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

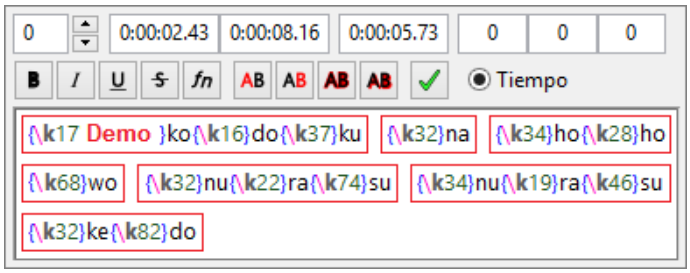
Add Tags Language: Lua

```
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**:

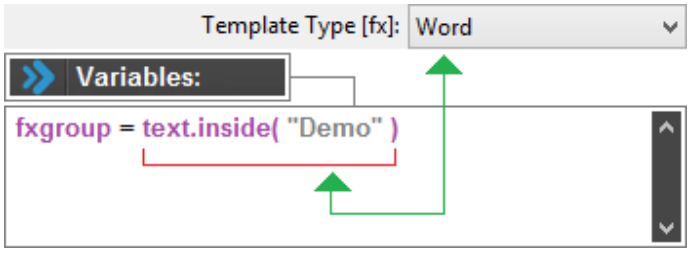
Ejemplo:

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

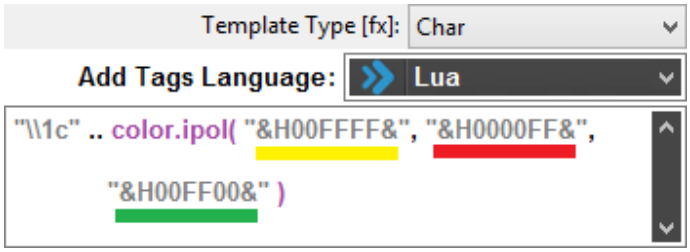
La ventaja de esta función es que puede verificar si un string del parámetro **inside** está, no solo en el texto de un objeto karaoke de la línea, sino que también dentro de sus tags. Esta ventaja la podemos usar para poner marcas en el script y luego aplicar uno o más efectos especiales solo en donde 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.

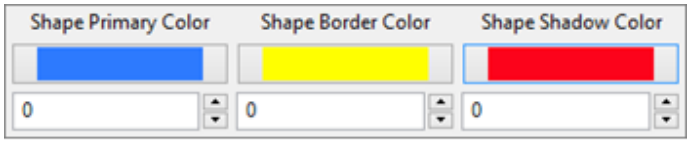
Ejemplo:

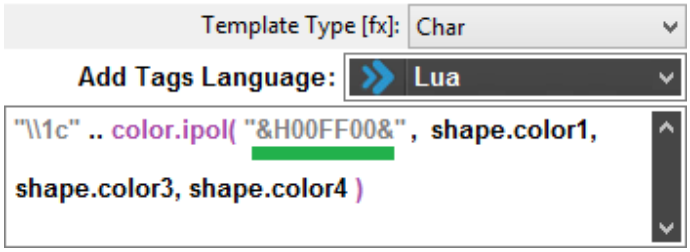


Al aplicar, notamos cómo cada una de las letras, dado que el **Template Type** es **Char**, hace el gradiente ente los tres colores, empezando por el amarillo, pasando por el rojo y llegando hasta el verde:

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:

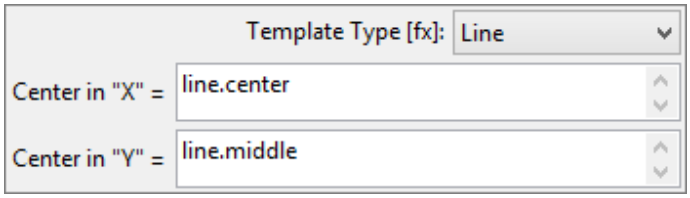
Recursos [KE]: → función

color.loop(...)

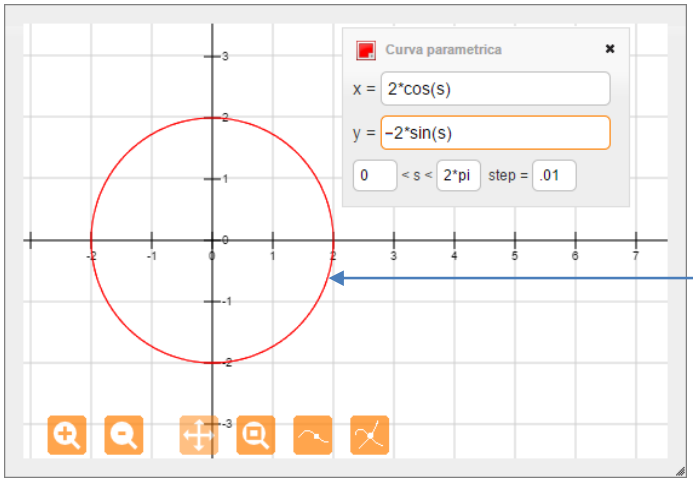
Esta función es similar a la anteriormente vista, pero con la diferencia de que crea el gradiente (**degradación**) entre todos los colores ingresados respecto al **loop** y no a los componentes karaokes de la línea.

