

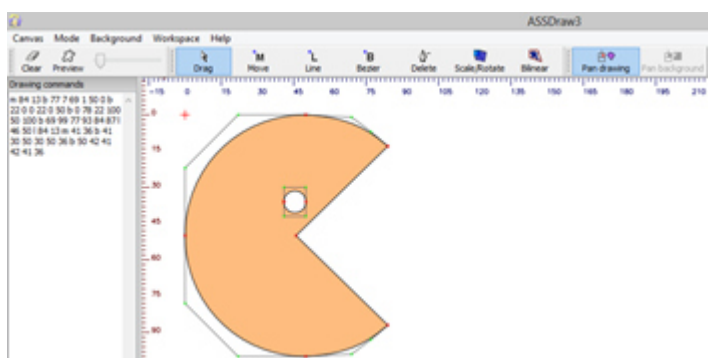
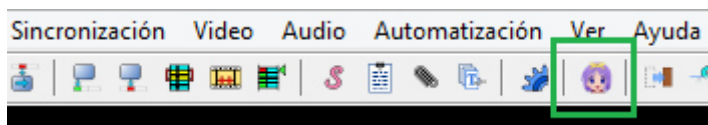
Librería Shape [KE] – Parte 1

En este artículo describe una nueva librería y cada vez estamos más cerca de terminarla 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.



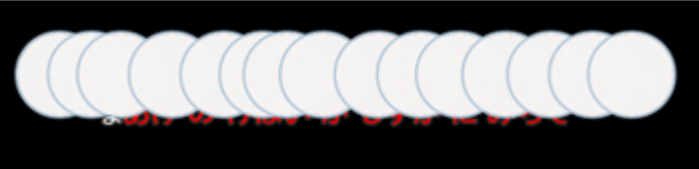
La Librería **shape** hace referencia a dichos dibujos y a las funciones relativas a ellos. Aparte de las funciones en esta Librería, el **Kara Effector** consta de un amplio listado de **shapes** (dibujos) prediseñadas para hacer uso de ellas y a continuación veremos el nombre, su tamaño en píxeles 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:

Return [fx]:	shape.circle
--------------	--------------











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










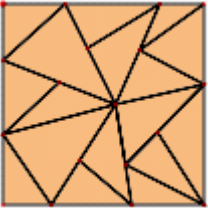
shape.shine4t <div>100 x 100 px</div> 	shape.trebol <div>100 x 106 px</div> 
shape.feather <div>100 x 100 px</div> 	shape.diamond <div>100 x 100 px</div> 
shape.gear <div>100 x 100 px</div> 	shape.bubble <div>100 x 100 px</div> 
shape.note1t <div>100 x 56 px</div> 	shape.note2t <div>100 x 100 px</div> 
Shapes Prediseñadas del Kara Effector	
shape.note3t <div>100 x 100 px</div> 	shape.note4t <div>100 x 100 px</div> 
shape.star <div>100 x 95 px</div> 	shape.star1t <div>100 x 95 px</div> 

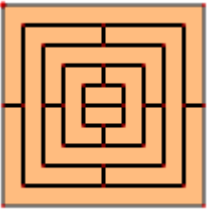
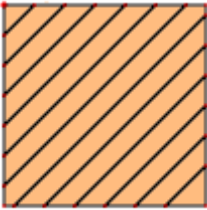

shape.star2t	shape.star3t
100 x 116 px	100 x 116 px
	
shape.star4t	shape.star5t
100 x 116 px	100 x 116 px
	
shape.star6t	shape.star7t
100 x 95 px	100 x 95 px
	
shape.star8t	shape.star9t
100 x 95 px	100 x 116 px
	
Shapes Prediseñadas del Kara Effector	
shape.star10t	shape.sakura
100 x 116 px	100 x 130 px
	

<div>shape.sakura1t</div> <div>100 x 116 px</div> <div></div>	<div>shape.sakura2t</div> <div>100 x 116 px</div> <div></div>
<div>shape.sakura3t</div> <div>100 x 116 px</div> <div></div>	<div>shape.sakura4t</div> <div>100 x 106 px</div> <div></div>
<div>shape.sakura5t</div> <div>100 x 100 px</div> <div></div>	<div>shape.sakura6t</div> <div>100 x 116 px</div> <div></div>
<div>shape.sakura7t</div> <div>100 x 116 px</div> <div></div>	<div>shape.snow1t</div> <div>100 x 108 px</div> <div></div>
<div>shape.snow2t</div> <div>100 x 96 px</div> <div></div>	<div>shape.snow3t</div> <div>100 x 94 px</div> <div></div>

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
	

<div>shape.flower21t</div> <div>100 x 116 px</div> <div></div>	<div>shape.flower22t</div> <div>100 x 116 px</div> <div></div>
<div>shape.flower23t</div> <div>100 x 116 px</div> <div></div>	<div>shape.flower24t</div> <div>100 x 116 px</div> <div></div>
<div>Shapes Prediseñadas del Kara Effector</div>	
<div>shape.flower25t</div> <div>100 x 116 px</div> <div></div>	<div>shape.flower26t</div> <div>100 x 130 px</div> <div></div>
<div>shape.flower27t</div> <div>100 x 116 px</div> <div></div>	<div>shape.flower28t</div> <div>100 x 116 px</div> <div></div>
<div>shape.flower29t</div> <div>100 x 116 px</div> <div></div>	<div>shape.cristal17</div> <div>100 x 100 px</div> <div></div>

shape.geometric10 100 x 100 px 	shape.diagonal13r 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.

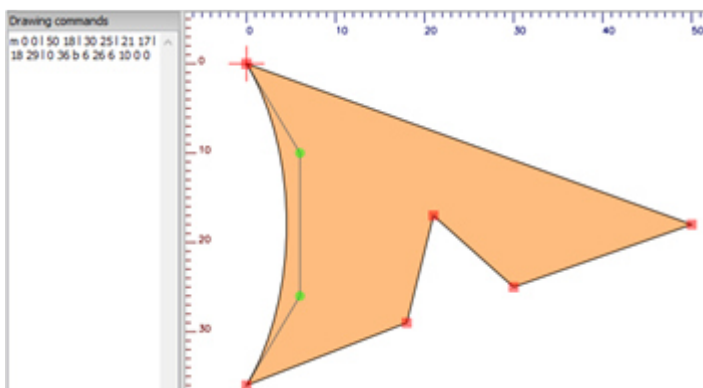
Con el anterior listado de Shapes prediseñadas del **Kara Effector**, comenzamos con la descripción de las funciones de la Librería **shape**, en donde algunas de ellas nos sirven para modificar y crear nuestras propias Shapes o, generar efectos directamente.

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 | 50 18 | 30 25 | 21 17 | 18 29 | 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 )
```

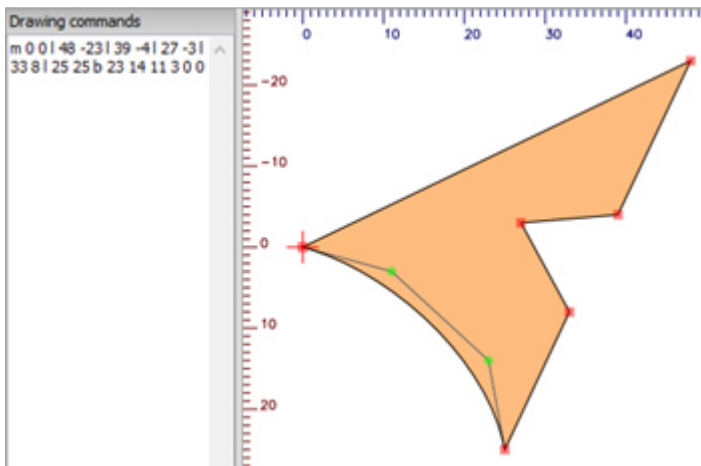
Opción 2:

```
Return [fx]:  
shape.rotate( "m 0 0 | 50 18 | 30 25 | 21 17 | 18 29 | 0 36  
b 6 26 6 10 0 0 ", 45 )
```

En vídeo:



En el **AssDraw3**:



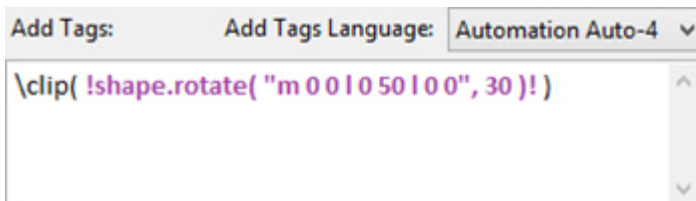
La **shape** ha rotado 45° respecto al punto **P = (0, 0)**, ya que en el ejemplo anterior se omitieron los parámetros **x** e **y**.

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 1 0 50 1 0 0)`

Como la **shape** “m 0 0 1 0 50 1 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:



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)**
-

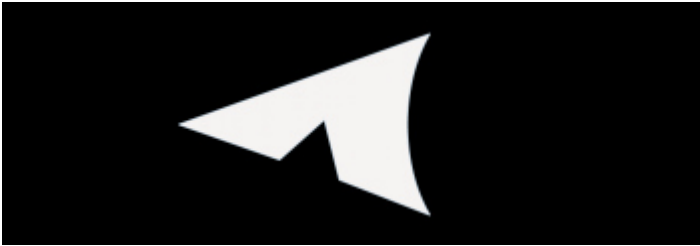
shape.reflect(Shape, Axis): refleja a la shape respecto al eje asignado en el parámetro **Axis**. Ejemplo:

Return [fx]:

```
shape.reflect( mi_shape, 'y' )
```

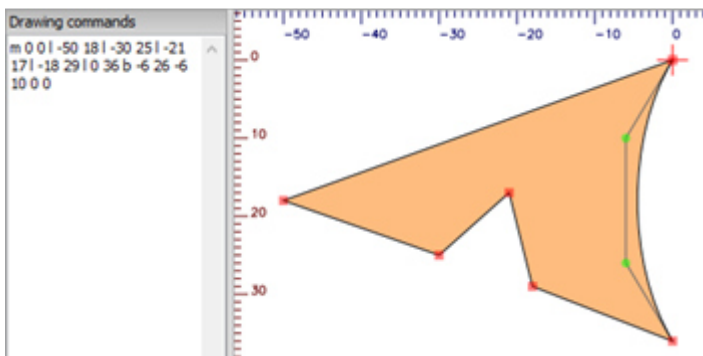
El parámetro **Axis** puede ser “x” para hacer referencia al eje “x”, o “y” para hacer referencia al eje “y”, como se acaba de usar en el ejemplo inmediatamente anterior.

Lo que resultaría así:



Effector [Fx] *m -0 0 | -50 18 | -30 25 | -21 17 | -18 29 | -0 36 b -6 26 -6 10 -0 0

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.

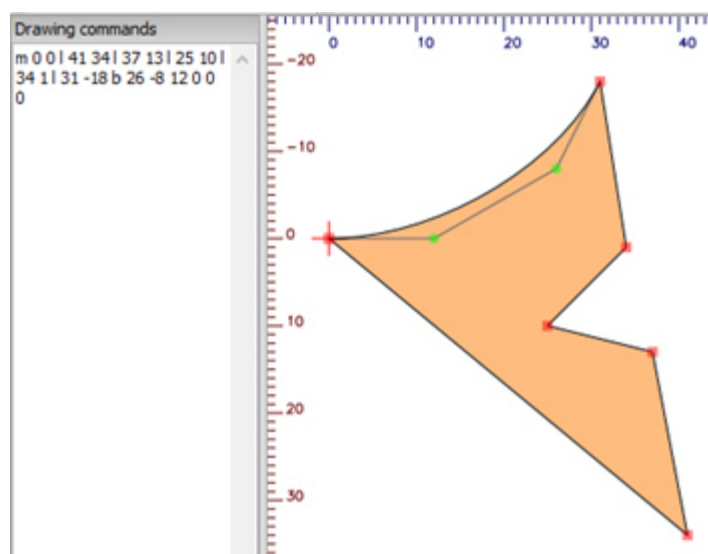
Veamos un ejemplo combinando las dos funciones hasta acá vistas de la Librería **shape**:

```
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":

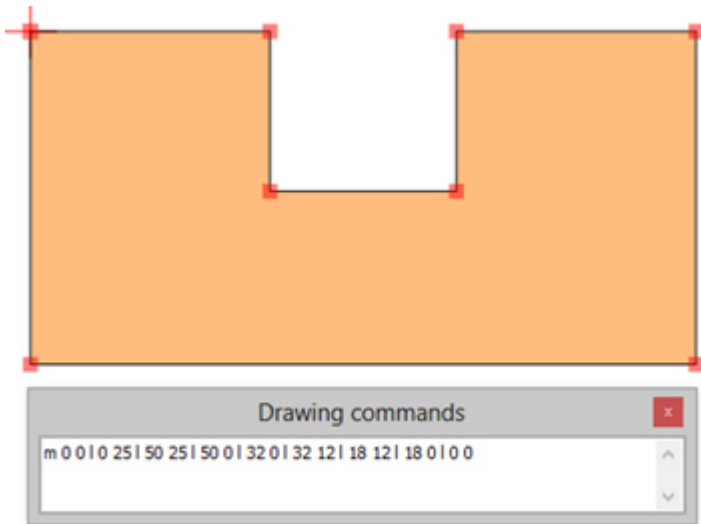
```
Return [fx]:  
shape.reflect( shape.rotate( mi_shape, 60 ), "x" )
```

Y en el **AssDraw3** podemos ver el suceso anteriormente descrito:

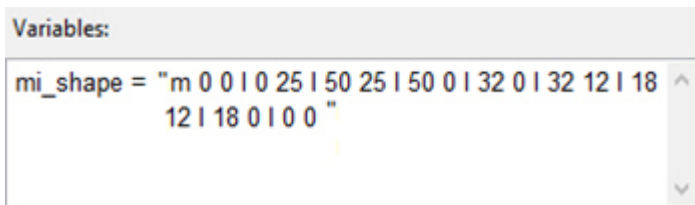


Son muchas las posibilidades al combinar tan solo estas dos funciones. Espero que puedan inventar sus propios ejemplos y que se puedan sorprender con los diferentes resultados.

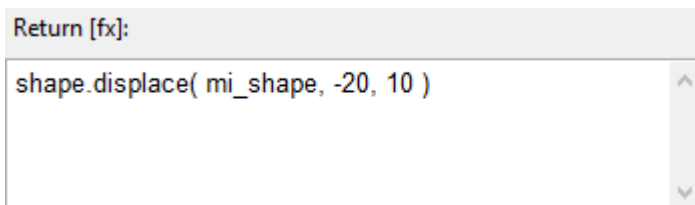
Para los siguientes ejemplos en las definiciones de las funciones de esta Librería, usaré esta simple **shape** que mide 50 X 25 px, pero ustedes pueden usar la que quieran y aun así los resultados deben ser visibles:



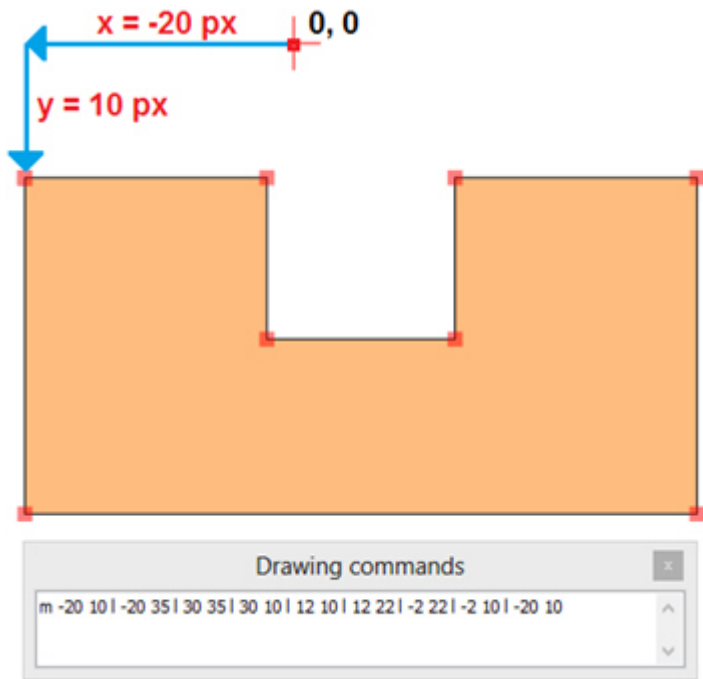
Y para mayor comodidad, la declaro en “**Variables**”:



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:



Retorna la misma **shape**, pero desplazada 20 pixeles a la izquierda ($x = -20$) y 10 pixeles hacia abajo ($y = 10$):

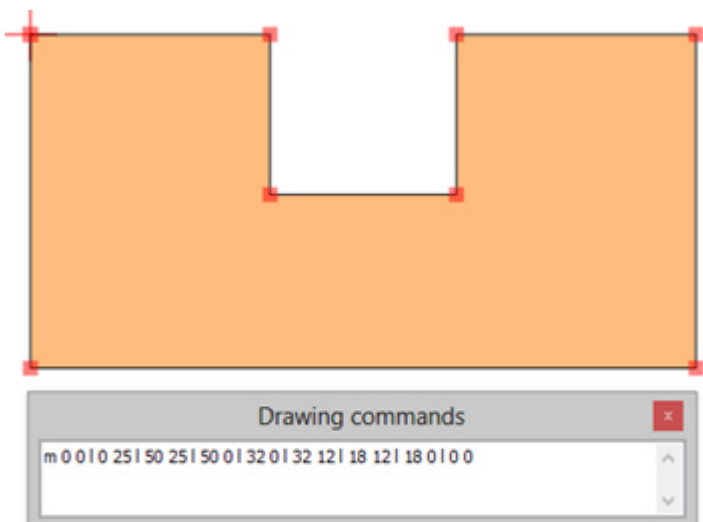


Recordemos que en el **AssDraw3** y en el formato .ass, el eje positivo de “y” es hacia abajo del eje “x” y el negativo, hacia arriba del mismo.

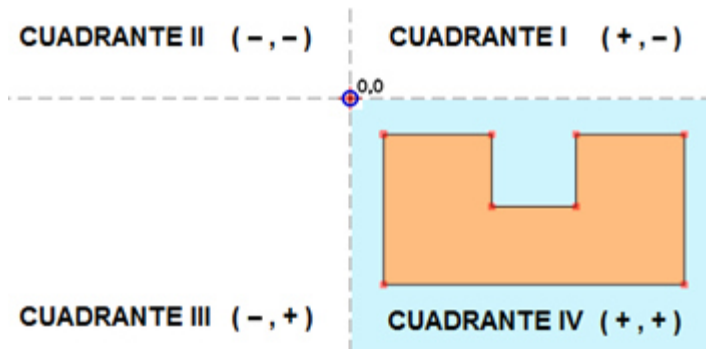
shape.origin(shape): desplaza la **shape** tantos pixeles respecto a ambos ejes cartesianos, hasta ubicarla en el **Cuadrante IV** del plano en el **AssDraw3**, respecto al punto de origen $P = (0, 0)$. Para el siguiente ejemplo usaré la **shape** desplazada del ejemplo anterior:

```
Return [fx]:
shape.origin( "m -20 10 | -20 35 | 30 35 | 30 10 | 12 10 | 12 22 | -2 22 | -2 10 | -20 10 " )
```

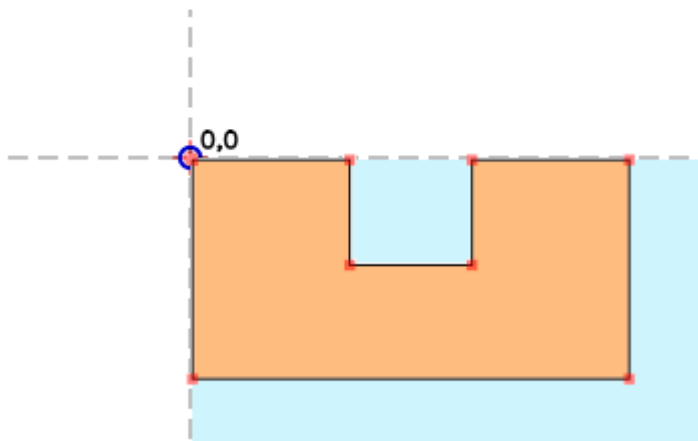
Entonces la función la ubicará en el **Cuadrante IV**:



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:



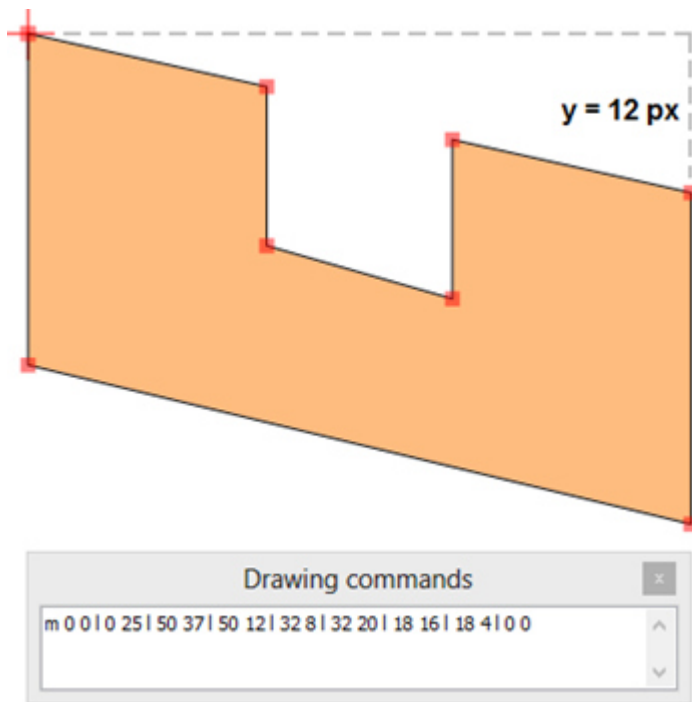
shape.oblique(shape, Pixels, Axis): deforma la **shape** de manera oblicua, en tantos pixeles indicados en el parámetro **Pixels**, respecto al eje asignado **Axis**.

