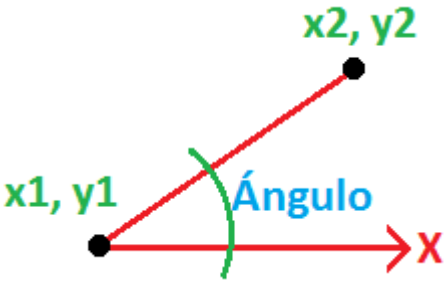
Librería “math” [KE]

La Librería “math” [LUA] 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 Effecto**r, pero es porque no son para usar en los Efectos, sino que son para ayudar a otras funciones a hacer su trabajo.

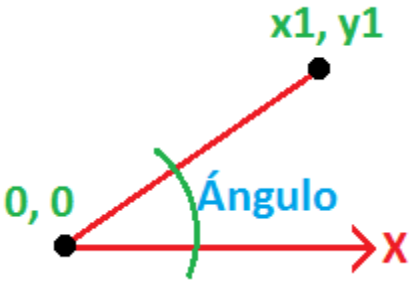
math.R(m, n)	Cumple prácticamente la misma función que **math.random**, pero con la diferencia que genera un número aleatorio por cada vez que se aplica el Efecto. La forma abreviada es: **R(m, n)**
math.Rfake(m, n, i)	Parecida a **math.random**, pero el random se genera una única vez para todos los 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)

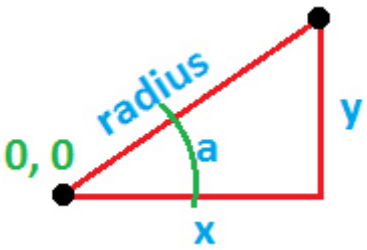
En este caso retorna el Ángulo que forma el segmento formado por los puntos P1 = (0, 0) y P=(x1, y1), y el tramo positivo del eje “x”:



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.

math.point(n, x_range, y_range, start_x, star_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**

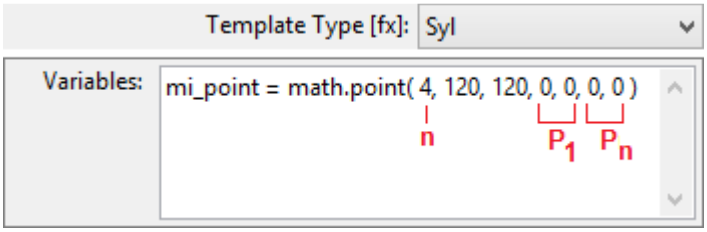
Con los parámetros **start_x** y **start_y** podemos decidir cuál será el primer punto de la **tabla** generada por la función, y sus valores por default equivalen a un punto generado al azar, dentro de los rangos seleccionados.

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).

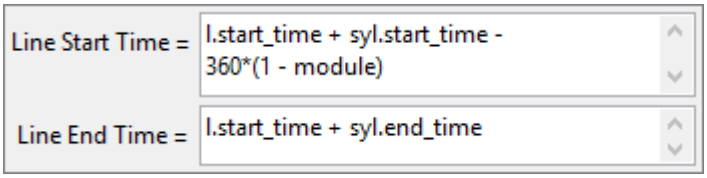
Ejemplo:

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

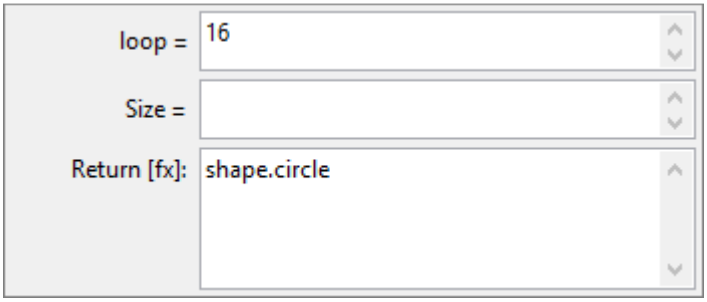
- n = 4
- rango en “x” = 120 px
- rango en “y” = 120 px
- punto inicial P1 = (0, 0)
- punto final P4 = (0, 0)