Ejemplo:

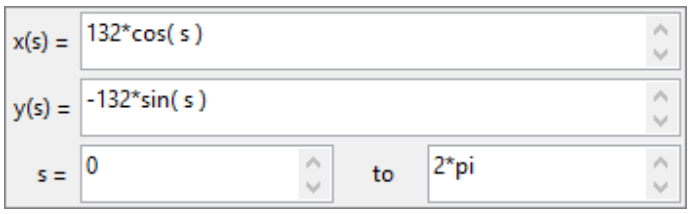
Seleccionamos un **Template Type: Line**



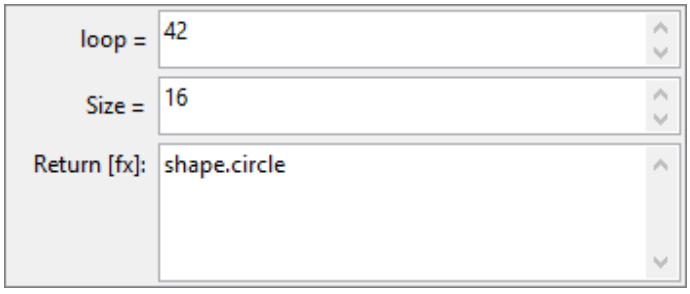
Ingresaremos la siguiente función paramétrica:



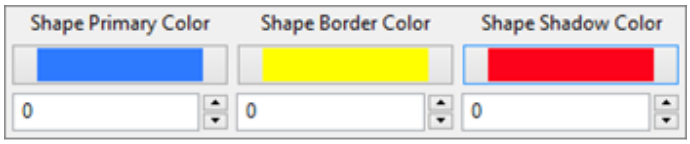
Aumentamos un poco la escala, ya que un círculo de 2 px de radio sería muy pequeño:

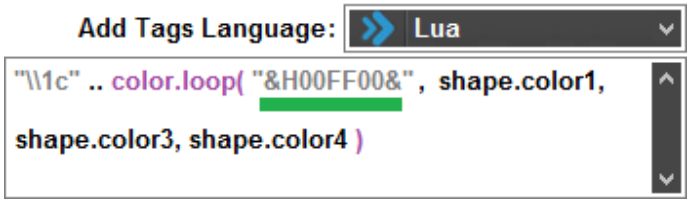


Aumentamos el **loop**, modificamos el tamaño de la shape y ponemos **shape.circle** en **Return [fx]:**



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.

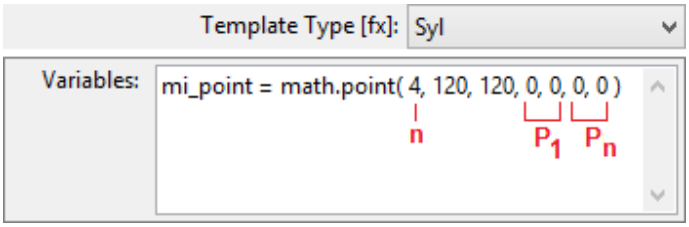
Con los parámetros **end_x** y **end_y** podemos decidir cuál será el último punto de la **tabla** generada por la función, y

sus valores por default equivalen a un punto de origen del sistema cartesiano P = (0, 0).

