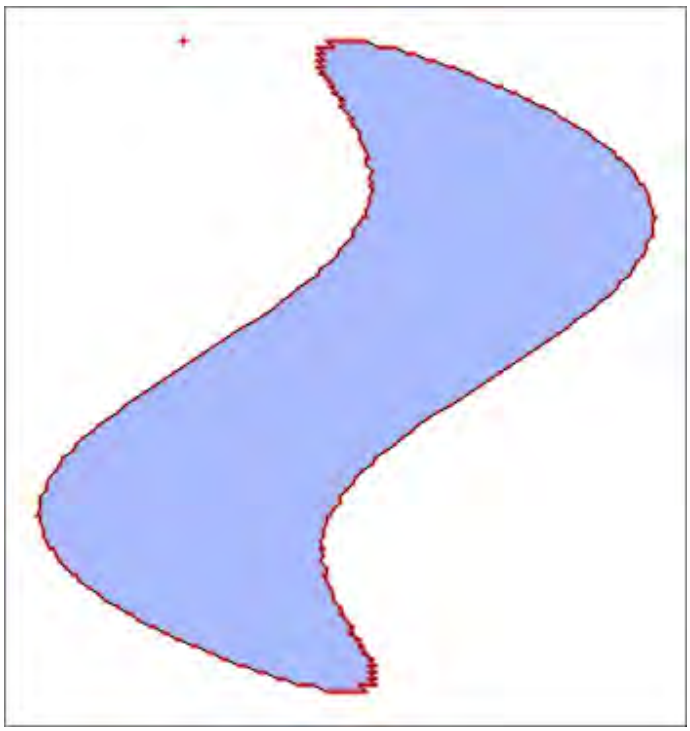
Return [fx]:

```
shape.modify( mi_shape, mi_modify )
```

Y convertiremos esta **shape**:



En esto:



Imaginen hacerle estas deformaciones al texto, sería algo genial para hacer nuestros fx. Pues es posible hacerlo y más adelante les mostraremos la forma de lograrlo, pero primero es importante que se vayan familiarizando con las funciones necesarias para ello.

Es todo por ahora para el **Tomo XXIII**, pero la **librería shape** continuará en el próximo **Tomo**. Intenten poner en práctica todos los ejemplos vistos en este **Tomo** y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario, exponer alguna duda o hacer alguna sugerencia. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXIV]

Kara Effector 3.2:

El **Tomo XXIV** es otro más dedicado a la librería **shape**, que como ya habrán notado, es la más extensa hasta ahora vista en el **Kara Effector**. El tamaño de esta librería nos da una idea de la importancia de las Shapes en un efecto karaoke, y es por ello que debemos tomarnos un tiempo en ver y conocer a cada una de las funciones y recursos disponibles para poder dominarlas.

Librería Shape [KE]:

shape.filter2(Shape, Filter, Split)

Esta función es la combinación de las funciones **shape.redraw** y **shape.modify** y tiene una modificación en la estructura de la función modificadora **Filter**.

El parámetro **Filter** es la función que modifica los puntos de la **shape** ingresada.

El parámetro **Split** es la longitud de los segmentos en los que se dividirá la **shape** ingresada.

La estructura de la función **Filter** es:

```
function( x, y )
    -- Instrucciones --
    return x, y
end
```

Noten que la estructura de la función es más simple que la de la función **shape.modify**, ya que la parte que retorna es simplemente: **return x, y**

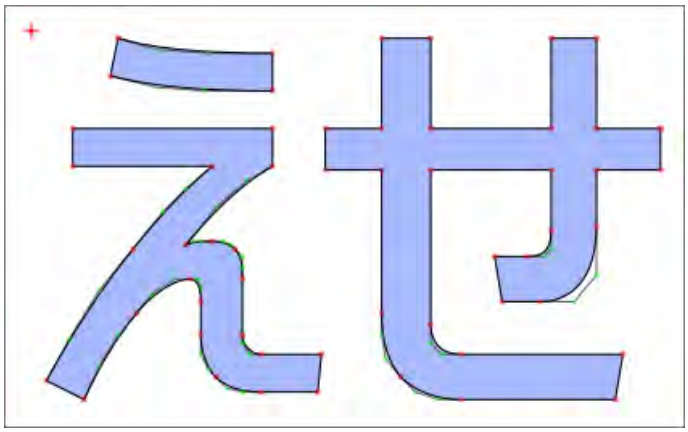
Ejemplo:

```
mi_modify = function( x, y )
        y = y + R(4,8)*(-1)^R(2)
        return x, y
end
```

Variables:

```
mi_modify = function( x, y )
        y = y + R(4,8)*(-1)^R(2)
        return x, y
end
```

Y usaremos esta shape para modificarla. Ya sabemos el procedimiento para definirla como una variable en la celda de texto "**Variables**":



Y en **Return [fx]** ponemos:

Return [fx]:

```
shape.filter2( mi_shape, mi_modify, 2 )
```

Y veremos cómo se ha modificado la **shape**:

Las coordenadas respecto al eje "y" se han modificado de manera considerable y hacen que la **shape** tenga un efecto distorsionado verticalmente.

Más adelante veremos cómo esta función nos ayudará a modificar el texto de nuestros karaokes de la misma manera que lo hace con la Shapes.

shape.from_audio(Audio, Width, Height_scale, Thickness, Offset_time)

Esta función convierte un archivo de audio en formato **.wav** en una animación a base de Shapes.

Hay una diversidad de programas para convertir un archivo de vídeo a **.wav**, lo mismo que archivos de audio. En lo personal uso **Format Factory**:

Y una vez tengamos nuestro archivo de audio en formato **.wav**, lo guardamos en la misma carpeta en donde esté nuestro archivo **Effector-utils-lib-3.2.lua**

El parámetro **Audio** es el nombre entre comillas, simples o dobles, de nuestro archivo .**wav**

Width es el ancho de la **shape** que simulará la frecuencia del audio. Su valor por default es **line.width**

Height_scale es la escala vertical de la **shape**. Su valor por **default** es 1/220

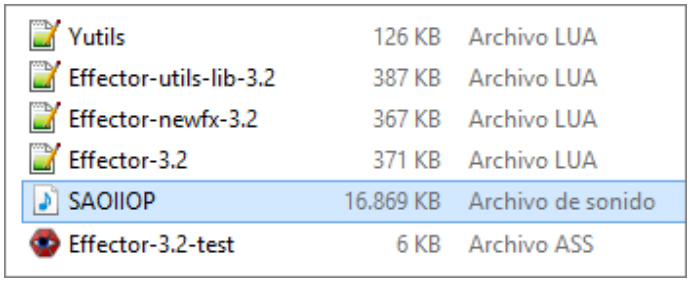
Thickness es el espesor de la **shape** y su valor por default es de 6 pixeles.

Offset_time es el tiempo adicional a la duración de la **shape** en pantalla, tanto al tiempo de inicio y final.

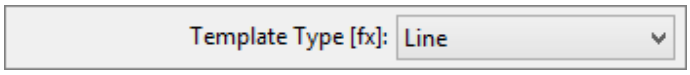
Esta función está pensada para ser usada en modo **Line**, es por ello que la duración en pantalla de las Shapes es la duración de la línea karaoke: **line.dur**

Ejemplo:

Como les mencionaba anteriormente, el archivo de audio de nuestro karaoke debe estar en la misma carpeta en la que se encuentra la librería principal del **Kara Effector**:



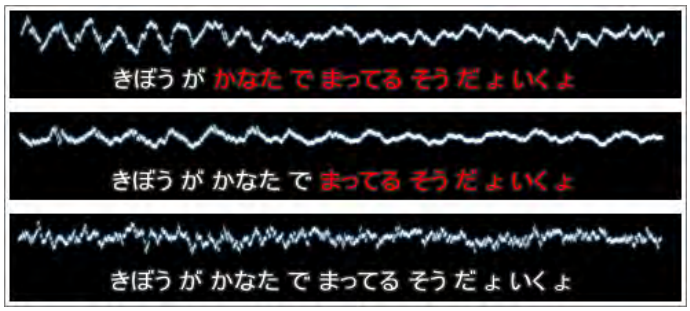
Ahora cambiamos el modo de nuestro fx a **Line**:



Return [fx]:

```
shape.from_audio( "SAOIIOP", l.width + 50, 1/160 )
```

Y ya en pantalla podemos ver cómo las Shapes hacen la animación de la frecuencia del audio de nuestro karaoke:



La implementación de esta función demanda cierta cantidad de recursos de nuestra PC, lo que hará que tarde cierto tiempo en aplicarse por completo. Esta función genera aproximadamente 120 líneas fx por cada línea karaoke y dependiendo de cada computadora, puede tardar desde 10 segundos hasta 2 minutos en aplicar completamente el efecto, así que no se preocupen si se tarda un poco en ello, ya que pueden ver el progreso de la aplicación al mismo tiempo que esperan que termine.

shape.bord(Shape, Bord, Size)

Esta función crea el borde de una shape ingresada con el tamaño del borde especificado y con el tamaño de la shape también seleccionado por el usuario.

El parámetro **Bord** es la medida en pixeles del tamaño del borde y su valor por default es 4px.

El parámetro **Size** es la medida en pixeles del tamaño de la **shape** que retornará la función y su valor por default son las dimensiones de la **shape** ingresada.

El parámetro **Size** puede ser o un número o una **tabla** con dos valores numéricos. Ejemplo:

Size = 120

Size = {80, 100}

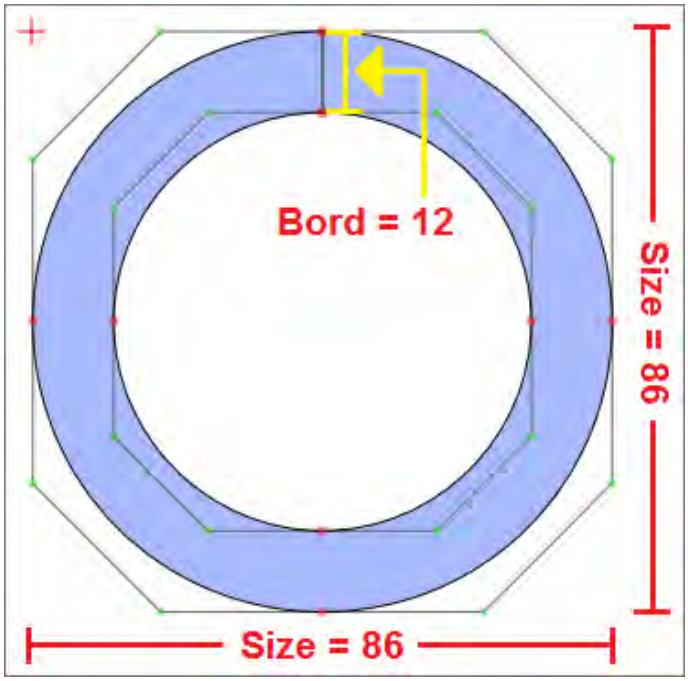
Para el caso cuando es un número, ambas dimensiones de la **shape** retornada, serán el mismo. Para el caso de ser una **tabla**, en ella podemos especificar el ancho y el alto a nuestro acomodo.

Ejemplo:

Return [fx]:

```
shape.bord( shape.circle, 12, 86 )
```

Lo que nos dará la **shape** del borde del círculo, de 86 px, tanto de ancho como de alto y con un borde de 12 px. Lo que en geometría se conoce como corona circular:

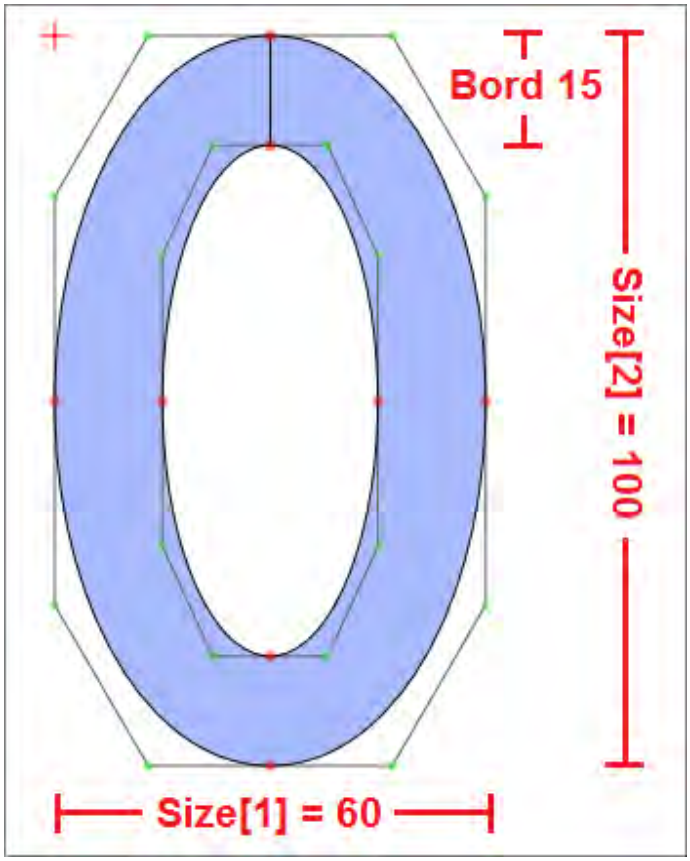


Ejemplo:

Return [fx]:

```
shape.bord( shape.circle, 15, {60, 100} )
```

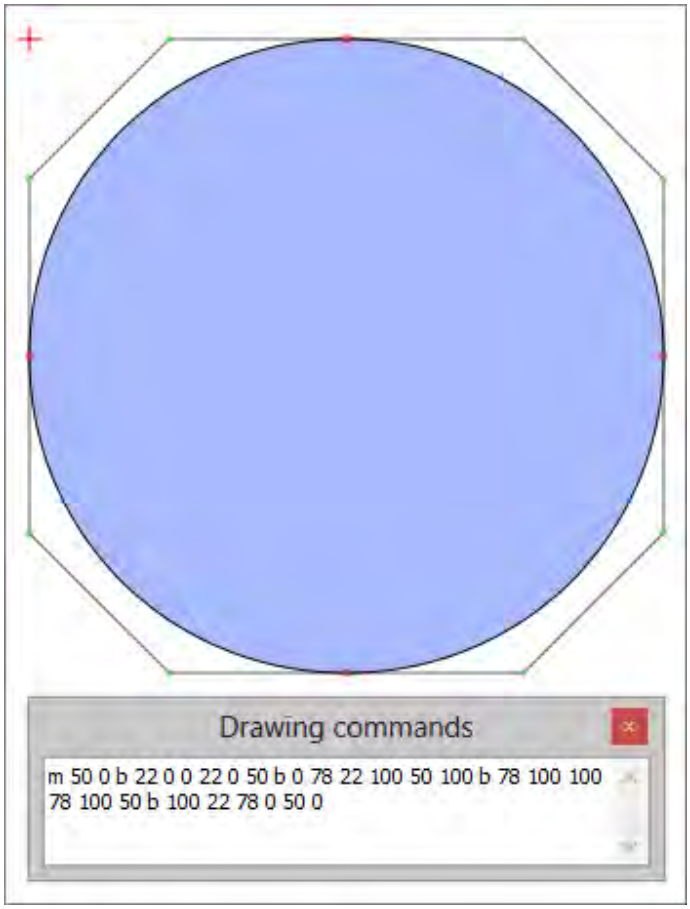
En este ejemplo, **Size** es una tabla, en donde el 60 indica el ancho de la **shape** y 100 será la altura medida también en pixeles:

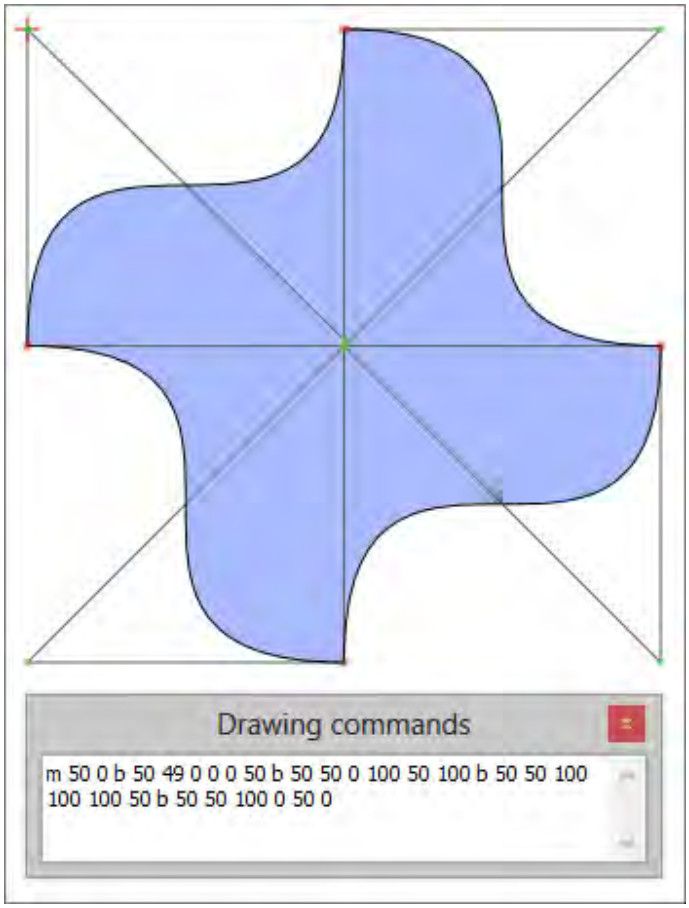


shape.morphism(Size, Shape1, Shape2)

Esta función retorna una tabla con la interpolación de las Shapes ingresadas. Los dos parámetros **Shape1** y **Shape2** deben cumplir con una muy especial particularidad para que la función cumpla con su objetivo, y es que una de esas Shapes debe ser una modificación en la posición de los puntos de la otra, es decir que no funcionará con dos Shapes que tengan distinta cantidad de puntos. En nuestro ejemplo usaremos las Shapes de nuestra derecha →

El parámetro **Size** debe ser un número entero mayor a 2 que indicará el tamaño de la tabla que se retornará y por ende indica la cantidad de Shapes que se crearán en la interpolación de una shape a la otra.





Ejemplo:

Definimos las dos Shapes anteriores en **Variables**:

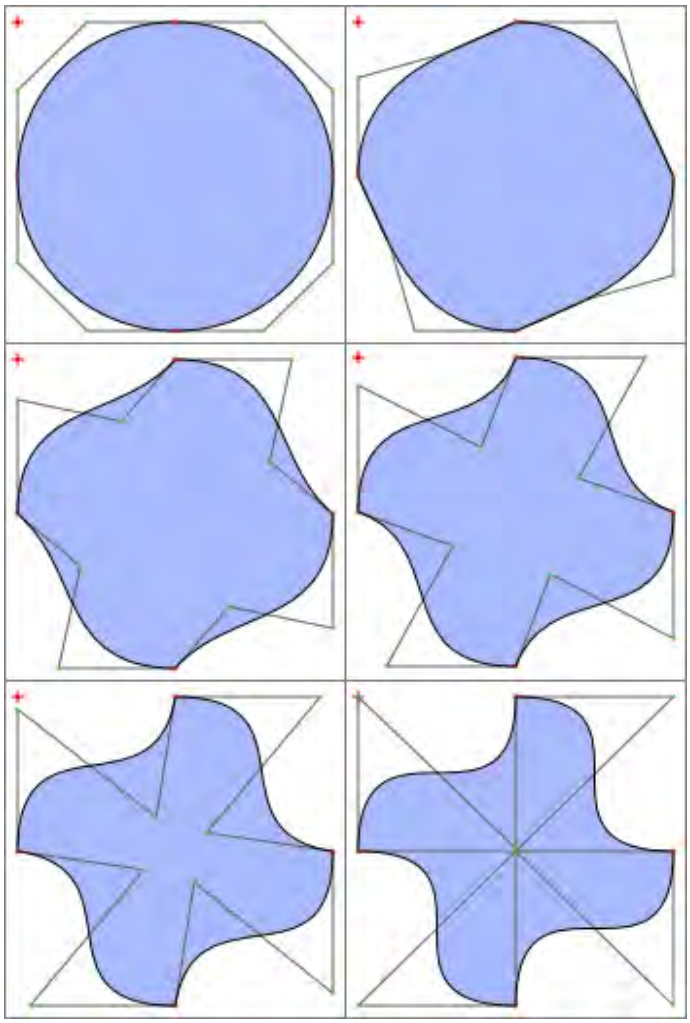
```
Variables:
mi_shape1 = "m 50 0 b 22 0 0 22 0 50 b 0 78 22 100 50 100 b 78 100 100 78 100 50 b 100 22 78 0 50 0 ";
mi_shape2 = "m 50 0 b 50 49 0 0 0 50 b 50 50 0 100 50 100 b 50 50 100 100 100 50 b 50 50 100 0 50 0 "
```

Como la función retorna una **tabla** de Shapes, no podemos hacer un ejemplo directo, ya que no la podríamos visualizar, entonces nos apoyamos de la función **table.show** para ver el contenido de la **tabla**:

Return [fx]:

```
table.show( shape.morphism( 6, mi_shape1,
                mi_shape2 ) )
```

Entonces la función hará en 6 Shapes, la transición de una a la otra:



Las Shapes contenidas en este tipo de tabla es ideal para ser usada en la función **shape.animated2**, para que la transición de una **shape** a la otra se vea una animación, una transformación de la una a la otra.

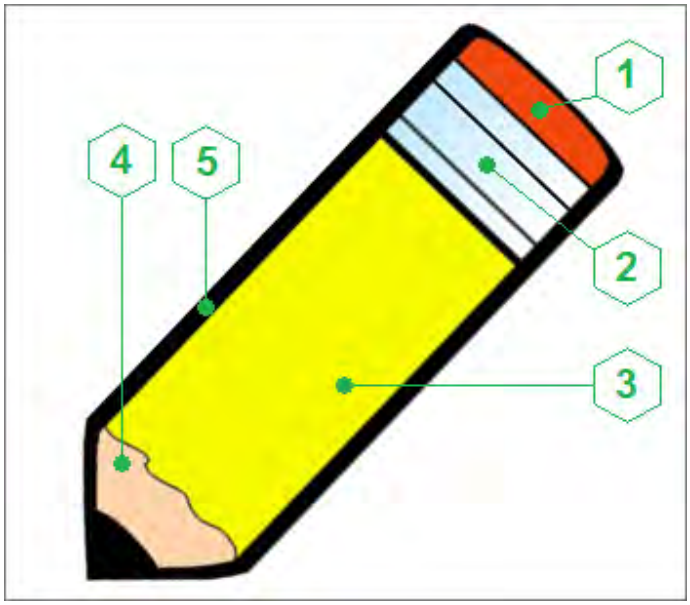
shape.divide(Shape, Mark)

Esta función divide las partes en las que se componen una shape más compleja, pera que sea más simple el darle los diferentes colores de la misma y que queden ubicadas en su posición de manera automática.

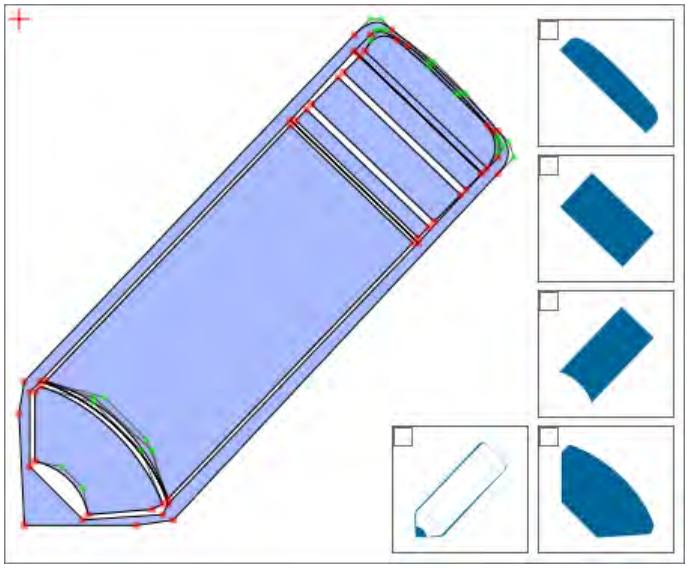
El parámetro **Mark** es opcional y hace referencia a 2 líneas paralelas que “enmarcan” a cada una de las Shapes que componen a la shape ingresada.

Ejemplo:

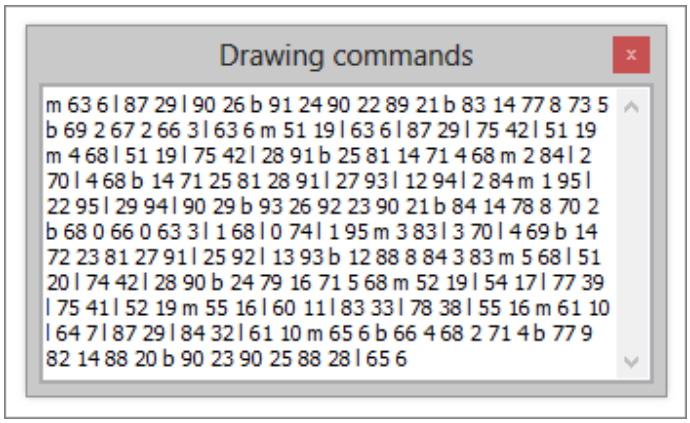
Para el siguiente ejemplo usaré la imagen de este lápiz que está compuesto por cinco Shapes individuales y tiene cinco colores diferentes:



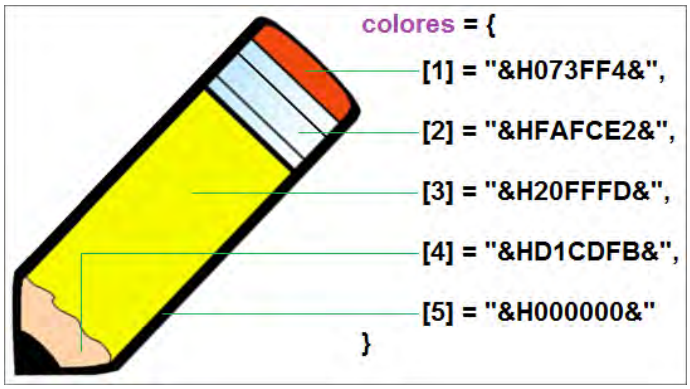
Dibujamos las cinco Shapes individuales en el mismo plano en el **ASSDraw3**. A la derecha de la imagen vemos lo que serían al dibujarlas en planos diferentes:



Este sería el código completo de toda la **shape** del lápiz al ser dibujado por partes, y con este código declaramos una variable: **mi_shape**

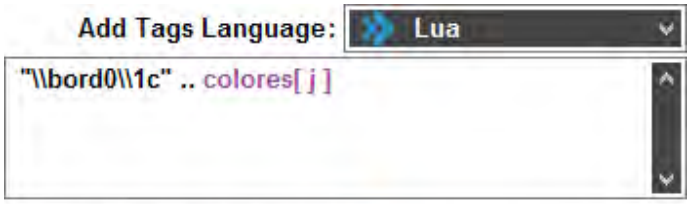


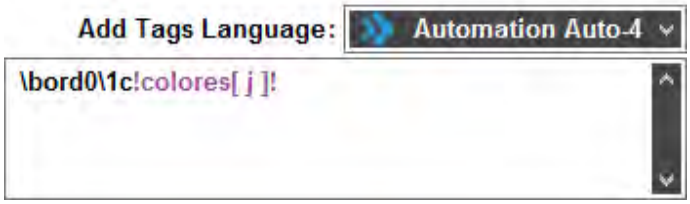
Junto con la variable de la **shape**, declaramos otra variable, una tabla que contenga los colores de cada una de las Shapes individuales, de tal manera que hagamos coincidir los colores con el orden en que dibujamos las Shapes:



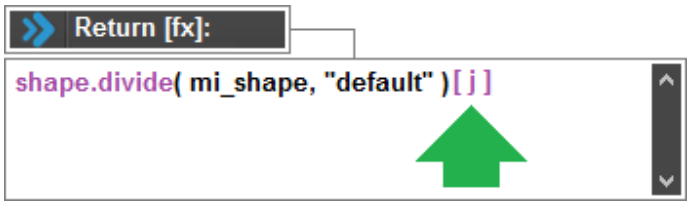
El siguiente paso es modificar el **loop** de nuestro efecto. El valor que debemos colocar será el de la cantidad de Shapes individuales que conforman a nuestra **shape**, en este caso será 5:

Hecho esto, asignamos los colores a las Shapes, y para ello ponemos lo siguiente en **Add Tags**. Recordemos las dos formas de hacerlo según el lenguaje:





Y por último, en **Return [fx]** llamamos a nuestra función, colocando la palabra "**default**" en el parámetro **Mark**:



Al aplicar, si seguimos los pasos correctamente, veremos cómo las 5 Shapes conforman al lápiz, con sus respectivos colores asignados:

Entonces, el parámetro **Mark** tiene las siguientes opciones:

- Si lo omitimos, la función no agregará nada a las Shapes individuales de la shape.
- Si ponemos la palabra "**default**", la función le pondrá dos líneas paralelas a cada una de las Shapes, con las dimensiones por default de la **shape** ingresada.
- Puede ser una tabla con cuatro valores numéricos, de manera que los dos primeros corresponden a la coordenada superior izquierda del marco y los dos siguientes, a la coordenada inferior derecha del mismo:

Es todo por ahora para el **Tomo XXIV**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXV]

Kara Effector 3.2:

En este **Tomo XXV** veremos una función más de la **librería shape** y haremos una corta pausa de la misma, ya que consideramos más importante ver otras funciones más de otras librerías que aún nos restan por ver, pero no se preocupen, hasta la fecha solo restan 3 funciones más de la **librería shape**, por documentar y ya llegará el momento de verlas con más detenimiento.

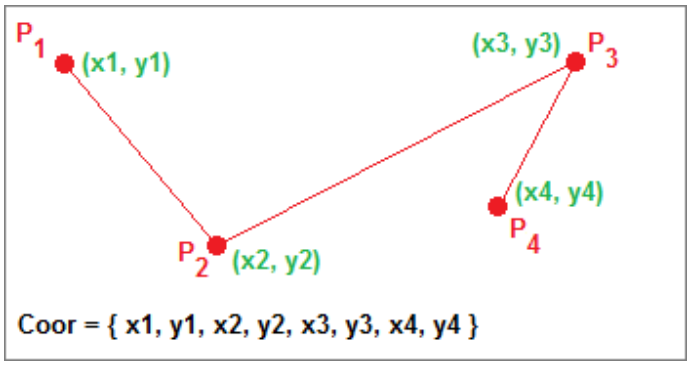
La nueva librería que empezaremos a ver a partir de este **Tomo XXV** es la **librería text**, que contiene una serie de funciones interesantes que sé que le sacarán el mayor provecho en sus proyectos.

Librería Shape [KE]:

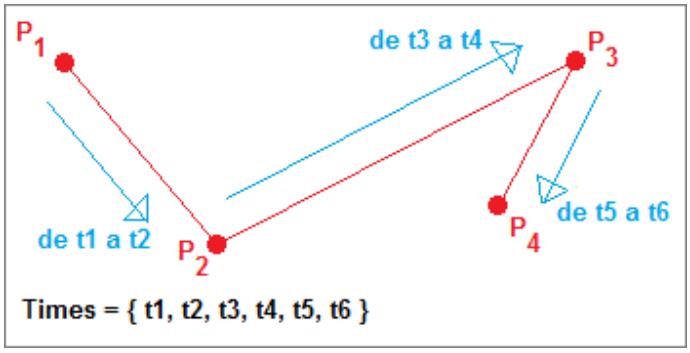
shape.Lmove2(Coor, Times)

Esta función es similar a **shape.Lmove**, pero con la diferencia que necesita dos tablas como parámetros para llevar a cabo su labor.

El parámetro **Coor** es la **tabla** de las coordenadas de los puntos en los que se desplazará nuestro objeto karaoke:



El parámetro **Times** es la **tabla** de los tiempos iniciales y finales de los movimientos entre punto y punto de la **tabla** del parámetro **Coor**:



Ejemplo:

Este será un sencillo ejemplo de su aplicación y puede ser usado en las líneas de traducción, de manera que usaremos el efecto con **Template Type: Translation Line**

Primero definimos nuestras dos **tablas** en la celda de texto "**Variables**", aunque este procedimiento no es del todo necesario, ya que podemos ingresar directamente las tablas en la función sin tener de definirlas, pero lo hago por simple organización:

Variables:

```
puntos = { line.center - 500, line.middle,
           line.center, line.middle,
           line.center + 500, line.middle };
tiempos = { 0, 200, line.dur - 200, line.dur}
```

Entonces tenemos:

- **P1 = line.center – 500, line.middle**
- **P2 = line.center, line.middle**
- **P3 = line.center + 500, line.middle**
- **t1 = 0**
- **t2 = 200**
- **t3 = line.dur – 200**
- **t4 = line.dur**

Y la función hará la siguiente:

- Moverá la línea de texto de P1 a P2 desde t1 a t2, o sea, desde los 0 ms hasta 200 ms.
- Desde t2 a t3, la línea de texto permanecerá estática, o sea que la línea no se moverá desde los 200 ms hasta line.dur – 200 ms.
- Moverá la línea de texto de P2 a P3 desde t3 a t4,o sea, desde line.dur – 200 ms hasta line.dur

Y para ello ponemos en **Add Tags** así:

Add Tags Language: Lua

```
shape.Lmove2( puntos, tiempos )
```

La cantidad de puntos que podemos ingresar en la función **shape.Lmove2** también es ilimitada, lo que conlleva a una cantidad ilimitada de movimientos lineales. Las aplicaciones son múltiples y todo dependerá de la imaginación.

La ventaja de esta función es que podemos hacer la cantidad de movimientos lineales que deseemos, y no necesariamente deben ser consecutivos ni durar la misma cantidad de tiempo, ya que controlamos los tiempos de cada desplazamiento, así como los lugares a los que se moverá.

Librería Text [KE]:

Esta librería, como su nombre lo indica, está enfocada en el texto de nuestros proyectos. Esta librería contiene una serie de funciones destinadas a la modificación, transformación y mejoramientos de los recursos de las líneas de texto.

text.upper(Text)

Esta función convierte el texto que le ingresemos en el parámetro **Text**, a mayúsculas.

El parámetro **Text** debe ser un string de texto, y su valor por default dependerá del **Template Type** seleccionado en el efecto que vayamos a aplicar:

Template Type [fx]	Texto por Default
Syl	
Line	line.text_stripped
Word	word.text
Syl	syl.text
Furi	furi.text
Char	char.text
Convert to Hiragana	hira.text
Convert to Katakana	kata.text
Convert to Romaji	roma.text
Translation Line	line.text_stripped
Translation Word	word.text
Translation Char	char.text
Template Line [Word]	line.text_stripped
Template Line [Syl]	line.text_stripped
Template Line [Char]	line.text_stripped

Ejemplo:

Template Type [fx]: Translation Line

Return [fx]:

```
text. upper( )
```

line.text_stripped por default

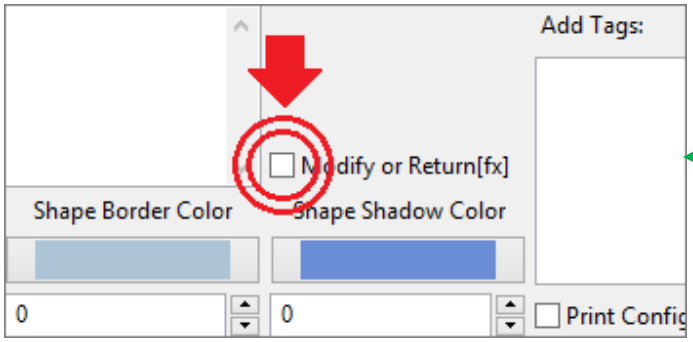
Y acá notamos cómo el texto se cambió todo a mayúsculas:

	Siento los pensamientos que dejaste atrás
	Me aferraré a ti y nunca te soltaré
	Dos corazones que se buscan conforman este sueño
[Fx]	✷MIS MEJILLAS SE MANCHAN CON LÁGRIMAS DE SOLEDAD
[Fx]	✷PERO PUEDO SENTIR LA LLEGADA DEL AMANECER
[Fx]	✷QUE ME ATRAE CON RUMBO HACIA LOS CIELOS
[Fx]	✷LA ESPERANZA ESPERA AL OTRO LADO, POR ESO VOLARÉ
[Fx]	✷ME PIERDO EN EL CAMINO CADA VEZ QUE TE BUSCO
[Fx]	✷SIENTO LOS PENSAMIENTOS QUE DEJASTE ATRÁS

Si ponemos algo distinto en el parámetro **Text**, la función lo transformará automáticamente en mayúsculas. Ejemplo:

- **text.upper("texto demo") → "TEXTO DEMO"**
- **text.upper("Ejemplo n° 2") → "EJEMPLO N° 2"**
- **text.upper("Aegisub") → "AEGISUB"**

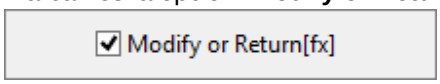
Otra forma de usar esta función es apoyándonos de esta herramienta del **Kara Effector**:



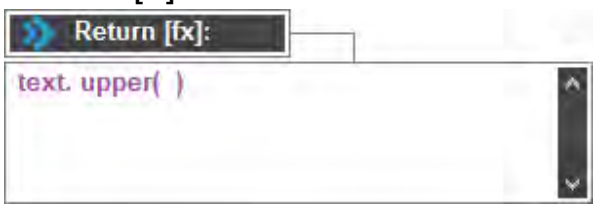
La opción "**Modify or Return[fx]**", al estar marcada, hace que el efecto en vez de generar nuevas líneas de fx, modifique a las líneas de nuestro script. Como su nombre lo indica, con esta opción decidimos si queremos modificar a nuestras líneas del archivo .ass o queremos generar un efecto a partir de líneas nuevas de fx.

Ejemplo:

- **Template Type**: Line o Translation Line
- Marcamos la opción "**Modify or Return[fx]**"



- En **Return [fx]** llamamos a nuestra función:



Y lo que sucederá es que no se generarán líneas de fx, sino que las líneas seleccionadas para que se le aplicará el efecto, se pasarán a mayúsculas:

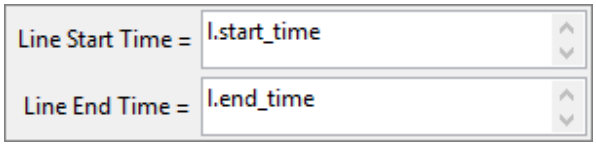
Estilo	Texto
Hiragana	✱す✱れ✱ち✱が✱う ✱い✱し✱き ✱て ✱が ✱ふ✱れ✱た ✱ょ ✱ね
Hiragana	✱つ✱か✱ま✱え✱る ✱ょ ✱し✱っ✱か✱り
Hiragana	✱も✱と✱め ✱あ✱う ✱こ✱こ✱ろ ✱そ✱れ ✱わ ✱ゅ✱め ✱の ✱あ
English	MIS MEJILLAS SE MANCHAN CON LÁGRIMAS DE SOLEDAD
English	PERO PUEDO SENTIR LA LLEGADA DEL AMANECER
English	QUE ME ATRAE CON RUMBO HACIA LOS CIELOS
English	LA ESPERANZA ESPERA AL OTRO LADO, POR ESO VOLARÉ
English	ME PIERDO EN EL CAMINO CADA VEZ QUE TE BUSCO
English	SIENTO LOS PENSAMIENTOS QUE DEJASTE ATRÁS
English	ME AFERRARÉ A TI Y NUNCA TE SOLTARÉ
English	DOS CORAZONES QUE SE BUSCAN CONFORMAN ESTE SUEÑ

Modify or Return [fx]

Esta opción del **Kara Effector** nos da la posibilidad de decidir si queremos modificar las líneas de nuestro script .ass o si queremos generar nuevas líneas fx.

Si marcamos esta opción, podemos modificar a nuestras líneas del script .ass de dos formas diferentes.

1. **Su contenido**:
   El contenido de las líneas se modificará solo con todo aquello que pongamos en la celda de texto **Return [fx].**
2. **Sus tiempos**:
   Podemos modificar los tiempos de inicio y final de las líneas de nuestro script .ass con las celdas de texto del **Kara Effector** destinadas para ello:



   Cualquier tiempo que añadamos o sustraigamos en estas dos celdas, modificarán los tiempos de las líneas del script .ass

Con esta opción marcada, solo funcionan estas tres celdas de texto de la **Ventana de Modificación** del **Kara Effector**, es decir que el resto de las herramientas en esta Ventana quedan inhabilitadas para modificar a la líneas de nuestro script .ass

text.lower(Text)

Es la función opuesta a **text.upper**, y convierte el string de texto ingresada en ella, a minúsculas.

Ejemplo:

- **text.lower("Texto Demo") → "texto demo"**
- **text.lower("EJEMPLO N° 2") → "ejemplo n° 2"**
- **text.lower("AEGISUB") → "aegisub"**

text.infx(select_fx)

Esta función aplica un efecto total o parcialmente, únicamente a las Sílabas previamente marcadas en nuestro script .ass

Hay dos formas diferentes con las que podemos marcar las Sílabas del script:

- **-fx**
- **+fx**

Estas marcas deben ir dentro de los tags de las Silabas.

La marca (**-fx**) selecciona únicamente a la sílaba que la contenga en su tag. Ejemplo

{\k19}ma{\k18-fx}yo{\k28}i {\k32}na{\k31-fx}ga{\k35-fx}ra

Sílabas marcadas:

- “yo”
- “ga”
- “ra”

La marca (**+fx**) selecciona a todas las Sílabas desde la marcada hasta el final de la línea karaoke, o hasta la siguiente marca (-fx) que se encuentre a su derecha:

{\k19}ma{\k18+fx}yo{\k28}i {\k32}na{\k31}ga{\k35}ra

De este modo, la marca (**+fx**) seleccionará a todas las Sílabas desde “yo” hasta la última, o sea la Sílaba “ra”.

{\k19}ma{\k18+fx}yo{\k28}i {\k32-fx}na{\k31}ga{\k35}ra

Hecho así, quedan seleccionadas las Sílabas desde “yo” hasta “na”.

{\k19}ma{\k18+fx}yo{\k28}i {\k32-fx}na{\k31}ga{\k35-fx }ra

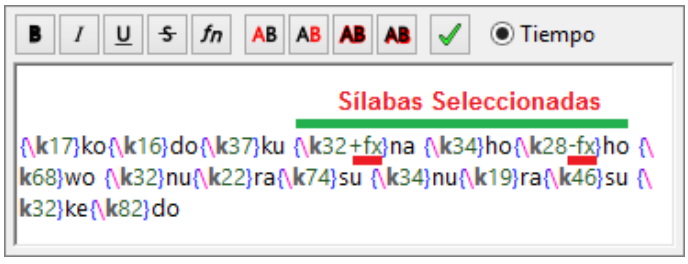
Y así, quedan seleccionadas las Sílabas desde “yo” hasta “na” y la Sílaba “ra”.

Una vez entendida la forma en las que podemos marcar las Sílabas de las líneas karaoke, veremos varios ejemplos de cómo usar la función **text.infx**, ya que tiene tres diferentes modos de uso.

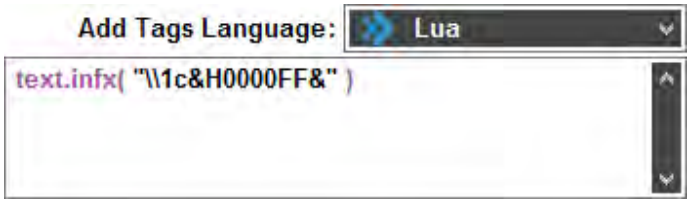
- **Modo 1**: aplicar un efecto parcial. Un **string**:

Ejemplo:

Marcamos algunas sílabas, no importa de cuál o de cuántas líneas karaokes:



Y en **Add Tags** agregamos algún **string** en la función:



Entonces al aplicar el efecto, la función solo agregará el string ingresado únicamente a las Sílabas seleccionadas previamente:

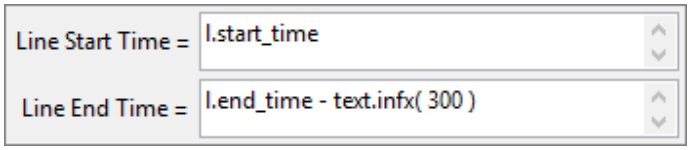
Notamos cómo el string (en este caso el color primario rojo) fue agregado solo a las Sílabas marcadas, a las demás no le agregó absolutamente nada.