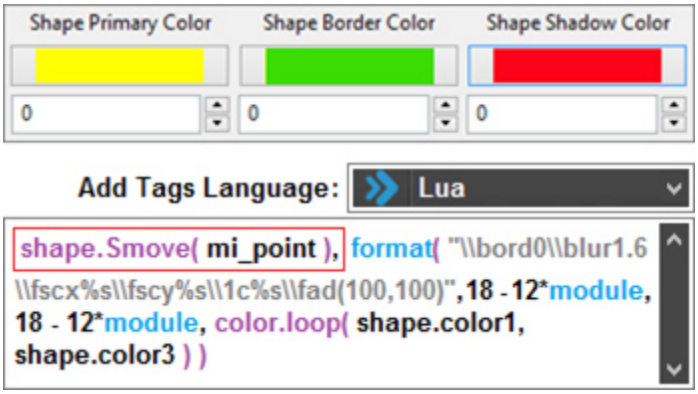
Modificamos los tiempos con la intensión de hacer un efecto tipo **hi-light**:



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.

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.

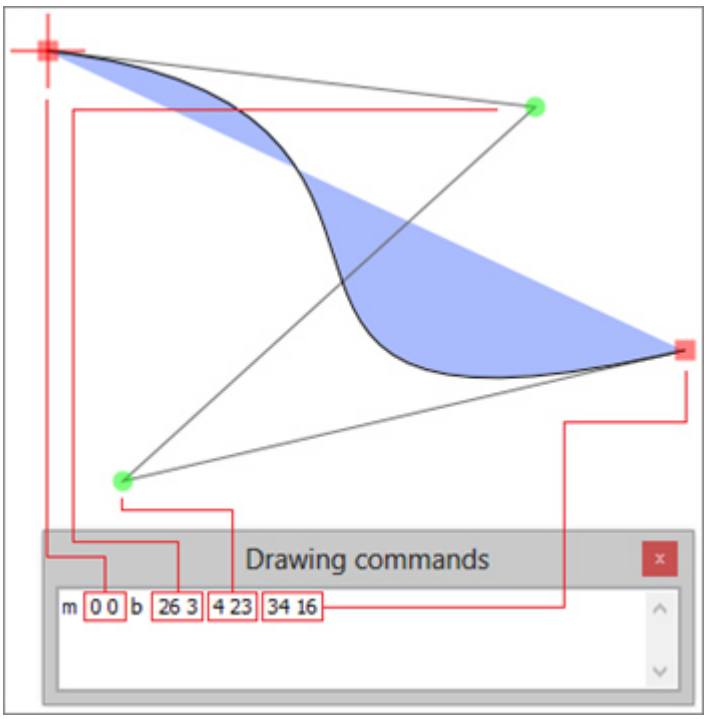
El parámetro **Return** decide lo que va a retornar la función, y tiene las tres siguientes opciones:

- **“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**.

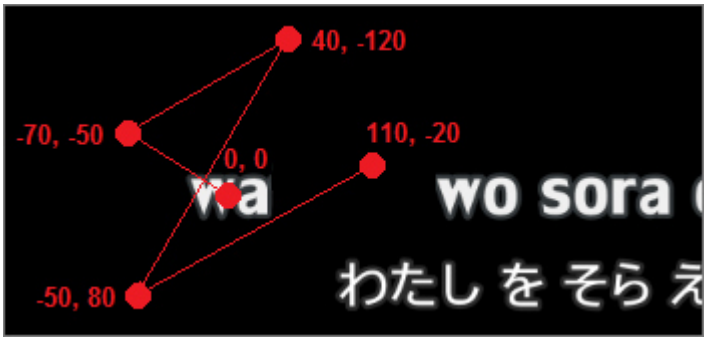
Ejemplo:

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

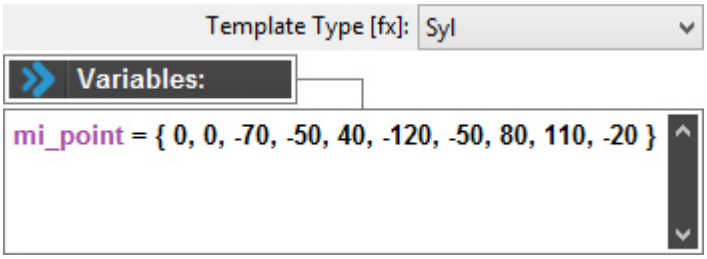


Ejemplo:

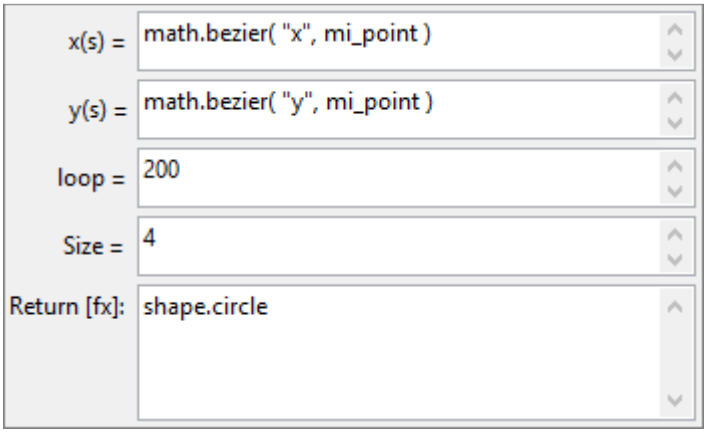
La siguiente imagen muestra cinco puntos con centro en una de las sílabas:



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



Y hacemos las siguientes configuraciones:

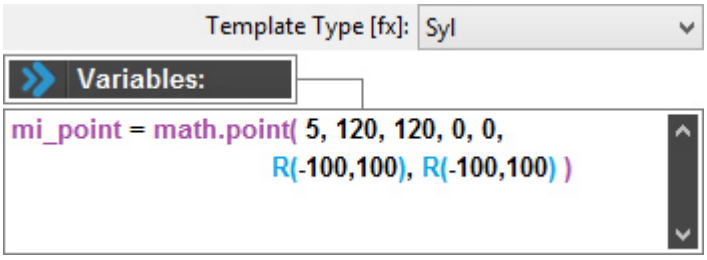


Cuando apliquemos, la función traza una curva **Bézier** que corresponde a los cinco puntos de la **tabla** que declaramos:

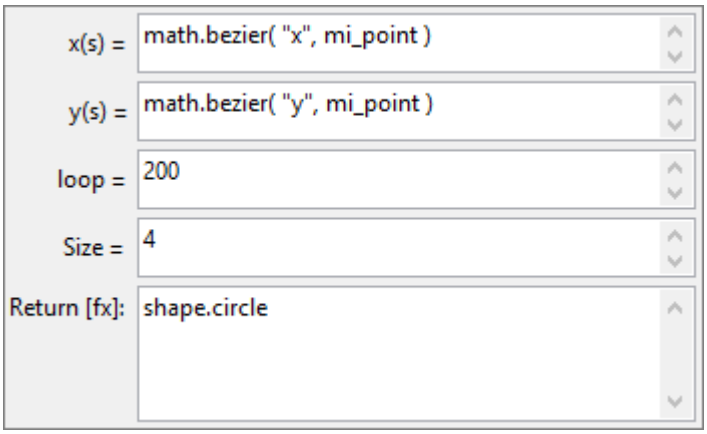
La desventaja de que los puntos sean fijos (constantes), es que la función siempre hará el mismo trazado de la curva **Bézier** en todas las sílabas.

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

Ejemplo:



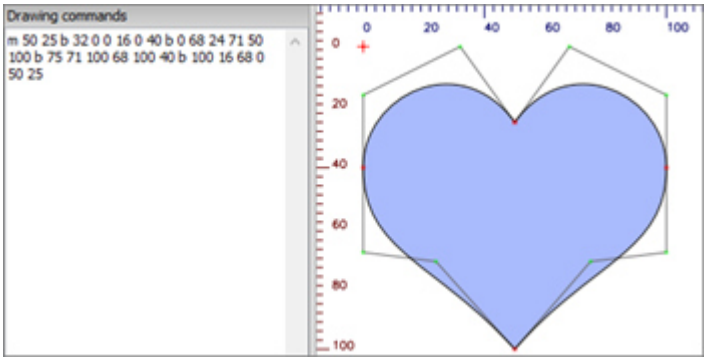
La función **math.bezier**
se llama en las celdas **x(s)** y **y(s)**. El **loop** también se puede modificar a gusto:



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

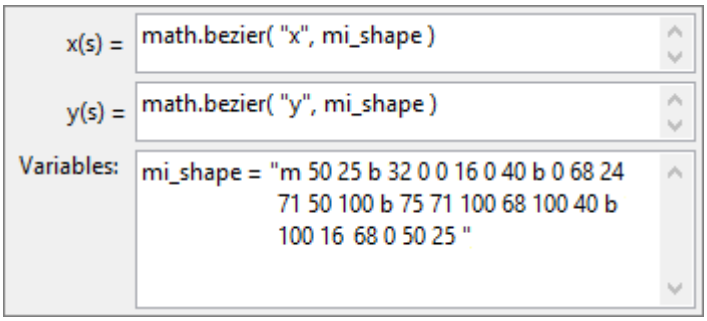
Conservando las configuraciones del "**Size**" y del "**Return [fx]**" del ejemplo anterior, cambiamos el segundo parámetro de la función, de una **tabla** de puntos a una **shape**, ampliamos las posibilidades:

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:

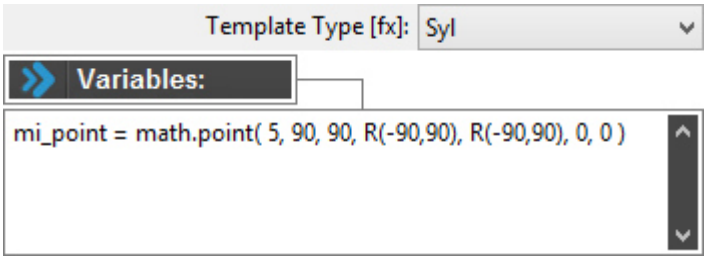
En este ejemplo aumentamos el valor de **max_space** de 1 a 1.5, lo que aumentará la distancia que separa a los objetos karaoke que trazan en contorno de la shape, y por ende disminuirá el **loop** generado por la función. El valor de **max_space** debe ser un número real mayor a cero y siempre éste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

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

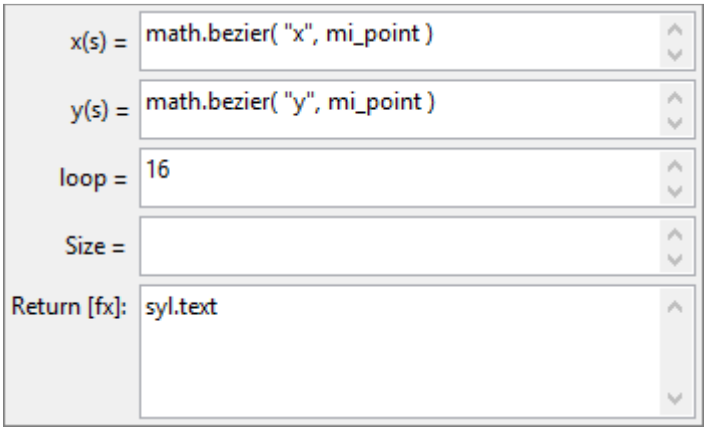
Para trazar una curva **Bézier** o trazar el contorno de una **shape** ingresada, no necesariamente debemos hacerlo con una **shape** en **Return [fx]**, también lo podemos hacer con el texto o con cualquier otro **string** que nos imaginemos.

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**:



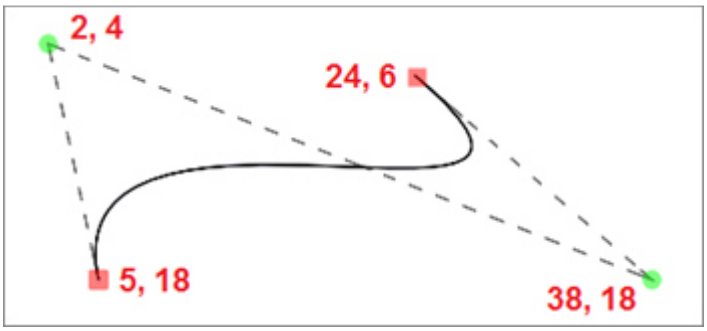
Al aplicar vemos cómo la curva es trazada por las sílabas:

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

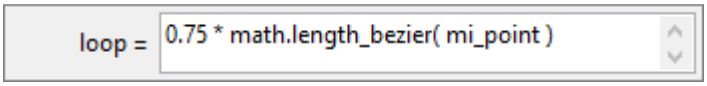
La **tabla** del parámetro **Points** de la función debe tener al menos un punto, lo que retornará 0 px como longitud, de ahí en adelante calculará la longitud de la **Bezier** generada.

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:

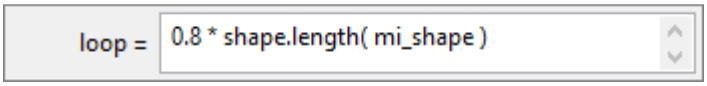
Para el ejemplo en el que usamos a la función **math.point**
para generar puntos al azar que dibujaran la curva **Bézier**, obviamente las generó de diferentes longitudes:

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:



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:



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.