- Ejemplo 1:

Return [fx]:

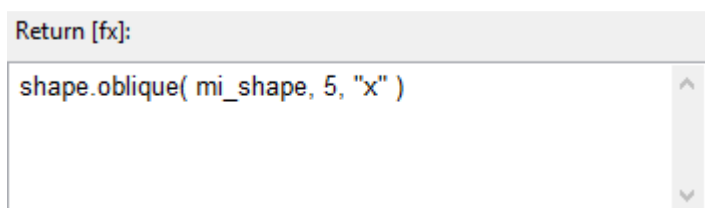
```
shape.oblique( mi_shape, 12, "y" )
```

Entonces la función deforma la **shape** 12 pixeles positivos respecto al eje “y”:

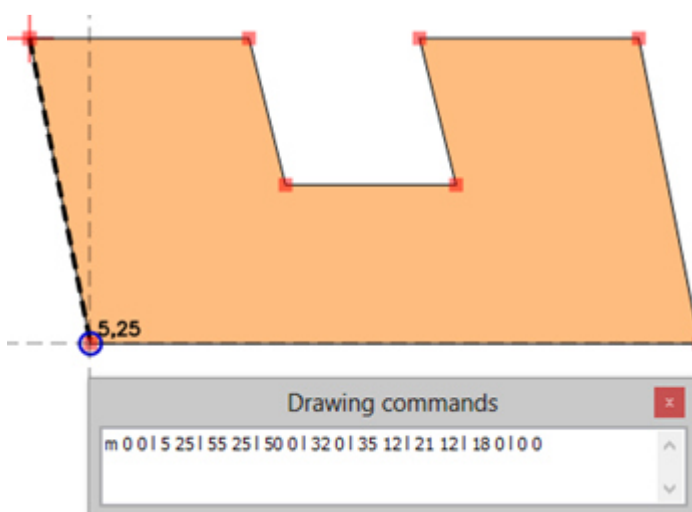


Todas las coordenadas en “x” se conservan, y las de “y” son desplazadas de forma progresiva hasta que aquellas que acompañan a las coordenadas “x” más alejadas del origen, se desplacen la cantidad de pixeles asignados en el parámetro **Pixels** (12 px). **Pixels** también puede ser un valor negativo, lo que deformaría la **shape** hacia arriba.

- Ejemplo 2:



En este caso, la **shape** se deformará 5 pixeles hacia la derecha, dado que **Pixels** es positivo:



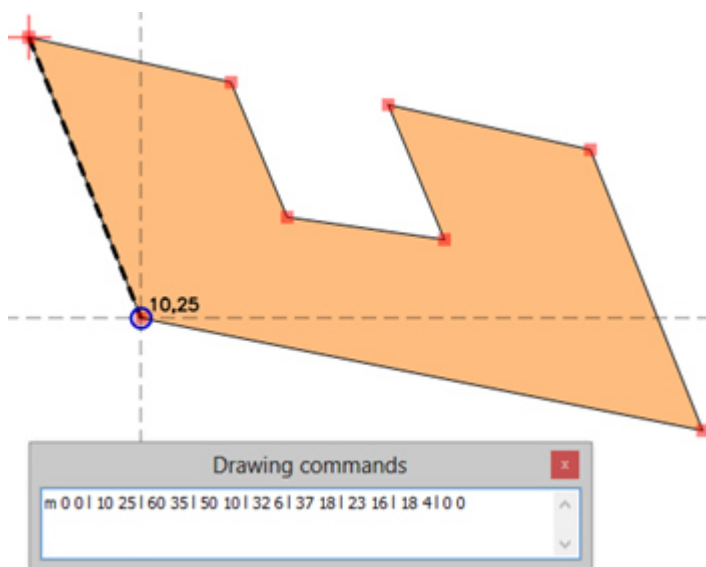
En el anterior ejemplo, las coordenadas que se conservan intactas en las **shape** son las de “y”, y las coordenadas de “x” se deforman de forma progresiva.

- 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).



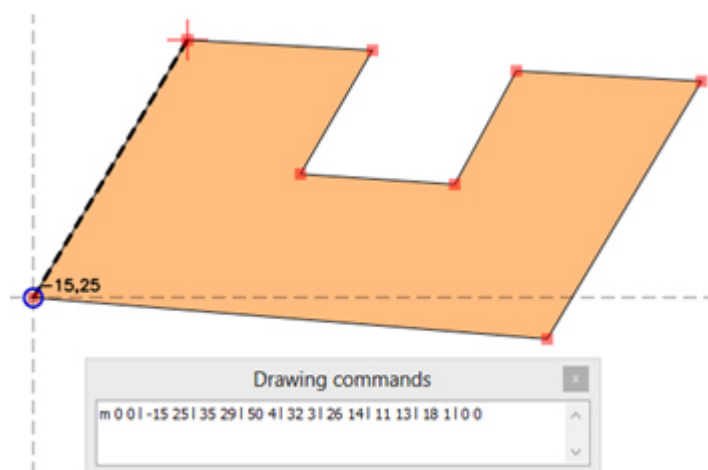
- Ejemplo 4:

Con este método podemos decidir la cantidad de pixeles en que se deformará la **shape** en ambos ejes:

Return [fx]:

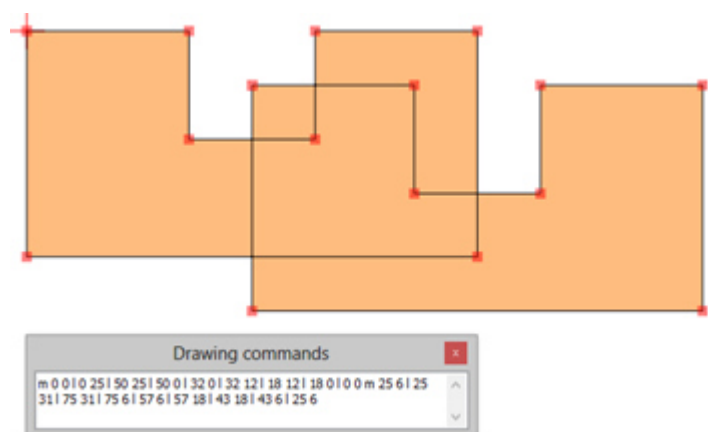
```
shape.oblique( mi_shape, { -15, 4 } )
```

Vemos cómo la **shape** se deformó 15 píxeles a la izquierda ($x = -15$ px) y 4 píxeles hacia abajo ($y = 4$ px):

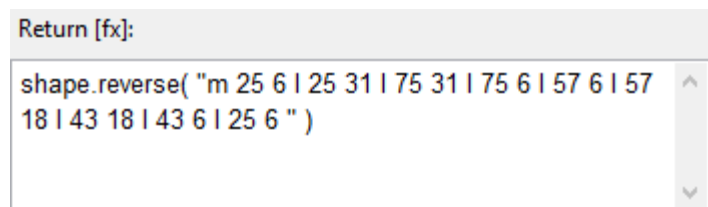


shape.reverse(shape): esta función reescribe la **shape** de manera que quede exactamente igual, pero dibujada a la inversa para que pueda ser sustraída de otra.

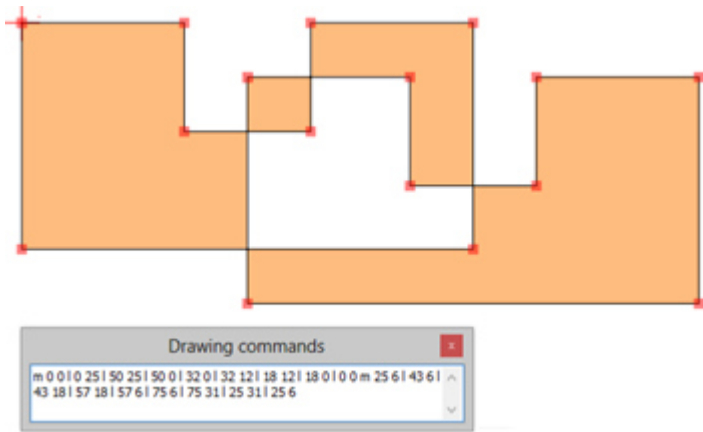
Para el siguiente ejemplo, dupliqué la misma **shape** y la desplacé varios píxeles respecto a la original, de manera que queden superpuestas como podemos ver en la siguiente imagen:



Ahora, aplicaremos la función a esa segunda **shape** para que sea redibujada de manera inversa:

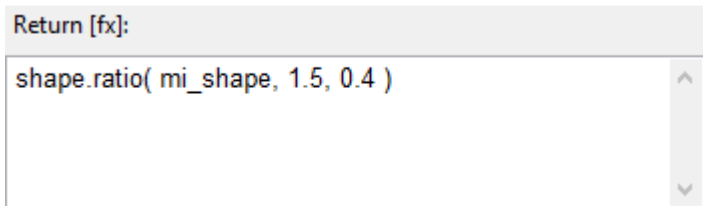


Las Shapes siguen siendo las mismas, pero el área que coincide entre ambas es sustraída, ya que una **shape** está dibujada en un sentido (sentido anti horario) y la otra a la inversa (sentido horario):

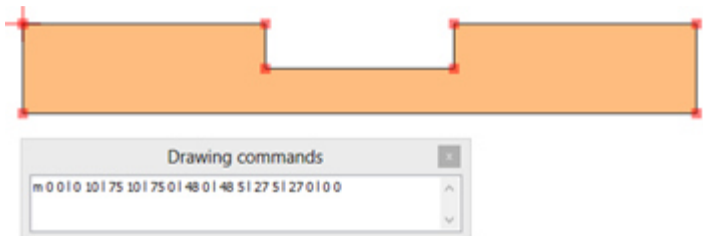


shape.ratio(shape, ratio_x, ratio_y): esta función redimensiona la **shape** en una proporción equivalente a **ratio_x** y **ratio_y**.