- **Modo 2**: aplicar un efecto parcial. Un **número**.

Ejemplo:

Entonces la función sumará los 50 a los 100 que están antes de ella, solo a las Sílabas marcadas, a las restantes le sumará 0. O sea que la función, a las Sílabas que no están marcadas les agregará **“\\fscy100”** y a las que sí lo están, les agregará **“\\fscy150”**:

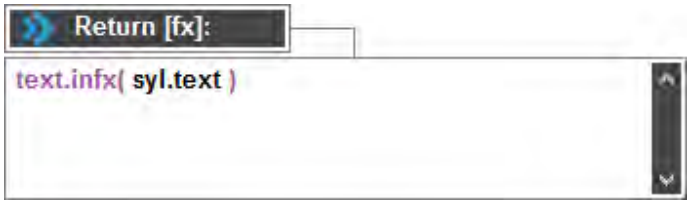
Con este modo de adicionar o sustraer un número con la función **text.infx**, también lo podemos hacer en las celdas de texto de tiempo. Ejemplo:



- **Modo 3**: aplicar un efecto total. Únicamente a las Sílabas marcadas:

Ejemplo:

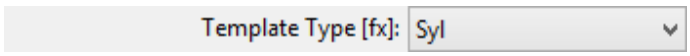
Para que un efecto se aplique únicamente a las Sílabas que previamente hemos marcado, lo que debemos hacer es usar la función en la calda de texto **Return** [fx], así:



Entonces el efecto que hayamos elegido solo se aplicará a la Sílabas marcadas, el resto no se tendrán en cuenta, y por ende no saldrán en pantalla:

En resumen, la función usada en cualquier celda de texto que no sea **Return [fx]** aplica el efecto de modo parcial a las sílabas marcadas, al resto solo se le aplicará el efecto general y no el que esté dentro de la función. Y al usar la función en **Return [fx]** como en el último ejemplo, el efecto general solo se aplicará a las Sílabas marcas y las demás no se tendrán en cuenta.

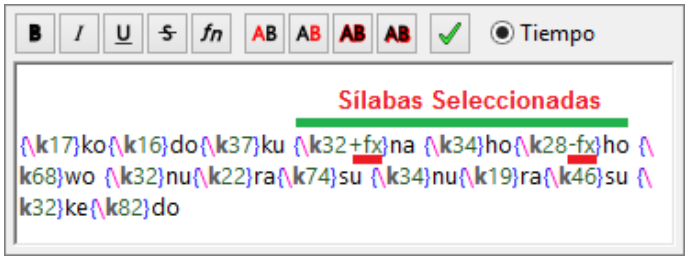
Solo resta decir que esta función, dado que marcamos las Sílabas, está diseñada únicamente para ser usada en el modo **Template Type: Syl**



text.outfx(select_fx)

Esta es la función opuesta a **text.infx**, es decir que nunca toma en cuenta a las Sílabas marcadas con (**+fx**) o con (**-fx**).

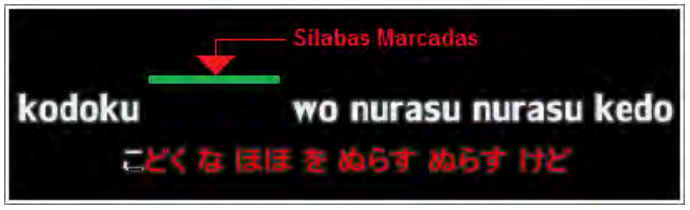
Ejemplo:



Return [fx]:

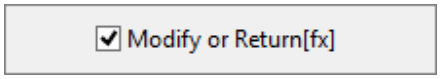
```
text.outfx( syl.text )
```

Y vemos el resultado opuesto de la función **text.infx**, es decir que no toma en cuenta a las Sílabas marcadas:



text.kara( )

Esta función está diseñada para ser usada con la opción "**Modify or Return [fx]**" marcada, y dentro de la celda de texto **Return [fx]**.



Return [fx]:

```
text.kara( )
```

Y lo que hace esta función es convertir las líneas normales de nuestro script .ass a líneas karaoke:

Ejemplo:

Convierte estas líneas:

English	Mis mejillas se manchan con lágrimas de soledad
English	Pero puedo sentir la llegada del amanecer
English	Que me atrae con rumbo hacia los cielos
English	La esperanza espera al otro lado, por eso volaré
English	Me pierdo en el camino cada vez que te busco
English	Siento los pensamientos que dejaste atrás
English	Me aferraré a ti y nunca te soltaré
English	Dos corazones que se buscan conforman este sueño

A líneas karaoke:

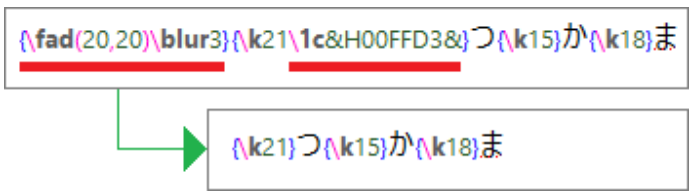
English	✱Mis ✱mejillas ✱se ✱manchan ✱con ✱lágrimas ✱de ✱soledad
English	✱Pero ✱puedo ✱sentir ✱la ✱llegada ✱del ✱amanecer
English	✱Que ✱me ✱atrae ✱con ✱rumbo ✱hacia ✱los ✱cielos
English	✱La ✱esperanza ✱espera ✱al ✱otro ✱lado, ✱por ✱eso ✱volaré
English	✱Me ✱pierdo ✱en ✱el ✱camino ✱cada ✱vez ✱que ✱te ✱busco
English	✱Siento ✱los ✱pensamientos ✱que ✱dejaste ✱atrás
English	✱Me ✱aferraré ✱a ✱ti ✱y ✱nunca ✱te ✱soltaré
English	✱Dos ✱corazones ✱que ✱se ✱buscan ✱conforman ✱este ✱sueño

```
{\k36}Me {\k145}aferraré {\k18}a {\k36}ti {\k18}y {\k91}nunca {\k36}te {\k127}soltaré
```

La función asigna un tiempo aproximado a cada palabra dependiendo de la cantidad de caracteres (letras) que tenga dicha palabra.

Y la otra función que cumple **text.kara( )** es remover todos los tags que no sean karaoke, de las líneas de karaoke:

Ejemplo:



text.tag(...)

Esta función agrega los tags que asignemos en sus parámetros, a cada uno de los caracteres (letras) del texto por default de un efecto.

Recordemos que el texto por default en un efecto depende del **Template Type**:

Template Type [fx]	Texto por Default
Syl	
Line	**line.text_stripped**
Word	**word.text**
Syl	**syl.text**
Furi	**furi.text**
Char	**char.text**
Convert to Hiragana	**hira.text**
Convert to Katakana	**kata.text**
Convert to Romaji	**roma.text**
Translation Line	**line.text_stripped**
Translation Word	**word.text**
Translation Char	**char.text**
Template Line [Word]	**line.text_stripped**
Template Line [Syl]	**line.text_stripped**
Template Line [Char]	**line.text_stripped**

Yo, por lo general uso esta función en los modos **Line** y **Translation Line**. Ya que esta función agrega tags a cada letra, entonces no tiene mucho sentido usarla en los modos **Char** y **Translation Char**.

Esta función no surge ningún efecto en los tres modos de **Template Line** (**[Word], [Syl] y [Char]**), y hay dos formas diferentes de agregar los tags a las letras:

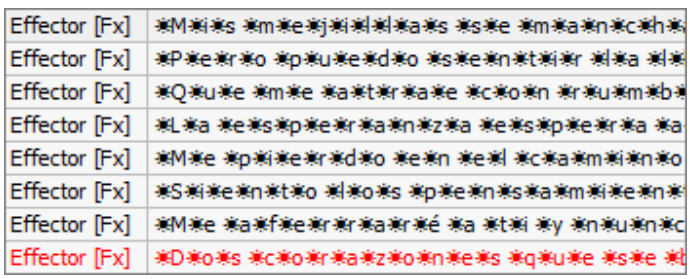
Ejemplo:

Template Type [fx]: Line

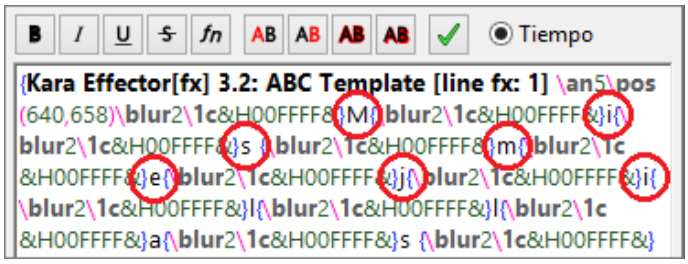
Return [fx]:

```
text.tag( "\\blur2", "\\1c&H00FFFF&" )
```

Como el ejemplo está hecho en modo **Line**, entonces en cada línea fx va la línea completa, y al aplicar el efecto con la función, entonces se le agregarán los tags ingresados a cada una de las letras de la línea de texto, como se nota en la siguiente imagen:



Acá vemos cómo los tags ingresados están antes de cada letra de la línea de texto:



La segunda forma de agregarle los tags a las letras, con esta función, es la más interesante:

Ejemplo:

Template Type [fx]: Line

Return [fx]:

```
text.tag( {"\\fscy", 120, 200},
          {"\\1c", "&H00FFFF&", "&H0000FF&"} )
```

Lo que hacemos es que, si un parámetro de la función es una tabla, debemos poner en el primer elemento de dicha tabla, al tag que queremos usar seguido de los dos valores a interpolar, el inicial y el final:

Vemos cómo el tag \fscy va creciendo desde 120 hasta 200, de letra a letra, y cómo el color primario \1c va cambiando de amarillo a rojo.

También se pueden combinar las dos formas de agregar los tags, y la cantidad de tags que se pueden añadir a la función es ilimitada.

Es todo por ahora para el **Tomo XXV**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXVI]

Kara Effector 3.2:

En este **Tomo XXVI** continuaremos viendo las funciones de la **librería text** que empezamos en el Tomo pasado. Esta librería contiene una serie de funciones interesantes que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también en la edición de los subtítulos.

Librería Text [KE]:

text.to_shape(Text, Scale, Tags)

Esta función, como su nombre lo indica, convierte un string de texto a **shape**.

El parámetro **Text** es el string de texto que queremos convertir a **shape** y su valor por default es el texto por default dependiendo el **Template Type**, como ya lo hemos visto en las anteriores funciones.

El parámetro **Scale** es un número entero positivo que hace referencia a las escala de la **shape** en la que se convertirá el texto. Las opciones recomendadas son:

- 1
- 2
- 4
- 8
- 16

De hecho, la escala puede ser cualquier otra potencia de 2, como 32, 64, 128 o superiores, pero eso hará que el texto se convierta en una **shape** demasiado grande, lo que generará un código muy extenso por línea, lo que hará que el script quedé muy pesado. Su valor por default es 1.

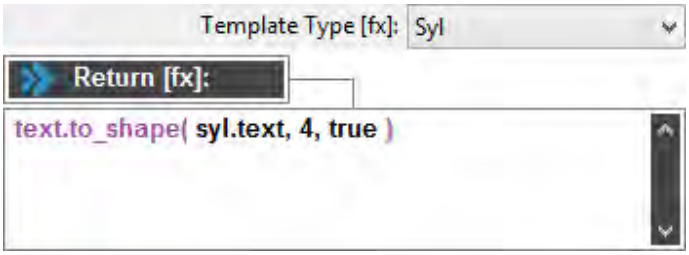
El parámetro **Tags** que añade 3 tags a la **shape** generada:

- **\an** ← Alineación respecto a la línea
- **\pos** ← Posición en la línea
- **\p** ← Proporción de la **shape**

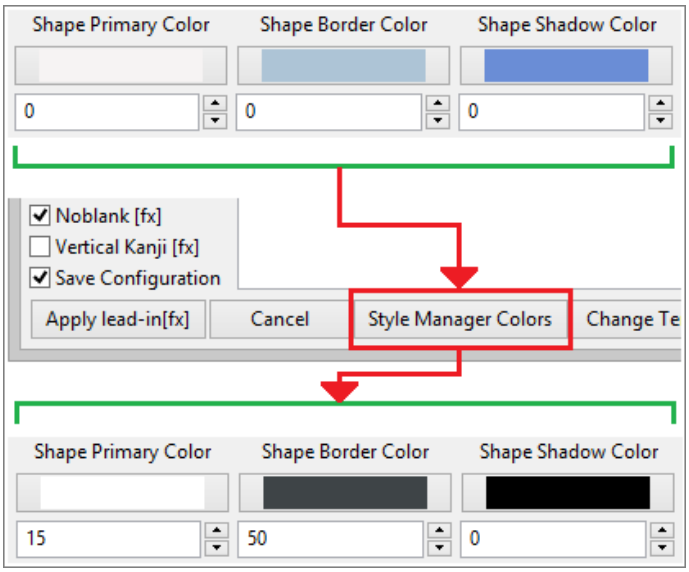
Estos tags se agregan a la **shape** con el fin ver en pantalla al texto convertido en **shape**, en su posición y alineación correcta sin necesidad de calcular otra cosa más en la **Ventana de Modificación** del **KE**. Este parámetro es un **boleano**, que de no estar en la función, se asumirá cono **nil** (nulo), o sea que no le añadirá ningún tag a la **shape** generada, es decir que **nil** es su valor por default. Y para añadir los tags mencionados, en el parámetro **Tags** debemos poner la palabra **true** (verdadero).

Ejemplo:

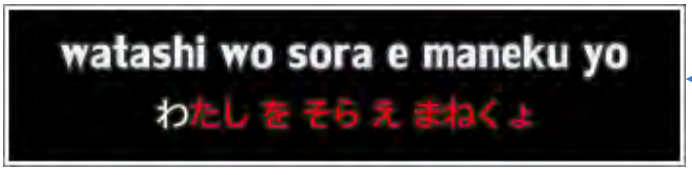
Llamamos a la función en **Return [fx]:**



Si aplicamos hasta este punto el ejemplo, el texto saldrá convertido en **shape** y por ende saldrá con los colores de las Shapes y no con los colores de los textos. Recordemos el procedimiento para convertir los colores a los asignados en el estilo:



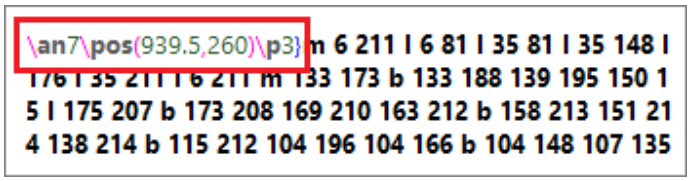
Ahora al aplicar, podremos ver el texto convertido a **shape**, pero con los colores del texto. En apariencia no se notan las diferencias entre el texto original y la shape a la que se transformó:



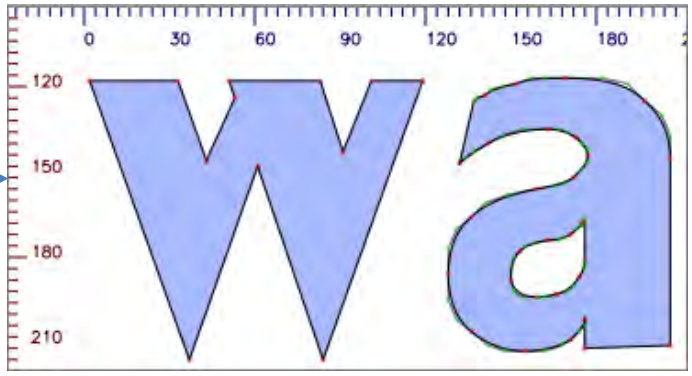
Al ver las líneas generadas, notamos que en lugar de cada Sílaba, está la **shape** de la misma:

	Dos corazones que se buscan conforman este sueño
Effector [Fx]	✱m 6 211 l 6 81 l 35 81 l 35 148 l 65 118 l 93 118 l 57 154
Effector [Fx]	✱m 85 81 l 85 211 l 56 211 l 56 206 b 54 207 52 209 49 2
Effector [Fx]	✱m 6 211 l 6 81 l 35 81 l 35 148 l 65 118 l 93 118 l 57 154
Effector [Fx]	✱m 84 144 l 84 211 l 62 211 l 62 150 b 62 146 61 143 58
Effector [Fx]	✱m 84 144 l 84 212 l 62 211 l 62 150 b 62 146 61 143 58
Effector [Fx]	✱m 84 144 l 84 212 l 62 211 l 62 150 b 62 146 61 143 58
Effector [Fx]	✱m 119 118 l 84 216 l 61 148 l 37 216 l 2 118 l 33 118 l 4

Y como pusimos "**true**" en el parámetro Tags, entonces la **shape** generada viene con los 3 tags que la posicionan:



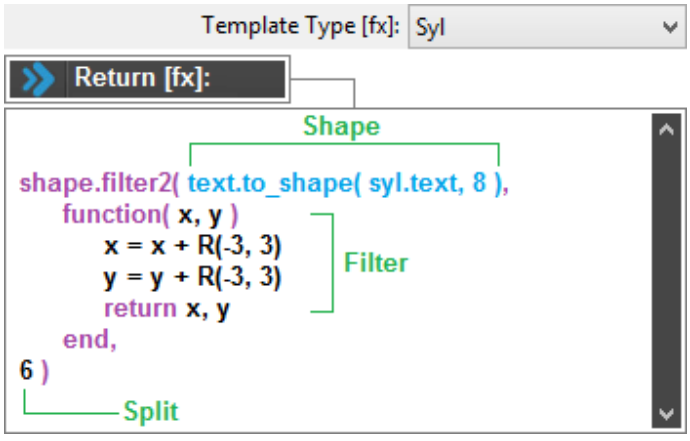
Al copiar el código de una de las Shapes generadas, y pegarlo en el **ASSDraw3**, veremos algo como esto:



La ventaja de poder convertir el texto en una **shape**, es que hecho una vez esto, le podemos aplicar cualquier función de la **librería shape**:

Ejemplo:

Para usar el texto como una shape convencional, debemos omitir al parámetro **Tags**, para que la función no retorne a los 3 tags junto con la **shape**:



shape.filter2:

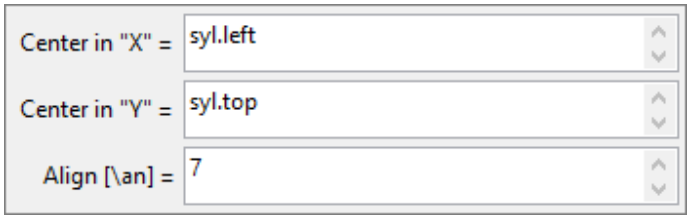
- **Shape**: el texto aumentado en un factor de 8.
- **Filter**: una función que hace que cada coordenada, tanto "x" como "y", queden desplazados de su posición original, por un valor aleatorio entre -3 y 3.
- **Split**: es el valor de la longitud de cada segmento recto en la que se dividirá la shape, en este caso al texto convertido en shape. Tiene en este ejemplo un valor de 6 px.

Y al aplicar sucederá lo siguiente:

El texto está convertido en una **shape**, que a su vez fue deformado por el filtro de la función **shape.filter2**, pero está 8 veces más grande de lo que debería estar y no tiene ni la alineación ni la posición correcta. Y para solucionar esto debemos hacer lo siguiente:

Ejemplo:

Modificamos los puntos de referencia a las posiciones **Left**, **Top** (Izquierda, Superior). Esta modificación depende del **Template Type**. Y cambiamos la alineación del texto a 7:



Ahora nos resta añadir el tag **\p** que es el que nos dará la proporción correcta de la shape para que tenga el tamaño real del texto, y lo debemos hacer así:

Return [fx]:

```
"{\\p4}" .. shape.filter2( text.to_shape( syl.text, 8 ),
        function( x, y )
            x = x + R(-3, 3)
            y = y + R(-3, 3)
            return x, y
        end,
    6 )
```

Y al aplicar veremos el texto deformado y en su posición y alineación correcta:

Y con un poco más de pixeles en la distorsión en el filtro:

En el anterior ejemplo vimos cómo el tag de proporción \p4 es el que le corresponde a una escala de 8 para el texto. La siguiente tabla nos mostrará la proporción correspondiente a cada una de las diferentes escalas:

Escala del Texto	Proporción \\p
1	1
2	2
4	3
8	4
16	5

Como las Shapes se modifican por medio de una función, de un filtro, las posibilidades son infinitas y todo dependerá de la creatividad de cada uno.

Para el siguiente ejemplo, ya no convertiremos a cada Sílaba una por una sino a la línea de texto completa, y para ello ya saber qué hacer, modificar el **Template Type**:

Ejemplo:

Recordemos que **minx** es el mínimo valor de “x” y que **w_shape** es el ancho medido en pixeles de la **shape**:

Template Type [fx]: Line

Return [fx]:

```
"{\\p4}" .. shape.filter2( text.to_shape( nil, 8 ),
        function( x, y )
            modx = (x - minx)/w_shape
            y = y + 60*sin(12*pi*modx)
            return x, y
        end,
    6 )
```

Al poner **nil** en el parámetro **Text** de **text.to_shape**, ésta lo asumirá por su valor por default, que en modo **Line** es: **line.text_stripped**

Y vemos cómo la línea de texto completa (en **shape**) se distorsiona gracias a la función del filtro:

text.do_shape(Text, Shape, Scale, Mode, Tags)

Esta función es algo similar a la anterior, y de hecho el principio es el mismo, ya que convierte el texto ingresado en **shape**. Esta función contiene dentro de sí a un filtro que hace que el texto, luego de haber sido convertido a **shape**, adopte la forma de otra **shape** ingresada en el parámetro **Shape**.

El parámetro **Text** es el texto para convertir a **shape** y su valor por default es el texto por default dependiendo el **Template Type**, como ya lo hemos visto en las anteriores funciones.

El parámetro **Shape** es la nueva figura que adoptará el texto una vez convertido en **shape**. Su valor por default es una **shape** recta equivalente a no aplicarle ningún filtro al texto, lo que en principio no tiene mucho sentido, ya que si usamos esta función es precisamente para convertir el texto en una **shape** nueva.

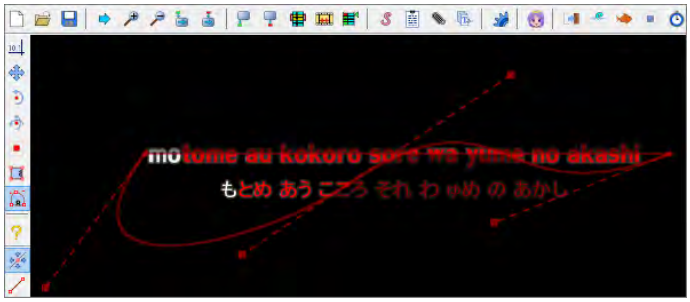
Los parámetros **Scale** y **Tags** cumplen exactamente la misma tarea que en la función anterior, que es la de darle proporción texto convertido en **shape** y añadirle los tags de posicionamiento al mismo.

El parámetro **Mode** es un número entero entre 1 y 4, para que la función haga la transformación en uno de sus cuatro modos habilitados. Su valor por default es 1.

Para los siguientes ejemplos, usaremos la función con un **Template Type: Line**, aunque con cualquiera es posible usar la función, ya que no importa el tipo de texto ingresado, ésta lo convertirá en shape.

Una forma sencilla de dibujar Shapes para esta función es usando la herramienta para recortar subtítulos en un área vectorial (**\clip**):

Y con solo dos bezier dibujé la siguiente curva usando dicha herramienta:



Este es el código de la anterior curva clip, el cual usaremos como **shape** para los siguientes ejemplos:

```
{\clip(m 204 298 b 31 532 375 474 558 352 845 160 815 418 1122 298)}
```

Y para mayor comodidad, definimos esta **shape** como una variable, que luego usaremos en la función:

Variables:

```
mi_shape = "m 204 298 b 31 532 375 474 558 352
           845 160 815 418 1122 298"
```

Ejemplo:

Usamos la función con **Mode** = 1

Template Type [fx]: Line

Return [fx]:

```
text.do_shape(
        line.text_stripped,     <-- Texto
        mi_shape,               <-- Shape
        8,                      <-- Escala
        1,                      <-- Modo
        true                    <-- Tags
)
```

Entonces la función convierte el texto a **shape** y luego lo obliga a adoptar la forma de la **shape** ingresada en el parámetro **Shape**:

Con **Mode** = 1, la función justifica al texto en el centro de la **shape**, de modo que queda repartido equitativamente, desde el centro hacia los extremos. El clip que dibujamos no se verá en la imagen, solo lo dejé para referencia.

Sin ese clip de referencia se debe ver así:

Con **Mode = 2**, el texto queda justificado a la derecha de la **shape** ingresada, según la longitud del mismo, que para este ejemplo sería: **line.width**

Con **Mode = 3**, el texto queda justificado a la izquierda de la **shape** ingresada:

Y con **Mode = 4**, el texto se deforma hasta alcanzar la longitud total de la **shape** ingresada:

Una quinta opción que tiene el parámetro **Mode** es la de ser una función filtro:

Ejemplo:

Template Type [fx]: Line

Return [fx]:

```
text.do_shape( nil, mi_shape, 8,
    function( x, y )
        y = y + R(-25, 25)
        return x, y
    end,
true )
```

Entonces la función asumirá la justificación de **Mode = 4**, y adicional a ello, aplicará el filtro a la **shape**:

Esta es otra función ideal para hacer algunos carteles pocos convencionales, como aquellos que tienen forma de arco:

O también para hacer logos que contengan texto que no esté en línea recta. Ejemplo:

Es todo por ahora para el **Tomo XXVI**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2: Effector Book Vol. II [Tomo XXVII]

Kara Effector 3.2:

En este **Tomo XXVII** continuaremos viendo las funciones de esta interesante **librería text**. Esta librería contiene una serie de funciones interesantes que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también en la edición de los subtítulos.

Librería Text [KE]:

text.bord_to_shape(Text, Scale, Tags, Bord)

Esta función convierte el borde del texto ingresado y lo convierte en una **shape**.

El parámetro **Text** es el string de texto al que la función le convertirá su borde en una **shape**, y su valor por default es el texto por default según el **Template Type**.

Los parámetros **Scale** y **Tags** cumplen con la misma tarea que en las dos funciones anteriores, darle proporción y agregar tags de posicionamiento. Sus valores por default son los mismos, es decir:

- **Scale** = 1
- **Tags** = nil

El parámetro **Bord** es un número que indica el grosor del borde hecho shape, su valor por default es 2.

Ejemplo:

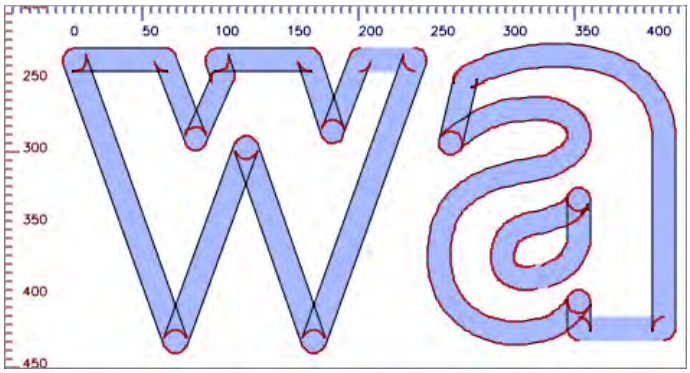
Template Type [fx]: Syl

Return [fx]:

```
text.bord_to_shape( syl.text, 8, true, 4 )
```

Al aplicar veremos algo como esto:

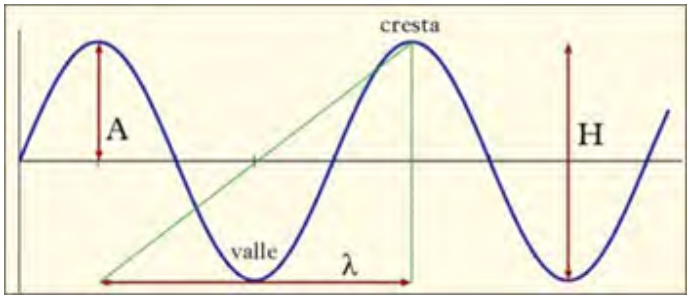
Una de las Sílabas vistas en el **ASSDraw3**:



Y el borde hecho **shape** ya queda apto para aplicarle la función de la librería **shape** que queramos.

text.deformed(Text, Deformed, Pixel, Axis)

Esta función convierte el texto ingresado en una **shape** y posteriormente lo deforma adaptando la forma de onda senoidal:



El parámetro **Deformed** es un número mayor que cero, que indica la cantidad de crestas y valles que tendrá la onda en la que se convertirá el texto. Su valor por default es 2.

El parámetro **Pixel** es la altura en pixeles de cada una de las crestas y valles que tendrá la onda, lo que en la imagen anterior corresponde a la letra "**A**". Su valor por default es **line.height**

El parámetro **Axis** tiene tres opciones:

- "**x**": dibuja la curva sobre el eje "x"
- "**y**": dibuja la curva sobre el eje "y"
- { **Deforme2**, **Pixel2** }: asume a **Deformed** y **Pixel** como valores en la deformación respecto al eje "x" y a **Deformed2** y **Pixel2** como valores para la deformación respecto al eje "y". O sea que de este modo se deforma el texto respecto a ambos ejes.

El valor por default del **Axis** es "x".

Para los siguiente ejemplos usaremos un **Template Type**: **Line**, aunque se puede usar esta función con cualquiera de los demás modos. Es solo que para fines ilustrativos es más visible ver la deformación sobre la línea de texto completa que en una letra o sílaba.

Ejemplo:

Template Type [fx]: Line

Return [fx]:

```
text.deformed( line.text_stripped, 8, 20, "x" )
```

Y al aplicar. Vemos cómo hay 8, entre crestas y valles, y la altura de cada una de ellas es de 20 px:

Ejemplo:

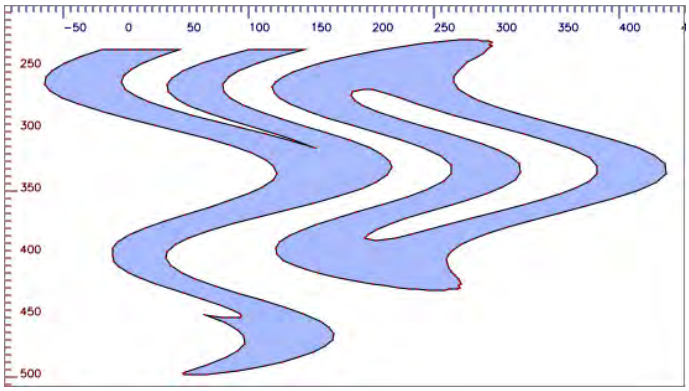
Template Type [fx]: Line

Return [fx]:

```
text.deformed( line.text_stripped, 5, 10, "y" )
```

Ahora el texto se deforma respecto al eje "y" con cinco, entre crestas y valles, y 10 px como su altura:

El poder controlar los valores en que se deformará el texto nos da muchas opciones y resultados. En la siguiente imagen vemos en el **ASSDraw3** la Sílaba "yo" deformada:



Ejemplo:

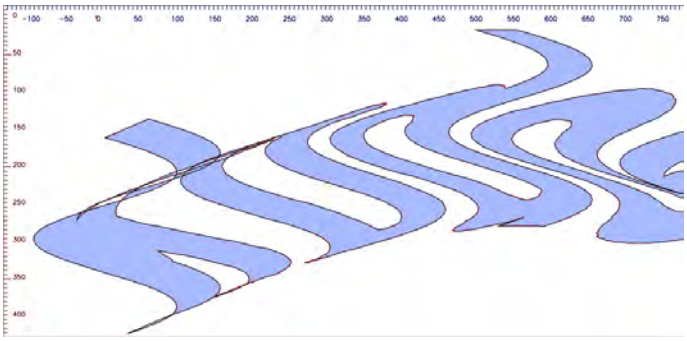
Template Type [fx]: Line

Return [fx]:

```
text.deformed( line.text_stripped, 6, 18, {3, 12} )
```

De este modo, se deforma el texto respecto a ambos ejes:

Visto en **ASSDraw3**, vemos un ejemplo de la deformación, en este caso pone: “kodo”



text.deformed2(Text, Mode)

Esta función retorna tres ejemplos más de cómo deformar un texto convertido a **shape**.

El parámetro **Text** es el string de texto a convertir a **shape** para deformar y su valor por default es el texto por default dependiendo del **Template Type**.

El parámetro **Mode** es un entero entre 1 y 3 que hará que la función retorne uno de los tres ejemplos de deformación. Su valor por default es 1.

Ejemplo:

- **Mode** = 1

Template Type [fx]: Syl

Return [fx]:

```
text.deformed2( syl.text, 1 )
```

Los puntos de la Sílaba convertida en **shape** se desplazan hasta el perímetro de un círculo de 80 px de diámetro:

- **Mode** = 2
  Los puntos de desplazan al perímetro de 3 círculos con diferentes diámetros:

- **Mode** = 3
  Los puntos de la **shape** se desplazan al perímetro de dos hexágonos concéntricos:

Las tres anteriores deformaciones son logradas gracias a las distintas funciones filtros que contienen cada una de ellos, es por eso que anteriormente les mencionaba que las posibilidades son infinitas a la hora de deformar una **shape** por medio de un filtro como función.

El **Mode** = 3 está basado en el siguiente filtro:

```
mi_filtro = function(x, y)
    local center_dx = minx + w_shape/2
    local center_dy = miny + h_shape/2
    local def_angle = math.angle(center_dx, center_dy, x, y)
    local def_angRE = (def_angle - 1)%60 + 1
    local def_ang3A = 180 - 60 - def_angRE
    local des_radiu = 200
    local des_dista = des_radiu*sin(rad(60))/sin(rad(def_ang3A))
    local des_radDE = (math.distance(center_dx,center_dy,x,y)<=80)
                       and des_dista/2 or des_dista
    x = center_dx + math.polar(def_angle, des_radDE, "x")
    y = center_dy + math.polar(def_angle, des_radDE, "y")
    return x, y
end
```

Este filtro nos ayudará a definir a una de las Shapes del siguiente ejemplo, también como variable. Y usaremos la función **shape.morphism** para ver a cada una de las Shapes, desde que el texto está normal hasta que se convierte en uno de los hexágonos del ejemplo anterior:

Ejemplo:

shape1 = shape.filter2(text.to_shape(“DE”, 8), nil, 6)

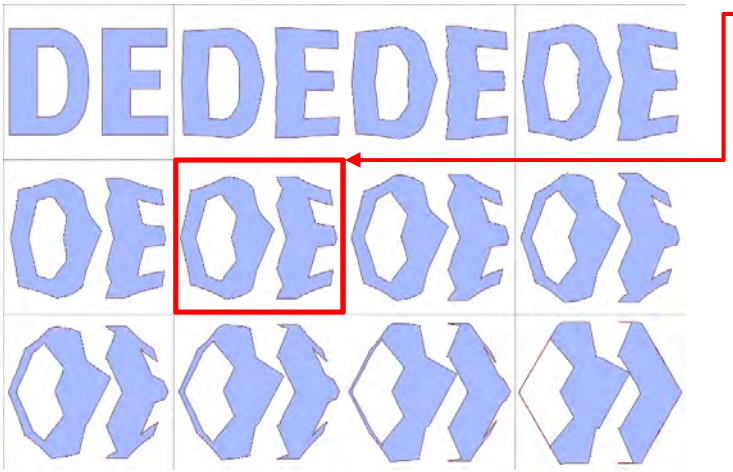
shape2 = shape.filter2(text.to_shape(“DE”, 8), mi_filtro, 6)

En la **shape1** el texto “DE” está normal, ya que en el filtro de la función pusimos “**nil**”. Y la **shape2**, como le pusimos el filtro del ejemplo anterior, entonces el texto “DE” ya está convertido en unos de los hexágonos vistos.

Ahora creamos la **tabla** con las interpolaciones entre una shape y la otra:

mi_tabla = **shape.morphism**(12, shape1, shape2)

Entonces veremos 12 Shapes de interpolación, de cómo el texto “DE” se transforma en uno de los hexágonos del ejemplo anterior:



Aplicando el anterior ejemplo, en un **Template Type: Syl**, cambiamos el texto “DE” por **syl.text**, así:

```
mi_filtro = function(x, y)
  local center_dx=minx + w_shape/2
  local center_dy=miny + h_shape/2
  local def_angle=math.angle(center_dx, center_dy, x, y)
  local def_angRE=(def_angle - 1)%60 + 1
  local def_ang3A=180 - 60 - def_angRE
  local des_radiu=200
  local des_dista=des_radiu*sin(rad(60))/sin(rad(def_ang3A))
  local des_radDE=(math.distance(center_dx,center_dy,x,y)<=80)
                  and des_dista/2 or des_dista
  x = center_dx + math.polar(def_angle, des_radDE, "x")
  y = center_dy + math.polar(def_angle, des_radDE, "y")
  return x, y
end;
shape_txt = text.to_shape( syl.text, 8 )
```

Y como vimos en las 12 Shapes de la transformación del texto “DE”, éste es legible como hasta la sexta o séptima **shape**. Sabido esto, adicionalmente hacemos lo siguiente:

Ejemplo:

Template Type [fx]: Syl

Center in "X" = syl.left

Center in "Y" = syl.top

Align [\an] = 7

Finalmente, ponemos esto en **Return [fx]:**

Return [fx]:

```
"{\\p4}" .. shape.morphism( 12,
    shape.filter2( shape_txt, nil, 6),
    shape.filter2( shape_txt, mi_filtro, 6)
)[6]
```

Y al aplicar veremos algo como esto:

Lo que hará que cada una de las Sílabas quede deformada de forma similar a la sexta **shape** del ejemplo anterior por haber puesto: **mi_tabla[6]**

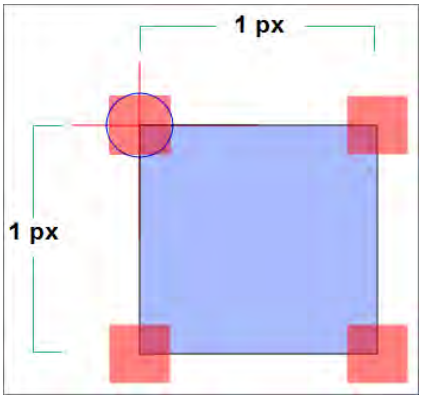
text.to_pixels(Text, Mode, Shape)

Esta función convierte al texto ingresado en pixeles. Esta función genera un loop equivalente a la cantidad de pixeles que tenga el texto ingresado, según los valores de escala y tamaño que tenga en el estilo del mismo.

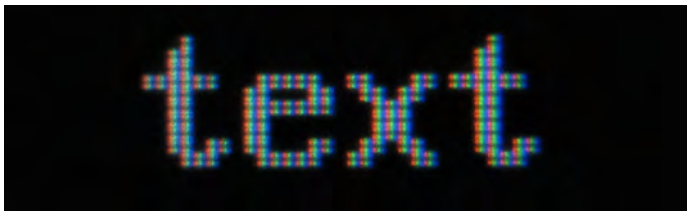
El parámetro **Text** es el texto a convertir en pixeles y su valor por default es el texto por default según el **Template Type**.

El parámetro **Mode** es un número entero entre 1 y 5 que hace referencia a las distintas posiciones y movimientos de cada uno de los pixeles que conforman el texto. Su valor por default es 1.

El parámetro **Shape** nos da la opción de que el texto no se convierta en pixeles (**shape.pixel**), sino en la **shape** que asignemos en este parámetro de la función. Su valor por default es **shape.pixel**:

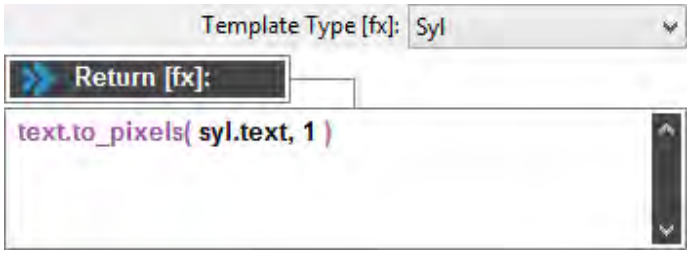


Lo que hace la función es, que por medio de Shapes de 1 x 1 px, remplazar el texto en su tamaño, forma y posición, por todos los pixeles que lo componen. Ejemplo:

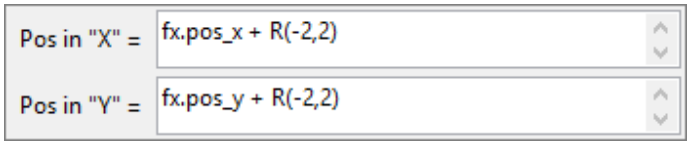


Ejemplo:

- **Mode = 1**



En **Modo = 1**, todos los pixeles que componen al texto salen en su posición exacta de tal manera que no se notará la diferencia entre el texto normal y el texto conformado por lo pixeles. Para notar a los pixeles que conforma al texto, añadí cierta cantidad de variación en la posición de los mismos, así:



Y obtendremos algo similar a esto:

Ejemplo:

- **Mode = 2**

Los pixeles se empiezan a desplazar desde el centro del texto, en todas direcciones, y conforman un círculo:

El tiempo total de este desplazamiento está determinado por **fx.dur**

Ejemplo:

- **Mode = 3**

Los pixeles se desplazan de forma muy similar a la anterior, pero ya no tienden a dibujar un círculo, sino una elipse vertical:

Ejemplo:

- **Mode = 4**

Los pixeles ahora se desplazan desde el centro del texto, describiendo tres elipses horizontales alineadas una arriba de la otra:

Ejemplo:

- **Mode = 5**

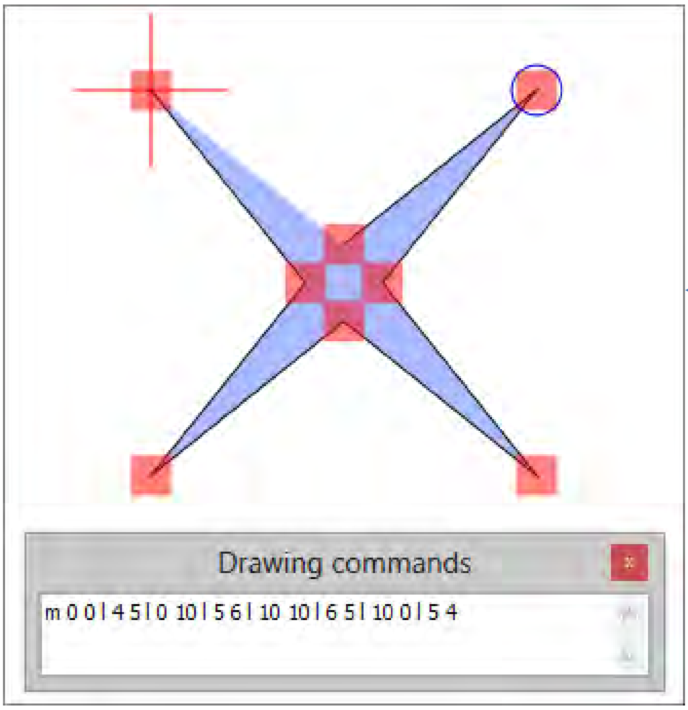
Los pixeles de desplazan describiendo una curva bezier seleccionada al azar:

A partir del modo 1, y con algo de imaginación, se pueden hacer muchas cosas con los pixeles de un texto que genera esta función. Recordemos que con solo variar el tiempo de inicio o el final de una línea fx, se puede modificar el **fx.dur**, lo que le dará una duración diferente a cada uno de los pixeles en el desplazamiento. Es cuestión de experimentar con todo aquello que hasta acá han aprendido.