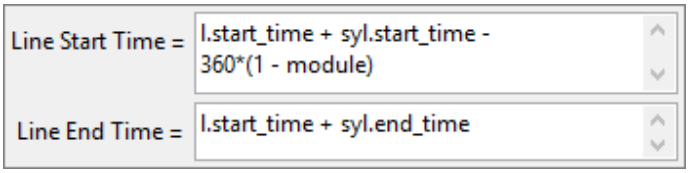
Ejemplo:

Definimos una variable utilizando la función para crear una **tabla** de cuatro puntos:

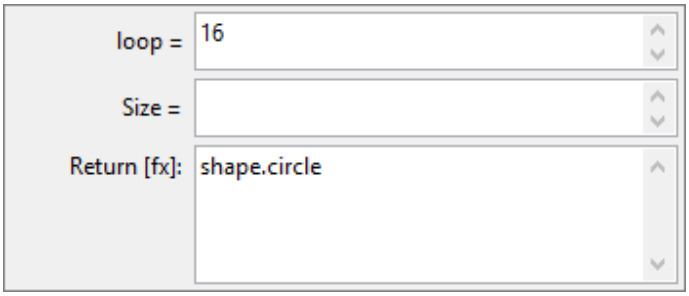
- n = 4
- rango en “x” = 120 px
- rango en “y” = 120 px
- punto inicial P1 = (0, 0)
- punto final P4 = (0, 0)



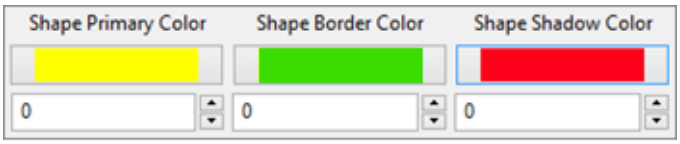
Modificamos los tiempos con la intensión de hacer un efecto tipo **hi-light**:

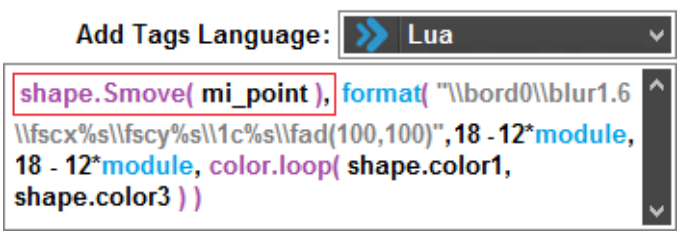


Aumentamos el **loop**, para este ejemplo en 16, y en **Return [fx]** ponemos **shape.circle**:



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.

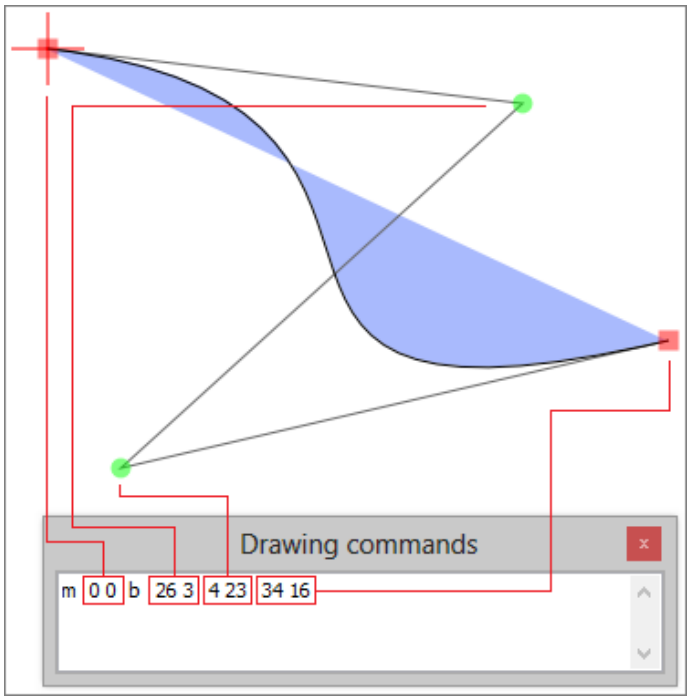
El parámetro **Return** decide lo que va a retornar la función, y tiene las tres siguientes opciones:

- **“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**.