- Ejemplo 1:



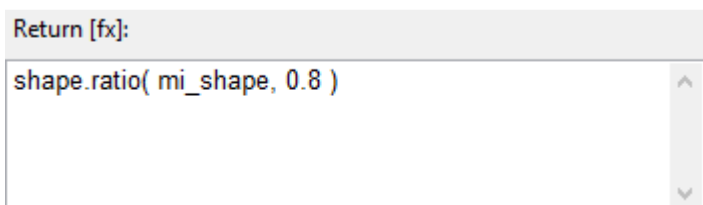
ratio_x = 1.5, es decir que la **shape** ahora es 1.5 veces más ancha de lo que era originalmente (150 %):



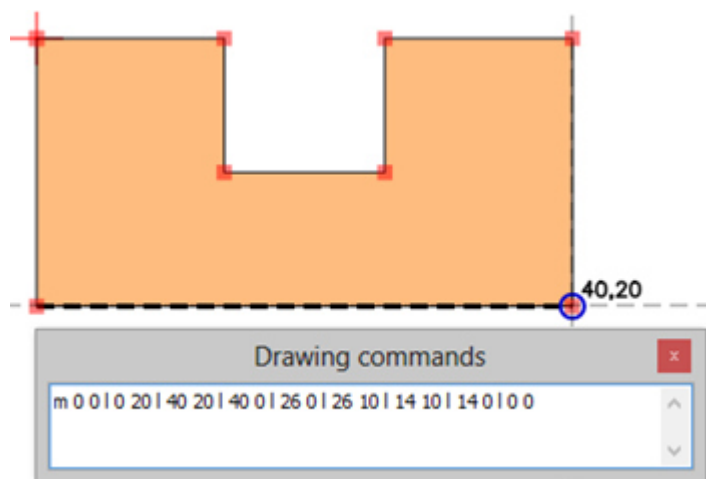
ratio_y = 0.4, es decir que la altura de la **shape** solo el 40% de la altura original.

- 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:



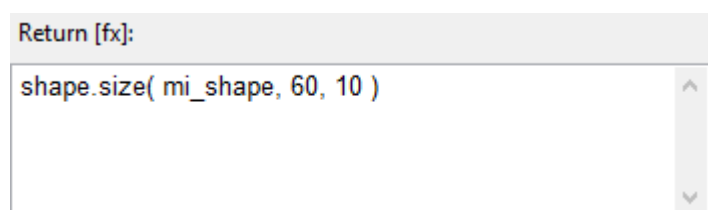
La **shape** que retorna es casi la misma, solo que su tamaño es un 80% (**ratio_x** = 0.8) del tamaño de la **shape** original:



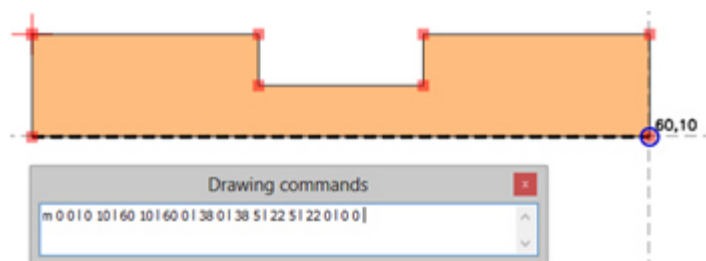
shape.size(shape, size_x, size_y): esta función es similar a la función **shape.ratio**, pero con la diferencia que redimensiona la **shape** de tal manera que el ancho de la misma determinado según el parámetro **size_x** en pixeles, y su altura será determinada por el parámetro **size_y**, también en pixeles.

- Ejemplo 1:

De este modo, la **shape** quedará midiendo 60 X 10 px:

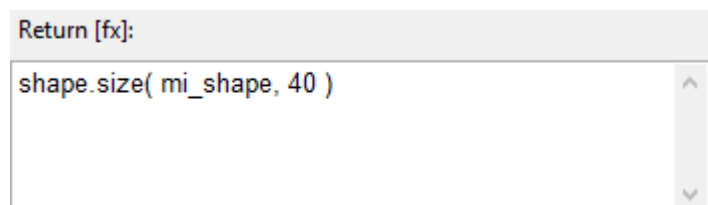


Como se evidencia, tanto en la imagen de la **shape** como en el código de la misma, las dimensiones de la **shape** son las ingresadas en la función (60 X 10 px):



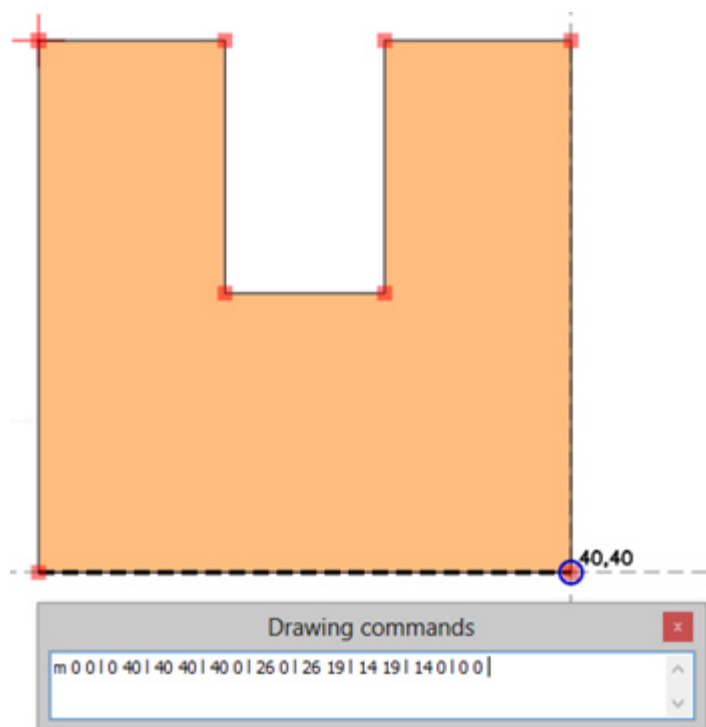
- Ejemplo 2:

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**:



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:

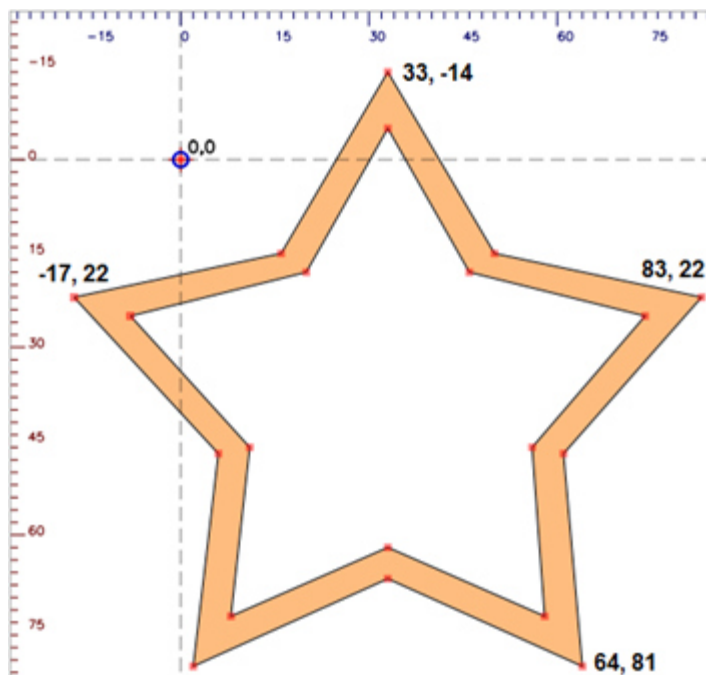
1. **minx**
2. **maxx**
3. **miny**
4. **maxy**
5. **w_shape**
6. **h_shape**

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 | 6 47 | 2 81 | 33 67 | 64  
81 | 61 47 | 83 22 | 50 15 | 33 -14 | 16 15 | -17 22 m -8 25 |  
20 18 | 33 -5 | 46 18 | 74 25 | 56 46 | 58 73 | 33 62 | 8 73 |  
11 46 | -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.

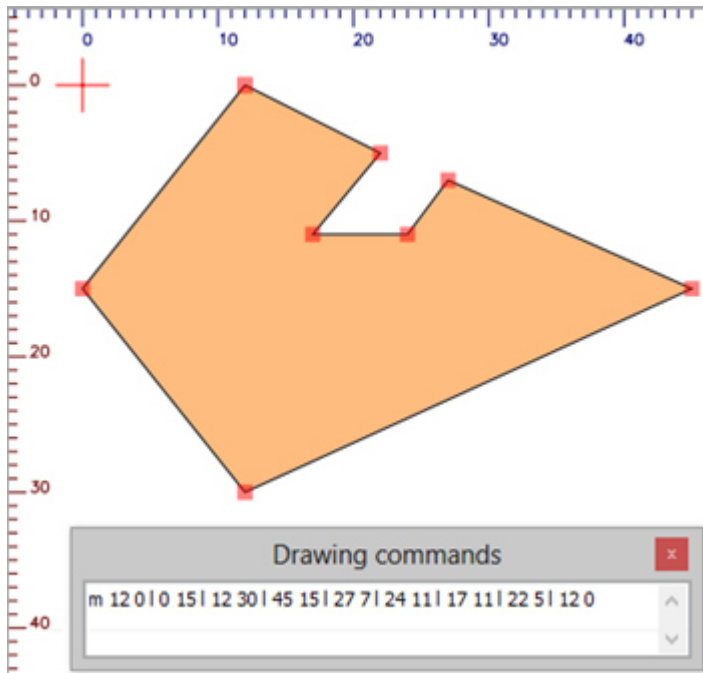
Las anteriores seis variable ya pueden ser usadas como valores numéricos en cualquier otra celda de texto de la ventana de modificación del **Kara**

Effector. Ejemplo:

Add Tags:	Add Tags Language:	Lua
<code>"\fscy" .. h_shape</code>		

Que a la postre retornará: `\fscy85`, dado que la altura de la **shape** ingresada era de 85 px.

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 | 0 15 | 12 30 | 45 15 | 27 7 | 24 11
            | 17 11 | 22 5 | 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**).