Hasta este punto, el parámetro **Shape** siempre se usó por default, lo que hacía que la función siempre "pixelará" el texto con la shape cuadrada de 1 x 1 (**shape.pixel**):

Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0
Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0
Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0
Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0
Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0
Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0
Romaji	lead-in	Effector [Fx]	m 0 0 l 0 1 l 1 1 l 1 0 l 0 0

Para el siguiente ejemplo, usaremos esta **shape** en el tercer parámetro de la función, con el fin que el texto ingresado en ella se descomponga por medio de esta **shape**:



Ejemplo:

Y lo seguiremos haciendo en **Template Type: Syl**, aunque esta función puede aplicarse en cualquiera de los modos de fx a aplicar.

Elegí la función en **Mode = 2**, pero ustedes pueden practicar con cualquiera de los ya vistos hasta acá, y pegamos el código de la shape en el tercer parámetro de la función:

Template Type [fx]: Syl

Return [fx]:

```
text.to_pilxels( syl.text, 2, "m 0 0 l 4 5 l 0 10 l 5 6
                 l 10 10 l 6 5 l 10 0 l 5 4 " )
```

Y al aplicar, el texto se descompuso en esta nueva **shape**, lo que da un efecto similar a un brillo:

lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4
lead-in	Effector [Fx]	m 0 0 l 4 5 l 0 10 l 5 6 l 10 10 l 6 5 l 10 0 l 5 4

Este hecho de poder elegir a nuestro antojo a la **shape** en la que "pixelará" el texto ingresado en la función, aumenta en forma exponencial nuestras posibilidades de nuevos e ingeniosos efectos. Tengan en cuenta que esta función genera un loop aproximado entre 7000 y 10000 líneas de fx por cada línea de karaoke o de traducción, a la que se le aplique estos efectos.

Es todo por ahora para el **Tomo XXVII**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXVIII]

Kara Effector 3.2:

En este **Tomo XXVIII** continuaremos viendo las funciones de esta interesante **librería text**. Esta librería contiene una serie de funciones interesantes que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también en la edición de los subtítulos.

Librería Text [KE]:

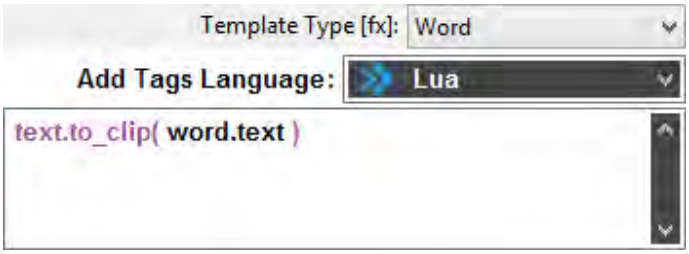
text.to_clip(Text, relative_pos, iclip)

Esta función retorna un clip con la forma exacta del texto ingresado. El parámetro **Text** es el string de texto a ser convertido en clip y su valor por default es el texto por default según el **Template Type** del efecto aplicado.

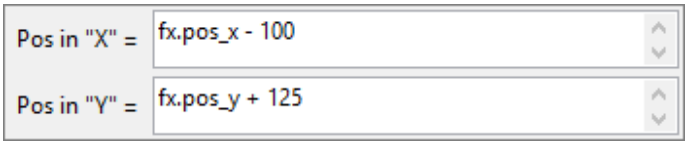
Los parámetros **relative_pos** e **iclip** son dos boleanos, el primero hace que el clip generado tenga la misma posición del texto original sin importar cuál sea su posición inicial y su valor por default es **nil**. El parámetro **iclip** decide si el texto ingresado se convertirá en un clip o en un iclip. Su valor por default es **nil**.

Ejemplo:

Para este ejemplo usamos un **Template Type: Word**



Y modificamos un poco las posiciones por default de cada Palabra generada por el efecto, por ejemplo así:

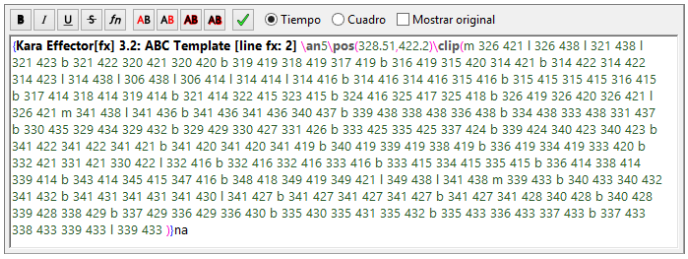


Es decir, 100 pixeles a la izquierda de su centro por default y 125 pixeles hacía abajo.

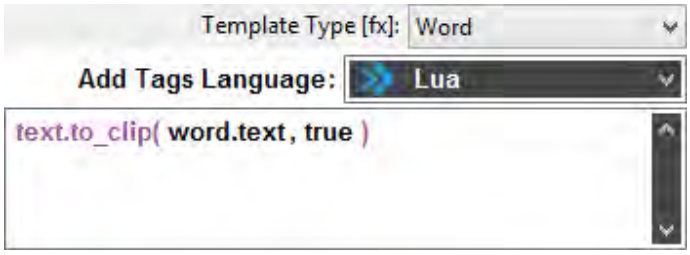
A aplicar, en apariencia solo veremos el texto en la posición que debería tener dada las modificaciones que previamente le indicamos.

Y vemos que los clip's son generados en la posición en la que se encuentra el texto sin importar que éste no esté en su posición por default: (**word.text, word.middle**)

Para poder apreciar el clip que genera esta función, es necesario que veamos en detalle una de las líneas fx generadas:



Si queremos que el clip generado tenga la posición por default del texto sin importar las modificaciones en las posiciones, ponemos true en el segundo parámetro de la función:



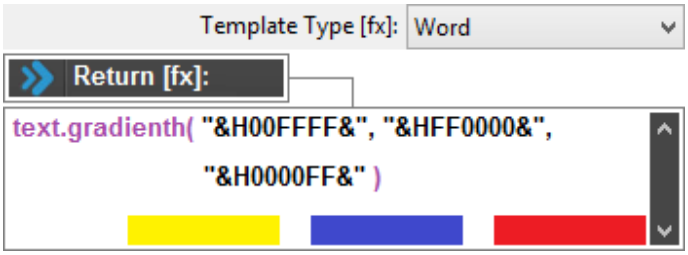
text.gradienth(...)

Crea un gradiente (degradado) horizontal entre todos los colores ingresados, sin la necesidad de usar el **VSFilterMod** y solo generando una única línea de fx.

La función convierte el texto por default según el **Template Type** del efecto, en un clip y luego genera dentro de él una serie de Shapes consecutivas de manera cada una tenga un color diferente y entre todas juntas conformen en gradiente horizontal.

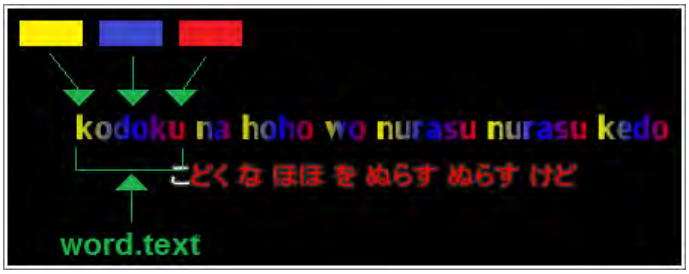
Ejemplo:

Para este ejemplo usaremos un **Template Type: Word**, y pondremos tres colores en la función, pero la cantidad total no tiene límites:



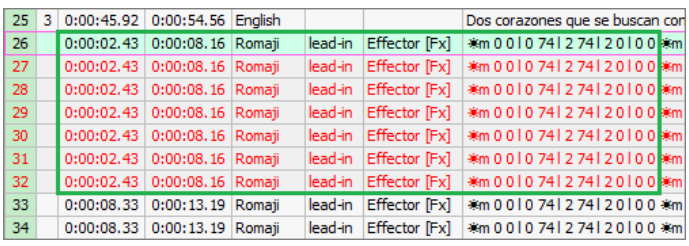
Recordemos que para crear un gradiente horizontal o vertical, la mínima cantidad de colores a ingresar en la función es de dos.

Al aplicar el ejemplo, notamos cómo cada palabra (Word) contiene un gradiente entre los tres colores ingresados:



Si ampliamos un poco la imagen veremos cómo la función crea el gradiente entre los tres colores, en cada palabra de la línea karaoke:

Una de las ventajas de esta función es que crea el gradiente horizontal entre todos los colores ingresados, generando una única línea de fx, en este ejemplo, por cada palabra de cada línea a la que se le aplica el efecto:



25	3	0:00:45.92	0:00:54.56	English			Dos corazones que se buscan cor
26		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
27		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
28		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
29		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
30		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
31		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
32		0:00:02.43	0:00:08.16	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
33		0:00:08.33	0:00:13.19	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m
34		0:00:08.33	0:00:13.19	Romaji	lead-in	Effector [Fx]	✱m 0 0 l 0 74 l 2 74 l 2 0 l 0 0 ✱m

Esta ventaja hace que el **script** sea más liviano y sea leído fácilmente, en el caso de los subtítulos en modo "**Softsub**".

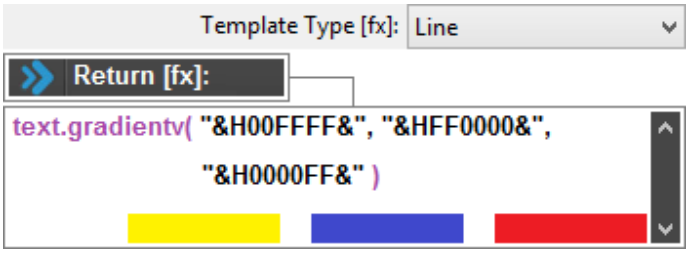
text.gradientv(...)

Crea un gradiente (degradado) vertical entre todos los colores ingresados, de manera similar a la función anterior, sin la necesidad de usar el **VSFilterMod** y solo generando una única línea de fx.

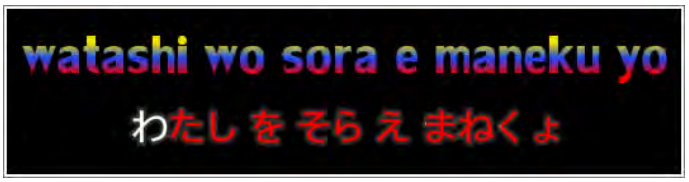
La función convierte el texto por default según el **Template Type** del efecto, en un clip y luego genera dentro de él una serie de Shapes consecutivas de manera cada una tenga un color diferente y entre todas juntas conformen en gradiente vertical.

Ejemplo:

Con un **Template Type: Line**, y en **Return [fx]** escribimos lo siguiente:



Y al aplicar:



Aunque aún restan varias funciones de la **librería text** por documentar, por ahora le daré prioridad a otras librerías, funciones y herramientas, que como todas las anteriores, nos ayudarán a sacarle un mayor provecho al **Kara Effector** y por ende a todos nuestros proyectos desarrollados en él.

Siguiendo con este orden de ideas, la siguiente librería en documentar será la **librería effector**.

Librería Effector [KE]:

Esta librería contiene un listado de funciones especiales, las cuales la mayoría son "**funciones administrativas**", ya que son funciones internas del **KE** que no generan efectos sino que hacen que él realice su tarea de forma correcta.

No vale la pena el entrar en detalle con las anteriormente mencionadas funciones administrativas, pero al igual las mencionaré. Por otra parte, hay unas cuantas funciones en esta librería que sí nos pueden ser de gran utilidad.

El listado de funciones que contiene la **librería effector** es el siguiente:

1. **effector.pos**
2. **effector.knj**
3. **effector.offset_pos**
4. **effector.import**
5. **effector.addfx**
6. **effector.savefx**
7. **effector.modify_pos**
8. **effector.new_pos**
9. **effector.default_val**
10. **effector.effect_offset**
11. **effector.decide**
12. **effector.print_error**
13. **effector.run_fx**
14. **effector.preprosses_styles**
15. **effector.preprosses_macro**
16. **effector.preprosses_lines**
17. **effector.macro_fx**

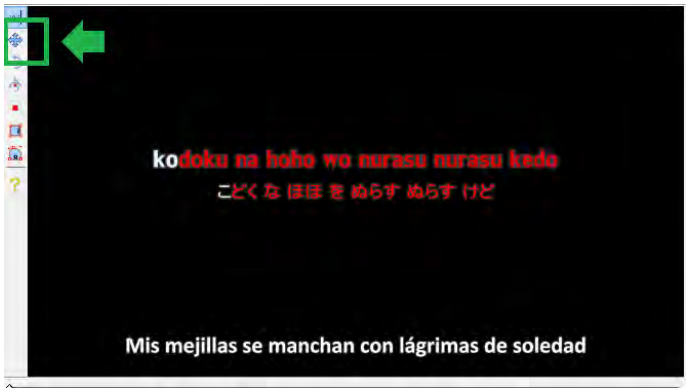
De las anteriores funciones, las que están en negro son las funciones administrativas, y las que están en azul son las que de una u otra manera le podemos sacar provecho.

effector.offset_pos( )

Es una función interna del **KE** que hace que las líneas de efecto generadas queden posicionadas de forma relativa a una posición previa hecha a las líneas del script en el vídeo.

Ejemplo:

Lo que debemos hacer es seleccionar la o las líneas que queremos reposicionar en el vídeo, para ello usaremos la herramienta para arrastrar los subtítulos del **Aegisub** que está en la esquina superior izquierda, al lado del vídeo:



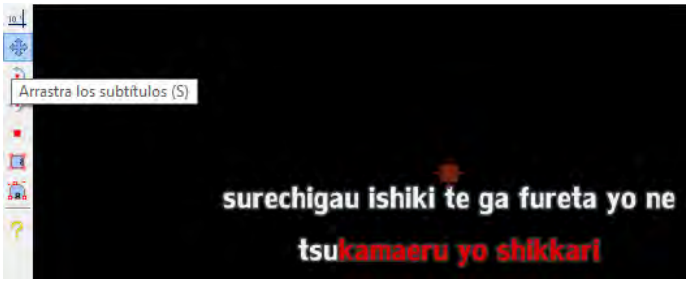
El procedimiento es simple, ya que una vez que pulsamos el botón para arrastrar los subtítulos, a cada elemento visible en el vídeo le sale un cuadrado con una cruz al centro que indica la posición y alineación del mismo, y con el clic derecho del mouse en él elegimos la nueva posición:

Una vez que hayamos desplazado la línea seleccionada a una nueva posición, la función **effector.offset_pos** hará internamente que esas coordenadas sean el punto de referencia al aplicar un efecto. Ejemplo:

Esta habilidad interna del **KE** es ideal, por ejemplo, para aquellas líneas que coinciden en tiempo y se superponen en el vídeo:

Ejemplo:

Las líneas tienen el mismo centro, y al coincidir en tiempo, se superponen:



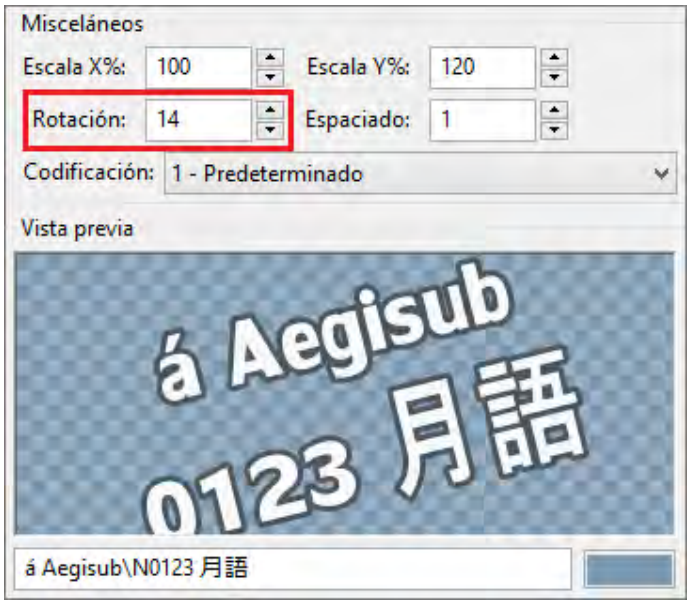
Entonces podemos mover las líneas de forma que ya no coincidan en su posición:

Y una vez que ya le hemos dado nuevas posiciones a las líneas en conflicto, ya podemos aplicar un efecto sin el temor de que las líneas de fx vayan a quedar superpuestas:

En resumen hasta acá, esta función hace que el **KE** redefina los centros que por default ya tiene cada línea del script dados por los valores de posición y alineación en el estilo de cada línea.

Otra habilidad que tiene esta función interna del **KE** es poder reposicionar a las líneas de karaoke previamente con la herramienta de rotación en los estilos de las mismas:

Ejemplo:



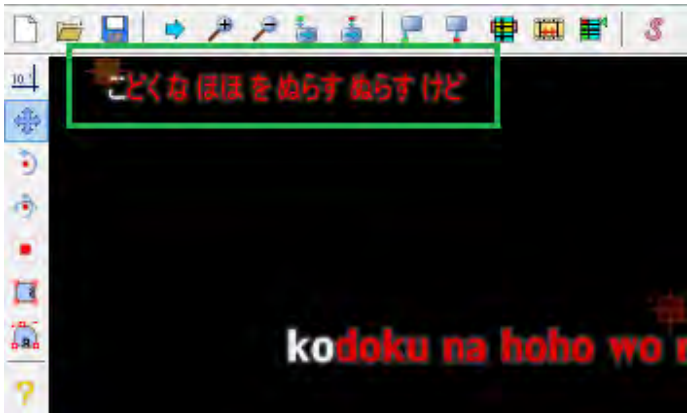
Y en vídeo veremos algo como esto:

Y al aplicar los efectos, el texto completo adoptará el ángulo de inclinación puesto en los estilos:

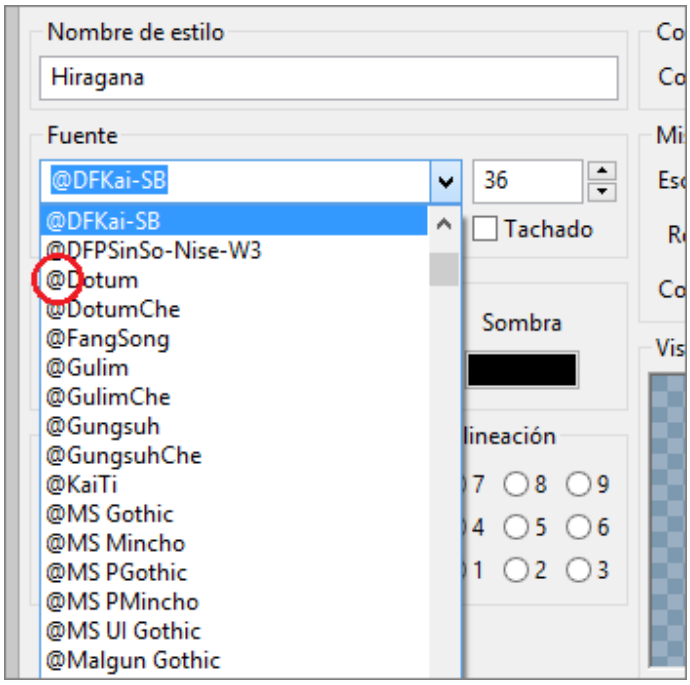
Una vez seleccionado el ángulo de inclinación de las líneas, podemos también modificar la posición en el vídeo con la herramienta de arrastre de subtítulos:

Y al aplicar los efectos seleccionados, la función reubicará automáticamente al texto para que las posiciones y ángulos de inclinación sean los mismos:

Otra de las utilidades de esta habilidad se usa para aplicar efectos a los **kanjis**, pero de forma vertical y utilizando las **fonts** que ya vienen prediseñadas para que el **kanji** se vea vertical con solo modificar la rotación del texto:

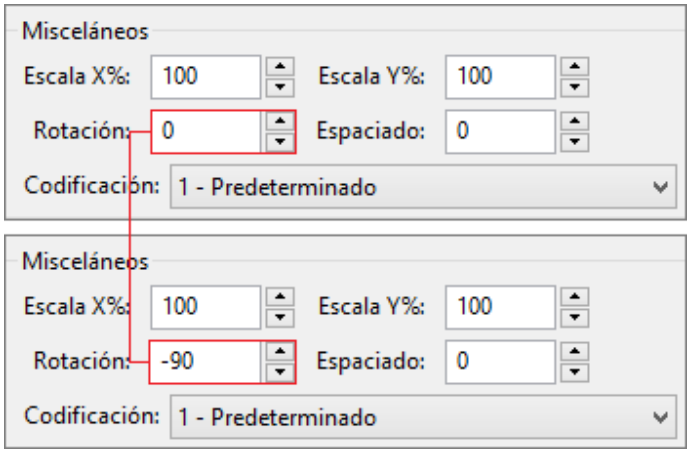


Las **fonts** que vienen prediseñadas para los **kanjis** en vertical, son la que en su nombre pone el símbolo de arroba (@) al inicio de su nombre:

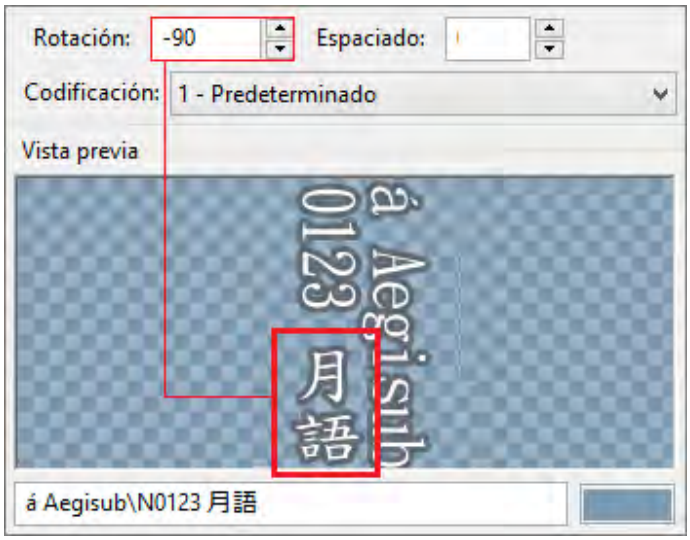


Seleccionamos una de esas **fonts** que más nos guste y se ajuste a nuestro proyecto, y vemos cómo el texto **kanji** está rotado en un ángulo de 90°:

Entonces modificamos la rotación del texto así:



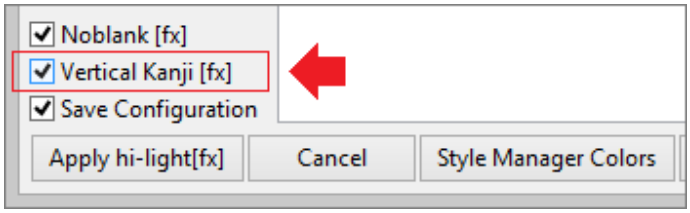
Ahora el **kanji** ya se ve bien en vertical aunque el resto del texto esté rotado -90°. Este hecho en particular no importa siempre y cuando en las líneas de texto solo haya **kanji**:



Hecho esto ya podemos ver en el vídeo cómo se ve el estilo del **kanji** en vertical sin aún haber aplicado ningún efecto:

Aplicando todos los efectos, la función hace los ajustes necesarios para que el texto **kanji** quede de forma vertical usando las **fonts** prediseñadas para ello:

La ventaja de esta habilidad es que hay un listado grande de **fonts** para **kanji** en vertical, y que de otro modo no podemos hacer con las **fonts** que son diseñadas pata texto horizontal. Recordemos que para aquellas fonts de texto horizontal usamos la opción de "**Vertical Kanji [fx]**" del **KE** y obtenemos los mismos resultados:



Es todo por ahora para el **Tomo XXVIII**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXIX]

Kara Effector 3.2:

En este **Tomo XXIX** continuaremos viendo las funciones de la **librería effector**. Esta librería contiene una serie de funciones interesantes que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también en la edición de los subtítulos.

Librería Effector [KE]:

1. **effector.pos**
2. **effector.knj**
3. **effector.offset_pos**
4. **effector.import**
5. **effector.addfx**
6. **effector.savefx**
7. **effector.modify_pos**
8. **effector.new_pos**
9. **effector.default_val**
10. **effector.effect_offset**
11. **effector.decide**
12. **effector.print_error**
13. **effector.run_fx**
14. **effector.preprosses_styles**
15. **effector.preprosses_macro**
16. **effector.preprosses_lines**
17. **effector.macro_fx**

De las anteriores funciones, las que están en negro son las funciones administrativas, y las que están en azul son las que de una u otra manera le podemos sacar provecho.

```
effector.import( file_fx )
```

Esta función nos permite importar un archivo .**lua** al **Kara Effector** para poder usar las funciones y variables que vienen en dicho archivo, al hacer uno o más efectos.

Esta función se usa en la celda de texto "**Variables**" y tiene una forma abreviada de llamarla:

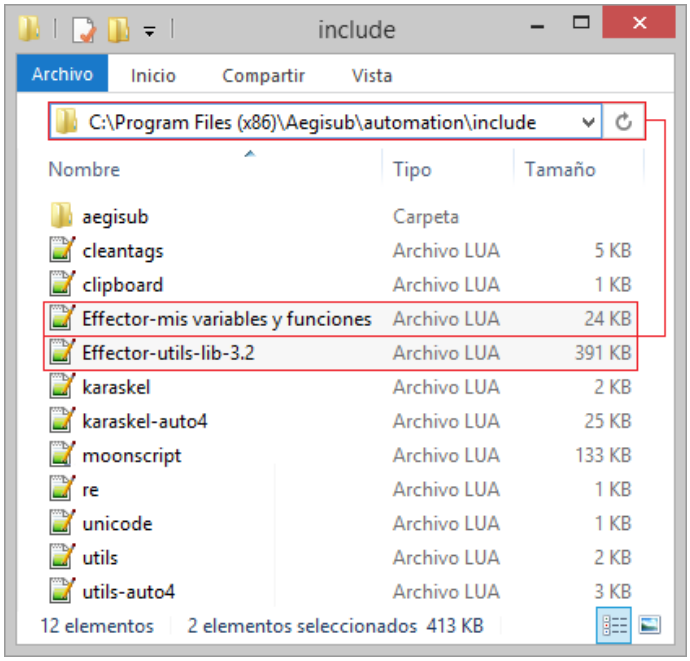
import(file_lua)

Hay dos formas de importar un archivo .**lua** al **KE**, una de ellas es guardar el archivo que queremos importar en la misma carpeta en donde esté guardado el archivo de la librería principal del **KE**: **Effector-utils-lib-3.2.lua**

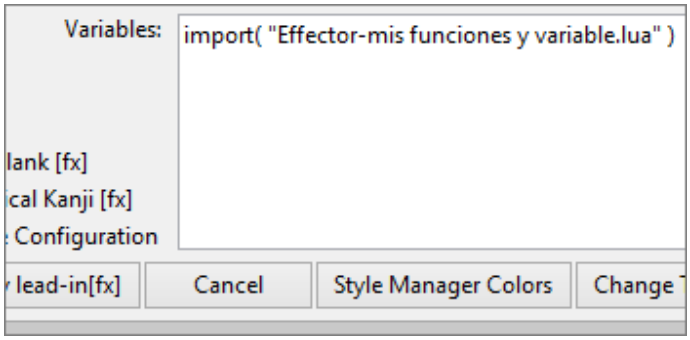
- **Método 1:**

El archivo **Effector-utils-lib-3.2.lua** del KE, la mayoría lo tenemos en la carpeta "**include**" del **Aegisub**, y es ahí en donde debemos guardar el archivo .**lua** que queremos importar al **Kara Effector**.

Ejemplo:

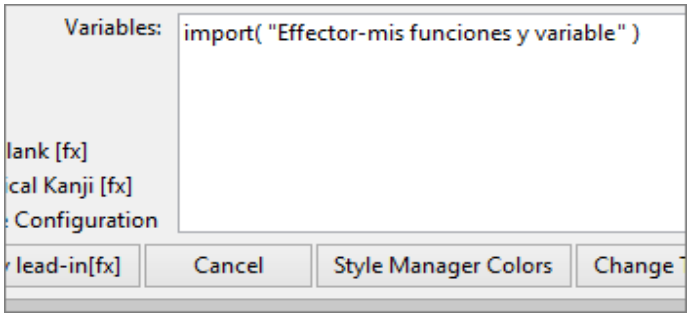


Y entonces, en la celda de textos "**Variables**" debemos poner así:



Obviamente no es necesario un nombre tan lago para el archivo .**lua** que vamos a importar, es solo un ejemplo.

Podemos omitir la extensión .**lua** del nombre del archivo y aun así poderlo importar satisfactoriamente:



- **Método 2:**

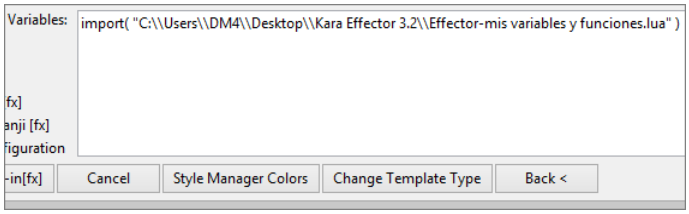
Con este método, no hay necesidad de guardar el archivo en la misma carpeta de la librería principal del **KE**, pero se debe poner la ruta completa de la posición del archivo dentro de la función:

Recordemos que las rutas deben ir con doble "\\".

Ejemplo:

import("C:\\Users\\DM4\\Desktop\\Kara Effector 3.2\\Effector-mis variables y funciones.lua")

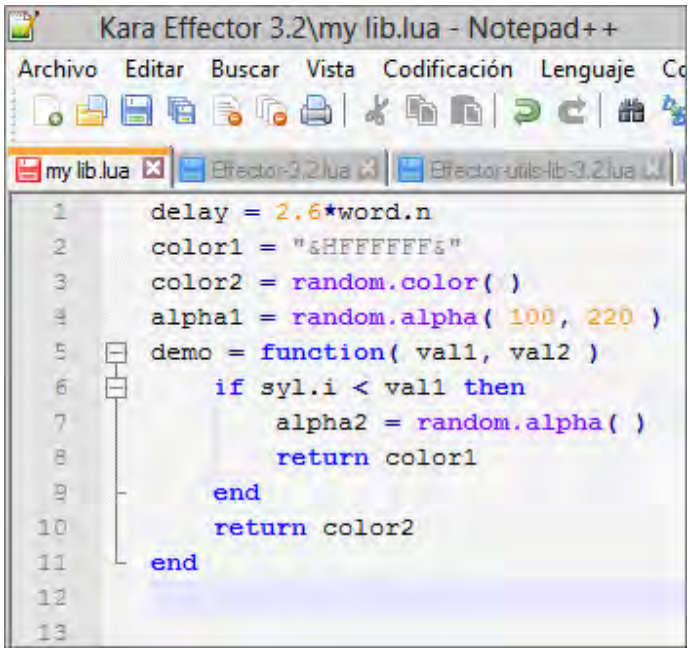
Y todo lo anterior debe ir en la celda de texto "**Variables**":



Una vez comprendido la forma de importar un archivo .**lua** al **KE**, ya podemos ver un ejemplo de cómo puede ser dicho archivo y las ventajas que tiene el poder importarlo.

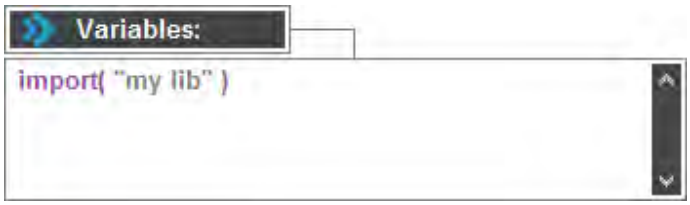
Ejemplo:

He creado un archivo llamado "**my lib.lua**" que contiene una serie de variables y una función sencilla, que he guardado en la misma carpeta en donde se encuentra la librería principal del KE:

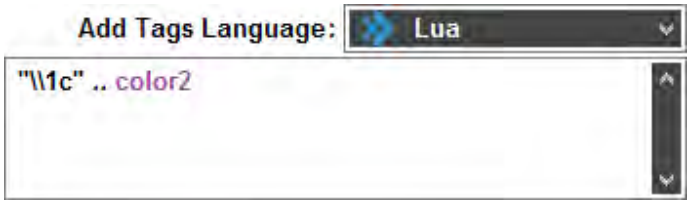


Obviamente al ser un archivo .**lua**, el leguaje a usar para declarar las variables y las funciones debe ser el **LUA**, lo que hasta este punto ya no debería ser un obstáculo tan difícil de superar como lo era al principio del aprendizaje.

Guardado el archivo, lo importamos al **KE** en la celda de texto "**Variables**" como acabamos de aprender:



Una vez importado el archivo .**lua**, ya podemos disponer de cualquiera de sus variables o funciones declaradas en él, en cualquier celda de texto de la Ventana de Modificación del **KE**, incluso en la misma celda "**Variables**". Ejemplo:



Y al aplicar el efecto, el **KE** reconocerá la variable "**color2**" que está en el archivo .**lua** creado, porque previamente lo ha importado:

El poder declarar variables y funciones en un archivo .**lua** independiente del **KE** nos da la ventaja de tener más espacio y comodidad para hacerlo. Otra de las ventajas es la siguiente:

- Se pude modificar el archivo .**lua**, y la modificación se asume en tiempo real en el efecto en que importemos el archivo, sin la necesidad de tener que recargar el **KE**.

Ejemplo:

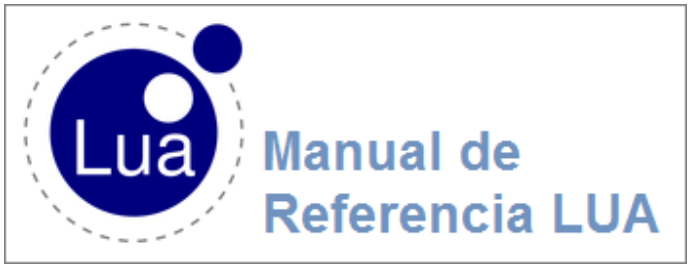
Modificamos la variable "**color2**":

```
1   delay = 2.6*word.n
2   color1 = "&HFFFFFF&"
3   color2 = "&HFFFFFF&"
4   alpha1 = random.alpha( 100, 220 )
5   demo = function( val1, val2 )
6       if syl.i < val1 then
7           alpha2 = random.alpha( )
8           return color1
9       end
10      return color2
11  end
12
13
```

Y sin tener que recargar el **KE**, aplicamos el efecto como en el ejemplo anterior, y la modificación se verá reflejada de forma inmediata:

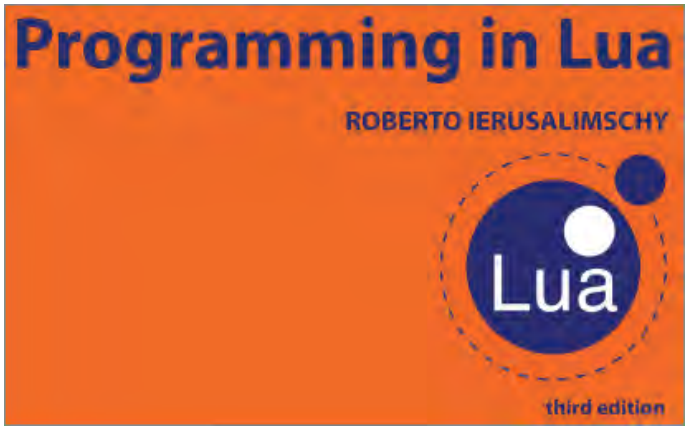
Saber y/o dominar el lenguaje de programación **LUA** no es un requisito para poder hacer efectos, ya que la razón del ser del **KE** es hacer sencillo el proceso de creación, edición y aplicación de los mismos. Para los que estén interesados en ampliar sus conocimientos en este lenguaje, recomiendo la siguiente web en español:

www.lua.org



O pueden descargar alguna de las ediciones en **PDF** del manual completo **LUA** en inglés:

- Programando en LUA 2da Edición
- Programando en LUA 3ra Edición



Al igual les dejaré ambos links en caso de que no funcionen los hipervínculos anteriores:

- 2da Edición: www.mediafire.com/?9aajra8ro6oukfa
- 3ra Edición: www.mediafire.com/?qf54ed3642rrn8o

Como les comentaba, este material es para aquellos que quieren profundizar en el lenguaje de programación **LUA** y aplicarlo en la creación de efectos, pero tengo pensado más adelante mostrarle unos cortos consejos de programación en **LUA** para hacer efectos en formato .ass

effector.addfx(library_fx, name_fx)

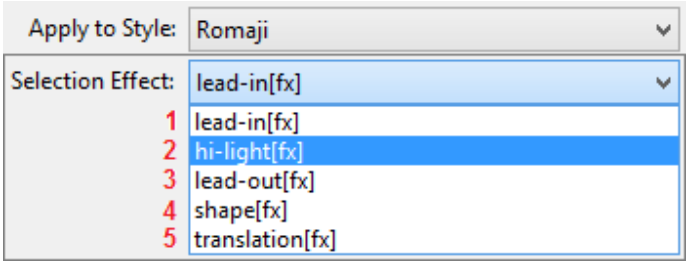
Esta función nos permite aplicar dos o más efectos al mismo tiempo, con la posibilidad de poder usar las variables y funciones declaradas en el primer efecto, en los demás adicionados. Su forma abreviada es:

addfx(library_fx, name_fx)

El parámetro **library_fx** es el nombre de alguna de las cinco librerías de efectos del **KE**:

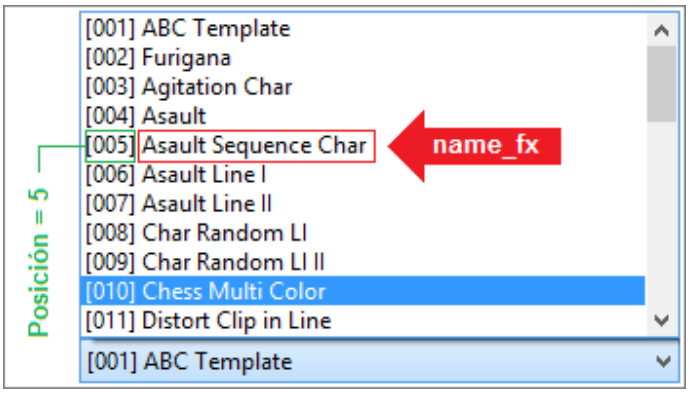
1. **"lead-in"**
2. **"hi-light"**
3. **"lead-out"**
4. **"shape"**
5. **"translation"**

Estas cinco librerías de efectos están en el mismo orden y hacen referencia a las opciones de la Ventana de Inicio del **Kara Effector**:



El parámetro **library_fx** puede ser el string del nombre de la librería de efectos o el número que indique su posición.

El parámetro **name_fx** es el string del nombre del efecto que queremos adicionar. Ejemplo:



Y de forma similar al parámetro **library_fx**, podemos poner el string del nombre del efecto a agregar o el número de la oposición que ocupa en su respectiva librería. Entonces podemos utilizar la función con cualquiera de las siguientes combinaciones:

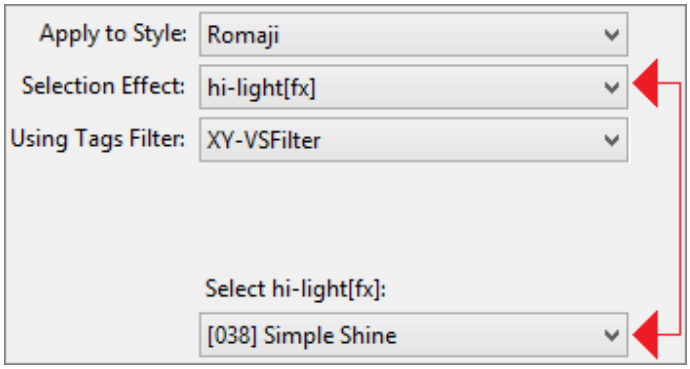
- **addfx(nombre_librería, nombre_efecto)**
- **addfx(nombre_librería, posición_efecto)**
- **addfx(posición_librería, nombre_efecto)**
- **addfx(posición_librería, posición_efecto)**

Esta función se usa en la celda de texto "**Variables**" y puede ser usada la cantidad de veces que así lo dispongamos.

Ejemplo:

Seleccionamos el primer efecto de la librería **lead-in[fx]** que es el efecto: **[001] ABC Template**

Y supongamos que queremos añadir a este lead-in el efecto "**Simple Shine**" de la librería hi-light[fx]:



Entonces tenemos la siguiente información del efecto que vamos a añadir:

- Librería: "**hi-light**"; Posición: 2
- Efecto: "**Simple Shine**"; Posición: 38

Y para añadirlo al lead-in[fx] "**ABC Template**", podemos hacerlo con algunas de las siguientes cuatro opciones en la celda de texto "**Variables**":

- **addfx("hi-light", "Simple Shine")**
- **addfx("hi-light", 38)**
- **addfx(2, "Simple Shine")**
- **addfx(2, 38)**

Ya sabemos que no importa cuál de las cuatro anteriores formas de hacerlo, la función añadirá el efecto al que ya hemos elegido, pero para el ejemplo usaré la primera, que es poniendo en nombre, tanto como el de la librería, como el del efecto:

Y al aplicar el lead-in, se aplicarán los dos efectos de forma simultánea, uno seguido del otro. Siempre se aplica primero el efecto de base, y luego todos los demás en el mismo orden en que los hayamos añadido:

Si queremos añadir más de un efecto a la vez, los debemos poner de la siguiente forma. Ejemplo:

Variables:

```
addfx( 2, 38 );  addfx( 4, 14 );  addfx( 3, 26 )
```

Lo que añadirá 3 efectos diferentes al que seleccionamos previamente en la Ventana de Inicio del **KE**.

El método para que una o más variables, al igual que las funciones declaradas en el primer efecto seleccionado en la Ventana de Inicio del **KE**, las podamos usar en los efectos a agregar, sin necesidad de tener que declararlas en ellos también, es el siguiente:

Ejemplo:

Variables:

```
delay = 420;  addfx( 2, 38 );  addfx( 4, 14 )
```

Al declarar la variable "**delay**" en este ejemplo, al inicio de la celda de texto "**Variables**", nos aseguramos de que esta también pueda ser usada en todos los efectos añadidos que así lo dispongamos.

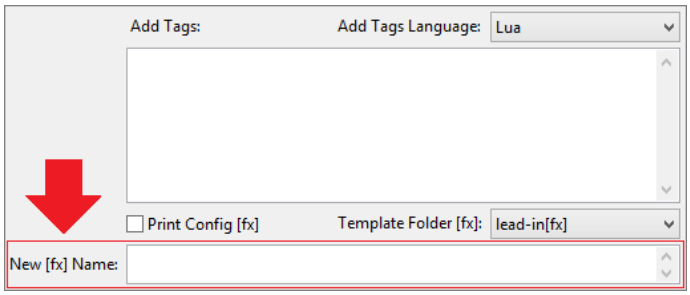
El aplicar más de un efecto al mismo tiempo es una ventaja cuando apenas estamos desarrollando el efecto total de un karaoke, cartel o logo; ya que no es evitará estar aplicando uno a uno a cada uno de los efecto individuales que lo componen.

effector.savefx(mode)

Es una función interna del **KE** que exporta una serie de información, tanto del script .ass, como de los efectos a aplicar. Consta de 12 modos:

1. **"export config"**
2. **"export times ms"**
3. **"export times"**
4. **"export lines"**
5. **"export text"**
6. **"export hira"**
7. **"export kata"**
8. **"export roma"**
9. **"export text_stripped"**
10. **"export hira_stripped"**
11. **"export kata_stripped"**
12. **"export roma_stripped"**

Toda la información que exporta esta función, lo hace en un archivo nuevo en formato .txt, y los modos de la función se deben ingresar en la celda de texto "**New [fx] Name**":



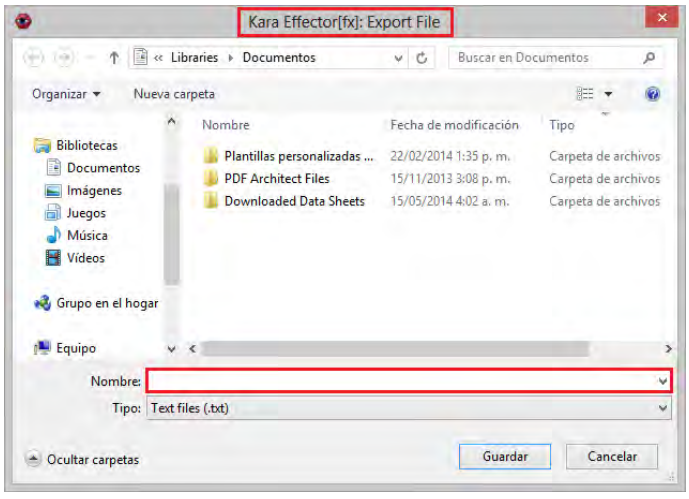
Lo que debemos hacer es escribir el modo en esta celda de texto, sin las comillas, y aplicar el efecto como siempre lo hacemos de forma normal.