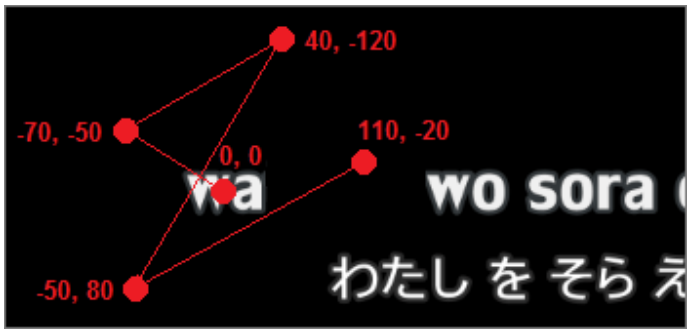
Ejemplo:

Un ejemplo sencillo es ver cómo el **ASSDraw3** traza una curva **Bézier** a partir de cuatro puntos:

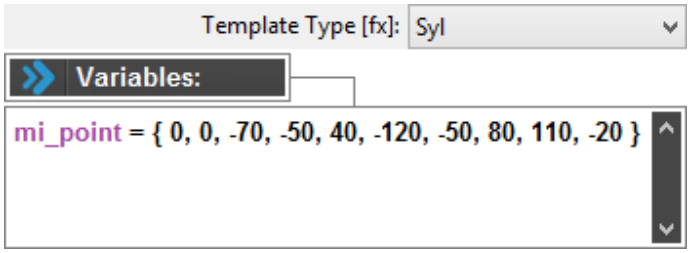


Ejemplo:

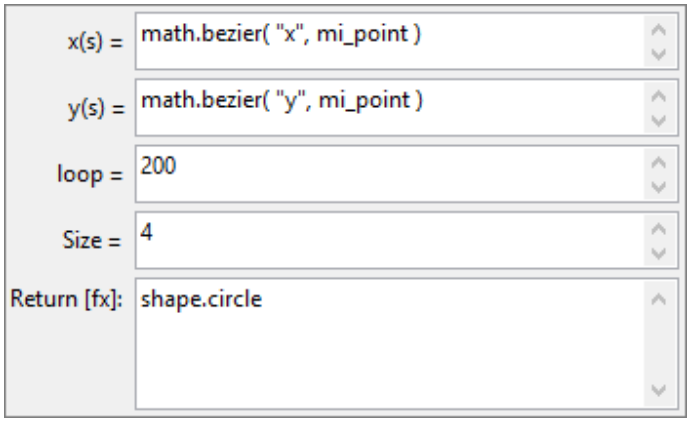
La siguiente imagen muestra cinco puntos con centro en una de las sílabas:



Con las coordenadas de esos puntos declaramos una **tabla** en la celda de texto **Variables**:



Y hacemos las siguientes configuraciones:

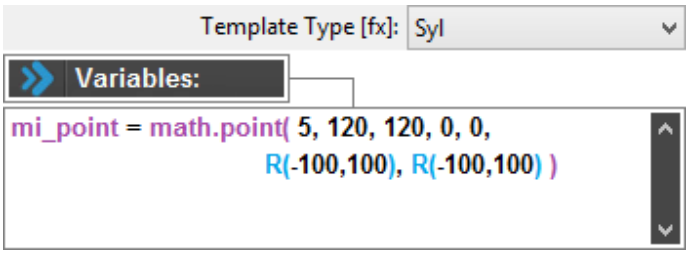


Cuando apliquemos, la función traza una curva **Bézier** que corresponde a los cinco puntos de la **tabla** que declaramos:

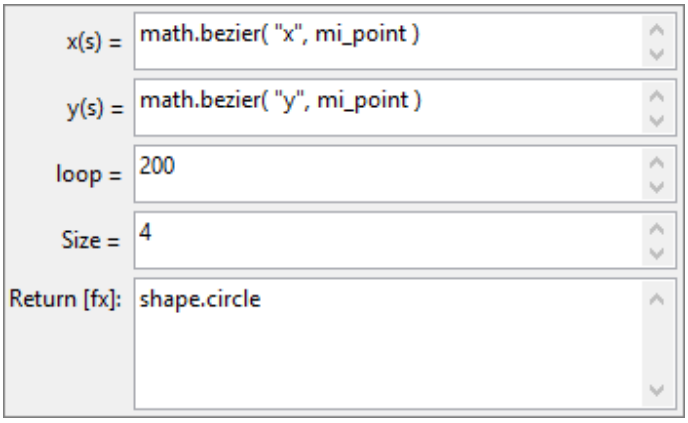
La desventaja de que los puntos sean fijos (constantes), es que la función siempre hará el mismo trazado de la curva **Bézier** en todas las sílabas.