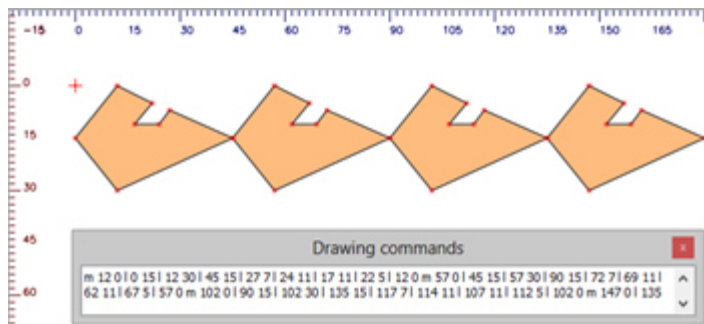
Esta función tiene tres modalidades distintas, y cada una tiene una serie de opciones que veremos a continuación en los siguientes ejemplos:

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°:

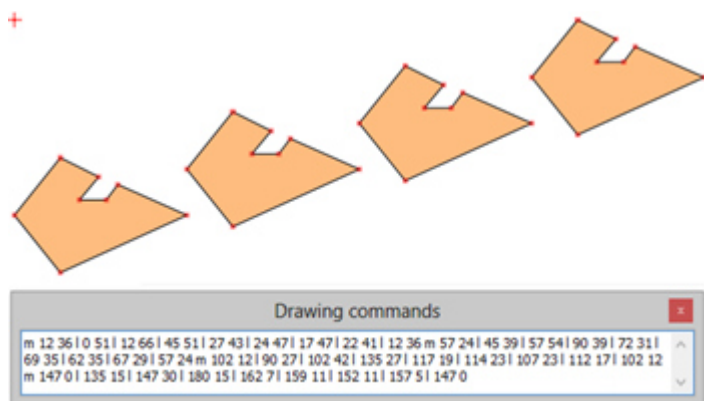


- **Ejemplo 2.** Aparte de la **shape** y la cantidad de repeticiones, el parámetro **A_mode** es 15, que es equivalente a un ángulo de 15°.

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 )
```

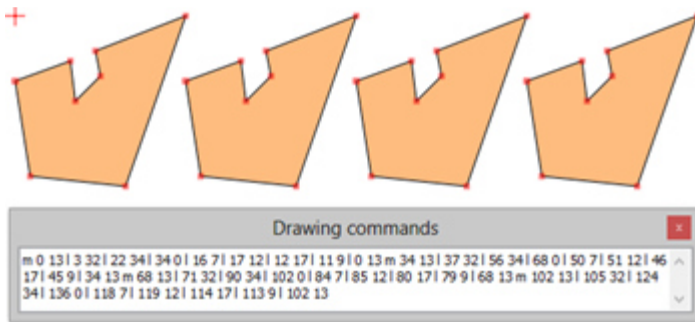
El Arreglo es Lineal y con un ángulo de 15°:



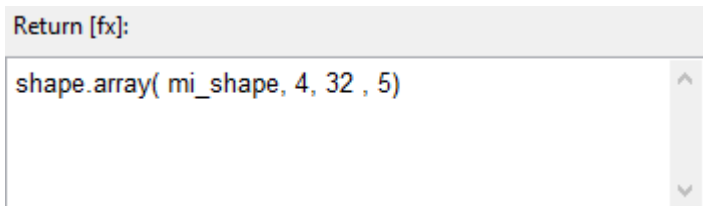
- **Ejemplo 3.** **A_mode** ahora es una **tabla** con dos valores numéricos, el primero indica el ángulo del arreglo lineal y el segundo, la rotación respecto al centro de la **shape**:

```
Return [fx]:
shape.array( mi_shape, 4, {0, 45} )
```

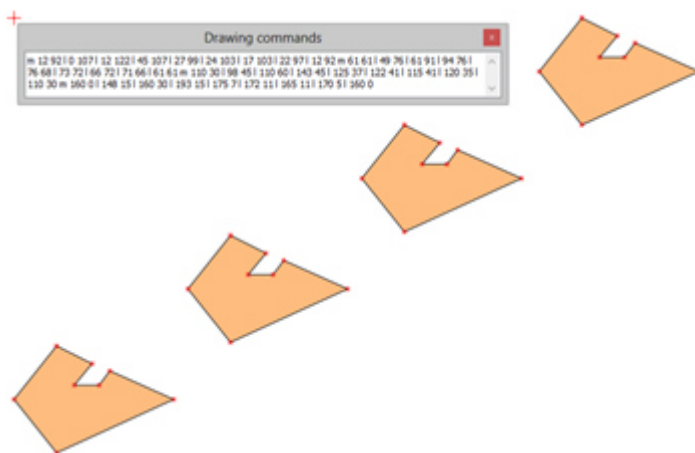
Entonces el ángulo del arreglo es 0 y la **shape**, antes de ser duplicada, se rotó un ángulo de 45°:



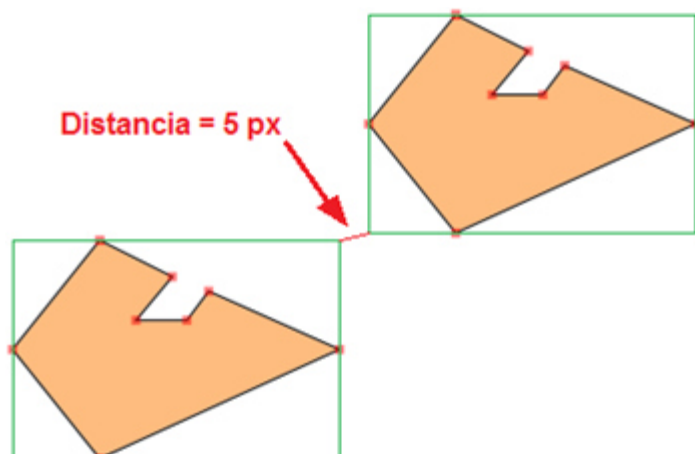
- **Ejemplo 4.** Incluimos al parámetro **Dxy** como valor numérico, que hace referencia a la distancia en px que separará a la **shape** dentro del arreglo:
 - loops : 4
 - Ángulo del Arreglo: 32°
 - Distancia Separadora: 5 px



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:



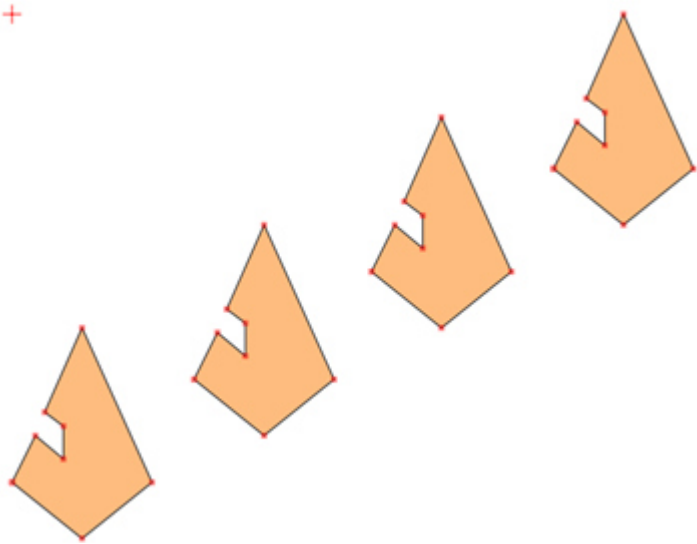
La distancia de separación también puede ser un valor numérico negativo, y lo que hará que el Arreglo Lineal quede más compacto y se acerquen tanto, una shape a la otra, hasta que se superpongan, si eso quisiéramos.

- **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:

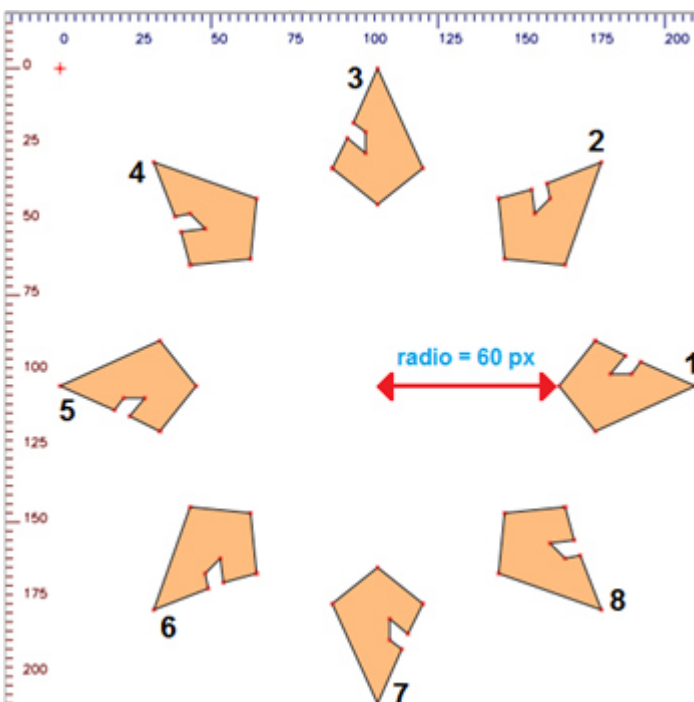
Return [fx]:

```
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



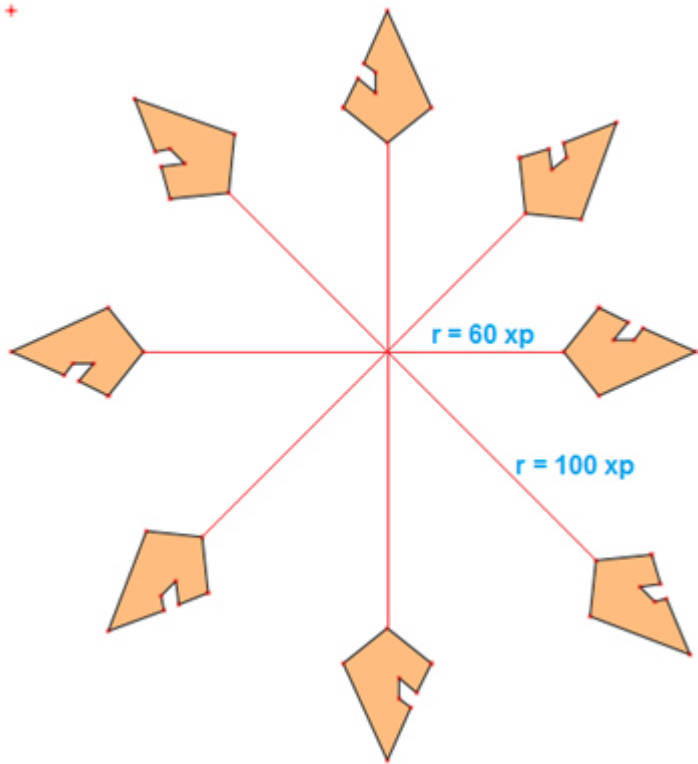
El radio es constante para cada una de las repeticiones del Arreglo (60 px), y cada una de las Shapes está rotada de tal manera que quedan orientadas en dirección del centro imaginario del Arreglo.