Modo 1:

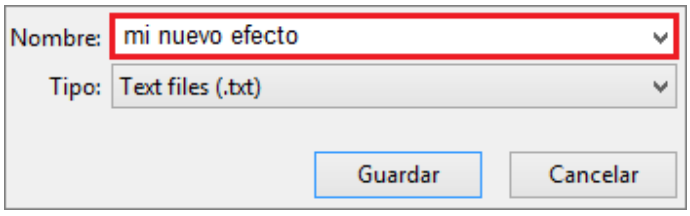
Exporta en un archivo .txt a todas las configuraciones del efecto, que posteriormente puede ser compartido con otros usuarios del **KE**.

Ejemplo:

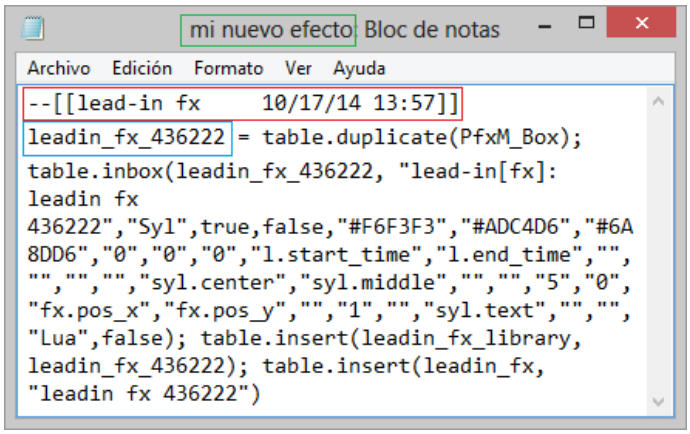
Al usar cualquiera de los doce modos de la función, nos debe salir un cuadro de dialogo como el de la siguiente imagen, en donde decidiremos el lugar de destino del archivo y el nombre del mismo:



Este es el método rápido de exportar las configuraciones del efecto en un archivo .txt, y una vez hallamos decidido en lugar de destino, le ponemos un nombre a dicho archivo:



Al abrir el archivo exportado, veremos algo como esto:



De la anterior imagen del archivo exportado se resaltan tres rectángulos de diferentes colores:

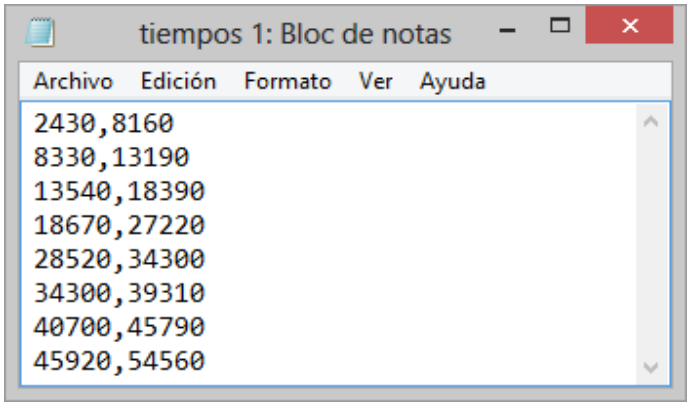
- En verde, el nombre del archivo .txt
- En rojo vemos 2 datos, primero la librería de efectos en la que se guardará el efecto una vez que hallamos pegado estas configuraciones en el **KE**, en el archivo **Effector-newfx-3.2.lua**. y segundo, la fecha y hora de creación del archivo. La fecha en formato Mes/Día/Año y la hora en HH:MM
- En azul, el nombre que por default el KE le asigna a las configuraciones de un efecto al cual no le hallamos puesto uno.

El método para asignarle un nombre al efecto exportado en un archivo .txt es el siguiente:

O sea, escribimos el modo "**export config**" y el nombre que le pondremos al efecto, separados por una coma (,). Ambos sin comillas.

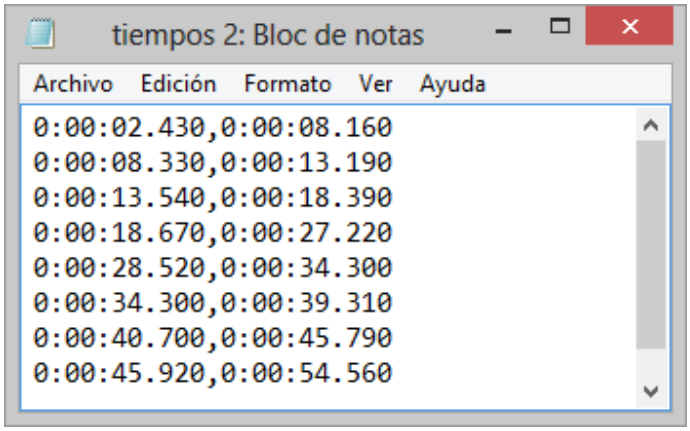
Modo 2:

Exporta un archivo .txt con los tiempos de inicio y final en ms de las líneas seleccionadas para aplicar un efecto:



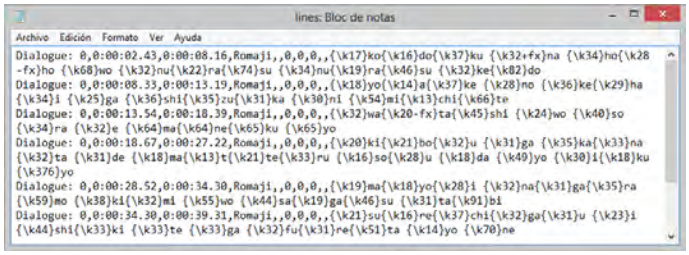
Modo 3:

Exporta un archivo .txt con los tiempos de inicio y final en formato HH:MM:SS.ms de las líneas seleccionadas para aplicar un efecto:



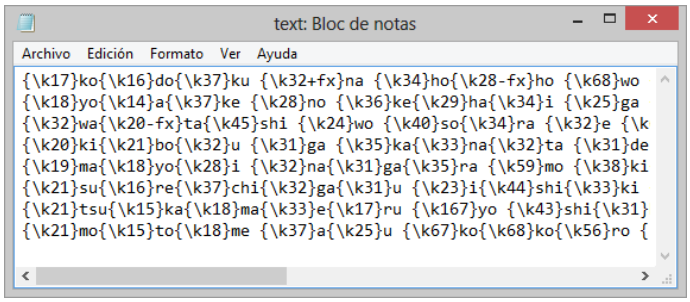
Modo 4:

Exporta un archivo .txt con la información completa de las líneas seleccionadas para aplicar un efecto, con el mismo formato de un script .ass:



Modo 5:

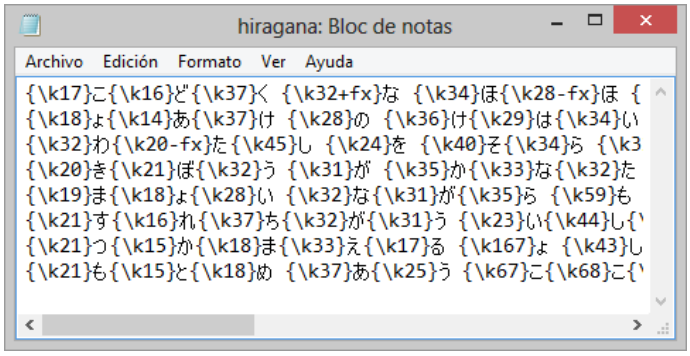
Exporta un archivo .txt con el contenido completo de las líneas seleccionadas para aplicar un efecto, incluidos los tags (etiquetas):



Modo 6:

New [fx] Name: **export hira**

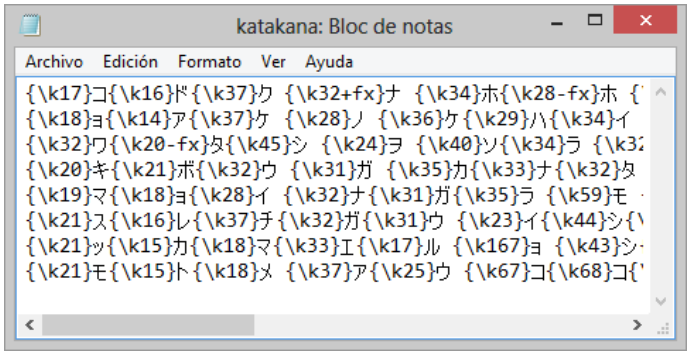
Exporta un archivo .txt con el **romaji** o el **katakana** de las líneas karaoke seleccionas, convertidos en **hiraganas:**



Modo 7:

New [fx] Name: **export kata**

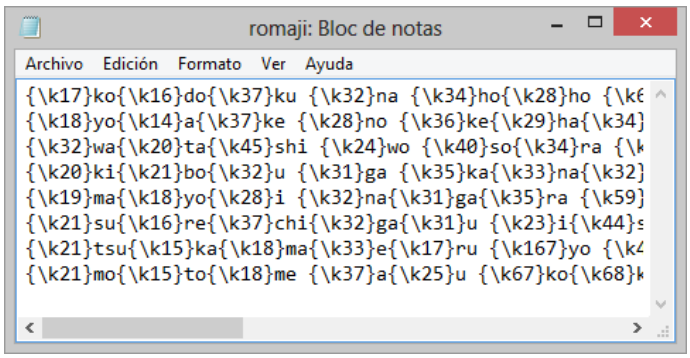
Exporta un archivo .txt con el **romaji** o el **hiragana** de las líneas karaoke seleccionas, convertidos en **katakanas:**



Modo 8:

New [fx] Name: **export roma**

Exporta un archivo .txt con el **hiragana** o el **katakana** de las líneas karaoke seleccionas, convertidos en **romajis:**



Modo 9:

New [fx] Name: **export text_stripped**

Este modo es similar al modo 5, pero exporta el contenido de las líneas de texto seleccionadas sin los tags (etiquetas).

Modo 10:

New [fx] Name: **export hira_stripped**

Este modo es similar al modo 6, exporta el contenido de las líneas de texto seleccionadas convertidas a **hiraganas**, pero sin los tags (etiquetas).

Modo 11:

New [fx] Name: **export kata_stripped**

Este modo es similar al modo 6, exporta el contenido de las líneas de texto seleccionadas convertidas a **katakanas**, pero sin los tags (etiquetas).

Modo 12:

New [fx] Name: **export roma_stripped**

Este modo es similar al modo 6, exporta el contenido de las líneas de texto seleccionadas convertidas a **romajis**, pero sin los tags (etiquetas).

No está demás aclarar que los modos que exportan a los **romajis**, **katanas** y **hiraganas** están pensados para las líneas de karaoke que tienen las sílabas separadas, es decir que no aplican a las líneas de traducción normales.

Es todo por ahora para el **Tomo XXIX**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo 30]

Kara Effector 3.2:

En este **Tomo 30** continuaremos viendo las funciones de la **librería effector**. Esta librería contiene una serie de funciones interesantes que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también en la edición de los subtítulos.

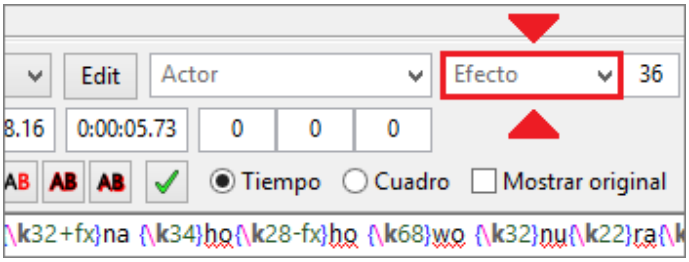
Librería Effector [KE]:

1. **effector.pos**
2. **effector.knj**
3. **effector.offset_pos**
4. **effector.import**
5. **effector.addfx**
6. **effector.savefx**
7. **effector.modify_pos**
8. **effector.new_pos**
9. **effector.default_val**
10. **effector.effect_offset**
11. **effector.decide**
12. **effector.print_error**
13. **effector.run_fx**
14. **effector.preprosses_styles**
15. **effector.preprosses_macro**
16. **effector.preprosses_lines**
17. **effector.macro_fx**

De las anteriores funciones, las que están en negro son las funciones administrativas, y las que están en azul son las que de una u otra manera le podemos sacar provecho.

effector.effect_offset( )

Es una función interna del **KE** que retorna una **tabla** con a lo más 32 valores numéricos ingresados en la celda "efecto" del **Aegisub**:



Los valores se ingresan separados por coma (,) o por punto y coma (;), así, Ejemplo:

En las líneas se verá algo como esto:

Final	Estilo	Efecto	Texto
0:00:08.16	Romaji		✱ko✱do✱ku ✱na ✱ho✱ho ✱
0:00:13.19	Romaji		✱yo✱a✱ke ✱no ✱ke✱ha✱i
0:00:18.39	Romaji	10, 24, 11.4, 86	✱wa✱ta✱shi ✱wo ✱so✱ra ✱
0:00:27.22	Romaji		✱ki✱bo✱u ✱ga ✱ka✱na✱ta
0:00:34.30	Romaji		✱ma✱yo✱i ✱na✱ga✱ra ✱m
0:00:39.31	Romaji		✱su✱re✱chi✱ga✱u ✱i✱shi✱
0:00:45.79	Romaji		✱tsu✱ka✱ma✱e✱ru ✱yo ✱
0:00:54.56	Romaji		✱mo✱to✱me ✱a✱u ✱ko✱ko
0:00:08.16	Hiragana		✱こ✱ど✱く ✱な ✱ほ✱ほ ✱を
0:00:13.19	Hiragana		✱ょ✱あ✱け ✱の ✱け✱は✱い

La **tabla** en la que se alojan estos valores se llama:

effect_val = { }

Los valores por default de cada uno de los 32 valores numéricos de la **tabla effect_val** es cero (0). Entonces los cuatro primeros valores, dependiendo de la línea, para este ejemplo, serían:

- **Línea 1: effect_val{ 0, 0, 0, 0 }**
- **Línea 2: effect_val{ 0, 0, 0, 0 }**
- **Línea 3: effect_val{ 10, 24, 11.4, 86 }**
- **Línea 5: effect_val{ 0, 0, 0, 0 }**
- **Línea 6: effect_val{ 0, 0, 0, 0 }**

Y así sucesivamente para el resto de las líneas, es decir que si no ponemos nada en la celda “efecto” de una línea en el script, todos los 32 valores de la tabla **effect_val** serán cero para esa línea.

Para ingresar diferentes valores por línea en la tabla, lo que debemos hacer es algo como esto:

Estilo	Efecto	Texto
Romaji		✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱nu✱ra
Romaji	5, 25, 25	✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱shi✱zu
Romaji	10, 24, 11.4, 86	✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱ma✱ne✱
Romaji	4, 8, 12, 16	✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de ✱ma✱t
Romaji		✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki✱mi ✱w
Romaji		✱su✱re✱chi✱ga✱u ✱i✱shi✱ki ✱te ✱ga
Romaji		✱tsu✱ka✱ma✱e✱ru ✱yo ✱shi✱k✱ka✱ri
Romaji		✱mo✱to✱me ✱a✱u ✱ko✱ko✱ro ✱so✱re
Hiragana		✱こ✱ど✱く ✱な ✱ほ✱ほ ✱を ✱ぬ✱ら✱す ✱

Para el ejemplo anterior, la **tabla** en la segunda línea tendrá los siguientes valores:

- **effect_val[1] = 5**
- **effect_val[2] = 25**
- **effect_val[3] = 25**

o sea que podemos acceder a cualquier valor de la tabla tan solo colocando el nombre de la tabla (**effect_val**) y el índice de su posición, desde 1 hasta el 32, sabiendo que el valor por default de cada uno de ellos es cero (0). Ejemplo:

- **effect_val[24] = 0**

Solo para a los tres primeros valores de la **tabla**, podemos acceder a dichos valores con otro nombre opcional son los siguientes:

- **fx.offset_x**
- **fx.offset_y**
- **fx.offset_z**

Estilo	Efecto	Texto
Romaji		✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱nu✱ra
Romaji	5, 25, 25	✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱shi✱zu
Romaji	10, 24, 11.4, 86	✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱ma✱ne✱
Romaji	4, 8, 12, 16	✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de ✱ma✱t
Romaji		
Romaji		
Romaji		
Romaji		
Hiragana		

fx.offset_x = effect_val[1]
fx.offset_y = effect_val[2]
fx.offset_z = effect_val[3]

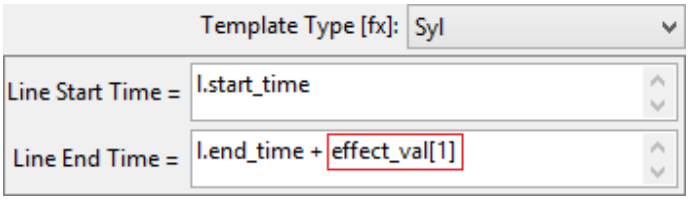
Cualquiera de los dos nombres para estos tres primeros valores numéricos es válido para poder usarlos en un efecto en la celda de texto de la Ventana de Modificación del **KE**, que necesitemos.

Ejemplo:

Un ejemplo sencillo es adicionar tiempo a una o más líneas seleccionadas:

Final	Estilo	Efecto	Texto
0:00:08.16	Romaji		✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱nu✱
0:00:13.19	Romaji	1000	✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱shi✱
0:00:18.39	Romaji		✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱ma✱
0:00:27.22	Romaji		✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de ✱ma
0:00:34.30	Romaji		✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki✱mi
0:00:39.31	Romaji		✱su✱re✱chi✱ga✱u ✱i✱shi✱ki ✱te ✱g
0:00:45.79	Romaji		✱tsu✱ka✱ma✱e✱ru ✱yo ✱shi✱k✱ka
0:00:54.56	Romaji		✱mo✱to✱me ✱a✱u ✱ko✱ko✱ro ✱so
0:00:08.16	Hiragana		✱こ✱ど✱く ✱な ✱ほ✱ほ ✱を ✱ぬ✱ら✱す

En la imagen anterior, colocamos 1000 en la celda “efecto” de la segunda línea de karaoke, y lo que haremos en el efecto seleccionado será adicionar ese valor como tiempo extra en **Line End Time**:



Entonces lo que la función hará será adicionar 1000 ms al tiempo final de la segunda línea y 0 ms al resto de ellas ya que ese es el valor por default de **effect_val[1]**.

Lo que podemos concluir como la aplicación de esta función es el poder guardar valores diferentes e independientes en las líneas del script que selecciones y poderlos usar a nuestra conveniencia en los efectos a aplicar.

Como ya les había mencionado antes, el resto de las funciones de la librería **effector** son funciones consideradas como administrativas, ya que no podemos usarlas en nuestros efectos de manera directa, sino que lo hacen de forma interna y hacen que el KE funcione de la manera que se supone que tiene que hacerlo. A continuación daré una corta descripción de cada una de ellas:

- **effector.pos**: es la función que hace que los objetos karaokes queden posicionados en el vídeo luego de aplicar nuestros efectos.
- **effector.knj**: es la función que permite ubicar de forma vertical a nuestros subtítulos, en especial los kanjis, hiraganas y katakanas.
- **effector.modify_pos**: esta función calcula cambios hechos en las celdas de posición de la Ventana de Modificación del **KE** y hace los ajustes necesarios para retornar el tag indicado. Las opciones de tags de posición que retornan las celdas de posición son:
  - **\pos**
  - **\move**
  - **\moves3**
  - **\moves4**
  - **\mover**
- **effector.new_pos**: esta función está incluida en algunas funciones del **KE** para redefinir los valores de posición, ejemplo la función **shape.Rmove**
- **effector.default_val**: esta función provee valores por default a las celdas de texto de la Ventana de Modificación del **KE**, con el fin de evitar un error en el caso de cometer un error sencillo.
- **effector.print_error**: esta función está incluida en casi todas las funciones del **KE** y es una alerta temprana de qué puede estar mal en el caso de cometer un error en un efecto. La función hace que veamos un mensaje indicando el parámetro de la función en el que está el error y nos dice qué tipo de valor deber ir en dicho parámetro.
- **effector.run_fx**: es la función que hace que se apliquen los efectos seleccionados.
- **effector.preprosses_styles**: esta función hace que veamos en lista a todos los estilos disponibles para aplicar efectos de los que están en el script.
- **effector.preprosses_macro**: esta función prepara la información que contienen las dos ventanas del **KE** y la organiza en las respectivas celdas.
- **effector.preprosses_lines**: esta función extrae toda la información de las líneas del script, estén seleccionadas o no, y guarda la información de cada una de ellas en tablas independientes para que sea usada en cualquier momento de un efecto.
- **effector.macro_fx**: es la función que hace posible la interfaz gráfica del **KE**, es la que diseña la posición y ubicación de todos los elementos en las dos ventanas de diálogos del **Kara Effector**.

Librería Aegisub [KE]:

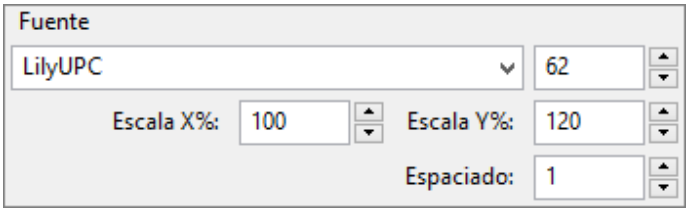
Es la librería del **KE** que contiene las funciones que hacen posible los diferentes modos de aplicar los efectos según las divisiones del texto en cada línea del script. Esta funciones hacen posible aplicar efectos tipo Syl, Word, Char y las demás opciones que hay **Template Type**.

La mayoría de las funciones de esta librería no son para hacer efectos directamente, son administrativas, y solo dos de ellas las podemos usar para obtener información de un string de texto asignado por nosotros mismos:

1. **aegisub.word(line_text, dur, i)**
2. **aegisub.word2s( )**
3. **aegisub.wordsi(words)**
4. **aegisub.syls2c( )**
5. **aegisub.line2W( )**
6. **aegisub.line2S( )**
7. **aegisub.line2C( )**
8. **aegisub.word2S( )**
9. **aegisub.word2C( )**
10. **aegisub.word2c( )**
11. **aegisub.wordci(words)**
12. **aegisub.width(string_txt)**
13. **aegisub.height(string_txt)**

aegisub.width(string_txt)

Esta función calcula en ancho medido en pixeles de un string de texto ingresado. Los cálculos que la función hace están basados en la información del estilo de la línea:



Los anteriores cinco datos del estilo de una línea, son los que determinan las dimensiones de un string de texto, lo que quiere decir estas dimensiones son relativas al estilo.

Ejemplo:

aegisub.width(string.char(R(50, 97))

Entonces la función retorna la medida en pixeles del ancho del carácter al azar que retorne **string.char**

aegisub.width(syl.text) = syl.width

aegisub.width(word.text) = word.width

aegisub.width("KE") → relativo al estilo

aegisub.height(string_txt)

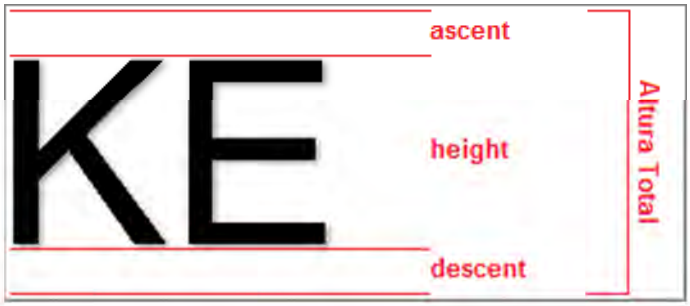
Esta función es similar a la anterior, pero con la diferencia de que retorna el alto de un string de texto en pixeles.

Ejemplo:

aegisub.height(syl.text) = syl.height

aegisub.height(word.text) = word.height

aegisub.height("KE") → relativo al estilo



Recursos [KE]:

He denominado al próximo tema como **Recursos** del **KE**, y son una serie de funciones, variables y múltiples valores y herramientas extras, con las que también podemos contar para ayudarnos en nuestros proyectos.

Recursos [KE]: → función

set_temp(ref, val)

Esta función es una creación del maestro **Pyointa**, cuya finalidad principal era el poder almacenar el valor de una variable definida con valor aleatorio, para hacer efectos con templates en **Automation Auto-4**. La función almacena a una variable en una **tabla** específica, para ser usada posteriormente. La **tabla** en donde son almacenadas las variables se llama:

temp = { }

El parámetro "**ref**" es el nombre de la variable que vamos a crear y el parámetro "**val**" es el valor que le daremos a dicha variable. El argumento "**ref**" debe ir entre comillas, ya sean dobles o sencillas.

Ejemplo:

set_temp("sizexy", R(80,120))
→ temp.sizexy = R(80,120)
→ temp = { sizexy = R(80,120) }

Ejemplo:

Add Tags Language: Automation Auto-4

```
\fscx!set_temp( "xy", R(80,120) )!\fscy!temp.xy!
```

Ejemplo:

Add Tags Language: Lua

```
format("\\fscx%s\\fscy%s",set_temp("xy", R(80,120)), temp.xy)
```

Esta función es equivalente a declarar una función en la celda de texto "**Variable**" y luego llamarla la cantidad de veces que sea necesaria. Este hecho hace que la función **set_temp** sea prácticamente obsoleta en el **Kara Effector**, pero aun así está incluida dentro de él, porque hay muchos que la usan y ya están familiarizados con ella.

Recursos [KE]: → función

remember(ref, val)

Es la función equivalente a la función **set_temp** que **Aegisub** estrenó en su versión **2.1.9** y la finalidad de esta es exactamente la misma, con la ligera diferencia de que ésta almacena las variables en una **tabla** llamada:

recall = { }

Ejemplo:

remember("angle", R(360))
→ recall.angle = R(360)
→ recall = { angle = R(360) }

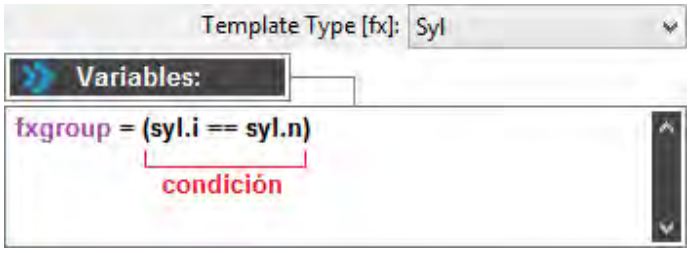
Recursos [KE]: → variable

fxgroup = false or true

Es una variable booleana a la que solo le podemos asignar los valores de falso o verdadero (false o true), usada en la celda de texto "**Variables**".

Esta variable es una función original del **Aegisub** que ha sido adaptada al **KE**, que restringe por medio de una o más condiciones dadas, los objetos a los cuales se les aplicará un efecto seleccionado.

Ejemplo:



La condición con la que definimos a la variable **fxgroup** es de igualdad, que es una condición de comparación, y solo es verdadera cuando **syl.i** es igual a la última sílaba de cada línea seleccionada para aplicar el efecto.

Entonces, cualquiera que haya sido el efecto aplicado, éste solo se aplicará a la última sílaba y no tendrá en cuenta al resto de ellas.

Recordemos las condiciones de comparación:

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

También tenemos a la disyunción (**or**) y la conjunción (**and**), que también arrojan un valor booleano entre falso (false) o verdadero (true), al comparar el valor de dos condiciones:

Disyunción (**or**):

- **true or true** → **true**
- **true or false** → **true**
- **false or true** → **true**
- **false or false** → **false**

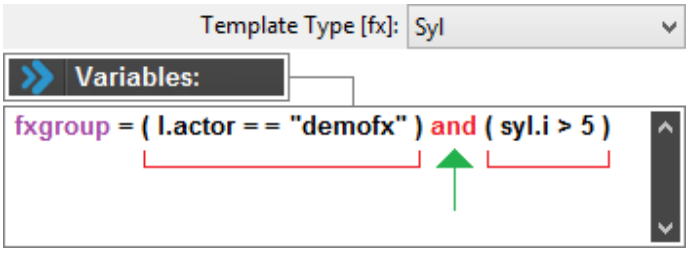
Conjunción (**and**):

- **true and true** → **true**
- **true and false** → **false**
- **false and true** → **false**
- **false and false** → **false**

Ejemplo:

Estilo	Actor	Texto
Romaji		✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱nu✱ra✱su ✱nu
Romaji	demofx	✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱shi✱zu✱ka ✱ni
Romaji		✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱ma✱ne✱ku ✱yo
Romaji		✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de ✱ma✱t✱te✱ru
Romaji		✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki✱mi ✱wo ✱sa✱g
Romaji		✱su✱re✱chi✱ga✱u ✱i✱shi✱ki ✱te ✱ga ✱fu✱re✱

Para este ejemplo, ponemos un nombre cualquiera a una o más líneas del script (no todas para que se note el fxgroup) en la celda “actor” del **Aegisub**, como se nota en la imagen anterior.

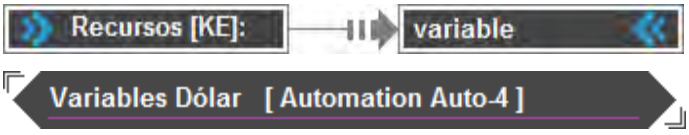


La primera condición es verdadera (true) solo para aquellas líneas que en la celda “actor” ponga: demofx

La segunda condición es verdadera (true) solo cuando las sílabas de las líneas son mayores que 5.

Y como las dos condiciones están relacionadas con una Conjunción (**and**), la variable **fxgroup** será verdadera solo cuando las dos condiciones sean verdadera. Es decir que al aplicar un efecto, éste solo tomará en cuanta a las sílabas mayores que 5 que pertenezcan a las líneas que tengan como actor a “demofx”:

El rectángulo naranja resalta el hecho que el efecto aplicado no tuvo en cuenta a las sílabas “**yo**”, “**a**”, “**ke**”, “**no**” y “**ke**”, porque a pesar de que éstas pertenecen a una línea con actor “demofx”, no cumplen con la segunda condición de que el número de su posición sea mayor que 5.



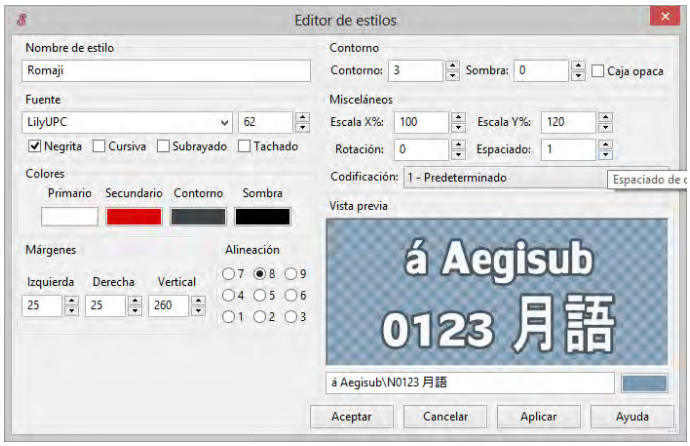
Son una serie de variables inspiradas en las variables Dólar del **Automation Auto-4** del **Aegisub**, y sirven para acceder a valores concernientes a las líneas y a las sílabas de las líneas de texto, pero acá en el **KE** he querido ampliar ese listado de variables para que abarquen a todos los modos de efectos dependiendo del **Template Type**.

La forma en la que el **KE** reconoce a dichas variables es cambiando el lenguaje de **Add Tags**. Ejemplo:

El primer grupo de variables dólar son las referentes a los estilos de cada una de las líneas del script:

variables Dólar de Estilos			
$layer	$margin_b	$fontsize	$color4
$style	$scale_x	$fontname	$alpha1
$actor	$scale_y	$spacing	$alpha2
$margin_l	$angle	$color1	$alpha3
$margin_r	$outline	$color2	$alpha4
$margin_v	$shadow	$color3	$align
$margin_t			

$layer, $style y $actor son variables de las líneas del script y el resto son las variables de todos los valores de los estilos de las líneas:



Luego tenemos las variables de tiempo:

variables Dólar de Tiempo			
$lstart	$wstart	$sstart	$fstart
$cstart	$fxstart	$start	
$lend	$wend	$send	$fend
$cend	$fxend	$end	
$lmid	$wmid	$smid	$fmid
$cmid	$fxmid	$mid	
$lkdur	$wkdur	$skdur	$fkdur
$ckdur	$fxkdur	$kdur	
$ldur	$wdur	$sdur	$fdur
$cdur	$fxdur	$dur	

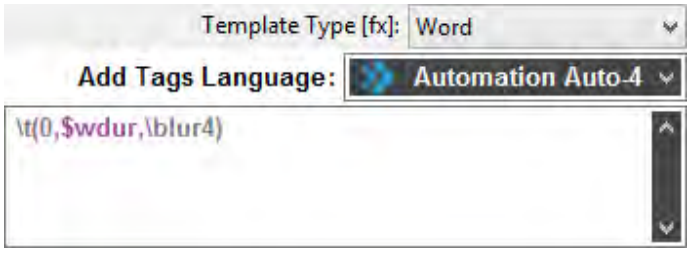
- **$lstart:** tiempo de inicio de la línea medido en ms, relativo al cero absoluto del vídeo.
- **$wstar:** tiempo de inicio de cada palabra medido en ms, relativo al tiempo de inicio de la línea a la que pertenece.
- **$sstart:** tiempo de inicio de cada sílaba medido en ms, relativo al tiempo de inicio de la línea a la que pertenece.
- **$fstart:** tiempo de inicio de cada **furigana** medido en ms, relativo al tiempo de inicio de la línea a la que pertenece.
- **$cstart:** tiempo de inicio de cada carácter medido en ms, relativo al tiempo de inicio de la línea a la que pertenece.
- **$fxstart:** es el tiempo de inicio del objeto karaoke por default dependiendo del **Template Type**. O sea que si aplicamos un karaoke con un modo Word, la variable **$fxstart** tomará el valor de **$wstart**.
- **$start:** esta variable tiene el mismo valor de **$fxstart**, es decir que ambas variables son la forma general de obtener el tiempo de inicio del objeto karaoke, dependiendo del **Template Type**.

Estas dos últimas variables las vemos en un recuadro rojo en la tabla de las variables, este recuadro indica las variables generales de cada categoría de variables.

En esta misma categoría de variables dólar de tiempo están las de tiempo final y tiempo medio y de duración total de los objetos karaokes:

- **$lend:** tiempo final de la línea. Es equivalente a **l.end_time**
- **$wmid:** tiempo medio de la palabra. Es equivalente a **word.mid_time**
- **$skdur:** duración de la sílaba en centésimas de segundos. Es equivalente a **syl.dur / 10**
- **$cdur:** duración del caracter medido en ms. Es equivalente a **char.dur**

Ejemplo:



variables Dólar de Contadores			
$li	$wi	$si	$fi
$ci	$fxi	$i	
$linen	$wordn	$syln	$furin
$charn	$fxn	$n	

Las variables de contadores nos dan dos valores, el índice de la posición y la cantidad de objetos karaoke.

- **$li:** es el contador de las líneas seleccionadas para aplicarle un efecto, equivalente a **line.i**
- **$wi:** es el contador de las palabras por línea, y es equivalente a **word.i**

- **$si:** es el contador de las sílabas por línea, y es equivalente a **syl.i**
- **$fi:** es el contador de los furiganas por línea, y es equivalente a **furi.i**
- **$ci:** es el contador de los caracteres por línea, y es equivalente a **char.i**
- **$fxi:** es el contador de los objetos karaokes en un efecto, dependiendo del **Template Type**.
- **$i:** es lo mismo que **$fxi**

El resto de las variables de esa categoría indican la cantidad de objetos karaokes, en el caso de **$linen**, ésta indica la cantidad total de líneas seleccionadas para aplicar un efecto, y las demás indican la cantidad de dicho objeto karaoke en cada línea seleccionada. Ejemplo:

- **$syln:** indica la cantidad total de sílabas en cada línea, de las seleccionadas para un efecto.

variables Dólar de Distancias Notables			
$lleft	$wleft	$sleft	$fleft
$cleft	$fxleft	$left	
$lright	$wright	$sright	$fright
$cright	$fxright	$right	
$ltop	$wtop	$stop	$ftop
$ctop	$fxtop	$top	
$lbottom	$wbottom	$sbottom	$fbottom
$cbottom	$fxbottom	$bottom	
$lcenter	$wcenter	$scenter	$fcenter
$ccenter	$fxcenter	$center	
$lx	$wx	$sx	$fx
$cx	$fxx	$x	
$lmiddle	$wmiddle	$smiddle	$fmiddle
$cmiddle	$fxmiddle	$middle	
$ly	$wy	$sy	$fy
$cy	$fxy	$y	

Esta categoría es la de las variables de distancias notables de los objetos karaokes. Son seis diferentes:

- **left (izquierda)**
- **center (centro)**
- **right (derecha)**
- **top (superior)**
- **middle (medio)**
- **bottom (inferior)**

También tenemos dos equivalencias respectivas para las variables de centro y medio, que son $x y $y para cada uno de los objetos karaokes. Ejemplo

- **$wcenter = $wx**

variables Dólar de Medidas Notables			
$lwidth	$wwidth	$swidth	$fwidth
$cwidth	$fxwidth	$width	
$lheight	$wheight	$sheight	$fheight
$cheight	$fxheight	$height	

Esta última categoría de variables dólar, es de las medidas notables de los objetos karaoke. Dichos valores vienen medidos en pixeles y equivalen al ancho y al alto.

- **$lwidth = l.width**
- **$wwidth = word.width**
- **$swidth = syl.width**
- **$fwidth = furi.width**
- **$cwidth = char.width**
- **$fxwidth = $width**
- **$lheight = l. height**
- **$wheight = word. height**
- **$sheight = syl. height**
- **$fheight = furi. height**
- **$cheight = char. height**
- **$fxheight = $height**

Podemos concluir que no hay una variable dólar de la cual no hayamos conocido su valor con anterioridad, pero como algunos de los usuarios del **KE** antes hacían efectos con los templates del **Aegisub**, entonces están más familiarizados con este tipo de notación de las variables y se les facilita seguir creando efectos en el **Kara Effector** con el lenguaje **Automation Auto-4**

Estas variables son algunas de las herramientas habilitadas para servirnos de apoyo a la hora de hacer efectos en lenguaje **Automation Auto-4**, pero no son las únicas, en el siguiente Tomo veremos unas cuantas más.

Es todo por ahora para el **Tomo 30** del **KE**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXI]

Kara Effector 3.2:

En este **Tomo XXXI** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también en la edición de los subtítulos.

Recursos [KE]:

Recursos [KE]: → herramienta

Signos de Admoración [!!]

El uso de los signos de admiración es heredado de los efectos hechos en el **Aegisub** con **Automation Auto-4** y lo que hacen es posibilitar el uso de cualquier operación matemática y el poder aplicar una o más funciones dentro de ellos.

Entonces, para empezar a usar los signos de admiración como una herramienta más del **KE**, debemos cambiar el lenguaje de **Add Tags** a **Automation Auto-4**:

Ejemplo:

Los signos de admiración hacen posible la multiplicación llevada a cobo entre 1.5 y **$scale_x**:

Template Type [fx]: Syl

Add Tags Language: Automation Auto-4

```
\fscx!1.5*$scale_x!\t(\fscx$scale_x)
```

Ejemplo:

Los signos de admiración hacen posible, para este caso, llamar a la función **retime** y modificar los tiempos de inicio y final del efecto aplicado:

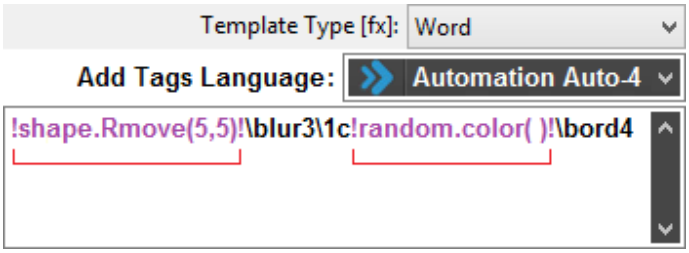
Template Type [fx]: Char

Add Tags Language: Automation Auto-4

```
!retime("start2syl", 50*(syl.i - syl.n -1), 0)! \fad(200,0)
```

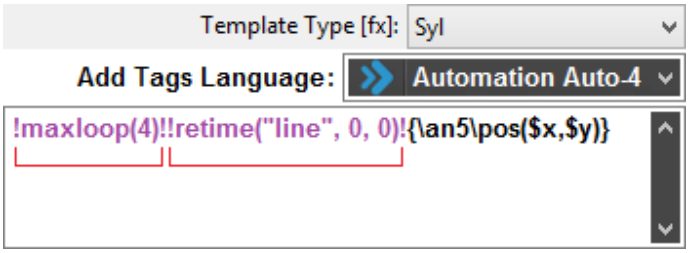
Ejemplo:

Llamar dos o más funciones independientes:



Ejemplo:

Llamar funciones consecutivas:



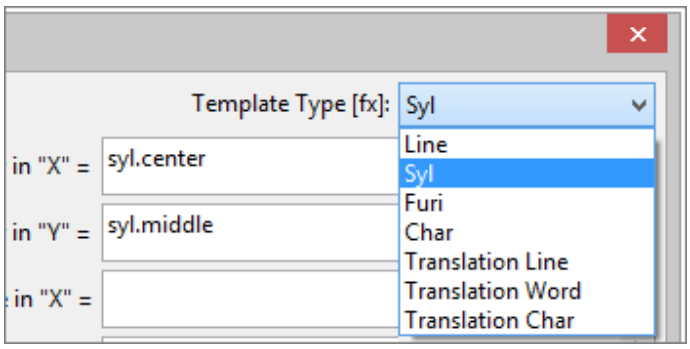
Podemos afirmar, en pocas palabras, que los signos de admiración para el **Automation Auto-4** son equivalentes a la función **string.format** para el lenguaje **LUA**, ya que ambos nos dan la opción de hacer operaciones y agregar funciones a los tags de nuestros efectos.

Recursos [KE]: herramienta

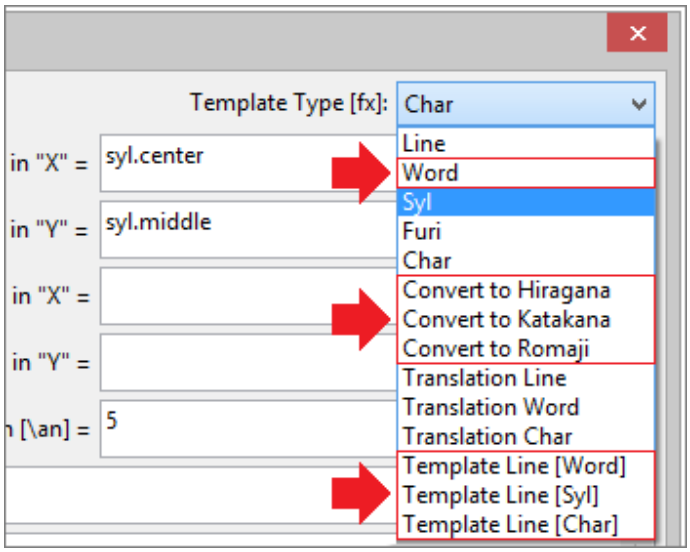
Nuevas Opciones de Template Type [Modos fx]

Desde la primera versión del **KE**, las opciones de los modos de los efectos (**Template Type**) han aumentado, lo que ha hecho que las posibilidades para nuestros efectos también lo hagan.