Para generar puntos al azar, volvemos a definir la **tabla** de puntos y nos apoyamos en la función **math.point**:

Ejemplo:



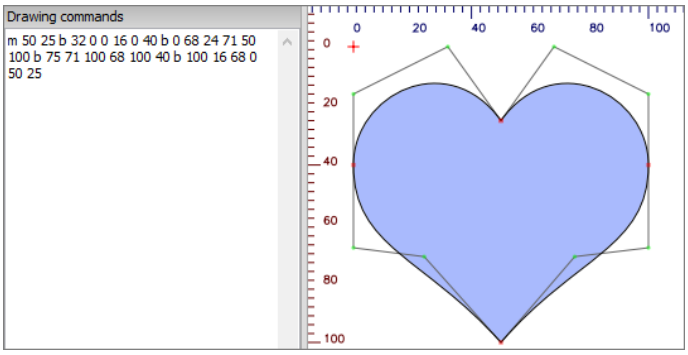
La función **math.bezier** se llama en las celdas **x(s)** y **y(s)**. El **loop** también se puede modificar a gusto:



Le cambié los colores para identificar la curva **Bézier** que se generó para cada una de las sílabas:

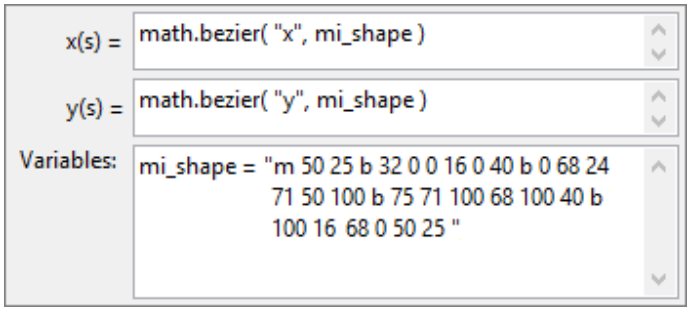
Conservando las configuraciones del "**Size**" y del "**Return [fx]**" del ejemplo anterior, cambiamos el segundo parámetro de la función, de una **tabla** de puntos a una **shape**, ampliamos las posibilidades:

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:

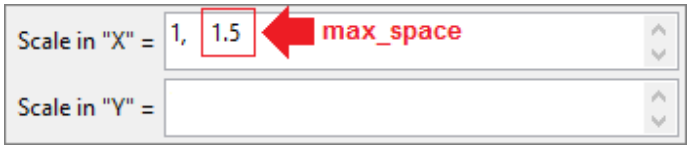
Cuando el segundo parámetro de la función es una shape, la función calcula el loop apoyándose en una variable interna del **KE**:

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:



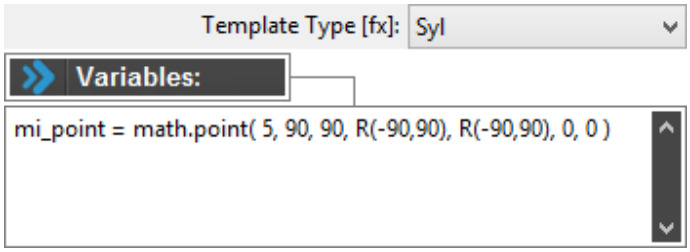
En este ejemplo aumentamos el valor de **max_space** de 1 a 1.5, lo que aumentará la distancia que separa a los objetos karaoke que trazan en contorno de la shape, y por ende disminuirá el **loop** generado por la función. El valor de **max_space** debe ser un número real mayor a cero y siempre éste será inversamente proporcional al **loop**.

Al aumentar la variable **max_space** en 5 px, es notorio la disminución del **loop**. Ejemplo:

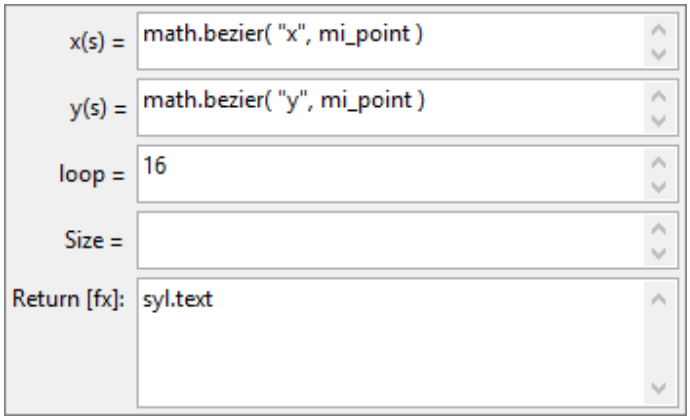
Para trazar una curva **Bézier** o trazar el contorno de una **shape** ingresada, no necesariamente debemos hacerlo con una **shape** en **Return [fx]**, también lo podemos hacer con el texto o con cualquier otro **string** que nos imaginemos.

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:

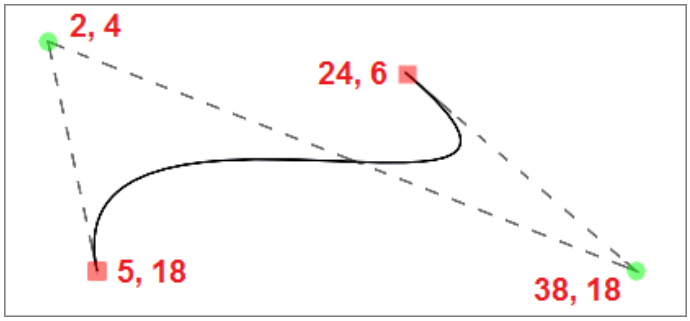
Recursos [KE]: → función

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