En un principio, el **Kara Effector** constaba de solo siete opciones de efectos en el **Template Type**, 4 dedicados a las líneas de karaoke y 3 para las de traducción, estas opciones eran: **Line**, **Syl**, **Furi**, **Char**, **Translation Line**, **Translation Word y Translation Char**.



Y en la actualidad (**KE versión 3.2.9.4**) tenemos 7 nuevos modos más, lo que nos da un total de 14, 8 para las líneas de karaoke y otras 6 para las líneas de traducción:

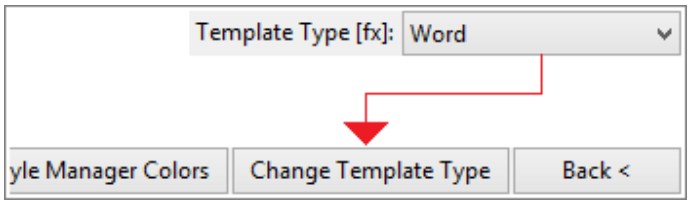


Modo: Word

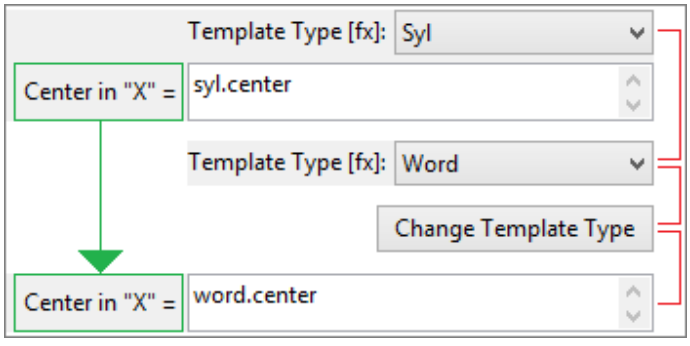
Este modo permite aplicar efectos a las líneas de karaoke, palabra por palabra, es decir que retorna una línea de fx por cada palabra que contenga cada una de las líneas de karaoke seleccionadas para aplicar un efecto.

Es modo es similar al modo "**Translation Word**", pero con la diferencia de que este último calcula la duración de cada palabra (**word.dur**) dependiendo de la cantidad de letras (caracteres) que cada una de ellas tenga; y el modo "Word" calcula la duración de la palabras dependiendo la duración individual de cada una de la sílabas que la componen.

Y como ya es sabido, luego de seleccionar el modo con el que aplicaremos nuestros efectos, debemos pulsar el botón "**Change Template Type**":



Ejemplo:



Del ejemplo anterior notamos cómo la variable en **"Center in X" = syl.center**, luego de seleccionar el **Template Type: Word** y pulsar el botón "**Change Template Type**", cambió a **"Center in X" = word.center**

Este modo convierte todo el texto de las líneas de karaoke (**Romaji** o **Katakana**) a **Hiraganas**. Este modo no afecta a las líneas de traducción, solo afecta a aquellas líneas separadas por sílabas que estén en **Romaji** o en **Katakana**.

Si el texto en una línea, a pesar de estar separado por sílabas, no está en **Romaji** o en **Katakana**, no se verá afectado por el modo y quedará tal cual.

Este modo convierte todo el texto de las líneas de karaoke (**Romaji** o **Hiragana**) a **Katakana**. Este modo no afecta a las líneas de traducción, solo afecta a aquellas líneas separadas por sílabas que estén en **Romaji** o en **Hiragana**.

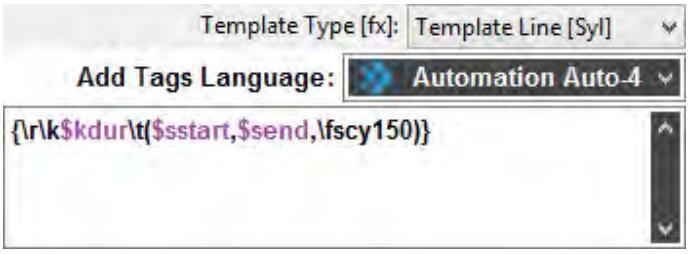
Si el texto en una línea, a pesar de estar separado por sílabas, no está en **Romaji** o en **Hiragana**, no se verá afectado por el modo y quedará tal cual.

Este modo convierte todo el texto de las líneas de karaoke (**Hiragana** o **Katakana**) a Romaji. Este modo no afecta a las líneas de traducción, solo afecta a aquellas líneas separadas por sílabas que estén en **Hiaragana** o en **Katakana**.

Si el texto en una línea, a pesar de estar separado por sílabas, no está en **Hiragana** o en **Katakana**, no se verá afectado por el modo y quedará tal cual.

Este modo es el conocido y poco usado "**template line**" del **Aegisub**, que aplicaba efectos a cada una de las sílabas de las líneas de karaoke, pero todo dentro de una única línea de fx retornada por cada línea seleccionada para aplicar un efecto.

Los tres diferentes modos de **Template Line** del **KE** tienen la particularidad de que en los dos tipos de lenguaje de **Add Tags** (**Lua** y **Automation Auto-4**) se pueden escribir los tags en leguaje **Automation Auto-4**, e incluso incluir las llaves ("{ }") en donde escribir a los mismos:

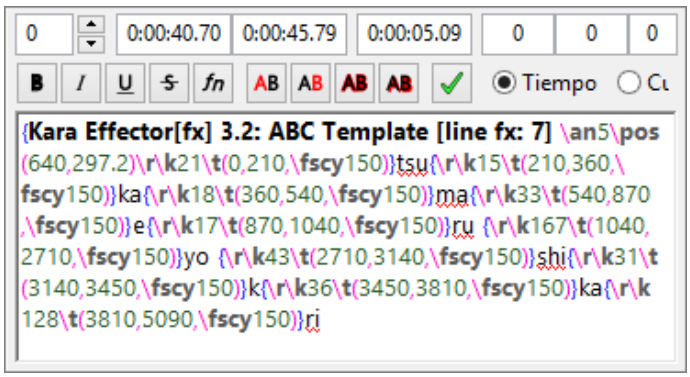


El tag **\r** cancela a todos los tags de las sílabas anteriores con el fin de no afectar a las sílabas siguientes. Lo que significa es \r = **reset**

Al aplicar, se genera una línea fx por cada línea karaoke:

Estilo	Actor	Efecto	Texto
English			✱Dos corazones que se buscan conf
Romaji	lead-in	Effector [Fx]	✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱n
Romaji	lead-in	Effector [Fx]	✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱s
Romaji	lead-in	Effector [Fx]	✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱ma
Romaji	lead-in	Effector [Fx]	✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de ✱
Romaji	lead-in	Effector [Fx]	✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki✱r
Romaji	lead-in	Effector [Fx]	✱su✱re✱chi✱ga✱u ✱i✱shi✱ki ✱te
Romaji	lead-in	Effector [Fx]	✱tsu✱ka✱ma✱e✱ru ✱yo ✱shi✱k✱
Romaji	lead-in	Effector [Fx]	✱mo✱to✱me ✱a✱u ✱ko✱ko✱ro ✱s

Al ver en detalle a una de las líneas generadas, notamos que el efecto se aplica a cada una de las sílabas, pero todo dentro de una única línea fx por cada línea de karaoke a la que le aplicamos el efecto:



Y un ejemplo del anterior efecto en el vídeo sería:

Para agregar tags en este modo, no es obligatorio es uso de las llaves (“{}”), pero igual el **KE** da la opción de ponerlos ya que hay muchos que aprendieron a hacer efectos de esta forma y ya están acostumbrados a ponerlas.

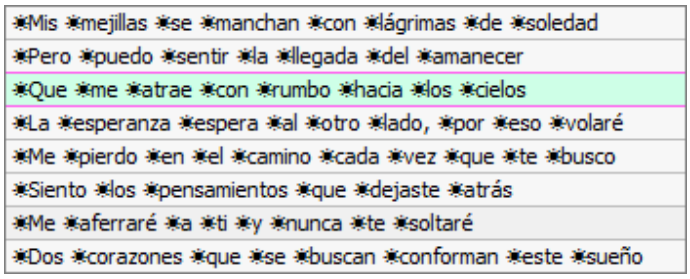
En este punto, el modo “**Template Line [Syl]**” es lo mismo que el “**template line**” del **Aegisub**, pero más adelante veremos que hasta acá llegan las similitudes entre los dos modos. En la web aún es posible hallar algunos **templates lines** del **Aegisub**, los cuales podríamos copiar y pegar tal cual en la celda de texto **Add Tags**, y el **KE** los reconocería y aplicaría sin ningún problema.

Modo: — **Template Line [Word]**

Este modo es similar al “**Template Line [Syl]**” con la obvia diferencia de que aplica los efectos palabra por palabra y todo dentro de una única línea fx por cada línea de karaoke a la que le apliquemos un efecto.

Ejemplo:

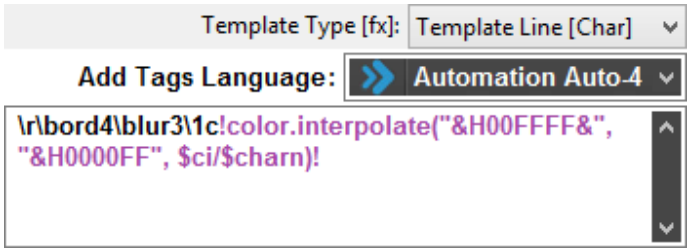
Los tres modos de “**Template Line**” del **KE** (**Word**, **Syl** y **Char**), pueden ser aplicados tanto a líneas de karaoke como a líneas de traducción, con la particularidad que si aplicamos un efecto en “**Template Line [Syl]**” a una línea de traducción, éste se aplicará palabra por palabra, como si fuera un “**Template Type [Word]**”.



Modo: — **Template Line [Char]**

Este modo es similar al “**Template Line [Syl]**” con la obvia diferencia de que aplica los efectos letra por letra y todo dentro de una única línea fx por cada línea de karaoke a la que le apliquemos un efecto.

Ejemplo:



Recursos [KE]: — **herramienta**

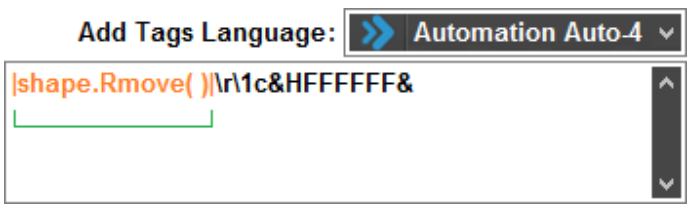
Barras Paralelas en Templates Line [‖]

Estas barras hacen posible agregar tags una única vez al inicio de las líneas fx generadas en los siguientes modos del **Template Type**:

- **Temaplate Line [Word]**
- **Temaplate Line [Syl]**
- **Temaplate Line [Char]**

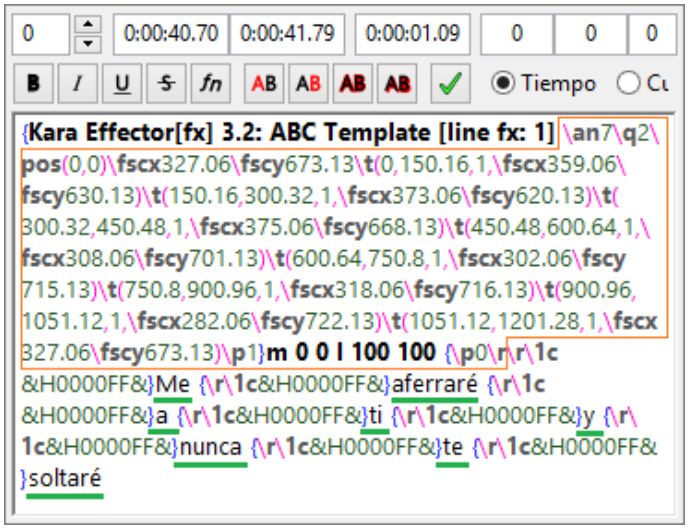
Esta herramienta es exclusiva del **Kara Effector**, y nos da la posibilidad de genera un efecto independiente a los objetos karaoke, al inicio de las líneas fx generadas por los tres anteriores modos enunciados.

Ejemplo:



De los tags agregados en el ejemplo anterior, lo que está en las barras paralelas (resaltado en color naranja) se aplicará una sola vez al inicio de la línea fx, y el resto de los tags (el texto en negro de la imagen) se aplicará en cada uno de los objetos karaoke dependiendo de cuál modo hayamos elegido entre los tres de **Template Line**. Para el ejemplo anterior se usó un **Template Line [Word].**

Es una de las líneas generadas veremos algo como esto:

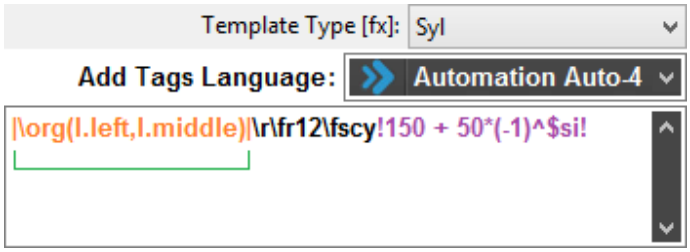


Los tags resaltados en naranja son los generados por la función **shape.Rmove** y están solo al inicio de la línea de fx, el resto de los tags se aplicaron palabra por palabra.

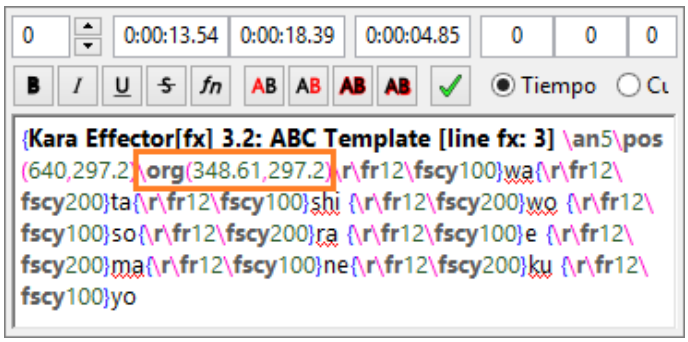
La barra (|) viene como uno de los caracteres de la mayoría de los teclados, pero para el caso de que no lo tengan o no lo encuentren en los suyos, la opción para sacarla es:

Alt + 124

Ejemplo:



El tag en las barras saldrá una sola vez al inicio de la línea:



Y al aplicar el efecto notamos los resultados:

En conclusión, el uso de las barras paralelas en **Add Tags** en los **Template Type**: "**Template Line [Word]**", "**Template Line [Syl]**" y "**Template Line [Char]**", es como aplicar un efecto **Templet Type: Line** dentro de ellos, ya que dichas barras aplican los tags una sola vez por línea fx generada.

Es todo por ahora para el **Tomo XXXI**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXII]

Kara Effector 3.2:

En este **Tomo XXXII** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también para la edición de las líneas de subtítulos.

Recursos [KE]:

Recursos [KE]: → Actualización

Función retime - Nuevos Modos [fx]

Esta función modifica los tiempos de inicio y final de las líneas fx generadas por un efecto. La función retime es una de las *funciones más usadas del* ***Automation Auto-4*** del **Aegisub**, y en su versión original consta de 10 modos:

1. **preline**
2. **line**
3. **postline**
4. **presyl**
5. **start2syl**
6. **syl**
7. **syl2end**
8. **postsyl**
9. **sylpct**
10. **set or abs**

En el **KE**, al haber más opciones para aplicar un efecto, los modos aumentaron en 29:

Función Retime - 29 Modos iniciales				
LINE	WORD	SYL	FURI	CHAR
preline	preword	presyl	prefuri	prechar
	start2word	start2syl	start2furi	start2char
line	word	syl	furi	char
	word2end	syl2end	furi2end	char2end
postline	postword	postsyl	postfuri	postchar
linepct	wordpct	sylpct	furipct	charpct
set or abs				

El **Kara Effector**, a partir de la versión **3.2.8** está provisto con 4 nuevos modos más:

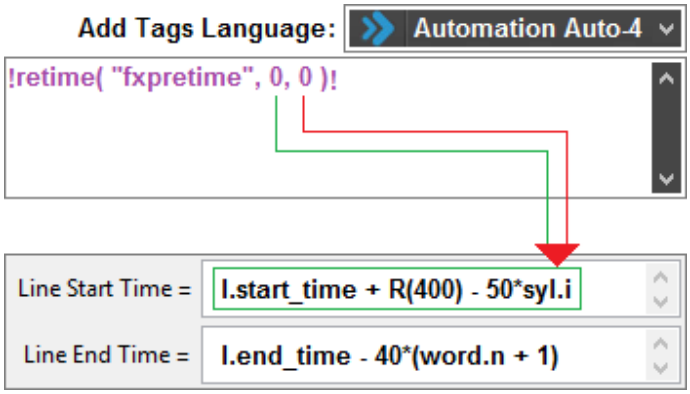
1. **fxpretime**
2. **fxtime**
3. **fxposttime**
4. **fxpct**

Modo: fxpretime

Retimea los tiempos de las líneas fx generadas por un efecto, con referencia al tiempo de inicio registrado en la celda "**Line Start Time**" de la Ventana de Modificación del **Kara Efector**.

Ambos ceros en la función equivalen al tiempo de inicio del efecto, en la celda "**Line Start Time**".

Ejemplo:



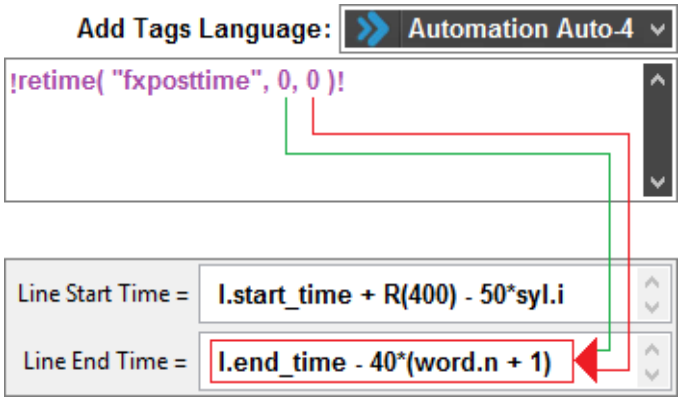
Este modo de la función es similar al modo "**preline**", pero con la diferencia de que no toma como referencia al tiempo de inicio de la línea karaoke, sino el de la línea fx.

Modo: fxposttime

Retimea los tiempos de las líneas fx generadas por un efecto, con referencia al tiempo de inicio registrado en la celda "**Line Start Time**" de la Ventana de Modificación del **Kara Efector**.

Ambos ceros en la función equivalen al tiempo final del efecto, en la celda "**Line End Time**".

Ejemplo:



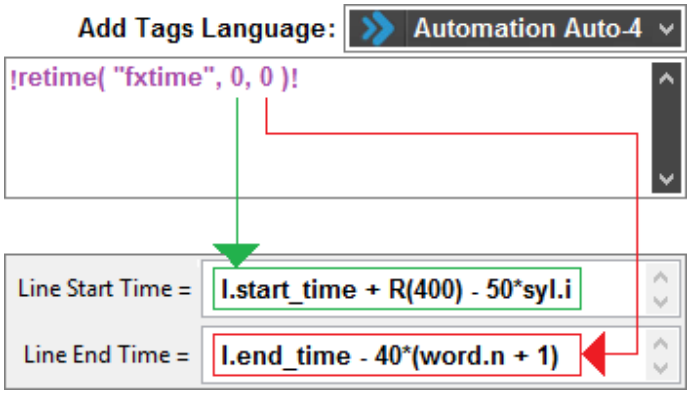
Este modo de la función es similar al modo "**postline**", pero con la diferencia de que no toma como referencia al tiempo final de la línea karaoke, sino el de la línea fx.

Modo: fxtime

Retimea los tiempos de las líneas fx generadas por un efecto, con referencia al tiempo de inicio registrado en la celda "**Line Start Time**" de la Ventana de Modificación del **Kara Efector**.

El primer cero en la función equivale al tiempo de inicio del efecto, en la celda "**Line Start Time**" y el segundo, al tiempo final del efecto, en la celda "**Line End Time**".

Ejemplo:



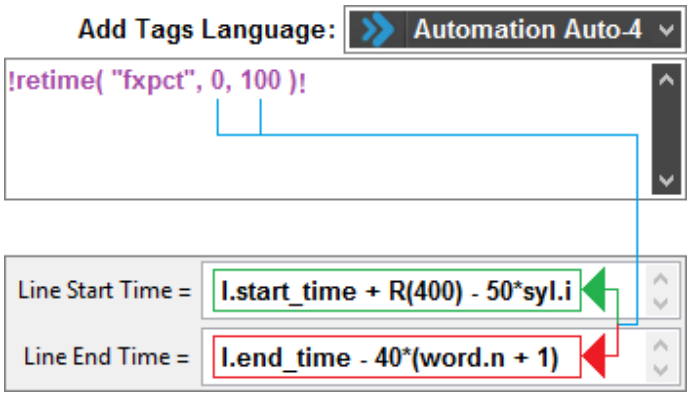
Este modo de la función es similar al modo "**line**", pero con la diferencia de que no toma como referencia a los tiempos de la línea karaoke, sino los tiempos de la línea fx.

Modo: fxpct

Retimea los tiempos de las líneas fx generadas por un efecto, con referencia al tiempo de inicio registrado en la celda "**Line Start Time**" de la Ventana de Modificación del **Kara Efector**.

El primer cero en la función equivale al 0% de la duración total de la línea fx (**fx.dur**). El 100 equivale al 100% de la duración total de la línea fx.

Ejemplo:



Este modo de la función es similar al modo "**linepct**", pero con la diferencia de que no toma como referencia a los tiempos de la línea karaoke, sino los tiempos de la línea fx.

Los cuatro ejemplos en los anteriores cuatro modos de la función **retime** están en lenguaje **Automation Auto-4**, lo que no quiere decir que esta función solo se puede usar en dicho lenguaje, ya que también la podemos usar en **LUA**.

Vistos estos cuatro nuevos modos de la función **retime**, el total de los modos es de 33, lo que hace que la tabla final de ellos quede de la siguiente manera:

Función Retime [KE] - 33 Modos totales				
LINE	WORD	SYL	FURI	CHAR
preline	preword	presyl	prefuri	prechar
	start2word	start2syl	start2furi	start2char
line	word	syl	furi	char
	word2end	syl2end	furi2end	char2end
postline	postword	postsyl	postfuri	postchar
linepct	wordpct	sylpct	furipct	charpct
set or abs				
FX	fxposttime	fxpct	fxtime	fxpretime

Recursos [KE]: → Actualización

text.bezier(Shape, mode, Accel, Offset_time)

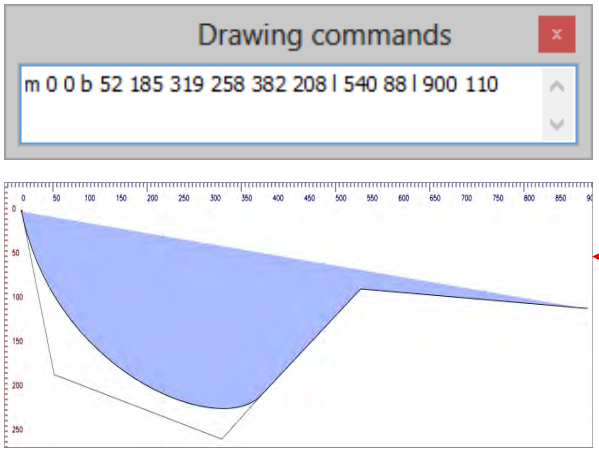
Esta función hace que el texto adopte la forma de una **curva Bézier**, de una **shape** ingresada en el primer parámetro o de un clip dibujado en las líneas del script.

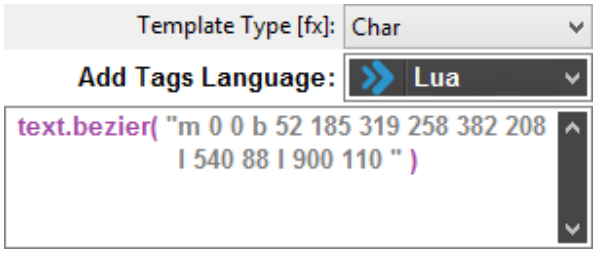
Esta función hace parte de la librería **text** y está disponible para la versión **3.2.9.4** o superior del **Kara Effector**, razón por la cual está en la sesión de recursos de actualización del **KE**.

El parámetro **Shape** tiene tres diferentes opciones, y cada una de ella con diversas características que a continuación veremos:

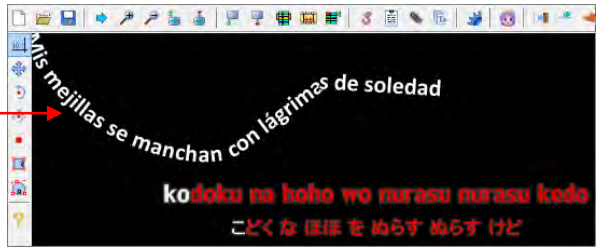
1. el código convencional de una **shape**.

Ejemplo:



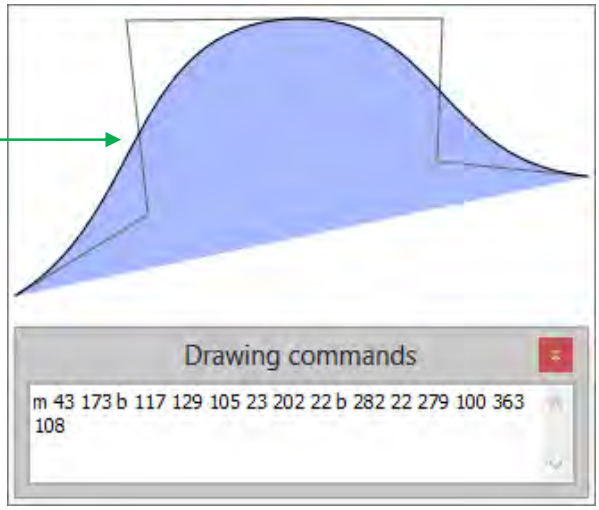


Y al aplicar:

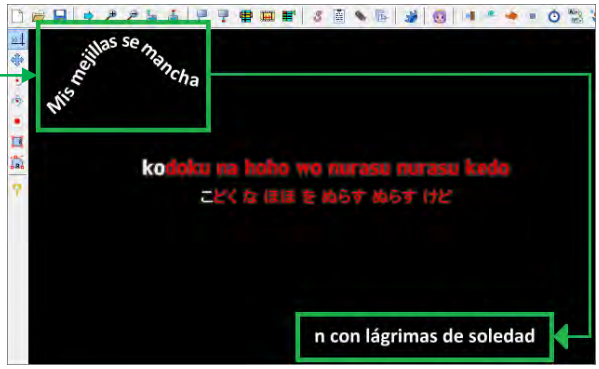


No importa la opción que elijamos en el parámetro **Shape**, debemos cerciorarnos que ésta mida igual o más que la longitud de la línea del script, o sea que su longitud sea mayor o igual que **line.width**.

Ejemplo:



Si ingresamos una shape que mida menos que la longitud de la línea a la que le apliquemos un efecto, el texto quedará en su posición por default como ni no hubiésemos usado la función:



Ejemplo:

Podemos usar alguna de las Shapes que traen por default el **KE** o una modificación de las mismas:

Variables:

```
mi_shape = shape.centerpos( shape.size(
    shape.circle, 340 ), 640, 360 )
```

una shape por default de KE

Y en **Add Tags** ponemos:

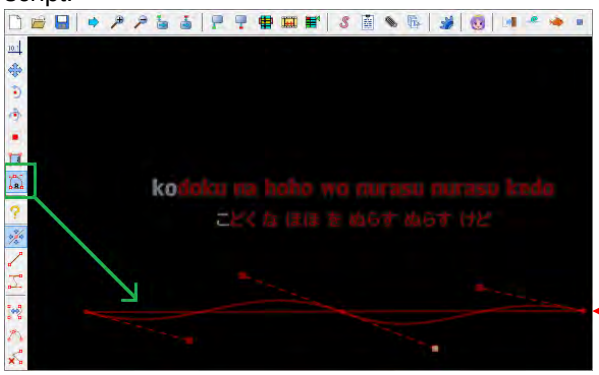
Template Type [fx]: Char
Add Tags Language: Lua

```
text.bezier( mi_shape )
```

2. el parámetro **Shape** puede ser, también, un clip dibujado en una o más líneas del script. Las líneas que no tengan un clip dentro de ellas, no se verán afectadas por la función.

Ejemplo:

Dibujamos un clip en una o más de las líneas del script:

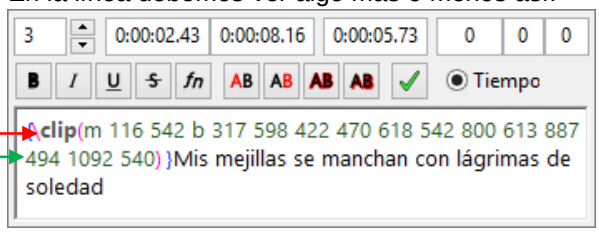


Y en **Add Tags** ponemos a alguna de estas dos opciones:

A. text.bezier(nil)
B. text.bezier( )

O sea que: **text.bezier(nil) = text.bezier( )**

En la línea debemos ver algo más o menos así:



Y al aplicar:

Y aquellas líneas en las que no dibujamos un clip, no se verá afectadas por la función.
La tercera opción del parámetro **Shape** es cuando es una **tabla** con mínimo 2 Shapes y máximo 4, y la veremos más adelante con el fin de ver primero el resto de los parámetros de la función.

El parámetro **mode** es un número entero entre 1 y 6, lo que nos da seis opciones diferentes de usar la función. Cada uno de ellos es independiente del parámetro **Shape**, es decir que no importa la manera en que le ingresemos la **shape** a la función:

mode = 1

Justifica el texto en el centro de la longitud de la shape:

Ejemplo:

Usamos como ejemplo el clip dibujado del ejemplo anterior, pero con cualquier **shape** ingresada, el **mode =1** hará que el texto quede justificado en el centro de la longitud de la misma:

Template Type [fx]: Char
Add Tags Language: Lua

```
text.bezier( nil, 1 )
```

Al aplicar, el texto (en este caso, los caracteres de la línea) adopta la forma del clip dibujado previamente y como lo había mencionado antes, éste queda justificado respecto al centro de la longitud total de la **shape**:

mode = 2

Es el modo por default de la función, y justifica el texto a la izquierda de la **shape**, o dicho de otra manera, coloca al texto desde el inicio de la **shape**:

Ejemplo:

Template Type [fx]: Char

Add Tags Language: Lua

```
text.bezier( nil, 2 )
```

mode = 3

Justifica el texto a la derecha de la **shape**, o dicho de otra manera, coloca al texto desde el final de la **shape**:

Ejemplo:

Template Type [fx]: Char

Add Tags Language: Lua

```
text.bezier( nil, 3 )
```

mode = 4

Hace que el texto se extienda a lo largo de la longitud total de la **shape**:

Ejemplo:

Template Type [fx]: Char

Add Tags Language: Lua

```
text.bezier( nil, 4 )
```

Los tres primeros valores del parámetro **mode** (1, 2 y 3) tienen una ventaja más que podemos usarla con la celda "**Actor**" del **Aegisub**, es un número que indica la cantidad de pixeles que necesitemos desplazar el texto una vez que éste adopta la forma de la **shape**.

No importa que en la celda "**Actor**" de una o más líneas del script ya hayamos escrito algo, lo único que debemos hacer es poner el valor que necesitemos al lado de lo que ahí ponga:

- Si la celda "**Actor**" está vacía solo ponemos el valor que necesitemos en la línea:

Ejemplo:

Estilo	Actor	Efecto	Texto
Romaji			✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱
Romaji			✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱
Romaji			✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱r
Romaji			✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de
Romaji			✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki

Estilo	Actor	Efecto	Texto
Romaji			✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱
Romaji	50		✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱
Romaji			✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱r
Romaji			✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de
Romaji			✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki

- Si en la celda "**Actor**" ya pone algo, solo debemos agregar un espacio seguido del valor que vamos a desplazar el texto sobre el perímetro de la shape:

Ejemplo:

Estilo	Actor	Efecto	Texto
Romaji			✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱
Romaji	50		✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱
Romaji	demo		✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱r
Romaji	intro		✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de
Romaji			✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki

Estilo	Actor	Efecto	Texto
Romaji			✱ko✱do✱ku ✱na ✱ho✱ho ✱
Romaji	50		✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱
Romaji	demo -86		✱wa✱ta✱shi ✱wo ✱so✱ra ✱
Romaji	intro		✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱
Romaji			✱ma✱yo✱i ✱na✱ga✱ra ✱mo

De cualquiera de las dos formas, la función reconoce el valor numérico que haya en la celda "**Actor**" y lo tomará como la distancia medida en pixeles para desplazar el texto según queramos.

En el **mode = 1**, dado que el texto queda justificado en el centro de la longitud total del perímetro de la shape, si el valor puesto en la celda "**Actor**" es positivo, lo desplazará hacia la derecha, en caso contrario, hacia la izquierda.

Ejemplo:

Si no hay un valor numérico en la celda "**Actor**", el valor del desplazamiento se asume como cero (0).

Hiragana			✱も✱と✱め ✱あ✱う ✱こ✱こ✱ろ ✱そ✱れ
English			✱Mis mejillas se manchan con lágrimas d
English			Pero puedo sentir la llegada del amanece

Es decir, que para **mode = 1**, el texto no se desplazará ni a la izquierda ni a la derecha.

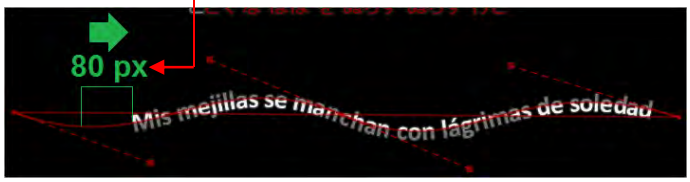
Si ponemos un valor positivo, se desplazará a la derecha:

Ejemplo:

Add Tags Language: Lua

```
text.bezier( nil, 1 )
```

Hiragana			✱も✱と✱め ✱あ✱う ✱こ✱こ✱ろ ✱
English	80		✱Mis mejillas se manchan con lágr
English			Pero puedo sentir la llegada del ar



Y para valores negativos, se desplazará a la izquierda:

Ejemplo:

Add Tags Language: Lua

```
text.bezier( nil, 1 )
```

Hiragana			✱も✱と✱め ✱あ✱う ✱こ✱こ✱ろ ✱
English	-50		✱Mis mejillas se manchan con lágr
English			Pero puedo sentir la llegada del ar

Entonces, con **mode = 1**, sí se tiene en cuenta el signo de la cantidad numérica que pongamos en la celda "**Actor**", todo depende de hacia dónde queramos desplazar al texto.

Para **mode = 2** y **mode = 3**, el valor numérico en la celda "**Actor**" debe positivo.

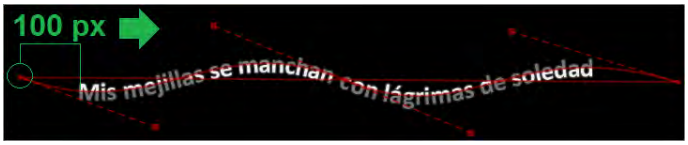
Ejemplo:

En **mode = 2**, el texto se desplazará tomando como punto de partida el inicio de la **shape**:

Add Tags Language: Lua

```
text.bezier( nil, 2 )
```

Hiragana			✱も✱と✱め ✱あ✱う ✱こ✱こ✱ろ ✱
English	100		✱Mis mejillas se manchan con lágr
English			Pero puedo sentir la llegada del ar



Ejemplo:

En **mode = 3**, el texto se desplazará tomando como punto de partida el final de la **shape**:

Add Tags Language: Lua

```
text.bezier( nil, 3 )
```

Hiragana			✱も✱と✱め ✱あ✱う ✱こ✱こ✱ろ ✱
English	125		✱Mis mejillas se manchan con lágr
English			Pero puedo sentir la llegada del ar

Continuaremos en el **Tomo XXXIII** con todo el resto de la documentación de la función **text.bezier**

Es todo por ahora para el **Tomo XXXII**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXIII]

Kara Effector 3.2:

En este **Tomo XXXIII** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también para la edición de las líneas de subtítulos.

Recursos [KE]:

En el **Tomo XXXII** empezamos a ver la función **text.bezier**, y vimos dos de las tres maneras de agregar una **shape** a la función, al igual que cuatro de los seis valores del parámetro **mode**. Es el momento de seguir en dónde nos habíamos quedado en el **Tomo** anterior.

En el **Tomo** anterior comentaba que en **mode = 1**, el valor numérico en la celda "**Actor**" determinada hacia dónde se desplazaba el texto; que si este valor era positivo, se desplazaría hacia la derecha y en caso contrario, hacia la izquierda. Una explicación más precisa de ese fenómeno es la siguiente:

mode = 1

- Si el valor numérico de la celda "**Actor**" es positivo, el texto se desplazará hacia el inicio de la **shape**, hacia su primer punto.
- Si el valor numérico de la celda "**Actor**" es negativo, el texto se desplazará hacia el final de la **shape**, hacia su último punto.

mode = 2

El valor numérico de la celda "**Actor**" debe ser un número positivo, entonces el texto se desplazará desde el inicio de la **shape**, hacia su último punto.

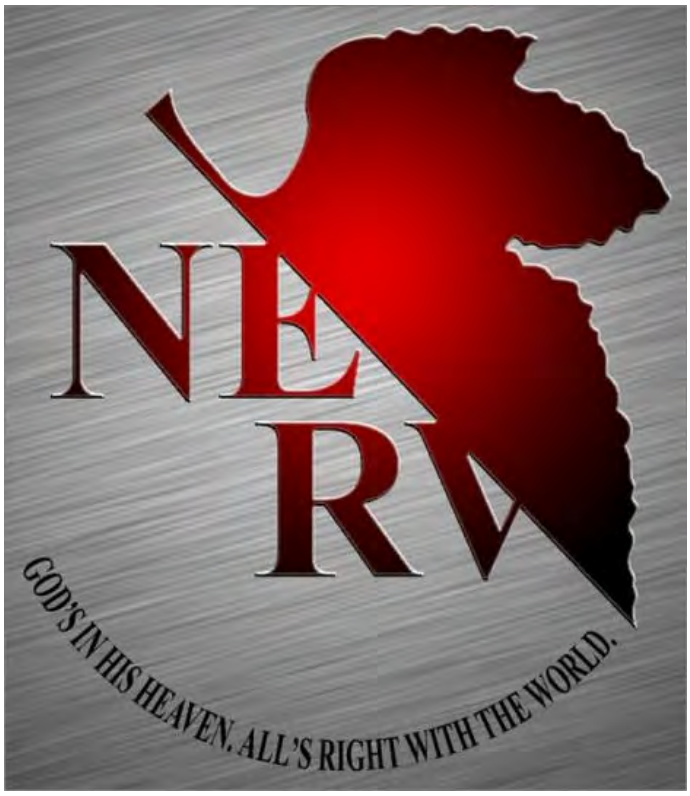
mode = 3

El valor numérico de la celda "**Actor**" debe ser un número positivo, entonces el texto se desplazará desde el final de la **shape**, hacia su primer punto.

Y para **mode = 4**, que es hasta donde vimos en el **Tomo** anterior, el valor numérico de la celda "**Actor**", no surte efecto alguno, no hace que el texto se desplace.

Para la siguiente habilidad de la función **text.bezier** usaré un logo en donde aparece texto en forma circular:

Ejemplo:



Primero modificamos el estilo del texto hasta que creamos que ya está acorde con el del vídeo:

Seleccionamos la herramienta para recortar subtítulos en un área vectorial y seleccionamos la herramienta para hacer rectas con ella.

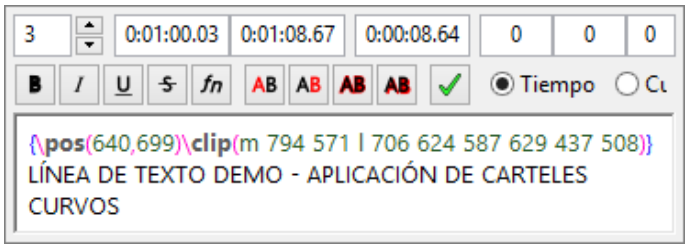
Debemos poner cuatro puntos de forma que cada uno de ellos quede en la misma circunferencia imaginaria que describe el texto original:

No importa en donde pongamos a cada uno de los puntos, pero si podemos conservar una distancia prudente entre ellos es más conveniente:

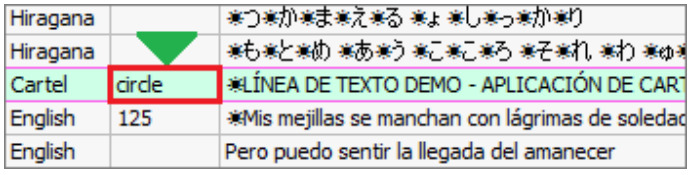
No es obligatorio que los puntos cumplan una secuencia en específico, pero sí debemos tratar que todos queden en la circunferencia del círculo:

El único punto es opcional, pero de estar, debe cumplir una simple condición, lo debemos poner en donde necesitemos que nuestro texto empiece:

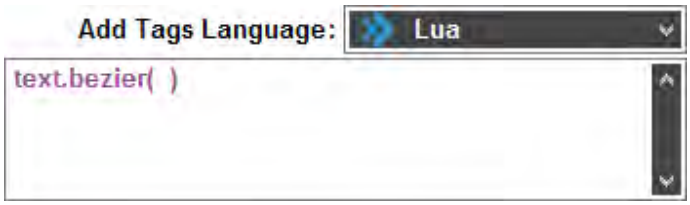
Hecho esto, nuestra línea en el script debe verse más o menos como en la siguiente imagen:



Ahora debemos hacer es escribir la palabra “circle” en la celda “**Actor**”, y si ya pone algo en esta celda, simplemente ponemos un espacio al final y la escribimos:



Y llamamos a nuestra función en **Add Tags**, sin ponerle la **shape**, para que tome los puntos creados en el clip:



Al aplicar, veremos cómo el texto adopta la forma espera:

Lo malo es que se ve encima del texto original del vídeo, pero cumple con los detalles que le ingresamos:

- La circunferencia imaginaria que describe el texto pasa justo pos los tres primeros puntos trazados en el clip.
- El texto inicia justo en donde pusimos el cuarto punto del clip.

Para remediar el hecho de que el texto generado se ve justo encima del texto original, debemos modificar el radio del círculo que lo genera, para ello debemos escribir en la celda "Efecto" del Aegisub la cantidad en pixeles que necesitemos se aumente o se reduzca el radio. Si la cantidad es positiva se aumenta, en caso contrario, se disminuye:

Ejemplo:

English			Me aferraré a ti y nunca te soltaré
English			Dos corazones que se buscan con
Cartel	circle	38	✱LÍNEA DE TEXTO DEMO - APLICA

Y al aplicar:

Quizá 30 px sea muy poco, pero eso ya depende del gusto de cada uno.

Al poner el cuarto punto en el clip dibujado, éste forma un ángulo con el centro del círculo:

Este ángulo que se forma con el centro del círculo y el cuarto punto del clip que señala el punto de inicio del texto, lo podemos modificar añadiendo o restando un valor en grados, con un segundo valor en la celda "Efecto".

Ejemplo:

Aumento del Radio (px)
Aumento del Ángulo (°)

English			Dos corazones que se buscan con
Cartel	circle	40 , 4.2	✱LÍNEA DE TEXTO DEMO - APLICA
Cartel			

En la anterior imagen, los dos números están separados por una coma, por no es obligatorio ponerla, con que estén separados por un espacio es suficiente.

Ahora aplicamos y se verán las diferencias:

Hicimos que el texto se desplazara 4.2° positivamente, es decir en forma inversa al movimiento de las manecillas del reloj.

Si ajustamos el radio y le restamos los ángulos necesarios, podemos hacer otra variante del mismo cartel.

Ejemplo:

- **Radio + 10 px**
- **Ángulo – 120°**

English				Me aferraré a ti y nunca te soltaré
English				Dos corazones que se buscan conforr
Cartel	circle	10	-120	✱LÍNEA DE TEXTO DEMO - APLICACI(
Cartel				

Estos dos valores salieron luego de un par de intentos a ensayo y error, y obviamente los valores serán diferentes dependiendo del cartel que estemos haciendo:

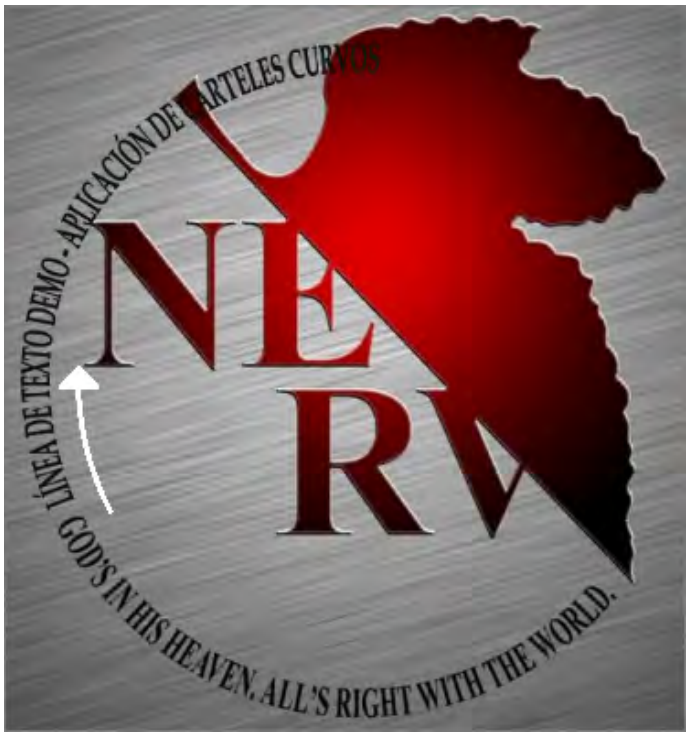
Siempre que pongamos en “**Actor**” la palabra “circle”, el texto se posicionará en la circunferencia en sentido inverso a la de las manecillas del reloj, como lo indica la flecha de la anterior imagen.

Si queremos que el texto se escriba en sentido horario, debemos escribir en “**Actor**” la palabra “icircle”:

Ejemplo:

English				Me aferraré a ti y nunca te soltaré
English				Dos corazones que se buscan conforma
Cartel	icircle	10	-3	✱LÍNEA DE TEXTO DEMO - APLICACIÓN
Cartel				

Ahora el texto está en el mismo sentido del movimiento de las manecillas del reloj:



Con “circle” e “icircle” en la celda de texto “**Actor**” y los dos valores que podemos usar en la celda “**Efecto**”, podemos lograr cualquier tipo de cartel que tenga forma circular. De aquí en adelante todo dependerá de la práctica:

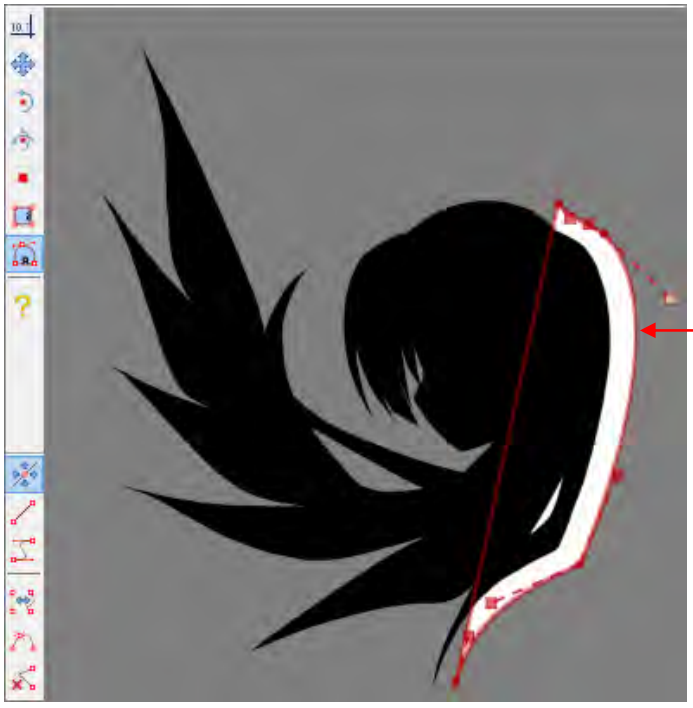
Ejemplo:

Otro ejemplo, este a dos colores:

Ejemplo:

Para los carteles que no sean perfectamente circulares o tengan otra forma libre, entonces dibujamos el contorno deseado con el clip:

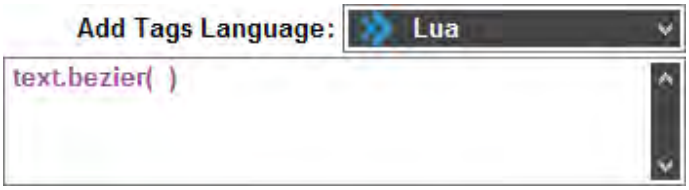
Ejemplo:



Y ya no es necesario escribir ni “circle” ni “icircle” en la celda “**Actor**”, solo necesitamos el clip:

English		Me aferraré a ti y nunca te soltaré
English		Dos corazones que se buscan conforman este su
Cartel		✱LÍNEA DE TEXTO DEMO - APLICACIÓN CARTEL

Con un **Template Type: Char**, ponemos en **Add Tags**:



Y los resultados no podrían ser mejores:

Cuando de hacer carteles y logos se trata, las posibilidades son muchas, pero de a poco vamos develando los secretos de las diferentes posiciones y rotaciones que el texto puede tener en ellos, pero hasta este punto solo hemos visto la forma de posicionar el texto de forma estática, no con un movimiento o desplazamiento respecto al tiempo. Para el próximo **Tomo** veremos carteles con dichas características.

Es todo por ahora para el **Tomo XXXIII**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXIV]

Kara Effector 3.2:

En este **Tomo XXXIV** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también para la edición de las líneas de subtítulos.

Recursos [KE]:

En el **Tomo XXXIII** vimos la forma de posicionar el texto de forma estática con la función **text.bezier**, y vimos dos de las tres maneras de agregar una **shape** a la función, al igual que cuatro de los seis valores del parámetro **mode**. Es el momento de seguir en dónde nos habíamos quedado en el **Tomo** anterior.

Recursos [KE]: → **Actualización**

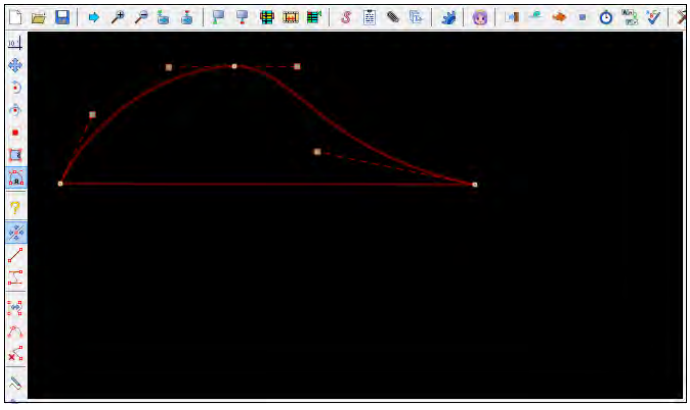
text.bezier(Shape, mode, Accel, Offset_time)

Una vez dominado el arte de posicionar el texto en cualquier forma no lineal, ahora veremos varios métodos de cómo darle movimiento a dicho texto y así poder ampliar nuestras posibilidades cuando desarrollamos un efecto.

Ya hemos visto varios ejemplos de cuando el parámetro **Shape** es el código de una **shape** normal o una de las Shapes por default del **KE**, también vimos ejemplos con clip's como remplazo de la **shape** o como marcador de puntos para la ubicación del texto en la circunferencia de un círculo.

Ahora veremos cuando el parámetro **Shape** es una **tabla** de Shapes.

Ejemplo:

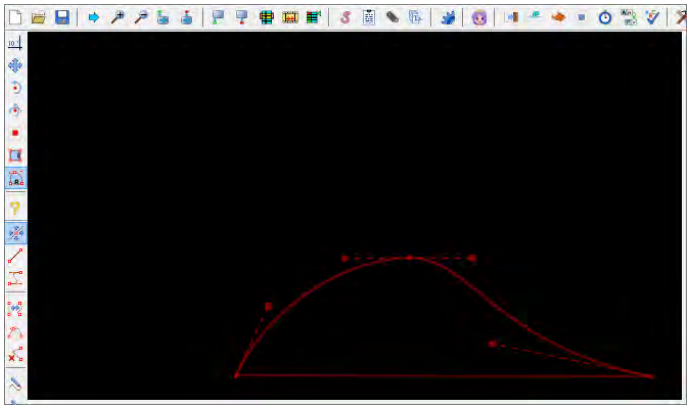


Empezamos creando la primera **shape**, la inicial, y con el código creamos una tabla en la celda "**Variables**":

Variables:

```
mi_shape = {
        [1] = "m 64 298 b 127 163 276 70 404 68 527
        69 566 236 874 300"
}
```

Luego ubicamos la segunda **shape**, que para este ejemplo será la **shape** de destino:



Entonces la **tabla** de Shapes se verá de la siguiente forma:

Variables:

```
mi_shape = {
        [1] = "m 64 298 b 127 163 276 70 404 68 527
        69 566 236 874 300",
        [2] = "m 408 672 b 471 537 620 444 748 442
        871 443 910 610 1218 674"
}
```

La idea es poner una **shape** que indique la posición inicial del texto y otra para la posición final.

Cuando la **tabla** de Shapes tiene 2 elementos, la función retorna un tag **\move**. Como ya lo había mencionado antes, la **tabla** debe tener entre 2 y 4 Shapes, lo que hace que la función retorne los siguientes tags:

- **2 shapes: \move**
- **3 shapes: \moves3**
- **4 shapes: \moves4**

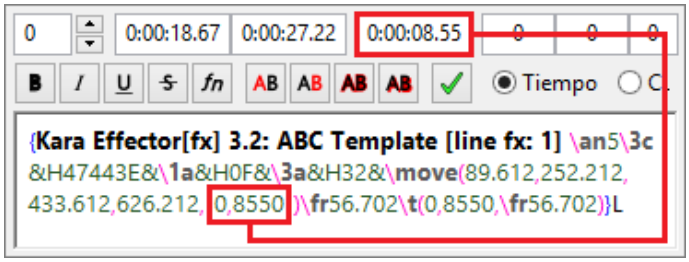
O sea que para más de 2 Shapes es necesario tener en el **Aegisub** al **VSFilterMod**. En este ejemplo solo usaremos 2, lo que nos debe retornar un tag **\move**:

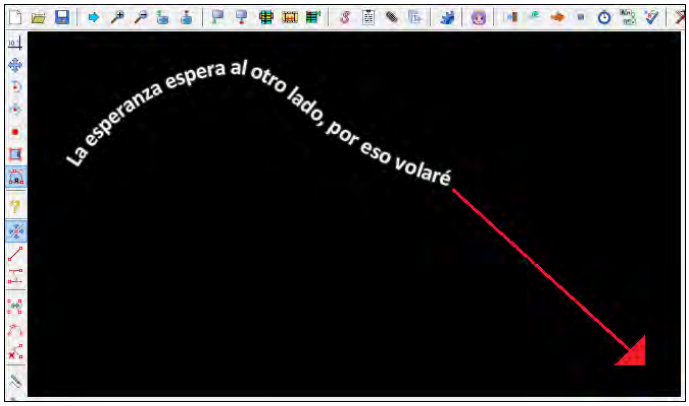
Template Type [fx]: Char

Add Tags Language: Lua

```
text.bezier( mi_shape )
```

Al aplicar, el texto inicia en la posición de la primera **shape** y a medida que trancurre el tiempo de la duración total de la linea fx, se va moviendo hacia la posición de la segunda **shape**:





Para modificar los tiempos de inicio y final del movimiento del texto generado, debemos entender la estructura que toma la función **text.bezier** cuando el parámetro **Shape** es una **tabla**:

mi_shape = { shape1, shape2, … }
text.bezier(mi_shape, mode, time_i, time, f)

El parámetro **mode** es un entero entre 1 y 4, **time_i** es el tiempo de inicio del movimiento y su valor por default es 0, y **time_fx** es el tiempo final del movimiento, donde su valor por default es **fx.dur**

Ejemplo:

Template Type [fx]: Char

Add Tags Language: Lua

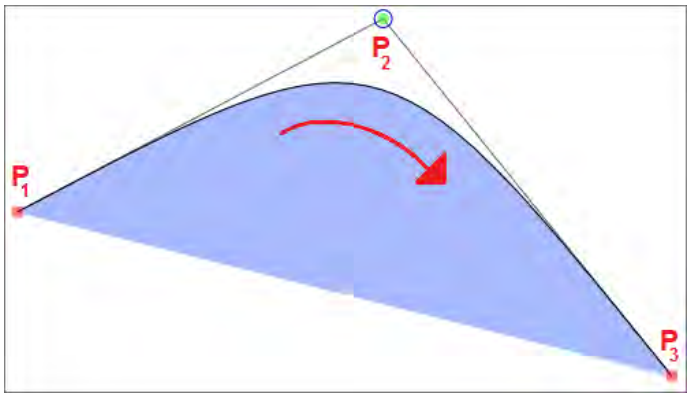
```
text.bezier( mi_shape, 1, 600, fx.dur - 400 )
```

- **mi_shape = tabla** de Shapes
- **mode = 1**: el texto queda alineado en el centro
- **time_i = 600**: el movimiento del texto empezará 600 ms luego de haber iniciado la línea fx.
- **time_f = fx.dur – 400**: el movimiento terminará 400 ms antes de que termine la línea fx.

Recordemos que para una **tabla** de dos Shapes, retorna un tag **\move**. Para más Shapes tenemos lo siguiente:

Para una **tabla** de tres Shapes, obtenemos un **\moves3**, y el movimiento tendría la siguiente particularidad:

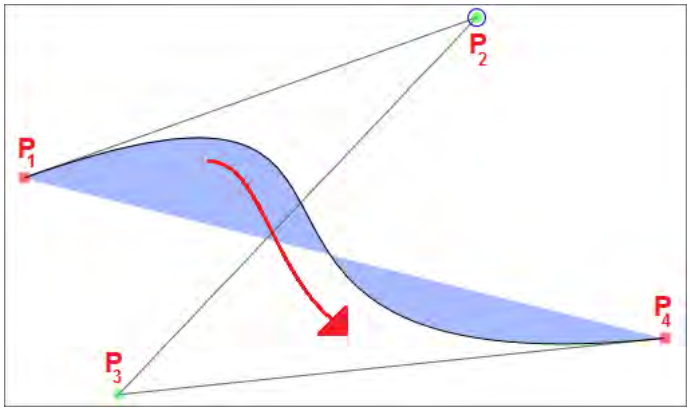
Ejemplo:



Un **\moves3** no hace un movimiento lineal entres los tres puntos ingresados, sino que hace una trayectoria **Bézier** entre ellos, como en la imagen anterior que se hace una curva entre los tres puntos.

Para una **tabla** de cuatro Shapes, obtenemos un **\moves4**, y el movimiento tendría la siguiente particularidad:

Ejemplo:



En el **\moves4** tampoco el movimiento es lineal entre sus puntos, sino que traza una curva **Bézier** entre sus cuatro puntos.

Sea cual sea la cantidad de Shapes que haya en nuestra **tabla**, no se nos debe olvidar tener en cuenta que cada una de ellas debe tener una longitud igual o mayor que el ancho de la línea a la que le apliquemos un efecto: **line.width**

Visto estos recientes ejemplos, solo nos resta por ver a los dos últimos valores del parámetro **mode** que son:

- **mode = 5**
- **mode = 6**

Estos dos valores de **mode** son válidos para cuando el parámetro **Shape** no es una **tabla**, o sea que es el código de una shape convencional, una shape por default del **KE** o un clip dibujado en la línea del script.

mode = 5

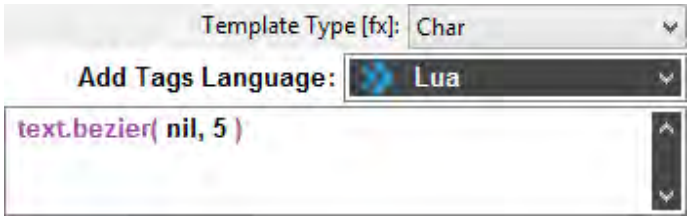
Crea una secuencia del texto y lo mueve respecto avanza el tiempo de la duración de la línea fx, desde su posición en **mode = 2** hasta **mode = 3**, o sea, desde el inicio de la **shape** hasta el final de la misma.

Lo que la función hace tanto en **mode = 5** y **mode = 6**, es una animación cuadro a cuadro de las diferentes posiciones del texto para dar la impresión de que éste se mueve a lo largo de la longitud total de la **shape** ingresada.

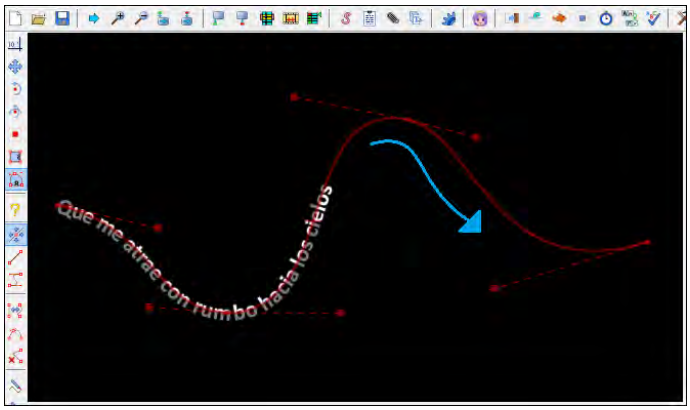
Ejemplo:

Dibujamos un clip vectorial en una de las líneas del script, asegurándonos de que éste sea más largo que la longitud del ancho de la línea. (**line.width**):

Hecho esto, ponemos:

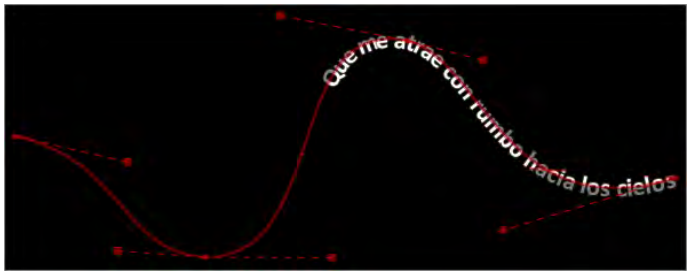


Y al aplicar veremos algo como esto (sin las curvas):



La primera posición del texto es en el inicio de la **shape**, y conforme avanza el tiempo, se irá desplazando a lo largo de todo el clip vectorial:

Así, hasta llegar al final del clip vectorial, que equivale al punto final de la **shape**:



La función crea el efecto cuadro por cuadro, y por default cada uno de ellos tiene la misma duración:

frame_dur = 41.708 ms

0:00:40.70	0:00:45.79	English			Me aferra
0:00:45.92	0:00:54.56	English			Dos coraz
0:00:13.54	0:00:13.58	English	lead-in	Effector [Fx]	✷Q
0:00:13.58	0:00:13.62	English	lead-in	Effector [Fx]	✷Q
0:00:13.62	0:00:13.66	English	lead-in	Effector [Fx]	✷Q
0:00:13.66	0:00:13.70	English	lead-in	Effector [Fx]	✷Q
0:00:13.70	0:00:13.74	English	lead-in	Effector [Fx]	✷Q

fx.dur = frame_dur

Al usar la función en **mode = 5** o **mode = 6**, podemos usar los otros dos parámetros restantes, así:

text.bezier(Shape, mode, Accel, Offset_time)

El parámetro **Accel** hace referencia a la aceleración del texto al moverse, al cambio de velocidad, y su valor por default es 1. Para valores mayores a 1, el texto acelera positivamente, en caso contrario, desacelera.

Para valores menores que 1, recomiendo uno entre 0.2 y 0.9; y para valores mayores que 1, entre 1.1 y 2.6

La elección de algunos de estos valores para la aceleración del texto dependerá de lo que estemos necesitando y es más algo de prueba y error que de una ciencia exacta.

El parámetro **Offset_time** hace referencia a un tiempo medido en milisegundos (**ms**) que se añadirá o restará de la duración por default (**41.708 ms**) de cada uno de los cuadros generados por la función. Su valor por default es 0.

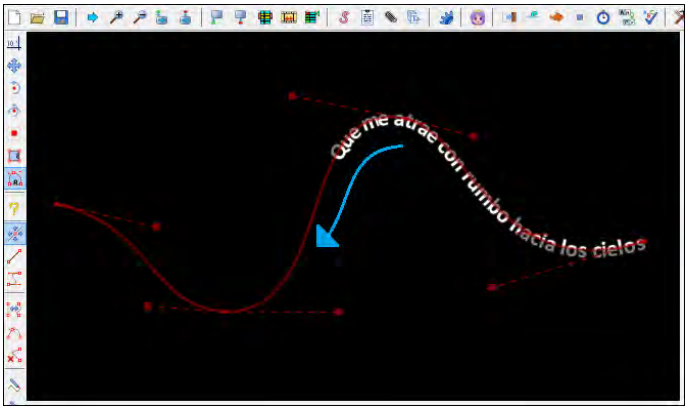
mode = 6

Crea una secuencia del texto y lo mueve respecto avanza el tiempo de la duración de la línea fx, desde su posición en **mode = 3** hasta **mode = 2**, o sea, desde el final de la **shape** hasta el inicio de la misma.

En este modo, la función hace que el texto se desplace de forma inversa a como lo hace con el modo anterior:

Ejemplo:

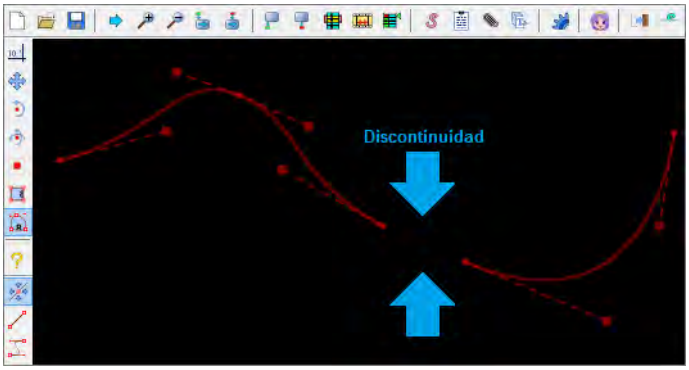
Lo que hará que el texto inicie en la parte final del clip y se vaya desplazando en función del tiempo, hacia el inicio del mismo, de forma inversa que en **mode = 5**:



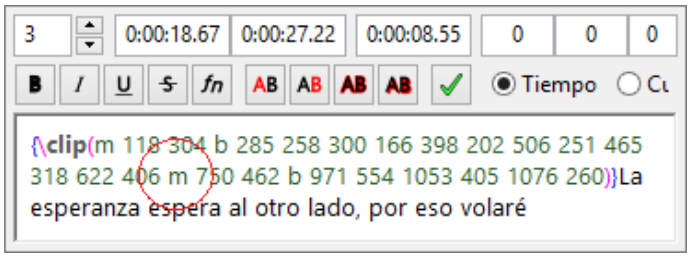
Hasta este punto, todo los ejemplos que hemos visto, han sido basado en Shapes continuas, de un solo trazo, pero la función también puede aplicarse con Shapes que no lo son.

Ejemplo:

Dibujamos una **shape** o un clip vectorial que contenga una o más discontinuidades, que sea algo notoria, como en la siguiente imagen:



En la línea del script veremos algo parecido a esto, ya que el clip vectorial está conformado por dos Shapes adheridas, una separada de la otra:



Al aplicar, el texto que no alcance a quedar en un tramo de la **shape**, saldrá en la parte restante:

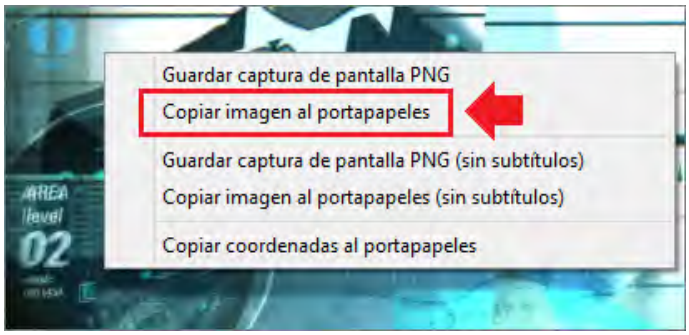
Visto sin el clip vectorial en pantalla:

Habrá ocasiones que el clip vectorial que usaremos como **shape** para la función, se extienda mucho más allá de los límites del vídeo, como la curva resaltada a continuación:

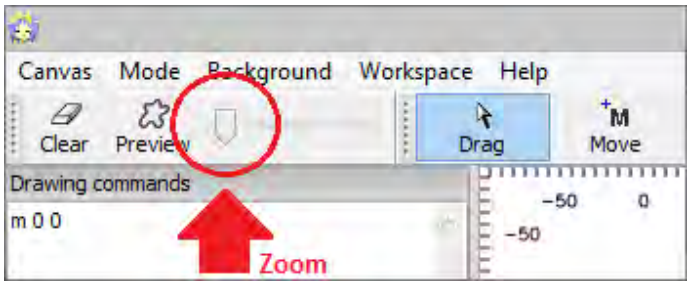
Ejemplo:

Entonces, uno de los trucos que podemos usar para poder dibujar esa parte de la **shape** que no se ve, es darle clic derecho sobre el vídeo, en el **Aegisub**:

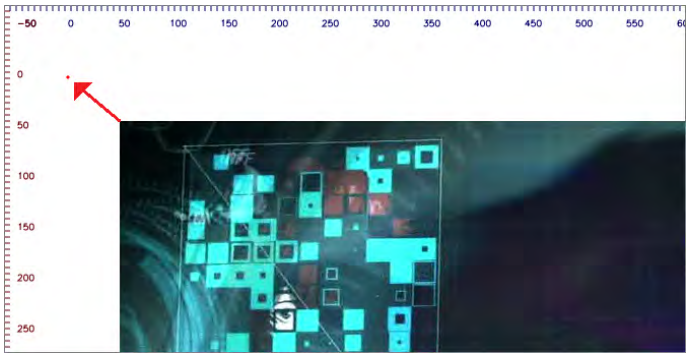
Y elegimos la opción de "**Copiar imagen al portapapeles**", lo que creará una captura instantánea del vídeo con sus dimensiones reales:



Luego abrimos el **ASSDraw3** y ponemos el zoom al mínimo, antes de pegar la captura del vídeo:

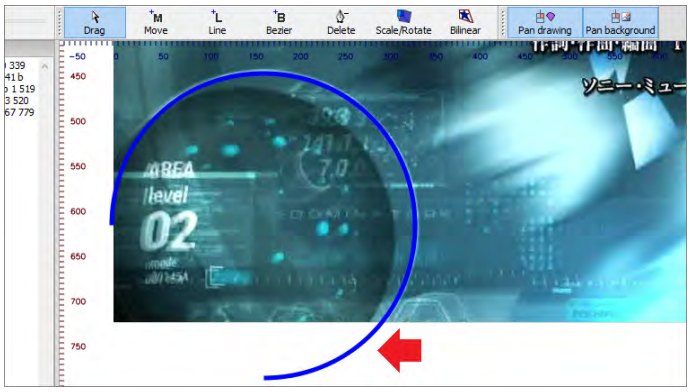


Pegamos la imagen que habíamos capturado del vídeo, que posteriormente tendremos que desplazar, de modo que la esquina superior izquierda de la captura quede justo en las coordenadas (0, 0):

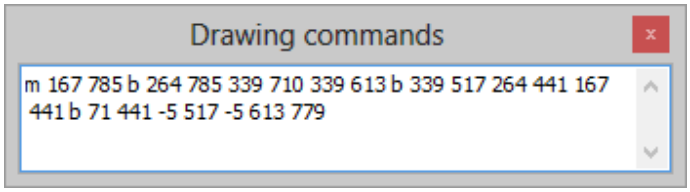


Hecho esto, desplazamos el lienzo de trabajo hasta que nos quede cómo para trazar la curva que deseamos extender por fuera de los límites del vídeo:

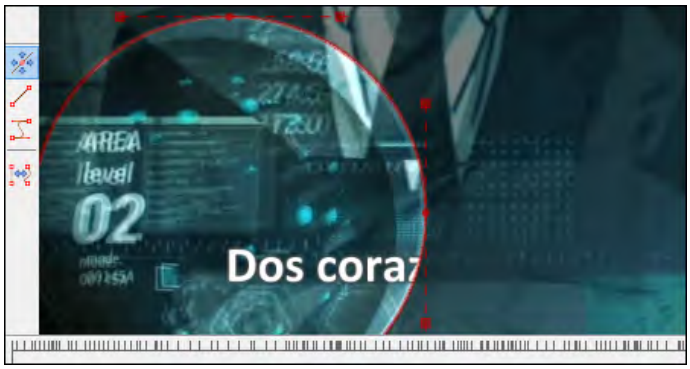
Ahora ya podemos trazar libremente la **shape** y superar los límites del vídeo, hasta dos necesitemos:



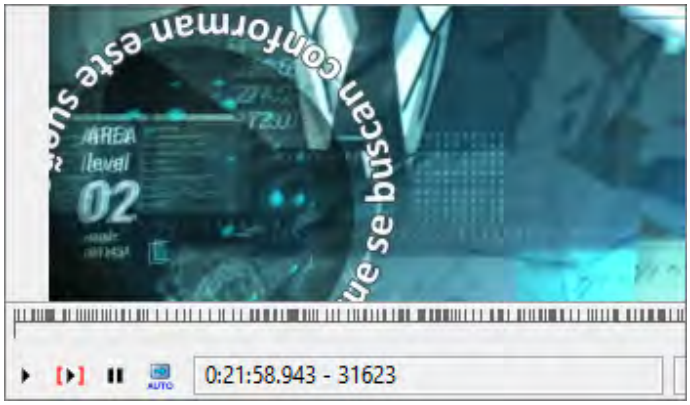
Ya podemos tomar el código de la **shape** y usarlo entre comillas, ya sean simples o dobles, para definir una variable en la celda de texto "**Variable**" o usarlo directamente como el primer parámetro de la función:



O si queremos, usamos el código de la **shape** recién creada y lo pegamos dentro de un clip vectorial, para que la función solo afecte a la línea o líneas que contengan a dicho clip:

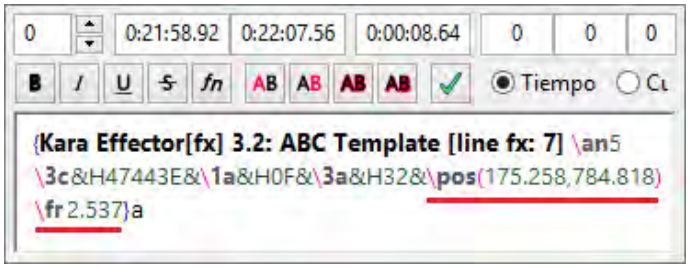


Y al aplicar, notamos cómo el texto está ligeramente por fuera de los límites del vídeo gracias a la extensión que le hicimos al trazar la shape. Así, al darle movimiento creamos el efecto de que el texto va apareciendo:



La función **text.bezier** retorna dos tags, uno de posición y otro de rotación:

Ejemplo:



El tag de posición retornado depende del modo en que usemos la función, y las cuatro opciones son: **\pos, \move, \moves3** y **\moves4**. Y el tag de rotación siempre será el **\fr** que es el mismo **\frz** (**font rotation in axis "z"**).

Con este ejemplo damos por terminada la documentación de la función **text.bezier**, que como ya habrán notado, tiene muchas aplicaciones que espero hayan sido de su agrado. No descarto más adelante agregarle más modos y opciones para ampliar aún más esas posibilidades.

Es todo por ahora para el **Tomo XXXIV**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXV]

Kara Effector 3.2:

En este **Tomo XXXV** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también para la edición de las líneas de subtítulos.

Recursos [KE]:

Recursos [KE]: → función

tag.module1(...)

Esta función es una entre una serie de cuatro de la librería tag enfocadas a aplicar tags respecto a los módulos del **KE**. Los módulos del **Kara Effector** son:

1. module
2. module1
3. module2
4. moduler

tag.module1 agrega una secuencia de tags a los efectos basado en el **module1** según el **Template Type**, que es una interpolación de 0 a 1 respecto al objeto karaoke. El **module1** solo es aplicable a los modos inferiores a **Line**:

Template Type	module1
Word	(word.i - 1) / (word.n - 1)
Syl	(syl.i - 1) / (syl.n - 1)
Furi	(furi.i - 1) / (furi.n - 1)
Char	(char.i - 1) / (char.n - 1)
Convert to Hiragana	(hira.i - 1) / (hira.n - 1)
Convert to Katakana	(kata.i - 1) / (kata.n - 1)
Convert to Romaji	(roma.i - 1) / (roma.n - 1)
Translation Word	(word.i - 1) / (word.n - 1)
Translation Char	(char.i - 1) / (char.n - 1)

Cada uno de los parámetros de esta función debe ser una **tabla** con tres elementos, en el siguiente orden:

1. **tag**
2. **valor inicial**
3. **valor final**

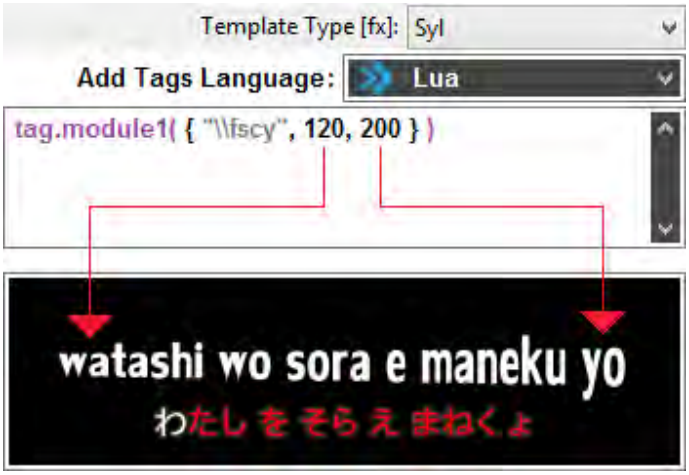
Ejemplo:

- **{ "\\1c", "&HFFFFFF&", "&H000000&" }**
- **{ "\\blur", 2, 6 }**
- **{ "\\fscx", 120, 250 }**

- **{ “\\alpha”, “&HFF&”, “&HAF&” }**
- **{ “\\frx”, 45, 135 }**
- **{ “\\fsp”, -5, 8 }**
- **{ “\\fry”, -90, 90 }**

Ejemplo:

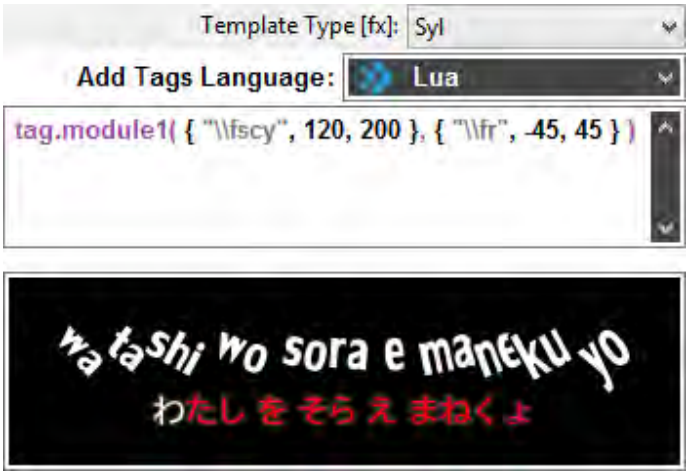
Notemos cómo el tamaño en el eje “**y**” (**\\fscy**) de las sílabas aumenta progresivamente desde el 120% de su tamaño por default hasta 200%; desde **syl.i = 1** hasta **syl.n**:



Para el siguiente ejemplo veremos cómo agregar dos o más parámetros en la función:

Ejemplo:

Ahora no solo crece progresivamente respecto al eje “y”, sino que también rota progresivamente respecto al eje “z” desde los -45° hasta los 45°:



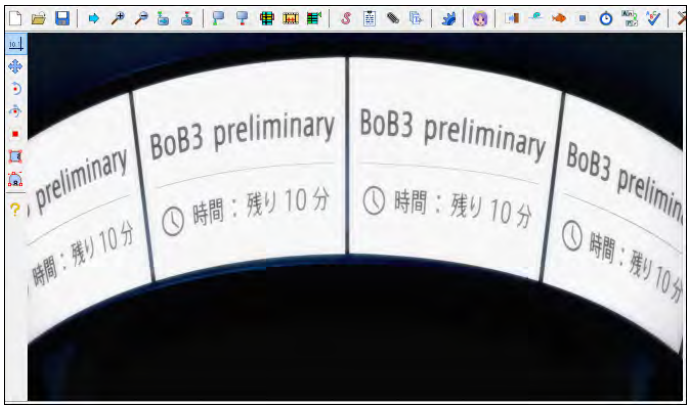
Esta función, al igual que las otras tres similares:

- **tag.module**
- **tag.module2**
- **tag.module3**

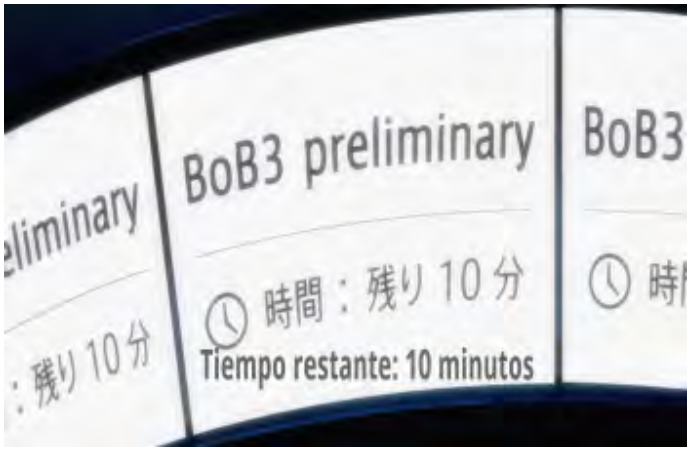
tiene la ventaja de poder ingresar la cantidad de parámetros que necesitemos, y como veremos en el próximo ejemplo, podemos complementar otras funciones con ella.

Ejemplo:

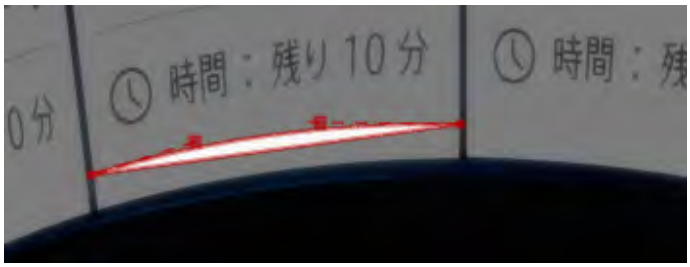
Supongamos que queremos hacer un cartel curvo como el de la siguiente imagen:



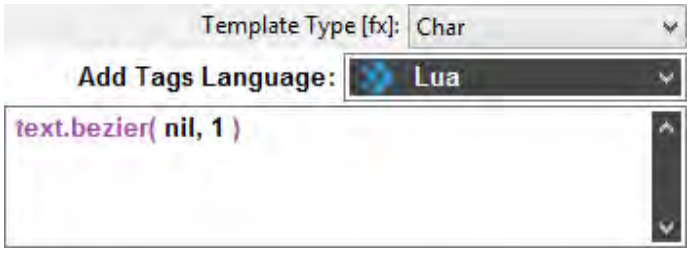
Empezamos poniendo el texto cerca, modificando el estilo del mismo hasta que quedemos satisfechos con el tamaño y color con los mismos:



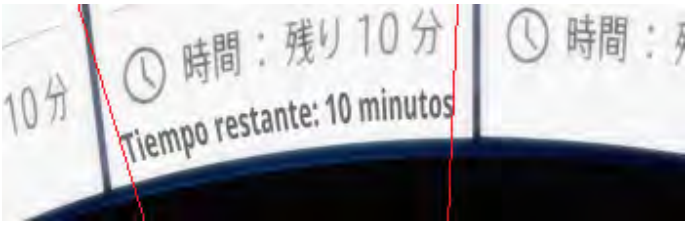
Luego trazamos el clip vectorial para que al usar la función **text.bezier**, el texto adopte su forma:



En el primer parámetro de la función ponemos “**nil**” para que se tome como shape a clip vectorial recientemente trazado:



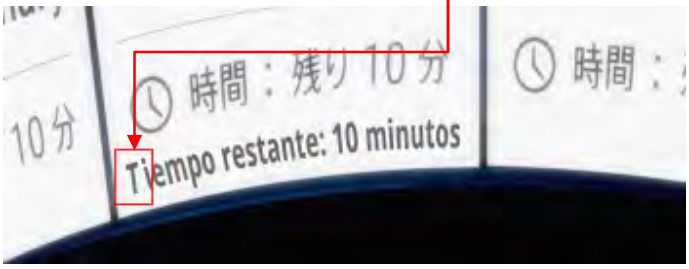
Al aplicar, notamos que efectivamente el texto adoptó la forma curva que esperábamos, pero si nos fijamos en las líneas rojas de la siguiente imagen, podemos ver que a pesar de que el texto tiene la inclinación correcta, no tiene la perspectiva misma que tiene el cartel en el vídeo:



Entonces, lo que debemos hacer con las líneas generadas es dar las perspectivas correctas a la primera y última letra de dichas líneas fx. En este caso usaremos el tag **\fax**:

Cartel			✱Tiempo restante: 10 minutos
Cartel	lead-in	Effector [Fx]	✱T
Cartel	lead-in	Effector [Fx]	✱i
Cartel	lead-in	Effector [Fx]	✱e
Cartel	lead-in	Effector [Fx]	✱m

Ubicamos la primera línea fx generada y manualmente le vamos dando valores al tag **\fax** hasta que la letra en esta línea tenga la perspectiva correcta. Este tag hace que el objeto karaoke se incline hacia la derecha para palores negativos, y hacia la izquierda para los valores positivos. Casi nunca estos valores se salen del rango [-1, 1]:



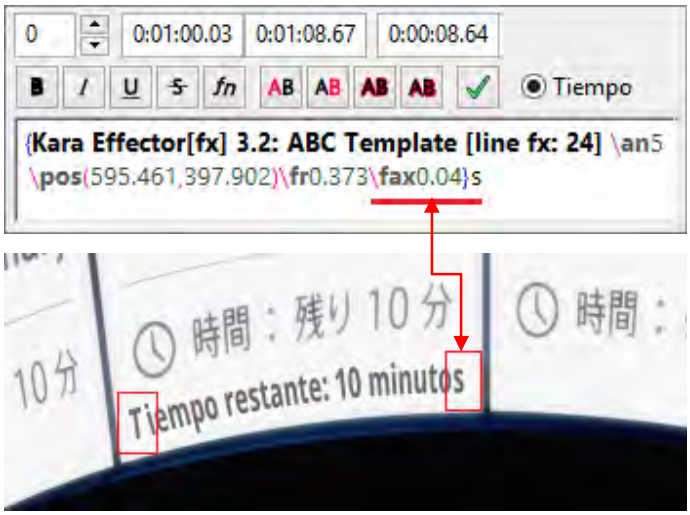
Una vez hayamos encontrado el valor correcto del \fax de la primera letra, lo usaremos más adelante como el valor inicial en la función **tag.module1**:

tag.module1({ "\\fax", -0.28, valor_final })

Ahora nos ponemos en la búsqueda del valor final y para ello seleccionamos a la última línea de fx generada:

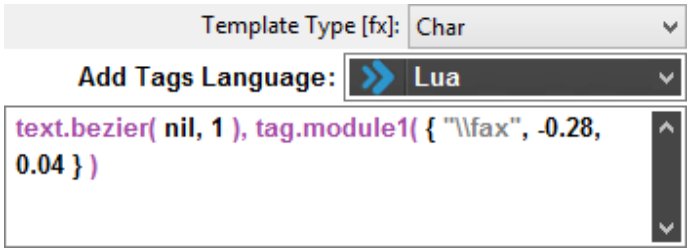
Cartel	lead-in	Effector [Fx]	✱t
Cartel	lead-in	Effector [Fx]	✱o
Cartel	lead-in	Effector [Fx]	✱s

Ponemos el tag **\fax** y le ponemos valores hasta dar con el correcto (no son más de tres intentos):



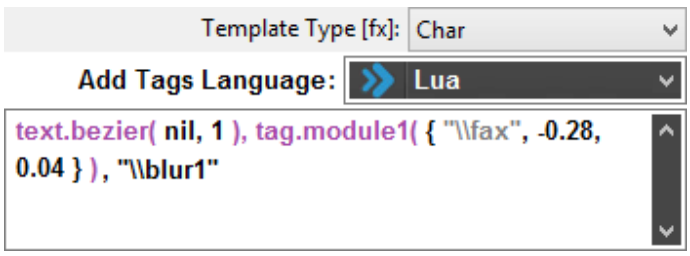
Entonces tenemos: **tag.module1({ "\\fax", -0.28, 0.04 })**

Ahora agregamos la función **tag.module1** en **Add Tags**, justo después de la función **text.bezier** que ya habíamos puesto:

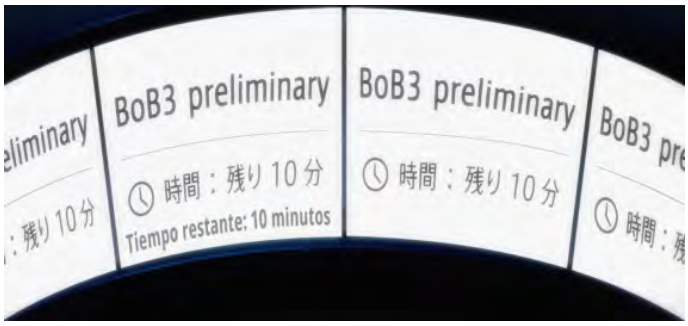


Ahora al aplicar, vemos que la función **tag.module1** sirvió como complemento de **text.bezier** para que el texto se adaptara mejor a las condiciones del cartel:

Una vez la perspectiva del texto nos deje satisfechos, ya solo es cuestión de cambiar lo que sea necesario para que el estilo sea lo más parecido al texto del cartel, como por ejemplo la opacidad. Ejemplo:



Y este sería el resultado final:



Aplicado a las cuatro secciones del cartel:

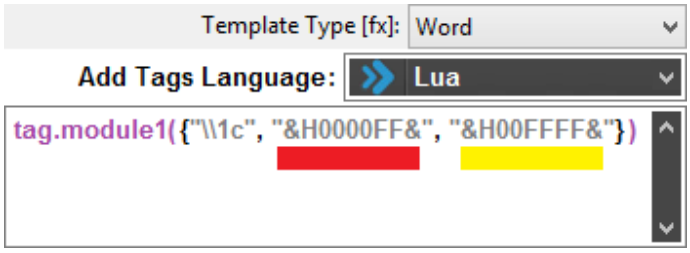
La función **tag.module1** interpola los valores de inicio y final ingresados en cada uno de sus parámetros desde el inicio hasta el final de la línea, sin importar el **loop**:

Ejemplo:

A un **Template Type: Word** le ponemos un cuadrado en la parte superior con un **loop** 3, con diferentes posiciones para para a cada uno de ellos:

En la anterior imagen se ve cómo cada palabra (**Word**) tiene encima de ella a los tres cuadrados (**loop 3**).

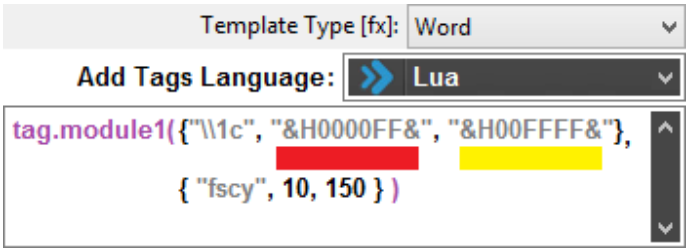
Aplicamos la función de tal manera que afectemos el color primario (**\\1c**) de los cuadrados:



Entonces la función interpola los colores desde el rojo hasta el amarillo, desde la primera palabra hasta la última y a través de cada uno de los tres **loops** de los cuadrados de 10 x 10 px, como se ve en la siguiente imagen:

Podemos agregar otro parámetro a la función para intentar dejar un poco más claro cómo se interpolan los valores de palabra en palabra y de **loop** en **loop**:

Ejemplo:

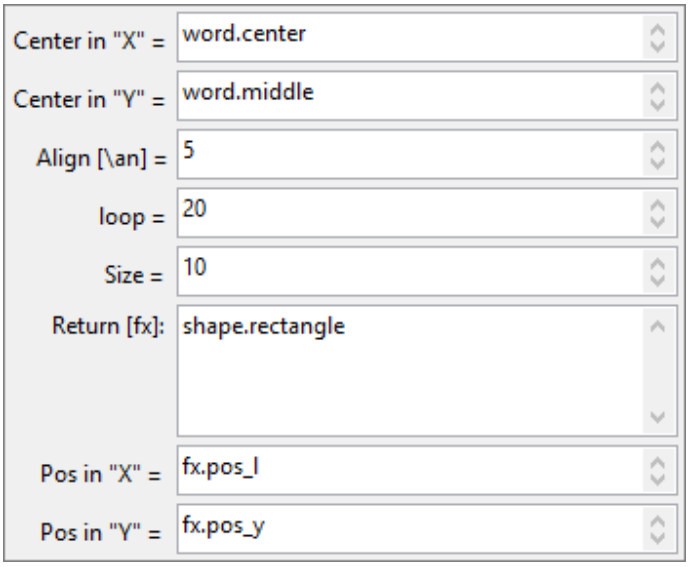


tag.module(...)

Esta función interpola los dos valores de uno o más tags de cada uno sus parámetros, en referencia a la variable **module** del **KE**.

A diferencia de la función anterior que interpolaba los dos valores de cada tag ingresado apoyada en la variable **module1**, es decir que lo hacía a lo largo de los elementos de la línea según el **Template Type**; la función **tag.module** interpola los dos valores en referencia a los **loops** de cada uno de los elementos de la línea.

Ejemplo:



Y como es de suponer, elegimos un **Template Type: Word** y en **Add Tags** ponemos lo siguiente:

Template Type [fx]: Word

Add Tags Language: Lua

```
format("\\org(%s,%s)", fx.pos_x, fx.pos_y),
tag.module( { "\\frz", 18, 360 } )
```

Al aplicar, los 20 cuadrados se dispondrán desde el centro de cada palabra, y partiendo desde la parte izquierda de las mismas, en un círculo de manera inversa al movimiento de las manecillas del reloj:

Podemos añadir otro tag, para ver cómo la función interpola sus valores respecto al **loop**, a diferencia de la función anterior que lo hacía a lo largo de la línea karaoke:

Template Type [fx]: Word

Add Tags Language: Lua

```
format("\\org(%s,%s)", fx.pos_x, fx.pos_y),
tag.module( { "\\frz", 18, 360 },
            { "\\1c", shape.color1, shape.color3} )
```

tag.module2(...)

Esta función interpola los valores de inicio y final de un tag ingresado en uno o más de sus parámetros, basado en la variable **module2** del **KE**, es decir, desde **line.i = 1**, hasta **line.n**

Ejemplo:

Template Type [fx]: Char

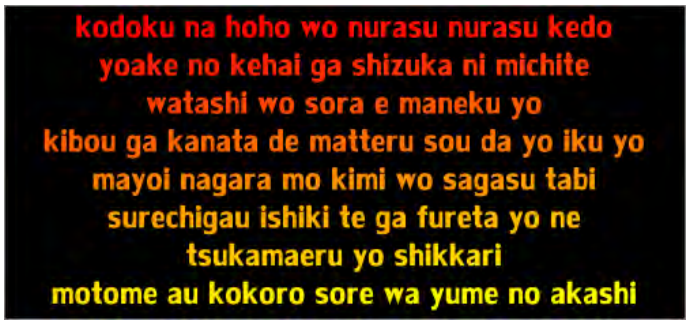
Add Tags Language: Lua

```
tag.module2({"\\1c", "&H0000FF&", "&H00FFFF&"})
```

Seleccioné todas las líneas de estilo "**Romaji**", que en este caso son 8:

Estilo	Texto
Romaji	✱m 0 0 l 0 100 100 100 100 0
Romaji	✱ko✱do✱ku ✱na ✱ho✱ho ✱wo ✱nu✱ra✱su ✱nu✱ra✱su ✱ke✱do
Romaji	✱yo✱a✱ke ✱no ✱ke✱ha✱i ✱ga ✱shi✱zu✱ka ✱ni ✱mi✱chi✱te
Romaji	✱wa✱ta✱shi ✱wo ✱so✱ra ✱e ✱ma✱ne✱ku ✱yo
Romaji	✱ki✱bo✱u ✱ga ✱ka✱na✱ta ✱de ✱ma✱t✱te✱ru ✱so✱u ✱da ✱yo ✱i✱ku ✱yo
Romaji	✱ma✱yo✱i ✱na✱ga✱ra ✱mo ✱ki✱mi ✱wo ✱sa✱ga✱su ✱ta✱bi
Romaji	✱su✱re✱chi✱ga✱u ✱i✱shi✱ki ✱te ✱ga ✱fu✱re✱ta ✱yo ✱ne
Romaji	✱tsu✱ka✱ma✱e✱ru ✱yo ✱shi✱k✱ka✱ri
Romaji	✱mo✱to✱me ✱a✱u ✱ko✱ko✱ro ✱so✱re ✱wa ✱yu✱me ✱no ✱a✱ka✱shi
Hiragana	✱こ✱ど✱く ✱な ✱ほ✱ほ ✱を ✱ぬ✱ら✱す ✱ぬ✱ら✱す ✱け✱ど
Hiragana	✱ょ✱あ✱け ✱の ✱け✱は✱い ✱が ✱し✱ず✱か ✱に ✱み✱ち✱て

Al aplicar el efecto, notamos que cada letra de cada una de las líneas va cambiando progresivamente, desde el rojo hasta el amarillo, desde **line.i = 1** hasta **line.n**:



tag.moduler(...)

Cuarta y última de las funciones de la **librería tag** enfocadas en aplicar gradualmente los valores de inicio y final de uno o más tags ingresados como alguno de sus parámetros.

Esta función interpola dichos valores respecto a la variable **moduler**, que es una variable con los valores equidistantes entre 0 y 1 según la cantidad de repeticiones hechas con la función **replay** del **KE**.

Ejemplo:

Seleccionamos el efecto **[001] ABC Template Hilight Syl** de la librería de efectos **hi-light [fx]** y comenzamos a hacer las siguientes modificaciones:

```
loop = 20
Size = 20
Return [fx]: shape.circle
```

Creamos dos nuevas variables:

```
Variables:
mi_angle = R(360);
mi_radius = R(60,80)
```

Modificamos las posiciones:

```
Pos in "X" = fx.pos_x + math.polar(var.syl.mi_angle,
var.syl.mi_radius*(1 - module), "x")
Pos in "Y" = fx.pos_y + math.polar(var.syl.mi_angle,
var.syl.mi_radius*(1 - module), "y")
```

Modificamos los tiempos del fx con el fin de crear un efecto tipo **pre-hilight**:

```
Line Start Time = l.start_time + syl.start_time - 400*(1 - module)
Line End Time = fx.start_time + 380
```

Y por último, añadimos unos cuantos tags para los estilos de la **shape** y llamamos a la función **tag.moduler**, así:

```
Add Tags:  Add Tags Language: Lua
"\\bord0\\blur2\\t(\\fscx5\\fscy5)\\fad(50,120)",
tag.moduler( {"\\1c", "&H00FFFF&", "&H0000FF&" } )
```

Entonces, a cada sílaba le llega una especie de rayo justo antes de ser karaokeada, y como no repetimos el fx ni una sola vez, solo se verá de color amarillo para todas las sílabas:

Ahora repetimos el fx con la función **replay**, de manera que a cada sílaba ahora le lleguen tres rayos en vez de uno:

```
Add Tags:  Add Tags Language: Lua
replay( 3 ), "\\bord0\\blur2\\t(\\fscx5\\fscy5)\\fad(50,120)",
tag.moduler( {"\\1c", "&H00FFFF&", "&H0000FF&" } )
```

Hecho de esta manera, ya podemos ver la interpolación de los colores entre el amarillo y el rojo, y los tres rayos que le llegan a cada una de las sílabas:

Se nota claramente la interpolación de los colores hecha por la función, un color diferente para cada rayo.

Es todo por ahora para el **Tomo XXXV**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXVI]

Kara Effector 3.2:

En este **Tomo XXXVI** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también para la edición de las líneas de subtítulos.

Recursos [KE]:

Recursos [KE]: → **función**

text.inside(inside, Text)

Esta función retorna **true** o **false** (verdadero o falso) al verificar si un **string** de texto del parámetro **inside**, hace parte parcial o total de otro **string** ingresado en el parámetro **Text**.

El parámetro **inside** es el string que la función buscará para determinar si el resultado es verdadero o falso.

El parámetro **Text** es el texto y/o **string** en el que la función buscará al parámetro **inside** para determinar si éste hace parte o no de él. Su valor por default es el texto por default dependiendo del **Template Type**.

La particularidad que tiene esta función de retornar solo **true** o **false** le impide que sea llamada directamente en un efecto ya que ambos son valores booleanos, pero sí la podemos usar como una condición de verdad dentro de otras funciones.

Ejemplo:

Template Type [fx]: Syl

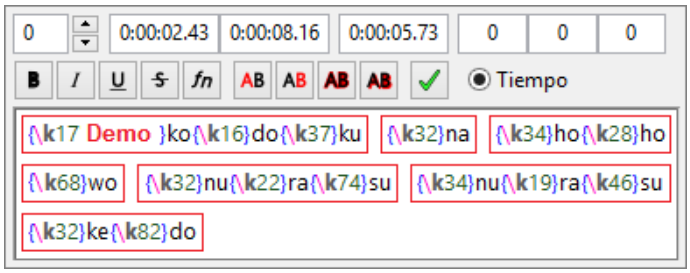
Add Tags Language: Lua

```
tag.only( text.inside( "o", syl.text ), "\\fscy200" )
```

La función **text.inside** verifica si la letra “o” hace parte de cada una de las sílabas, en caso de ser verdadero, aplica el tag de la función **tag.only**:

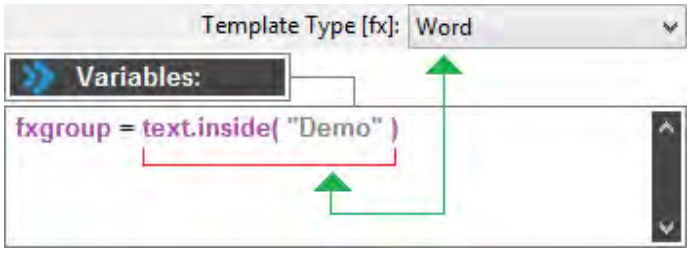
Ejemplo:

Ponemos una marca en una de los tags karaoke de una o más sílabas:



En la imagen anterior se resaltan las palabras de una línea karaoke, y en el tag karaoke de la sílaba "**ko**" puse una palabra "**Demo**".

Esa marca la usaremos para aplicar un efecto únicamente a la palabra que contenga dicho **string**:



Y al aplicar veremos cómo se generó una única línea fx correspondiente a la palabra "**kodoku**" que era la única a la que le incluí el **string** "**Demo**":

Estilo	Actor	Efecto	Texto
English			Me pierdo en el camino cada vez que
English			Siento los pensamientos que dejaste
English			Me aferraré a ti y nunca te soltaré
English			Dos corazones que se buscan confor
Romaji	lead-in	Effector [Fx]	✱kodoku

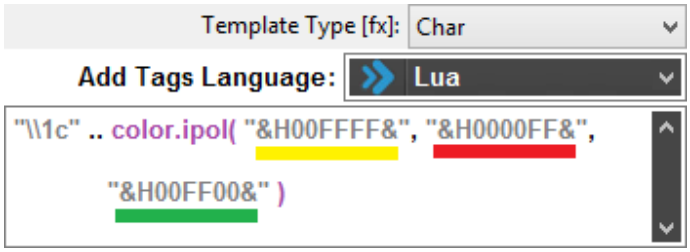
La ventaja de esta función es que puede verificar si un string del parámetro **inside** está, no solo en el texto de un objeto karaoke de la línea, sino que también dentro de sus tags. Esta ventaja la podemos usar para poner marcas en el script y luego aplicar uno o más efectos especiales solo en donde las pusimos. La cantidad de marcas y la diversidad de ellas, depende solo de la necesidad de cada uno.

Recursos [KE]: → función

color.ipol(...)

Esta función crea un gradiente (**degradación**) entre dos o más colores ingresados en ella, para posteriormente asignarle un único color a cada uno de los elementos de la línea.

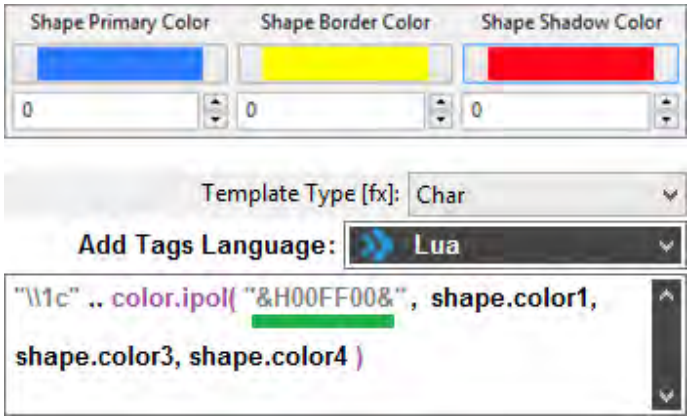
Ejemplo:



Al aplicar, notamos cómo cada una de las letras, dado que el **Template Type** es **Char**, hace el gradiente ente los tres colores, empezando por el amarillo, pasando por el rojo y llegando hasta el verde:

Ejemplo:

Podemos sacar provecho de uno o más de los colores que en principio son para las Shapes:



Usamos 4 colores, primero el verde, y luego los tres colores que modificamos en la segunda Ventana del **KE**, y luego de aplicar veremos algo muy similar a esto:

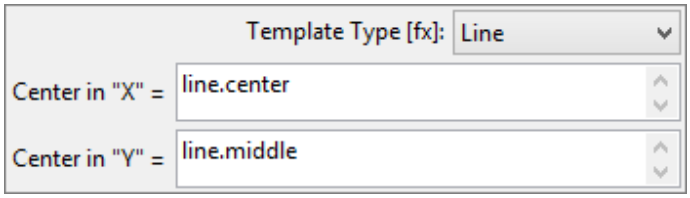
Recursos [KE]: → función

color.loop(...)

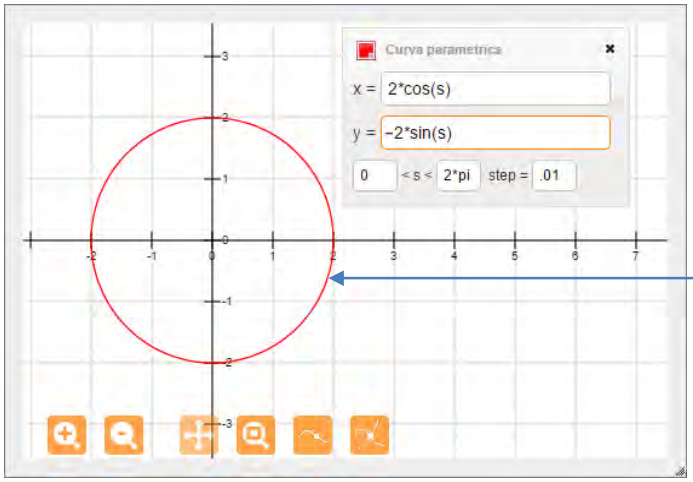
Esta función es similar a la anteriormente vista, pero con la diferencia de que crea el gradiente (**degradación**) entre todos los colores ingresados respecto al **loop** y no a los componentes karaokes de la línea.

Ejemplo:

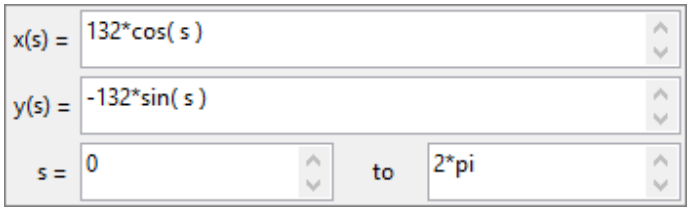
Seleccionamos un **Template Type: Line**



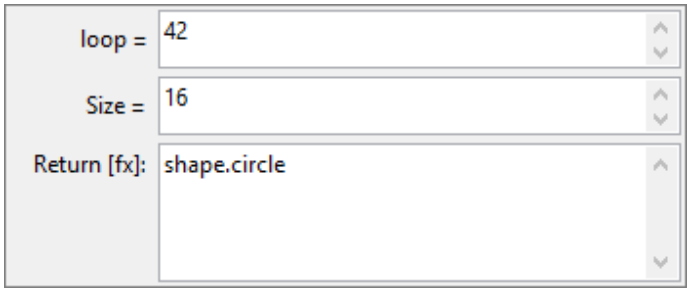
Ingresaremos la siguiente función paramétrica:



Aumentamos un poco la escala, ya que un círculo de 2 px de radio sería muy pequeño:



Aumentamos el **loop**, modificamos el tamaño de la shape y ponemos **shape.circle** en **Return [fx]:**



Y usando los mismos cuatro colores del ejemplo anterior, llamamos a la función **color.loop**:

Los resultados son:

Entonces la función genera el gradiente a través de los 42 **loops** que habíamos puesto de la **shape shape.circle**, empezando por el verde, pasando por el azul y el amarillo y terminando en el rojo.

Recursos [KE]: → función

math.point(n, x_range, y_range, start_x, start_y, end_x, end_y)

Esta función crea una **tabla** con una **n** cantidad de puntos aleatorios bajo ciertas condiciones controladas por el resto de los parámetros o por una serie de valores por default.

El parámetro **n** es un número entero mayor a cero, es la cantidad de puntos que contendrá la **tabla** generada. Cada dos elementos de la tabla equivalen a un punto. Ejemplo:

tabla = { Px1, Py1, Px2, Py2, Px3, Py3, ... }

Los parámetros **x_range** y **y_range** son 2 números reales que corresponden a la máxima distancia del punto (0, 0) en la que aleatoriamente se generarán los puntos. Su valor por default es el mismo para ambos: **2.5*l.fontsize**

Con los parámetros **start_x** y **start_y** podemos decidir cuál será el primer punto de la **tabla** generada por la función, y sus valores por default equivalen a un punto generado al azar, dentro de los rangos seleccionados.

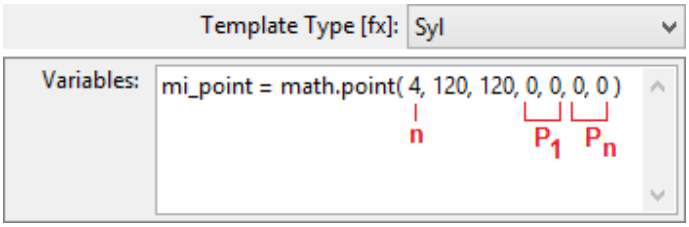
Con los parámetros **end_x** y **end_y** podemos decidir cuál será el último punto de la **tabla** generada por la función, y

sus valores por default equivalen a un punto de origen del sistema cartesiano P = (0, 0).

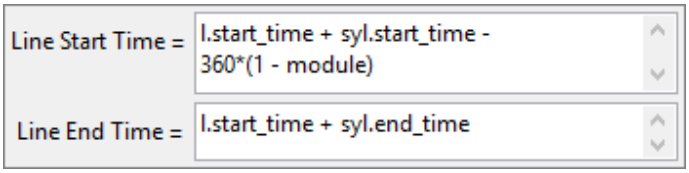
Ejemplo:

Definimos una variable utilizando la función para crear una **tabla** de cuatro puntos:

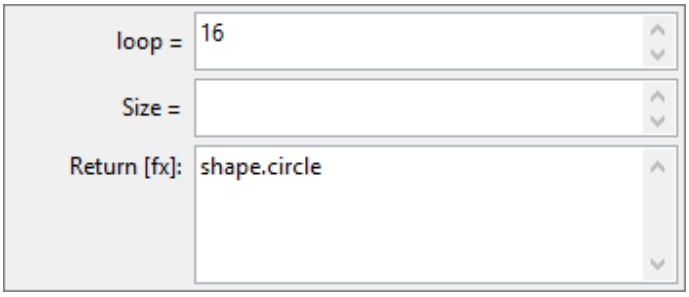
- n = 4
- rango en “x” = 120 px
- rango en “y” = 120 px
- punto inicial P1 = (0, 0)
- punto final P4 = (0, 0)



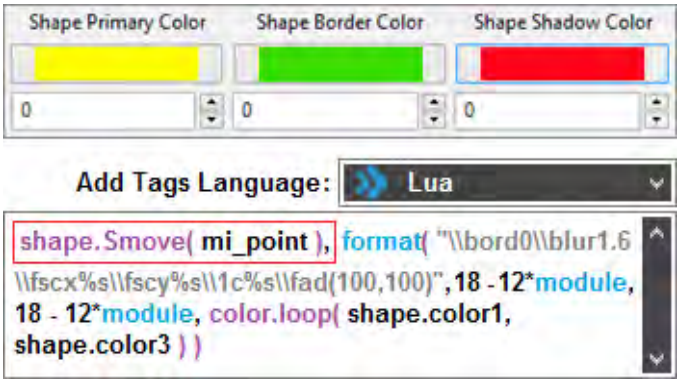
Modificamos los tiempos con la intensión de hacer un efecto tipo **hi-light**:



Aumentamos el **loop**, para este ejemplo en 16, y en **Return [fx]** ponemos **shape.circle**:



Hechas estas configuraciones, usaremos la **tabla** generada dentro de la función **shape.Smove**, también usaremos la función que vimos hace poco, **color.loop** con el color primario y el de borde de la **shape**:



Los cuatro puntos creados por la función **math.point** hacen que la función **shape.Smove** genere una curva **Bézier** con ellos y luego genere un movimiento siguiendo dicha curva:

Más adelante veremos otras ventajas de poder generar n cantidad de puntos de forma aleatoria, y la forma de sacarle provecho para nuestros efectos.

Recursos [KE]: → **función**

math.bezier(Return, Point_or_Shape)

Esta función crea la trayectoria de una curva **Bézier** formada por todos los puntos de una **tabla** ingresada en el parámetro **Point_or_Shape**, o por el código de una **shape** ingresada en el mismo parámetro.

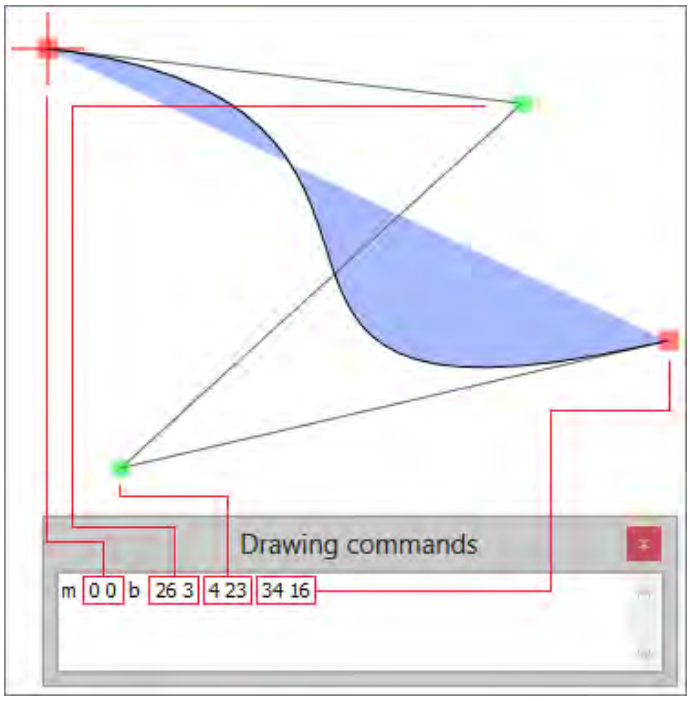
El parámetro **Return** decide lo que va a retornar la función, y tiene las tres siguientes opciones:

- **“x”**: retorna la coordenada “x” de la posición
- **“y”**: retorna la coordenada “y” de la posición
- **nil**: retorna ambas posiciones

El parámetro **Point_or_Shape** puede ser una **tabla** con dos o más puntos cartesianos o el código .ass de una **shape**.

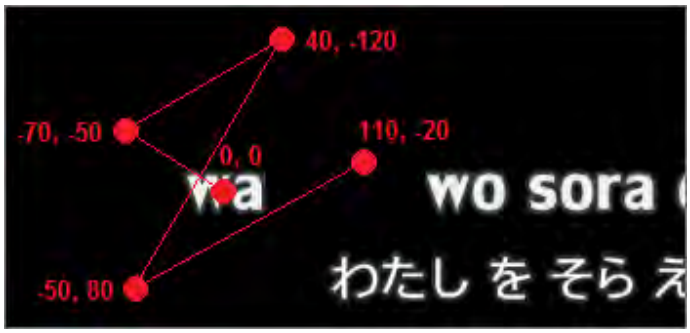
Ejemplo:

Un ejemplo sencillo es ver cómo el **ASSDraw3** traza una curva **Bézier** a partir de cuatro puntos:

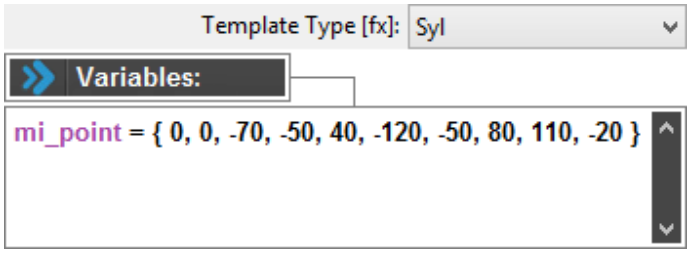


Ejemplo:

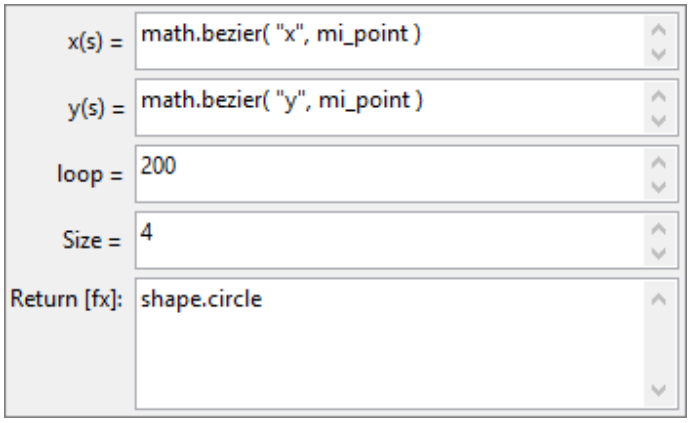
La siguiente imagen muestra cinco puntos con centro en una de las sílabas:



Con las coordenadas de esos puntos declaramos una **tabla** en la celda de texto **Variables**:



Y hacemos las siguientes configuraciones:

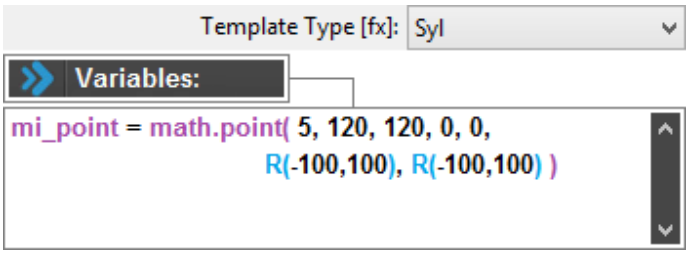


Cuando apliquemos, la función traza una curva **Bézier** que corresponde a los cinco puntos de la **tabla** que declaramos:

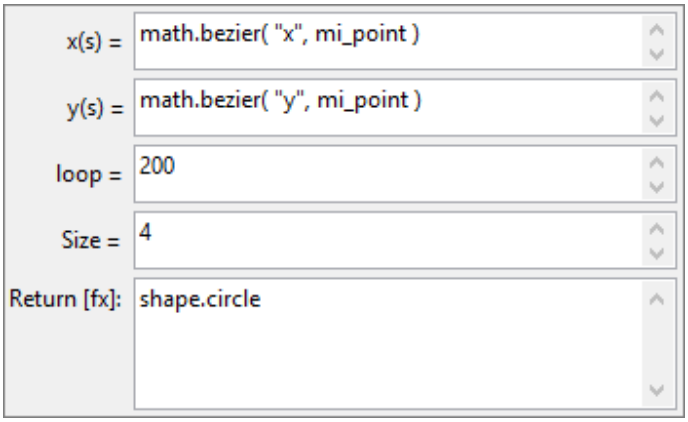
La desventaja de que los puntos sean fijos (constantes), es que la función siempre hará el mismo trazado de la curva **Bézier** en todas las sílabas.

Para generar puntos al azar, volvemos a definir la **tabla** de puntos y nos apoyamos en la función **math.point**:

Ejemplo:



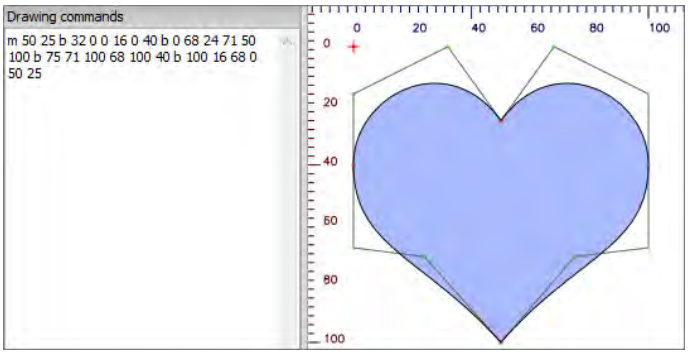
La función **math.bezier** se llama en las celdas **x(s)** y **y(s)**. El **loop** también se puede modificar a gusto:



Le cambié los colores para identificar la curva **Bézier** que se generó para cada una de las sílabas:

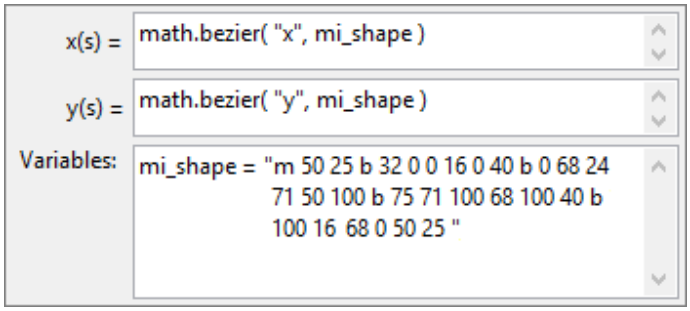
Conservando las configuraciones del "**Size**" y del "**Return [fx]**" del ejemplo anterior, cambiamos el segundo parámetro de la función, de una **tabla** de puntos a una **shape**, ampliamos las posibilidades:

Ejemplo:



Cuando el segundo parámetro de la función era una **tabla**, debíamos asignar el valor del **loop** para dibujar el trazo de la curva **Bézier** generada, pero cuando dicho parámetro es una **shape**, la función calcula internamente la longitud de la misma y asigna un **loop** de forma automática basándose en el valor obtenido. Lo que quiere decir que al usar la función con una **shape** como segundo parámetro, ya no importa lo que pongamos en la celda de texto "**loop**" ya que este valor lo decide la función.

Usamos el código de la **shape** para definir una variable y posteriormente usarla dentro de la función:



Y la función copiará el contorno de la **shape** a la misma escala y en su misma posición, ahora relativa al centro del objeto karaoke de la línea, en este ejemplo, la sílaba:

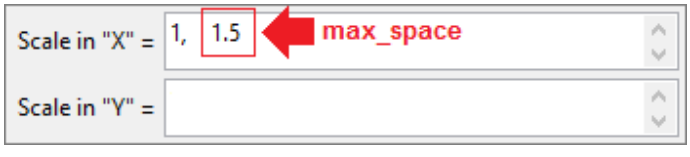
Cuando el segundo parámetro de la función es una shape, la función calcula el loop apoyándose en una variable interna del **KE**:

max_space = 1 ← valor por default (1 px)

Esta variable interna del **KE** indica la distancia en pixeles que separa a cada uno de los objetos karaokes que trazan al contorno de la **shape** ingresada, asumiendo que éstos tienen un área de **1 px^2**.

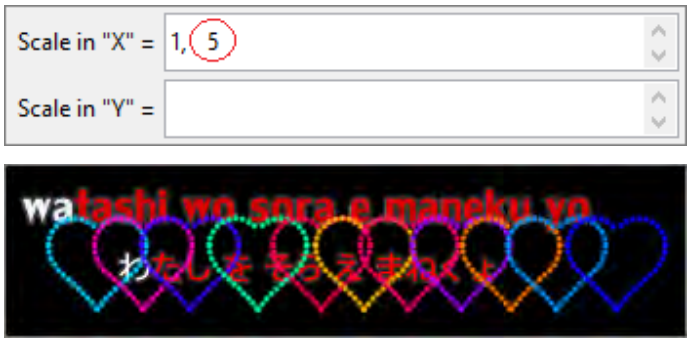
El valor de esta variable la podemos modificar desde el segundo argumento de la celda de texto **Scale in "X":**

Ejemplo:



En este ejemplo aumentamos el valor de **max_space** de 1 a 1.5, lo que aumentará la distancia que separa a los objetos karaoke que trazan en contorno de la shape, y por ende disminuirá el **loop** generado por la función. El valor de **max_space** debe ser un número real mayor a cero y siempre éste será inversamente proporcional al **loop**.

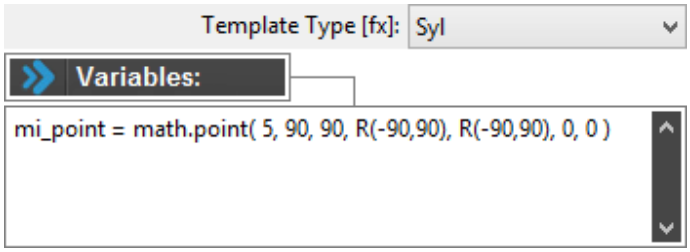
Al aumentar la variable **max_space** en 5 px, es notorio la disminución del **loop**. Ejemplo:



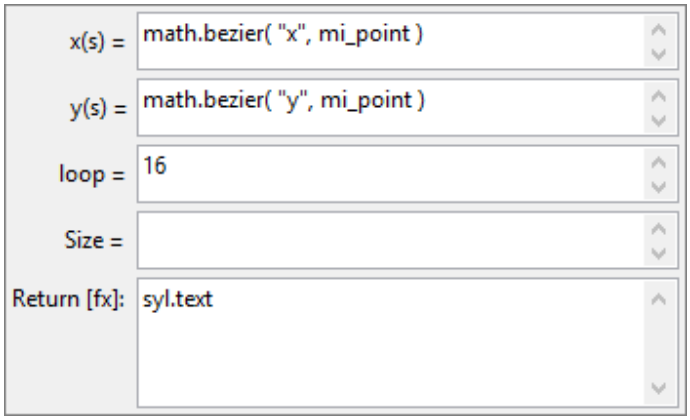
Para trazar una curva **Bézier** o trazar el contorno de una **shape** ingresada, no necesariamente debemos hacerlo con una **shape** en **Return [fx]**, también lo podemos hacer con el texto o con cualquier otro **string** que nos imaginemos.

Ejemplo:

Con un **Template Type: Syl** definimos una **tabla** de puntos en la celda de texto "**Variables**":



Dejamos el **loop** con una cantidad constante, 16 para este caso, y en **Return [fx]** dejamos el texto por default en vez de por alguna **shape** que trace la curva **Bézier** generada por los puntos de la **tabla**:



Al aplicar vemos cómo la curva es trazada por las sílabas:

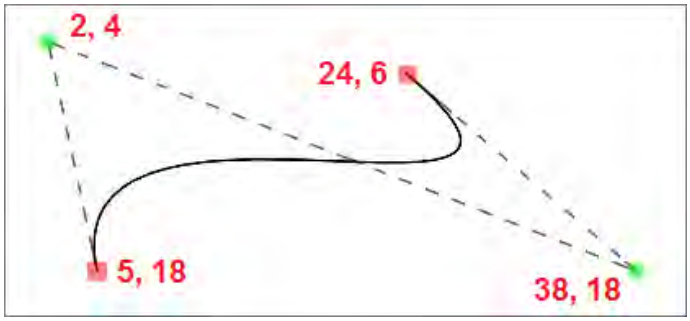
Recursos [KE]: → función

math.length_bezier(Points)

Esta función retorna la longitud medida en pixeles de una curva **Bézier** teniendo como referencia a los puntos de la **tabla** del parámetro **Points**.

Ejemplo:

Los siguientes cuatro puntos dibujan la curva **Bézier** que vemos en la gráfica:



Definimos los puntos en una **tabla**:

mi_point = { 5, 18, 2, 4, 38, 18, 24, 6 }

Por último, usamos la función para determinar la longitud de esta curva **Bézier** en particular:

math.length_bezier(mi_point) = 32.505 px

La **tabla** del parámetro **Points** de la función debe tener al menos un punto, lo que retornará 0 px como longitud, de ahí en adelante calculará la longitud de la **Bezier** generada.

El conocer la longitud de una curva **Bézier**, sin importar el número de puntos que la generan, nos ayudará a decidir un mejor número para el **loop** de los ejemplos anteriores.

Ejemplo:

Para el ejemplo en el que usamos a la función **math.point** para generar puntos al azar que dibujaran la curva **Bézier**, obviamente las generó de diferentes longitudes:

Pero para todas ellas el **loop** que usamos siempre fue de 200. Si por algún motivo la curva fuese lo suficientemente grande, entonces esos 200 **loops** no alcanzarían y el trazo de la **Bézier** no sería continuo. Es en estos casos que la función **math.length_bezier** nos es de gran utilidad, así:

```
loop = 0.75 * math.length_bezier( mi_point )
```

Y lo que quiere decir este simple ejemplo es que el **loop** será equivalente al 75% de la longitud de la curva **Bézier** que lleguen a generar los puntos de la **tabla mi_point**.

Para usar la longitud de una shape en el **loop**, podemos poner en dicha celda así:

```
loop = 0.8 * shape.length( mi_shape )
```

Es decir, que el **loop** será equivalente al 80% de la longitud de la **shape** declarada en la celda de texto "**Variables**" o de la **shape** que le ingresemos directamente.

Es todo por ahora para el **Tomo XXXVI**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.2:
Effector Book
Vol. II [Tomo XXXVII]

Kara Effector 3.2:

En este **Tomo XXXVII** continuaremos viendo más de los Recursos disponibles en el **Kara Effector**, que espero que con la ayuda de esta documentación, le puedan sacar el máximo provecho a la hora de llevar a cabo sus proyectos, no solo karaokes, sino también para la edición de las líneas de subtítulos.

Recursos [KE]:

Recursos [KE]: → Actualización

Nuevos Modos de la Función R [KE]:

Los nuevos modos acá documentados de la función **R** están disponibles a partir de la versión **3.2.9.5** del **KE**, y tienen la finalidad de ampliar las posibilidades a la hora de generar valores numéricos aleatorios.

La primera actualización consiste en un tercer parámetro de la siguiente forma:

R(Val1, Val2, Step)

El parámetro **Step** es un número entero no mayor a la diferencia entre **Val2** y **Val1**, y lo que hace es marcar la distancia entre los valores que retornará la función.

Ejemplo:

R(40, 90, 10)

En este caso, lo que hace el **Step = 10** es que la función retorne un valor aleatorio entre 40 y 90, pero de 10 en 10, o sea que los posibles valores que podrían ser retornados por la función **R** son:

- 40
- 50
- 60
- 70
- 80
- 90

Ejemplo:

R(-21, 35, 7)

Ahora el **Step** tiene un valor de 7 unidades, entonces el valor retornado es un valor aleatorio entre -21 y 35, pero de 7 en 7.

El valor por default del parámetro **Step** es 1, y en el caso de usar este parámetro en la función, debe ser mayor que cero.

Usemos o no el parámetro **Step** en la función, ésta siempre retornará números enteros, y basado en esta particularidad, vienen las siguientes actualizaciones de la función:

- **Rd**
- **Rc**
- **Rm**

Rd hace que la función **R** retorne valores redondeados en décimas.

Ejemplo:

Rd(2, 10)

La función retorna un número aleatorio entre 2 y 10 con una precisión de hasta una décima:

- 2.6
- 5.4
- 9
- 3.1
- 10

Rc hace que la función **R** retorne valores redondeados en centésimas.

Ejemplo:

Rc(-3, 5)

La función retorna un número aleatorio entre -3 y 5 con una precisión de hasta una centésima:

- -1.63
- 1.48
- 0
- 0.31
- 4.92

Rm hace que la función **R** retorne valores redondeados en milésimas.

Ejemplo:

Rm(1, 2.43)

La función retorna un número aleatorio entre 1 y 2.43 con una precisión de hasta una milésima:

- 2.326
- 1
- 1.934
- 2

También podemos usar el parámetro **Step** en la función, en los tres anteriores modos documentados, lo que hará que las posibilidades aumenten aún más.

Todas estas actualizaciones de la función **R** nos servirán de apoyo para muchas de las funciones ya documentadas y para algunas más que aún no hemos visto, y que de a poco aprenderemos a sacarle el máximo provecho.

Recursos [KE]: — Actualización

Abreviaciones de algunos tags [KE]:

Las siguientes cinco abreviaciones hacen posible que al llamar un solo tag, podamos aplicar dos o más de ellos al mismo tiempo. Las abreviaciones son las siguientes:

- **\fscxy**
- **\frxy**
- **\frxz**
- **\fryz**
- **\frxyz**

\fscxy es equivalente a los tags \fscx y \fscy al tiempo, y hay dos formas diferentes de hacerlo. Ejemplo:

LUA:

- "\\fscxy120" = \fscx120\fscy120
- "\\fscxy80" = \fscx80\fscy80

Automation Auto-4:

- \fscxy200 = \fscx200\fscy200
- \fscxy25 = \fscx25\fscy25

O sea que siempre que usemos valores constantes en esta abreviatura, dichos valores serán los mismos para los tags a los que equivale. En el caso de que queramos poner un valor aleatorio (random), debemos poner la función junto con la abreviatura. Ejemplo:

LUA:

- "\\fscxyR(100,200)" = \fscx132\fscy157
- "\\fscxyRd(10,50)" = \fscx34.9\fscy18.7

Automation Auto-4:

- \fscxyRm(0,1) = \fscx0.854\fscy0.051
- \fscxyRc(23,37) = \fscx25.67\fscy33.78

Entonces, al adjuntarle a la abreviación la función random, garantizamos que a los tags equivalentes les corresponda un valor diferente para cada uno de forma aleatoria.

Las otras cuatro abreviaturas corresponden a los siguientes tags:

- **\frxy = \frx\fry**
- **\frxz = \frx\frz**
- **\fryz = \fry\frz**
- **\frxyz = \frx\fry\frz**

Y de forma similar a los ejemplos de vistos en la abreviatura \fscxy, son aplicables las mismas condiciones, tanto para los valores constantes, como para los valores aleatorios al usar la función **R** o alguna de sus nuevas modificaciones.

Estas cinco abreviaciones no consisten un efecto en sí mismas, pero nos ayudan a ahorrar trabajo en los mismos. Disponibles en el **KE** versión **3.2.9.5** o superior.

Recursos [KE]: → Actualización

tags progresivos [KE]:

Esta es otra de las actualizaciones que se pueden usar a partir de la versión **3.2.9.5** del **Kara Effector**, y lo que hace es interpolar todos los valores dentro de una tabla adjunta a un tag, respecto al **module1** y al **module**, según así lo dispongamos.

Ejemplo:

Template Type [fx]: Char

Add Tags Language: Lua

```
"\\fscy{200, 50, 120}"
```

Dado que es un **Template Type: Char**, cada una de los caracteres de la línea tomarán los valores en el tag \fscy empezando desde el 200%, pasando por el 50% hasta llegar al 120%:

La interpolación de los diferentes porcentajes en la escala de la Font respecto al eje “y” se hizo en relación al **module1**.

Ejemplo:

Template Type [fx]: Char

Add Tags Language: Automation Auto-4

```
\1c{"&H00FFFF&", "&H0000FF&", "&H16C047&",
    "&HFFFFFF&"}
```

Los colores de la tabla adjunta son:

- **&H00FFFF& = Amarillo**
- **&H0000FF& = Rojo**
- **&H16C047& = Verde**
- **&HFFFFFF& = Blanco**

De los dos ejemplos vistos anteriormente podemos deducir que al adjuntar normalmente la **tabla** al tag, los valores de la misma se interpolarán respecto al **module1**. Ahora para que los valores de la tabla se interpolen respecto al **module** (respecto al **loop**) debemos añadir el signo menos (-) justo en medio del tag y la **tabla**:

Ejemplo:

Template Type [fx]: Syl

Pos in "X" = `fx.pos_x + math.polar( 360*j/maxj, 70, "x" )`

Pos in "Y" = `fx.pos_y + math.polar( 360*j/maxj, 70, "y" )`

loop = 12

Return [fx]: shape.circle

Y en **Add Tags** ponemos:

Add Tags Language: Automation Auto-4

```
\fscxy-{10, 40}
```

Aplicando, con los tiempos de un **hi-light**, veremos esto:

Al usar una de las abreviaciones vistas hace poco se ahorra aún más trabajo, dado que:

\fscxy-{10, 40} = \fscx-{10, 40}\fscy-{10, 40}

Entonces los valores de la **tabla** se aplicarán a dichos tags de forma progresiva, interpolándose desde el 10% hasta llegar al 40%, como se puede ver en la imagen anterior.

A este mismo ejemplo le podemos aplicar la cantidad de tags progresivos, abreviaciones o normales que queramos, todo depende de los resultados que estemos necesitando, ejemplo:

\fscxy-{10, 40}\1c-{“&H00FFFF&”, “&H0000FF&”}

Las combinaciones posibles son infinitas, solo resta probar.

Recursos [KE]: Actualización

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:

Add Tags Language: Automation Auto-4

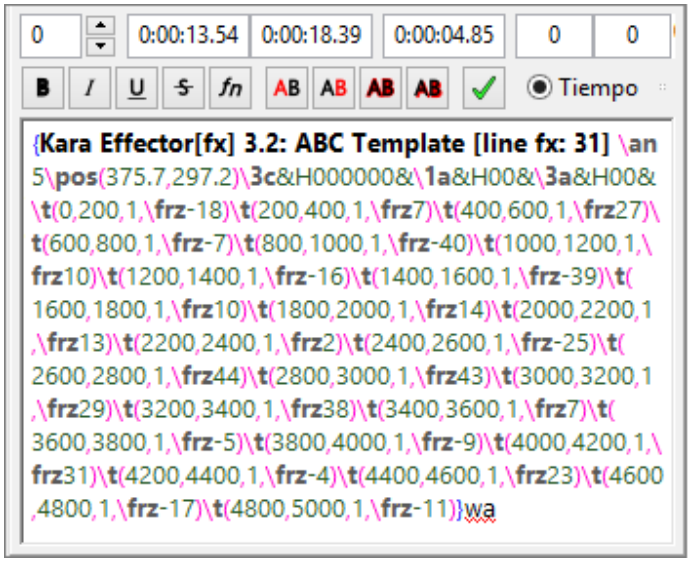
```
tag.oscill( fx.dur, 200, "\\frz( R(-45, 45) )" )
```

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:

Así se vería una de las líneas generadas:



Ejemplo:

Add Tags Language: Automation Auto-4

```
tag.oscill( fx.dur, 200, "\\fax( Rc(-0.2, 0.2) )
        \\fay( Rc(-0.2, 0.2) )" )
```

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:

Add Tags Language: Automation Auto-4

```
tag.oscill( fx.dur, 320, "\\fr( 20*(-1)^(i + syl.i) )" )
```

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

Es todo por ahora para el **Tomo XXXVII**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.2** y visitarnos en el **Blog Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.blogspot.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

Kara Effector 3.3

El **Effector Book**, que es una especie de manual o tutorial del uso del **Kara Effector**, está de regreso luego de un tiempo de receso por otras ocupaciones que tuve en el transcurso de este largo año. Desde la publicación del **Tomo 37** (**XXXVII**), el **Kara Effector** ha venido evolucionando de forma exponencial, y por eso la necesidad de tener un material de apoyo que nos guie y ayude a sacarle el mayor provecho a dichas evoluciones.

En este **Tomo**, el **38**, veremos las funciones que se han actualizado, en qué consisten y cómo se aplican, y ejemplo de la forma en la que podemos usarla; también veremos la evolución que han tenido las celdas de texto de la **Ventana de Modificación** del **KE**, para que sea aún más simple el poder agregar funciones, tags, entre otras herramientas disponibles para nuestros fx.

Recursos [KE]:

La primera función que me interesa que vean la forma en la que ha evolucionado es la función **R** (es la versión **math.random** del **KE**), que no es otra que aquella que nos permite obtener un número aleatorio o al azar, siguiendo una serie de parámetros o argumentos predeterminados, o por default si así lo queremos.

Recordemos que esta función retorna un número aleatorio que se encuentre entre los argumentos **val1** y **val2**, inclusive, en un paso determinado por el tercer argumento, **Step**.

Ejemplo:

R(0, 100, 10)

De este modo, la función retornará un número entero entre 0 y 100 de a 10 en 10, es decir que los posibles resultados serían:

0, 10, 20, 30, 40, 50, 60, 70, 80, 90 o **100**

Partiendo de este hecho, y de que la función **R** es una de las más usadas a la hora de desarrollar un efecto, hemos ampliado su rango de operación, para así ampliar nuestras posibilidades, agregando una serie de letras para que la función cumpla con otros requisitos.

Las actualizaciones son 16 en total, y veremos ejemplos de cada una de ellas para comprender en qué consisten:

- **Rs(Val1, Val2, Step)**: la letra "**s**" es de "**signo**", es decir que esta función retorna un número aleatorio entre **Val1** y **Val2**, y el signo de dicho número también será asignado de forma aleatoria. Recordemos que los tres argumentos de esta función son opcionales, de manera que si solo ponemos un solo argumento, el random original se generará entre el 1 y dicho argumento y el valor del **Step** será 1 por

default. Si ponemos dos argumentos, el random se llevará a cabo entre esos dos valores inclusive, y el **Step** también será 1.

Ejemplo:

Rs(12, 18)

Retornará un número aleatorio entre 12 y 18 con cualquiera de los dos signos, positivo o negativo. Entonces los posibles resultados de este ejemplo serían:

-**18, -17, -16, -15, -14, -13, -12**

12, 13, 14, 15, 16, 17 o **18**

- **Rr(Val1, Val2, Step)**: la letra "**r**" es de "**ratio**". La función retornará un número aleatorio que esté entre los primeros dos argumentos inclusive, pero el resultado final dependerá del tamaño del vídeo en dónde se aplique el efecto, es decir que si el vídeo es de 720p el resultado no se modificará, pero si es más pequeño, el resultado disminuirá proporcionalmente; y pasaría lo mismo si el vídeo es de mayor resolución que el ya mencionado.

Ejemplo:

Rr(8, 10)

Los tres posibles resultados de este ejemplo en un vídeo de 720 x 1280 px son 8, 9 o 10, pero si el vídeo fuera de 1080 x 1960 px, cualquiera de esos tres resultados se multiplica por el "ratio", es decir, la razón entre 1080 y 720:

1080 / 720 = 1.5

O sea que los tres posibles resultados para el vídeo de 1080p vendrían a ser:

- 8 * 1.5 = 12
- 9 * 1.5 = 13.5
- 10 * 1.5 = 15

Esta actualización tiene su aplicación, por ejemplo, al designar aleatoriamente el borde con el tag \bord; ya que un \bord3 para un vídeo de 720p puede que sea muy poco para un vídeo de 1080p, o muy poco para uno de 480p.

- **Rd(Val1, Val2, Step)**: la letra "**d**" es de "**décima**". La función retorna un valor entre los dos primeros argumentos, pero redondeados al primer decimal.

Ejemplo:

Rd(-1, 2)

La función retornaría un número entre -1 y 2, pero tendría en cuenta los números redondeados al primer decimal, o sea que los resultados serían:

-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.8 o **2**

- **Rc(Val1, Val2, Step)**: la letra "**c**" es de "**centésima**". Es similar a la actualización anterior, pero con la diferencia que el valor retornado estará redondeado al segundo decimal, es decir que la cantidad de opciones posibles se amplían 100 veces más que en la función normal.

- **Rm(Val1, Val2, Step)**: la letra "**m**" es de "**milésima**". Redondea el número a la tercera cifra decimal.

Para las siguientes actualizaciones, lo único que debemos hacer es combinar las letras aprendidas en las actualizaciones anteriores de la función **R**, de la siguiente forma:

- **Rsr** o **Rrs**: Random + signo + ratio
- **Rdr**: Random + décima + ratio
- **Rcr**: Random + centésima + ratio
- **Rmr**: Random + milésima + ratio
- **Rds**: Random + décima + signo
- **Rcs**: Random + centésima + signo
- **Rms**: Random + milésima + signo
- **Rdrs**: Random + décima + ratio + signo
- **Rcrs**: Random + centésima + ratio + signo
- **Rmrs**: Random + milésima + ratio + signo

Y la última de las actualizaciones de la función **R** es una que nos permite ingresarle una **tabla** y obtendremos uno de sus elementos seleccionados de forma aleatoria:

- **Re(tabla)**: la letra "**e**" es de "**elemento**". Retorna un elemento seleccionado aleatoriamente de la **tabla** que le ingresemos como su único argumento. Tiene la ventaja que también lo puede hacer en **tablas** que no sean indexadas:

  Ejemplo:

```
my_table = {
    [1] = 4,
    [2] = "\\fry-45",
    Size = Rr( 80, 100 ),
    80,
    -12
}
Re( my_table )
```

  Así, de este modo, la función podría retornar cualquiera de los cinco elementos de la **tabla** "**my_table**", es decir: **4, \\fry-45, Rr(80, 100), 80** o **-12**

Con todas estas modificaciones, creo que quedan cubiertas muchas o gran parte de todas las posibilidades a la hora de generar un valor aleatorio. Lo que viene de acá en más es poner en práctica y experimentar con lo aprendido e ir desarrollando nuestros propios efectos y estilos.

El tema que veremos a continuación, considero que es uno de los más importantes para todos aquellos que les gusta desarrollar y llevar a cabo aquellos efectos que se les vienen en mente o que se presentan en los diferentes vídeos y que desean de alguna manera poder "imitarlos" para sus propios proyectos. El tema consiste es una serie de abreviaciones, convenciones y un par de trucos más que el **KE** tiene en sus librerías para que lo que antes eran simples tags, ahora se conviertan en funcionen, y por qué no, en efectos en sí mismos.

herramienta

tags abreviados y tags funciones

Parte de esta herramienta ya la empezamos a ver en el Tomo anterior, y no está de más que los recordemos y ver en qué consistía cada uno de ellos:

	tags abreviados		
	tag	**Ejemplo (valor constante)**	**resultado**
1	\\fscxy	"\\fscxy147"	\fscx147\fscy147
2	\\frxy	"\\frxy-23"	\frx-23\fry-23
3	\\frxz	"\\frxz12"	\frx12\frz12
4	\\fryz	"\\fryz345"	\fry345\frz345
5	\\frxyz	"\\frxyz-360"	\frx-360\fry-360\frz-360

Estas mismas cinco abreviaciones de tags también las podemos usar con la función **R** y todas sus nuevas actualizaciones que recién vimos:

	tags abreviados	
	tag	**Ejemplo (valor aleatorio)**
1	\\fscxy	"\\fscxyR(100, 200, 10)"
2	\\frxy	"\\frxyRs(20, 50)"
3	\\frxz	"\\frxzRcs(-125, 215)"
4	\\fryz	"\\fryzRd(22, 43)"
5	\\frxyz	"\\frxyzRs(360)"

Ahora también podemos usar, y no solo con estos cinco tags sino también con todos, operaciones y/o funciones, con solo añadir un paréntesis después de los mismos:

Ejemplo:

- "\\fscxy(120 + Rs(40))"
- "\\fr(45 * (-1) ^ syl.i)"

Add Tags Language: Lua

```
"\\fr( 45 * (-1) ^ syl.i )\\fad(200,0)"
```

Add Tags Language: Automation Auto-4

```
\frxyz( 120 –  Rs( 360 ) + 0.4 * fx.dur )\fad(200,0)\1a&HFF&
```

Entonces, cualquier tag lo podemos usar de las anteriores tres formas que acabamos de ver:

- Valor constante
- Valor aleatorio
- Operación y/o función

Esto nos ahorrará un poco de trabajo y hace que sea más simple el uso de cada uno de ellos. Y con el fin de seguir ahorrando tiempo y trabajo, el **KE** ya consta de otras abreviaciones que podemos usar de las tres formas ya mencionadas. Las primeras tres nuevas abreviaciones básicas que veremos son las siguientes:

	nuevos tags abreviados	
	tag	**equivalencia**
1	\\faxy	\fax + \fay
2	\\xybord	\xbord + \ybor
3	\\xyshad	\xshad + \yshad

Ejemplo:

Add Tags Language: Lua

```
tag.oscill( fx.dur, 80, "\\faxyRms( 0.2 )" ), "\\bord0\\shad0\\fad(0,120)"
```

Las anteriores ocho abreviaciones son las consideradas "básicas", y a partir de ellas se abre un mundo de posibilidades ante nosotros, que espero poder explicarlas todas y no olvidar ninguna por el camino. Empecemos viendo un resumen de las abreviaciones vista hasta este momento y unas nuevas que ya merecen ser mencionadas:

	tags abreviados	
	tag	**equivalencia**
1	\\fscxy	\fscx + \fscy
2	\\frxy	\frx + \fry
3	\\frxz	\frx + \frz
4	\\fryz	\fry + \frz
5	\\frxyz	\frx + \fry + \frx
6	\\faxy	\fax + \fay
7	\\xybord	\xbord + \ybord
8	\\xyshad	\xshad + \yshad

La primera abreviación (**\\fscxy**) agrupa los dos tags de porcentajes de tamaño, y cuando la usamos con un valor aleatorio, cada uno de los dos tags adquiere valores diferentes entre ellos, según los parámetros ingresados, ejemplo:

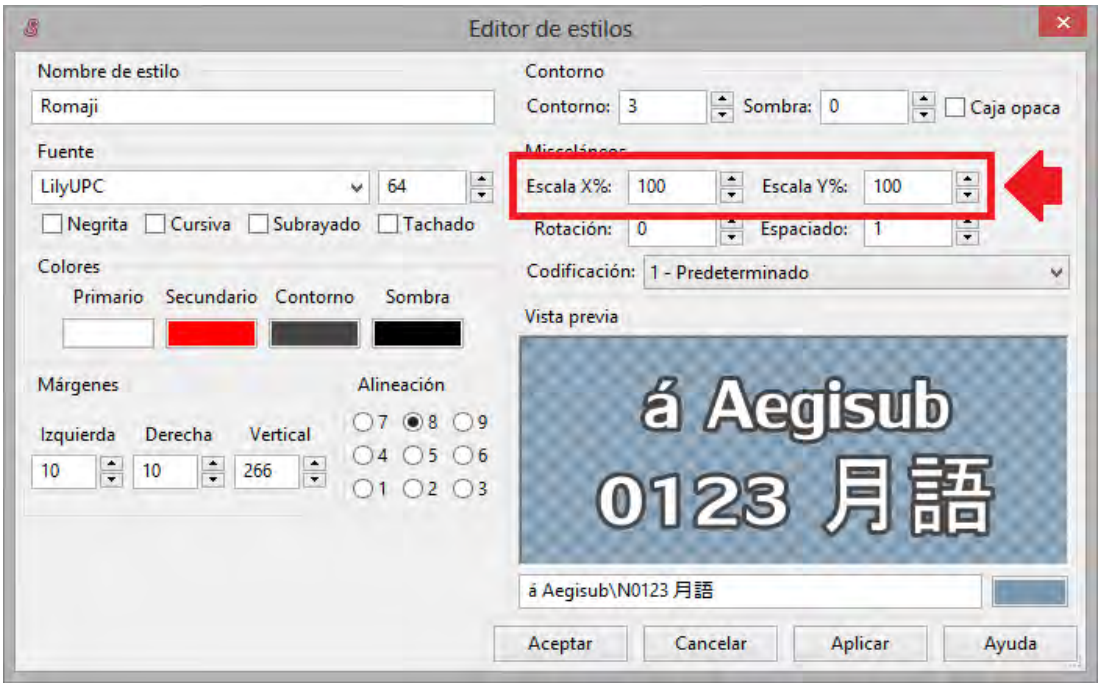
- \\fscxyR(80, 120) → \fscx104\fscy86

En el ejemplo anterior, 104 y 86 son posibles resultados aleatorios de la función **R**, cuyo valor mínimo sería 80 y valor máximo 120. El ejemplo anterior no garantiza que el valor aleatorio sea el mismo para ambos tags, y para que ello suceda, es necesario agregar la letra "i" al tag justo antes de la función **R**:

Ejemplo:

- \\fscxyiRc(40, 60) → \fscx47.18\fscy47.18

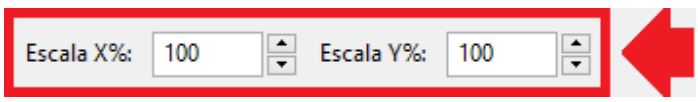
Lo que nos quiere indicar el ejemplo anterior es que el valor aleatorio que arroje la función **Rc** para el tag \fscx, será el mismo que para el tag \fscy, según los parámetros de la función random. Esta habilidad que le da la letra "i" a la abreviación \fscxy garantiza que la variación de tamaño de objeto karaoke sea siempre proporcional, siempre que los valores de escala en el Estilo de la línea a la que le apliquen el fx, sean iguales entre sí:



Para el caso en que estos dos valores no sean iguales entre sí en un Estilo, veremos las siguientes cuatro adaptaciones de los tags de escala de tamaño:

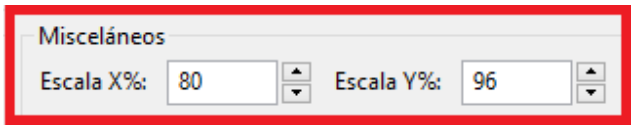
- \\fscxr
- \\fscyr
- \\fscxyr
- \\fscxyir

La letra "r" hace referencia al "ratio" o "razón" de la variación respecto a los valores de escala en un determinado Estilo de Línea



Ejemplo:

Supongamos que para cierto Estilo de Línea los valores de escala son los siguientes:



Entonces:

- \\fscx**r**1 = \\fscx80 porque 80 x 1 = 80
- \\fscy**r**1 = \\fscy96 porque 96 x 1 = 96
- \\fscx**r**0.5 = \\fscx40 porque 80 x 0.5 = 40
- \\fscy**r**1.2 = \\fscy115.2 porque 96 x 1.2 = 115.2
- \\fscx**r**2.5 = \\fscx200 porque 80 x 2.5 = 200
- \\fscxy**r**1.5 = \\fscx120\\fscy144 porque 80 x 1.5 = 120 y 96 x 1.5 = 144
- \\fscxy**r**0.4 = \\fscx32\\fscy38.4 porque 80 x 0.4 = 32 y 96 x 0.4 = 38.4

Así que el valor que coloquemos después de la "r" se multiplicará por el valor de escala original del Estilo de Línea al que le apliquemos el fx, entonces, si por ejemplo queremos que ambas escalas aumenten al doble de su valor original, lo único que tenemos que poner sería:

- \\fscxy**r**2

Si queremos que ambas escalas se modifiquen aleatoriamente de forma proporcional usamos \\fscxyir

Ejemplo:

- \\fscxyir**Rd**(1, 2) → \\fscxy**r**1.6 → \\fscx**r**1.6\\fscy**r**1.6 → \\fscx128\\fscy153.6

Para el próximo Tomo, seguiremos viendo más de las abreviaciones de los tags que podemos usar en el **Kara Effector** y que hará que hacer efectos sea una tarea aún más sencilla de lo que podríamos llegar a imaginarnos, así que estén pendiente de futuras entregas para que estén al tanto de todas la novedades.

Es todo por ahora para el **Tomo XXXVIII**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.3** y visitarnos en la **Web Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012

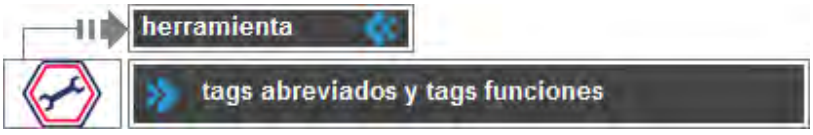
Kara Effector 3.3

En este **Tomo 39** continuaremos viendo más de los tags abreviados, modificados y añadidos en el **Kara Effector**, con el fin de saber cómo aprovecharlos y aplicarlos a nuestros proyectos. Todos estos tags nos harán el trabajo un poco más simple e incluso nos ayudarán a descubrir nuevas combinaciones y efectos, dada las múltiples posibles combinaciones que se pueden realizar entre ellos.

Recursos [KE]:

Comenzaremos viendo una tabla con el resumen de todos los tags abreviados vistos hasta este momento, con el fin de tenerlos presentes y de continuar viendo los que aún nos hacen falta:

	tags abreviados	
	tag	**equivalencia**
1	\\fscxr	\fscx
2	\\fscyr	\fscy
3	\\fscxyr	\fscx + \fscy
4	\\fscxyi	\fscx + \fscy
5	\\fscxyir	\fscx + \fscy
6	\\fscxy	\fscx + \fscy
7	\\frxy	\frx + \fry
8	\\frxz	\frx + \frz
9	\\fryz	\fry + \frz
10	\\frxyz	\frx + \fry + \frx
11	\\faxy	\fax + \fay
12	\\xybord	\xbord + \ybord
13	\\xyshad	\xshad + \yshad



Los siguientes tags que veremos incluyen a los tags de rotación y al tag \org, y son:

- \\frx**o**
- \\fry**o**
- \\frz**o** o \\fr**o**
- \\frxy**o**
- \\frxz**o**
- \\fryz**o**
- \\frxyz**o**

Y precisamente, lo que hace la letra "o" en los anteriores tags, es agregar al tag \org con los parámetros **fx.pos_x** y **fx.pos_y**, o sea, las coordenadas de la posición por default del objeto karaoke.

Ejemplo:

- \\frxz**o**Rs(360) → \org(fx.pos_x, fx.pos_y)\frx-24\frz238
- \\t(0,300,\\fry**o**90) → \org(fx.pos_x, fx.pos_y)\t(0,300,\fry90)
- \\frx**o**R(60, 120)\\t(\\fr**o**360) → \org(fx.pos_x, fx.pos_y)\frx72\t(\frz360)

Cuando las coordenadas del tag \org son las de las posición por default del objeto karaoke, cualquier rotación que se haga, se hará respecto a dicho origen, como si esa fuera la aguja de un compás con el cual trazamos un círculo.

Siempre que efectuemos una rotación, si no está el tag \org, ésta se hará respecto al tag \pos o a las dos primeras coordenadas del tag \move, pero como ya se habrán dado cuenta, el **KE** tiene algunas funciones que generan un tag \pos, como la función **shape.Rmove**, que casi siempre genera un \pos(0,0), por lo que si no especificamos el \org, las rotaciones se harán respecto al punto P = (0, 0) o a cualquier otro del tag \pos.

Con estas últimas siete abreviaciones ya completamos un total de veinte, pero no son las únicas modificaciones que podemos hacerle a los tags para generar nuevas combinaciones. Las siguientes abreviaciones incluyen a los tags "animados" ya conocidos e incluso los anteriores veinte que acabamos de aprender, y aparte de esto le sumamos el tag \t (transformación). Se dice que un tag es "animado" si se puede ingresar dentro de un tag \t y es afectado por éste:

	tags animados							
	xy-vsfilter			vsfiltermod			abreviaciones	
1	c	xbord	fax	1vc	3img	rndz	fscxr	faxy
2	1c	ybord	fay	2vc	4img	distort	fscyr	xybord
3	2c	bord	fr	3vc	fsc		fscxyr	xyshad
4	3c	xshad	frx	4vc	frs		fscxyi	frxo
5	4c	yshad	fry	1va	fsvp		fscxyir	fryo
6	alpha	shad	frz	2va	jitter		fscxy	frzo - fro
7	1a	blur	fs	3va	z		frxy	frxyo
8	2a	be	fsp	4va	rnd		frxz	frxzo
9	3a	fscx	clip	1img	rndx		fryz	fryzo
10	4a	fscy	iclip	2img	rndy		frxyz	frxyzo

Y si a cualquiera de los anteriores 72 tags animados le agregamos la letra "t", el tag se ingresará inmediatamente dentro de un tag \t:

Ejemplo:

- \\frz**t**45 = \t(\frz45)
- \\blur**t**R(2, 5) = \\t(\\blurR(2, 5)) → por ejemplo: \t(\blur3)
- \\frxzo**t**Rs(120) = \\org(fx.pos_x, fx.pos_y)\\t(\\frxRs(120)\\frzRs(120))
- \\3c**t**&H0000FF& = \t(\3c&H0000FF&)

Como podrán notar en los ejemplos anteriores, los tags son ingresados en un tag \t sin tiempos, es decir que la transformación se lleva a cabo a lo largo de la duración total de la línea fx (**fx.dur**):

\\t(\\tags) = \\t(0,fx.dur,\\tags)

La anterior tabla de tags animados la debemos tener siempre presente, ya que son todos aquellos tags que podemos "transformar" conforme el tiempo transcurre. Y para continuar con la misma temática de las transformaciones, veremos las siguientes abreviaciones también aplicables a los anteriores 72 tags animados:

- \\tag-
- \\tag~

Al agregar el signo menos (-) al final de un tag, el **KE** le añade una transformación por default que contiene el mismo tag, pero con el valor inverso:

Ejemplo:

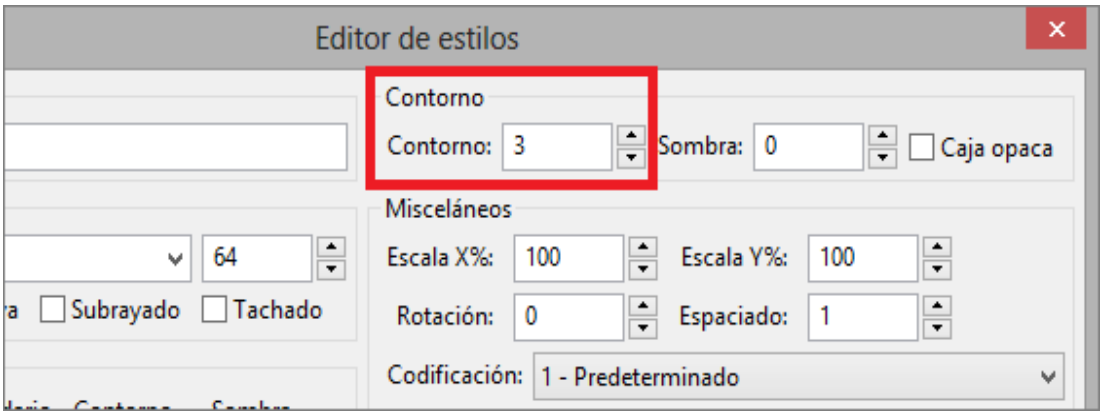
- \\frx86- → \fr86\t(\fr-86)
- \\yshad-4.2- → \yshad-4\t(\yshad4)
- \\frxzR(20, 40)- →\frx36\frz24\t(\frx-36\frz-24)
- \\fryoRs(15)- →\org(fx.pos_x, fx.pos_y)\fry-11\t(\fry11)

Y como ya se habrán imaginado, no todos los tags tienen un valor inverso, para todos aquellos con esta característica el **KE** le asigna su valor por default o uno nulo (0).

Al agregar el signo (~) al final de un tag, el **KE** le añade una transformación por default que contiene el mismo tag, pero con el valor por default o nulo:

Ejemplo:

Si para un Estilo de Línea le ponemos un contorno de 3, ese 3 será su valor por default:



Entonces para los siguientes ejemplos tenemos:

- \\bord4~ →\bord4\t(\bord3)
- \\bordRd(5, 8)~ →\bord6.7\t(\bord3)
- \\blur3~\\bord6~ →\blur3\bord6\t(blur0\bord3)

La siguiente tabla nos ayudará a recordar cuáles son los valores por default e inversos que el **KE** le asigna a cada uno de los tags animados, al agregarles un signo (~) o un signo (-) al final de los mismos:

	tags animados					
	xy-vsfilter			vsfiltermod		
	tag	valor default (\\tag~)	valor inverso (\\tag-)	tag	valor default (\\tag~)	valor inverso (\\tag-)
1	c	text.color1	text.color1	1vc	text.color1	text.color1
2	1c	text.color1	text.color1	2vc	text.color2	text.color2
3	2c	text.color2	text.color2	3vc	text.color3	text.color3
4	3c	text.color3	text.color3	4vc	text.color4	text.color4
5	4c	text.color4	text.color4	1va	text.alpha1	text.alpha1
6	alpha	&H00&	&H00&	2va	text.alpha2	text.alpha2
7	1a	text.alpha1	text.alpha1	3va	text.alpha3	text.alpha3
8	2a	text.alpha2	text.alpha2	4va	text.alpha4	text.alpha4
9	3a	text.alpha3	text.alpha3	1img	---	---
10	4a	text.alpha4	text.alpha4	2img	---	---
11	xbord	l.outline	-1 * xbord	3img	---	---
12	ybord	l.outline	-1 * ybord	4img	---	---
13	bord	l.outline	l.outline	fsc	l.scale_x	l.scale_x
14	xshad	l.shadow	-1 * xshad	frs	0	-1 * frs
15	yshad	l.shadow	-1 * yshad	fsvp	0	-1 * fsvp
16	shad	l.shadow	l.shadow	jitter	---	---
17	blur	0	0	z	0	-1 * z
18	be	0	0	rnd	0	-1 * rnd
19	fscx	l.scale_x	l.scale_x	rndx	0	-1 * rndx
20	fscy	l.scale_y	l.scale_y	rndy	0	-1 * rndy
21	fax	0	-1 * fax	rndz	0	-1 * rndz
22	fay	0	-1 * fay	distort	---	---
23	fr	l.angle	-1 * fr			
24	frx	0	-1 * frx			
25	fry	0	-1 * fry			
26	frz	l.angle	-1 * frz			
27	fs	l.fontsize	l.fontsize			
28	fsp	l.spacing	-1 * fsp			
29	clip	---	---			
30	iclip	---	---			

Como acabamos de ver, la transformación generada siempre es una por default, o sea sin tiempos dentro del tag \t. Para controlar el tiempo en que la transformación retorna el tag a su valor por default o a su valor inverso, lo único que debemos hacer es colocar el tiempo en milisegundos (ms) luego del signo (~) o del signo (-), o si queremos realizar una operación para obtener dicho tiempo, entonces colocamos la operación o función dentro de paréntesis después de los signos:

Ejemplo:

- \\fryzo90-300 →\org(fx.pos_x, fx.pos_y)\fry90\frz90\t(0,300,\fry-90\frz-90)
- \\blur3~(fx.dur/3) →\blur3\t(0,fx.dur/3,\blur0)
- \\xyshadRs(10)-200 →\xshad-7\yshad9\t(0,200,\xshad7\yshad-9)

Con esta adaptación al final de los signos (~) y (-), podemos controlar el tiempo exacto en que la transformación se realizará, lo que amplía aún más nuestras posibilidades a la hora de hacer un efecto. Estas abreviaciones tienen muchas aplicaciones en los efectos **lead-in** y **hi-light** sobre todo, pero el tipo de efecto en las que las podemos usar solo depende de cada uno de nosotros. Intenten hacer sus propias combinaciones y de a poco se irán familiarizando a ellas.

Ya con los conceptos de valores inversos y por default de los tags animados, es hora de ver unas abreviaciones un poco más complejas e igual de prácticas como las anteriores. Para las siguientes abreviaciones debemos colocar unas letras especiales entre el nombre del tag y su valor, dichas letras son:

- **mr** → \tag + \t(0,fx.dur/2,\tag-) + \t(fx.dur/2,fx.dur,\tag)
- **md** → \tag + \t(0,fx.dur/2,\tag~) + \t(fx.dur/2,fx.dur,\tag)
- **mrd** → \tag + \t(0,fx.dur/2,\tag-) + \t(fx.dur/2,fx.dur,\tag~)
- **k** → \t(0,fx.dur/4,\tag)+\t(fx.dur/4,3*fx.dur/4,\tag-) + \t(3*fx.dur/4,fx.dur,\tag~)

Ejemplo:

- \\frxmr-45 → \frx-45\t(0,fx.dur/2,\frx45)\t(fx.dur/2,fx.dur,\frx-45)
- \\blurmd5 → \blur5\t(0,fx.dur/2,\blur0)\t(fx.dur/2,fx.dur,\blur5)
- \\frmrdR(20) → \fr17\t(0,fx.dur/2,\fr-17)\t(fx.dur/2,fx.dur,\fr0)
- \\xyshadkRs(8)
- \\frxyzomdRds(120, 240)
- \\fscxyrkRc(0.75, 1.55)

En la siguiente tabla vemos las modificaciones vistas hasta ahora, que podemos hacerle a la mayoría de los tags animados (ya que hay algunas excepciones que pronto abordaremos):

	modificaciones de un tag animado			
	añadido	**valor normal del tag**	**valor aleatorio de la función R ***	**operación o función**
tag		\tag<valor>	\tagR( )	\tag(operación)
	t	\tagt<valor>	\tagtR( )	\tagt(operación)
	~	\tag<valor>~	\tagR( )~	\tag(operación)~
	-	\tag<valor>-	\tagR( )-	\tag(operación)-
	mr	\tagmr<valor>	\tagmrR( )	\tagmr(operación)
	md	\tagmd<valor>	\tagmdR( )	\tagmd(operación)
	mrd	\tagmrd<valor>	\tagmrdR( )	\tagmrd(operación)
	k	\tagk<valor>	\tagkR( )	\tagk(operación)

* La modificación de un valor aleatorio con la función **R** no aplica para todos los tags.

Los tags de colores son aquellos para los que la modificación de un valor aleatorio con la función **R** no aplica, o no de manera convencional, pero el **KE** tiene también la solución para ello, y es la siguiente convención:

- \\1c**R**( ) → \\1c(**random.color( )**)

Ejemplo:

- \\3c**R**({ 60, 120 }) = \\3c(**random.color({ 60, 120 })**)
- \\4c**R**(nil, 80) = \\4c(**random.color(nil, 80)**)
- \\1c**R**(40, nil, { 60, 90 }) = \\1c(**random.color(40, nil, { 60, 90 })**)
- \\3c**R**( ) = \\3c(**random.color( )**)

A parte de la anterior convención, tenemos las siguientes para "llamar" a los colores y alphas propios de las ventanas del **KE**:

convenciones de colores y alphas del KE			
convención	**equivalencia**	**convención**	**equivalencia**
TC1	text.color1	**SC1**	shape.color1
TC2	text.color2		
TC3	text.color3	**SC3**	shape.color3
TC4	text.color4	**SC4**	shape.color4
TA1	text.alpha1	**SA1**	shape.alpha1
TA2	text.alpha2		
TA3	text.alpha3	**SA3**	shape.alpha3
TA4	text.alpha4	**SA4**	shape.alpha4
\\txt.c	text.color	**\\shp.c**	shape.color
\\txt.a	text.alpha	**\\shp.a**	shape.alpha
\\txt.s	text.style	**\\shp.s**	shape.style

Ejemplo:

- \\3c**SC1** = \\3c(**shape.color1**) = **format**("\\3c%s", **shape.color1**)
- \\1c**TC4** = \\1c(**text.color4**) = **format**("\\1c%s", **text.color1**)
- \\3c**R**( )\\3c**t**SC3 = **format**("\\3c%s\\t(\\3c%s)", **random.color( )**, **shape.color3**)

También, para mayor practicidad, ahora el **KE** puede darle valores decimales entre 0 y 255 a los tags y a las funciones alpha, para que no nos compliquemos tanto al tener que calcular los mismos valores en base hexadecimal (16).

Ejemplo:

- \\1a**86** = **format**("\\1a%s", **ass_alpha(86)**)
- \\3a**255~** = **format**("\\3a%s\\t(\\3a%s)", **ass_alpha(255)**, **text.alpha3**)
- \\4a**92~**300 = **format**("\\4a%s\\t(0,300,\\4a%s)", **ass_alpha(92)**, **text.alpha4**)

En los anteriores ejemplos, el **KE** convierte automáticamente esos valores en base decimal (10), a base hexadecimal (16), que es el formato .ass de los valores alphas.

Y como lo mencionaba anteriormente, también podemos ingresar valores en base decimal (10) dentro de las funciones alpha, que de forma normal las usaríamos así:

Ejemplo:

- **alpha.module**(“&HAA&”, “&HFF&”)

Esta misma función, aplicada con valores decimales sería:

- **alpha.module**(170, 255)

Lo que evidentemente es más simple de hacer y de aplicar. También nos da la opción dentro de las funciones de combinar ambos casos: **alpha.module**(170, “&HFF&”)

Esta misma habilidad de poner valores decimales a los tags y funciones alpha también es aplicable a los tags del **VSfiltermod**.

Ejemplo:

- \\1va(0,0,255,255)
- \\3va255 = \\3va(255,255,255,255) = \\3va(&HFF&,&HFF&,&HFF&,&HFF&)

Hay una adaptación más hecha especialmente para aplicarla a los tags alpha del **xy-vsfilter**, y es la de poder darle el mismo valor a más de un tag:

Ejemplo:

- **13**a&A4& → \1a&HA4&\3a&HA4&
- **142**a200 → \1a200\4a200\2a200 → \1a&HC8&\4a&HC8&\2a&HC8&
- **31**a255 → \3a255\1a255 → \3a&HFF&\1a&HFF&

Es todo por ahora para el **Tomo XIL**. Intenten poner en práctica todos los ejemplos vistos y no olviden descargar la última actualización disponible del **Kara Effector 3.3** y visitarnos en la **Web Oficial**, lo mismo que en los canales de **YouTube** para descargar los nuevos Efectos o dejar algún comentario. Pueden visitarnos y dejar su comentario en nuestra página de **Facebook**:

- www.karaeffector.com
- www.facebook.com/karaeffector
- www.youtube.com/user/victor8607
- www.youtube.com/user/NatsuoKE
- www.youtube.com/user/karalaura2012