- **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} )
```

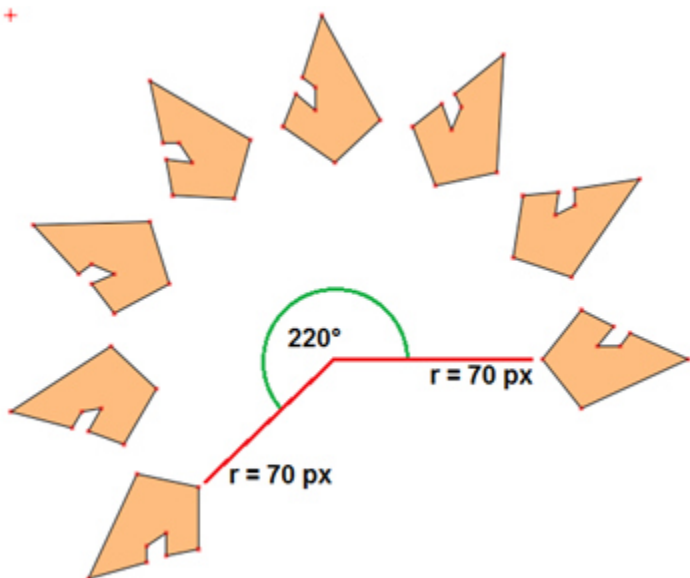
Los radios del Arreglo van aumentando progresivamente desde el Radio Inicial (60 px) hasta el Radio Final (100 px):



- **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} )
```

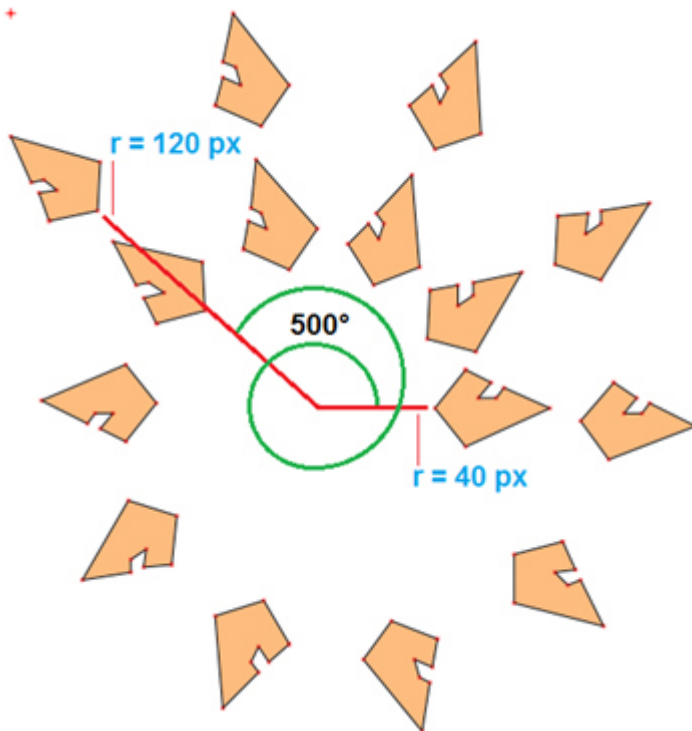


- **Ejemplo 3.1.** Aumentamos el valor del ángulo del Arreglo de tal manera que exceda los 360° de la circunferencia y, ponemos al Radio Inicial y al Radio Final con diferentes valores para evitar que las Shapes se superpongan:

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°



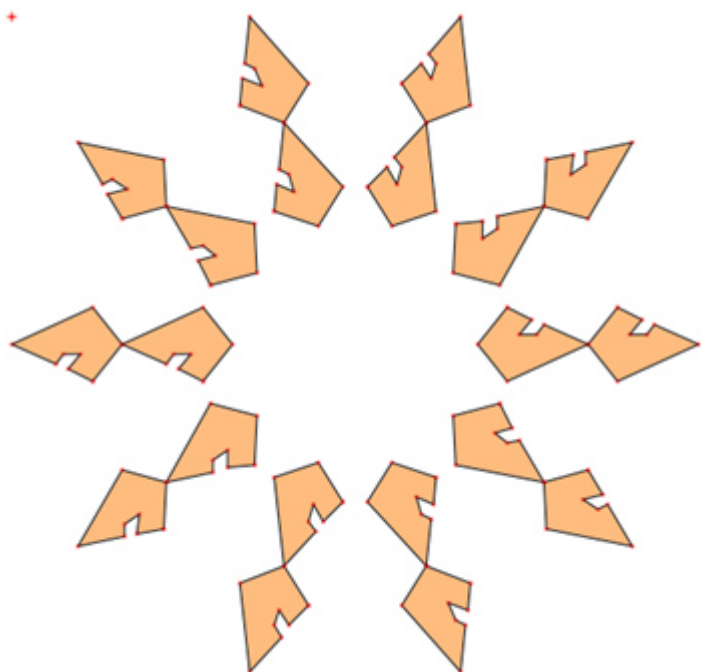
- **Ejemplo 4.** El parámetro **loops** es una **tabla** con dos valores numéricos, el primero indica las veces que se repite la **shape** en el Arreglo Radial, y el segundo indica la veces que se repite el Arreglo de forma y tamaño mayormente progresivo:

Return [fx]:

```
shape.array( mi_shape, {10, 2}, "radial", 50 )
```

- loops del Arreglo: 10
- Repeticiones: 2
- loops Total: $10 \times 2 = 20$
- 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:

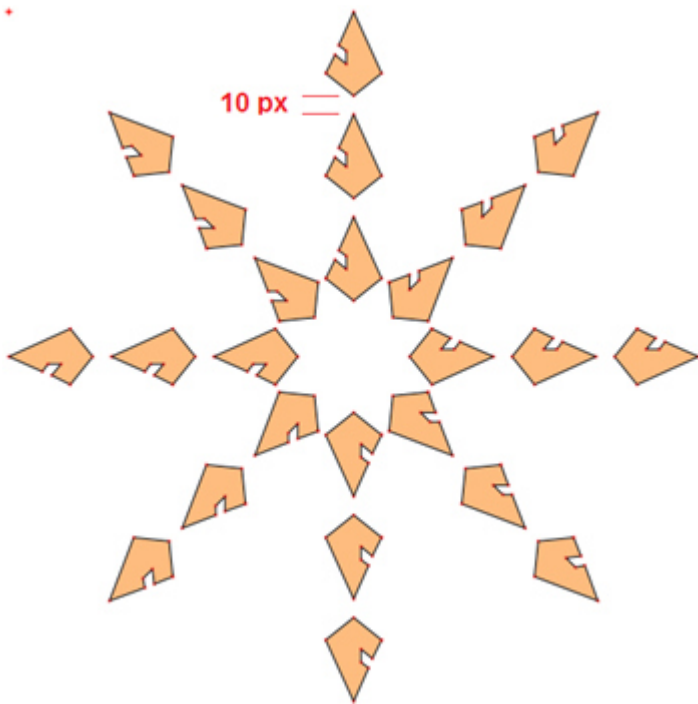


- **Ejemplo 5.** El parámetro **Dxy** es ahora una **tabla** con dos valores numéricos, el primero equivale al radio interior del primer Arreglo y el segundo, a la distancia en pixeles que separará a cada uno de los Arreglos:

Return [fx]:

```
shape.array( mi_shape, {8, 3}, "radial", {30, 10} )
```

- loops del Arreglo: 8
- Repeticiones: 3
- loops Total: $8 \times 3 = 24$
- 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 | 0 15 | 12 30 | 45 15 | 27 7 | 24 11  
| 17 11 | 22 5 | 12 0 ", shape.size(shape.circle, 24) }
```

Círculo de 24 px

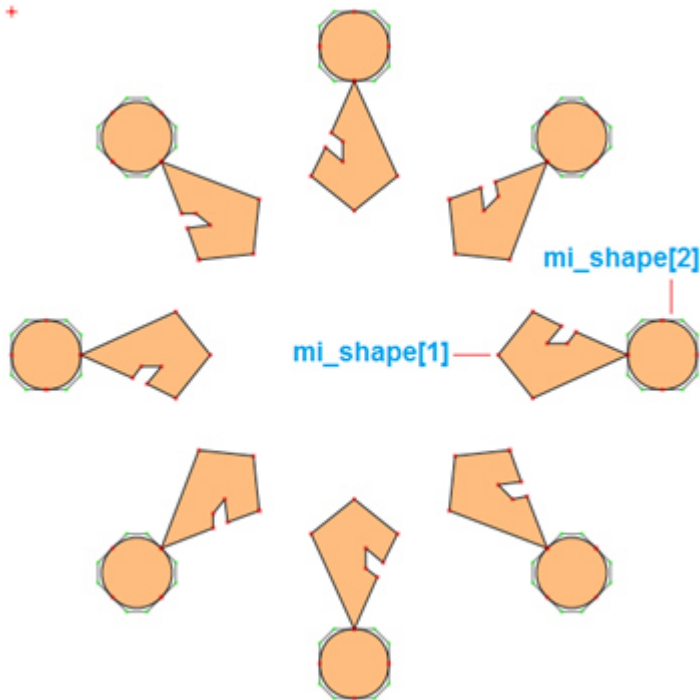
La **tabla mi_shape** de ser completamente de Shapes para poder ser usada dentro la función, así que pueden usar las de estos ejemplos o las que ustedes quieran. La cantidad de Shapes de la **tabla** es ilimitada.

- **Ejemplo 6.** **mi_shape** es una **tabla** de Shapes:
 - loops del Arreglo: 8
 - Repeticiones: 2
 - loops Total: $8 \times 2 = 16$
 - 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]**.

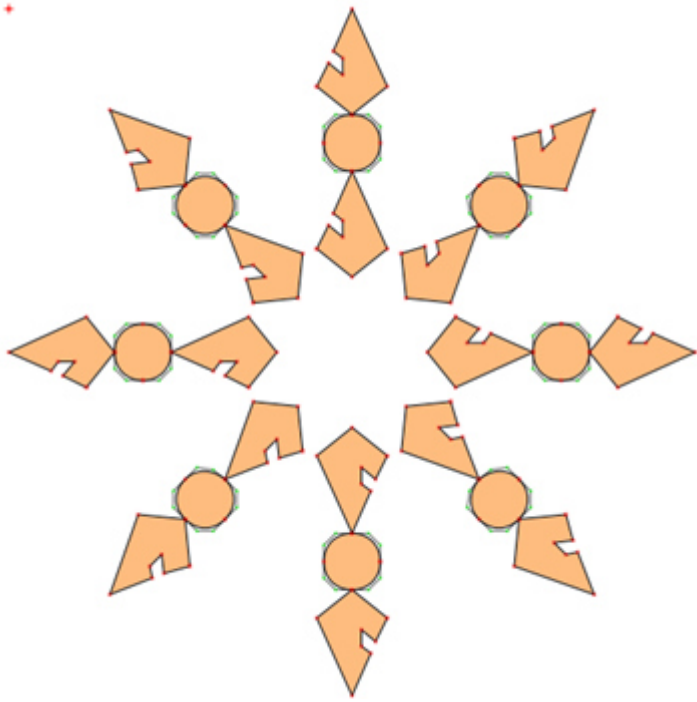
La distancia que separa un Arreglo respecto a otro es 0 px, por default, pero también la podemos modificar.

- **Ejemplo 6.1. mi_shape** es una **tabla** de Shapes:
 - loops del Arreglo: 8
 - Repeticiones: 3
 - loops Total: $8 \times 3 = 24$
 - 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**:

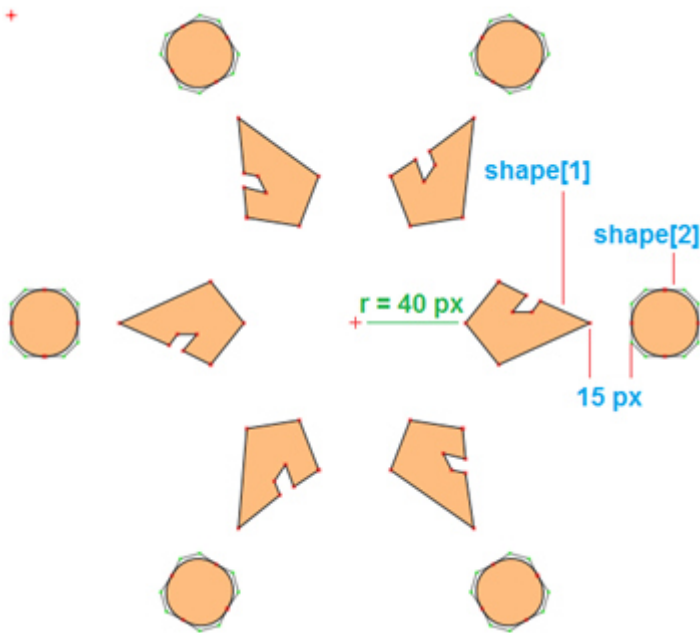


- **Ejemplo 7.** **mi_shape** es una **tabla** de Shapes y en **Dxy**, en forma de **tabla**, modificamos la distancia que separará a cada uno de los Arreglos:

Return [fx]:

```
shape.array( mi_shape, {6, 2}, "radial", {40, 15} )
```

- loops del Arreglo: 6
- Repeticiones: 2
- loops Total: $6 \times 2 = 12$
- Modo: "radial"
- Radio Interior del primer Arreglo: 40 px
- Distancia Separadora: 15 px



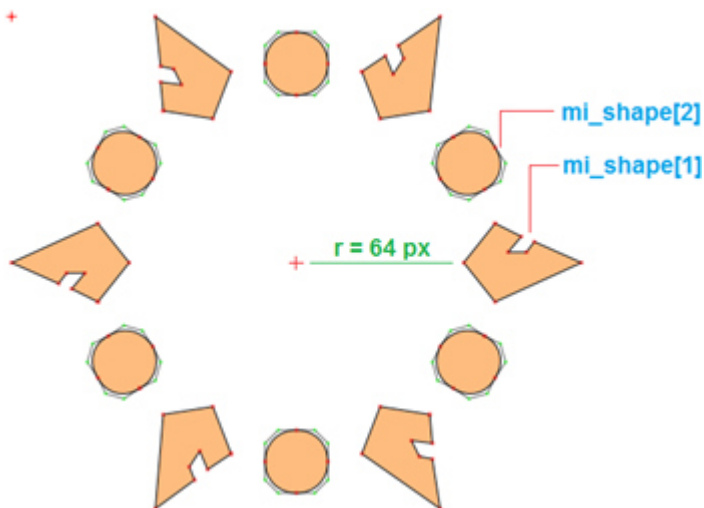
- **Ejemplo 8.** Ingresamos la **tabla mi_shape**, el **loops** vuelve a ser un valor numérico, que para este ejemplo es 6:

Return [fx]:

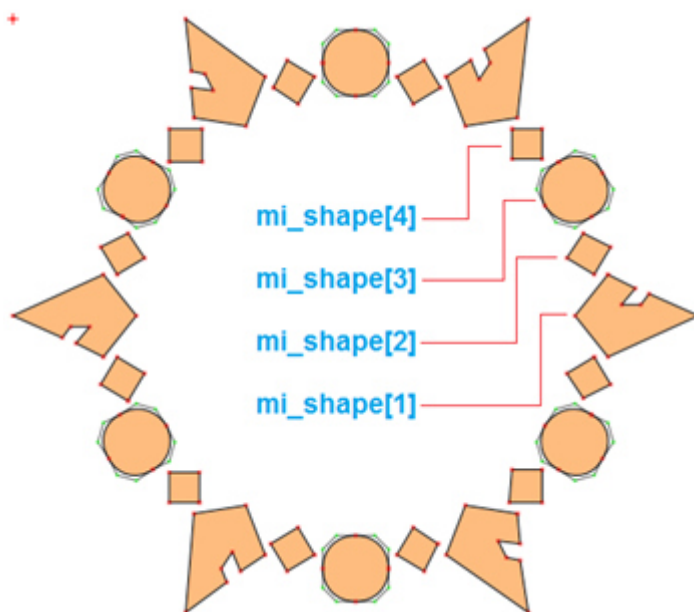
```
shape.array( mi_shape, 6, "radial", 64 )
```

- loops de cada **shape** del Arreglo: 6
- Tamaño de la **tabla**: 2
- loops Total: $6 \times 2 = 12$
- 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.

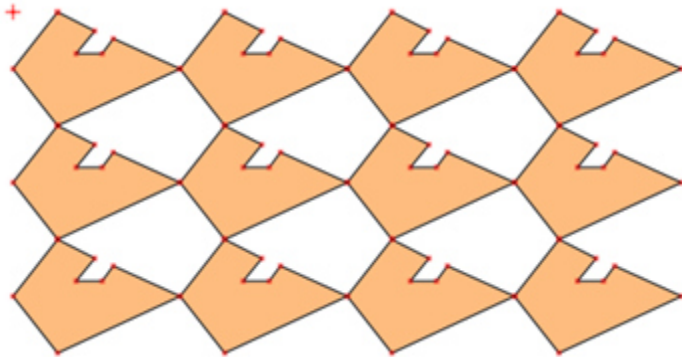
Ejemplo 1. Nuevamente iniciamos con **mi_shape** como una **shape**, declarada en forma de variable. El parámetro **loops** siempre debe ser una **tabla** con dos valores numéricos en donde se dan la cantidad de repeticiones, el primer número indica las repeticiones de la **shape** respecto al eje “x” (a lo ancho) y el segundo número indica la cantidad de repeticiones respecto al eje “y” (a lo alto). Por último, en **A_mode** ponemos la palabra “**array**”:

```
Variables:
mi_shape = "m 12 0 | 0 15 | 12 30 | 45 15 | 27 7 | 24 11 |
            17 11 | 22 5 | 12 0 "
```

Y en **Return [fx]** ponemos así:

```
Return [fx]:  
shape.array( mi_shape, {4, 3}, "array" )
```

Se generará el siguiente arreglo:

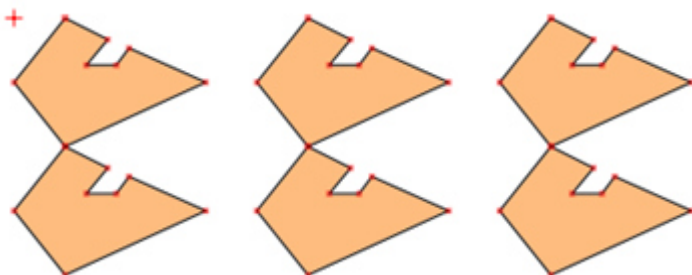


- loop Horizontal: 4
- loop Vertical: 3
- loops Total: $4 \times 3 = 12$
- Modo: "array"
- Distancia Separadora: 0 px (por default)

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: $3 \times 2 = 6$
- Modo: "array"
- Distancia Separadora Horizontal: 12 px
- Distancia Separadora Vertical: 0 px (default)

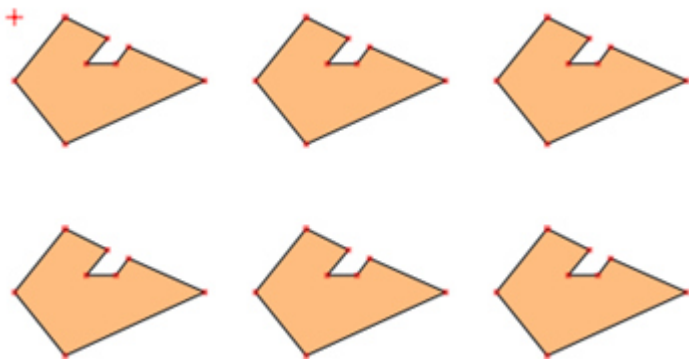


Ejemplo 3. Modificamos a **Dxy** a una **tabla** con dos valores numéricos, el primero indicará la distancia separadora horizontal y el segundo la vertical:

Return [fx]:

```
shape.array( mi_shape, {3, 2}, "array", {12, 20} )
```

- loop Horizontal: 3
- loop Vertical: 2
- loops Total: $3 \times 2 = 6$
- Modo: "array"
- Distancia Separadora Horizontal: 12 px
- Distancia Separadora Vertical: 20 px

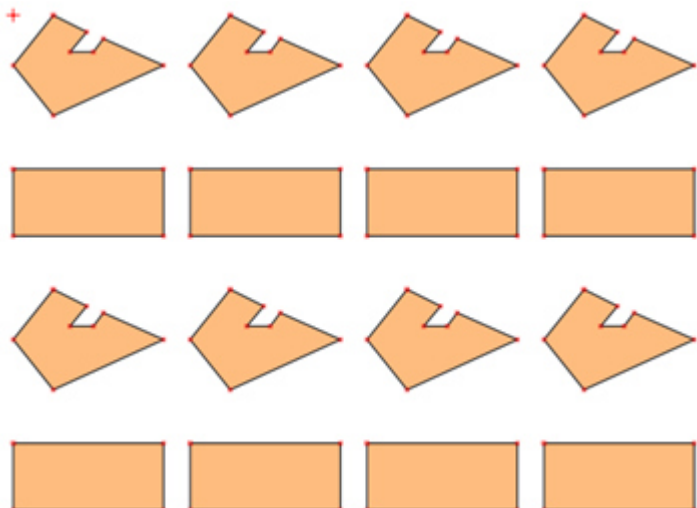


Ejemplo 4. **mi_shape** la usamos ahora como una **tabla** de Shapes, la cantidad de elementos de dicha **tabla** es decidida por cada quien:

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**:

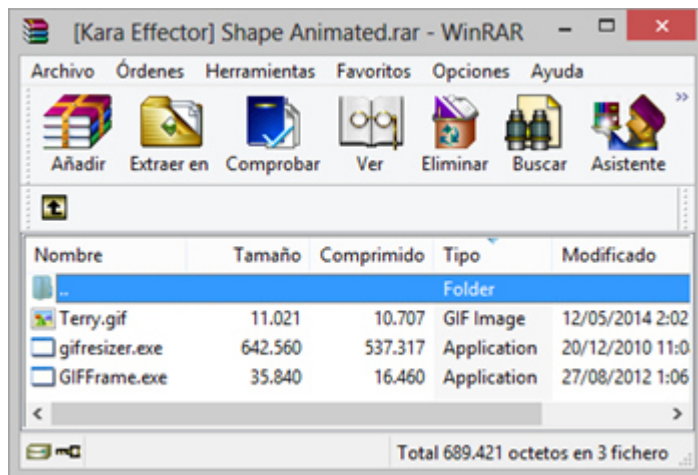


shape.animated(dur, frame_dur, frames, Sx, Sy): retorna Shapes rectangulares que contienen imágenes en formato **PNG** para hacer efectos de animaciones.

Esta función es de uso exclusivo para el filtro **VSFilterMod**, ya que está basado en el tag **\img**, 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**.

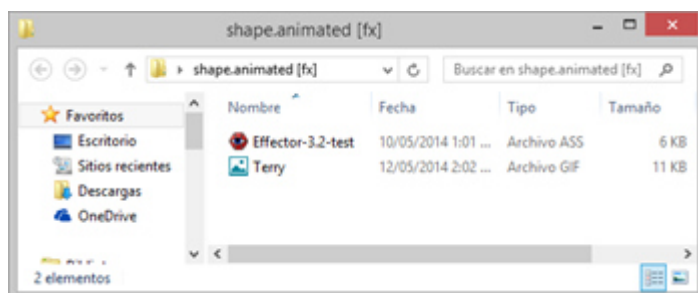
Para el siguiente ejemplo he subido un **.rar** que contiene tres archivos, un archivo **GIF** y dos aplicaciones que no requieren de instalación, que nos ayudarán a manipular y modificar los archivos **GIF**:

[\[Kara Effector\] 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 necesitamos, 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.

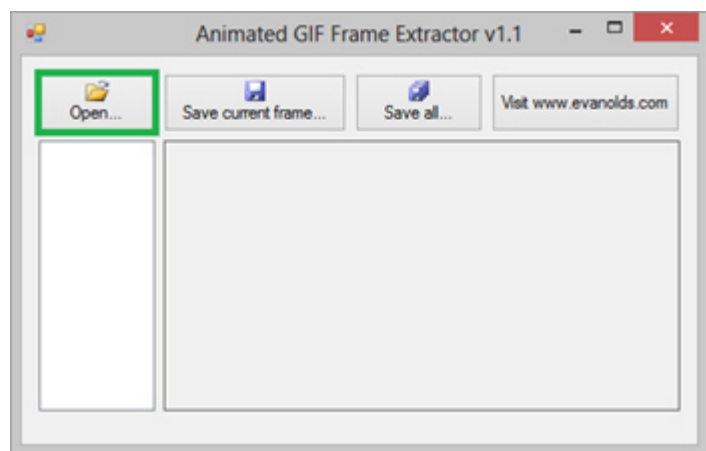
El primer paso para usar la función **shape.animated** es tener el archivo **.ass** y el archivo **GIF** en la misma carpeta. No importa el nombre ni la ubicación de la carpeta, lo que importa es que ambos archivos estén en la misma:



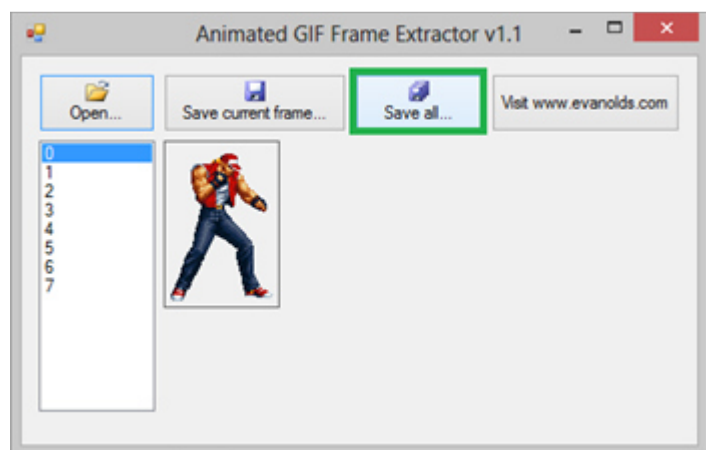
En el caso que nuestra **GIF** tenga un tamaño superior o inferior al que necesitamos (aunque no es recomendable ampliar un **GIF**, ya que la imagen pixela y empieza a perder calidad), abrimos la aplicación **GIFResizer** y en la parte inferior le damos las dimensiones en pixeles que se ajuste a nuestras necesidades. Esta aplicación crea un nuevo archivo, en tal caso se debe guardar en la misma carpeta del archivo **.ass** y del **GIF** original:



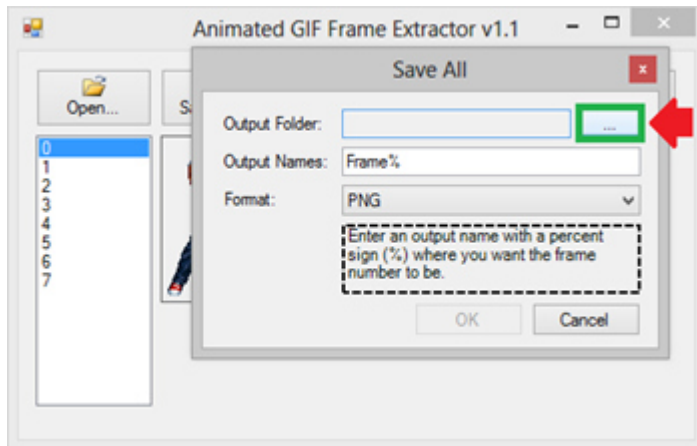
Para este ejemplo, no hubo la necesidad de cambiar el **GIF** de tamaño, pero en cualquiera de los dos casos, el tercer paso es abrir la aplicación **GIFFrame** y le damos al botón que pone “**Open...**”:



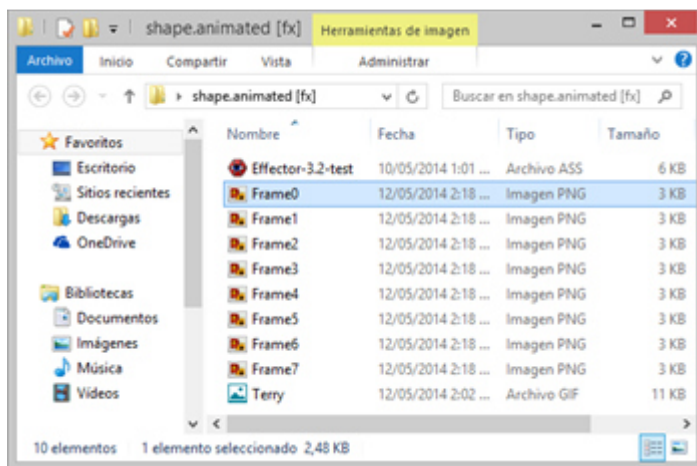
Ubicamos el **GIF** que vamos a utilizar, y pulsamos el botón “**Save All...**”:



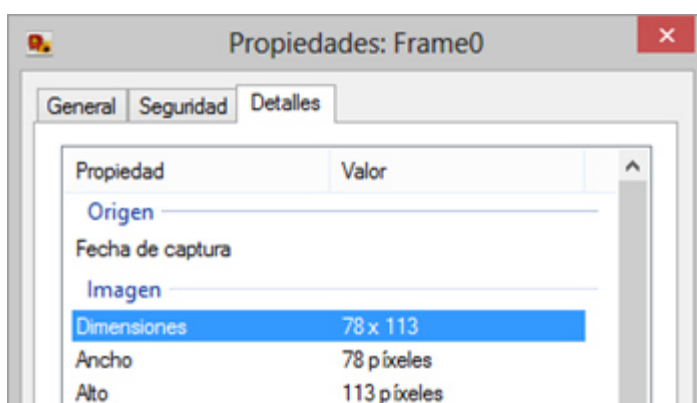
Y en donde pone “**Output Folder**” ubicamos la carpeta en donde están el archivo **.ass** que usaremos y el **GIF**, y en donde pone “**Format**” siempre debe poner “**PNG**”:



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):

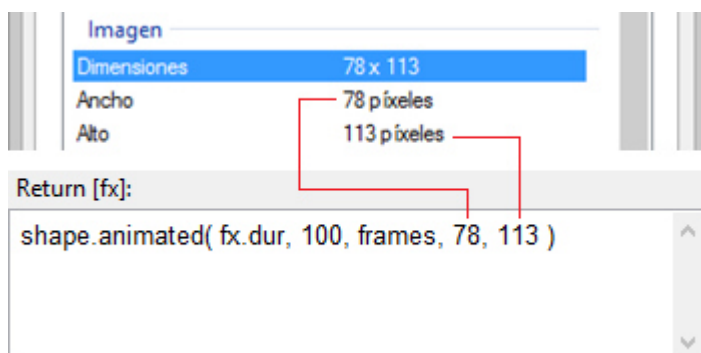


Hasta este punto, podemos decir que hemos hecho todos los pasos preliminares para usar la función y generar un efecto de animación. Los pasos siguientes ya los debemos realizar desde el **Kara Effector** directamente, y uno de ellos es crear una tabla en la celda de texto “**Variables**” que contenga, entre comillas, el nombre de cada una de las imágenes y su extensión (**.png**):

```
Variables:
frames = { "Frame0.png", "Frame1.png", "Frame2.png",
"Frame3.png", "Frame4.png", "Frame5.png",
"Frame6.png", "Frame7.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**.

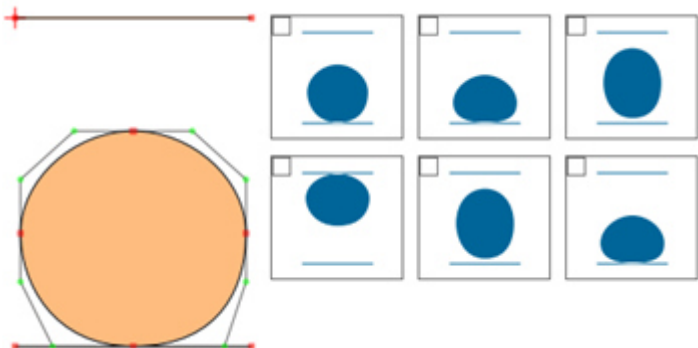
El resto de las configuraciones dependerá del efecto que queremos hacer, configuraciones como el **Template Type**, las posiciones y los tiempos. Para este ejemplo usé un **Template Type: Line**, con posición izquierda de la línea y con los tiempos por default de la misma:



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** las 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 | 44 0 m 0 61 | 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 | 44 0 m 0 61 | 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 | 44 0 m 0 61 | 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 | 44 0 m 0 61 | 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 | 44 0 m 0 61 | 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 | 44 0 m 0 61 | 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:

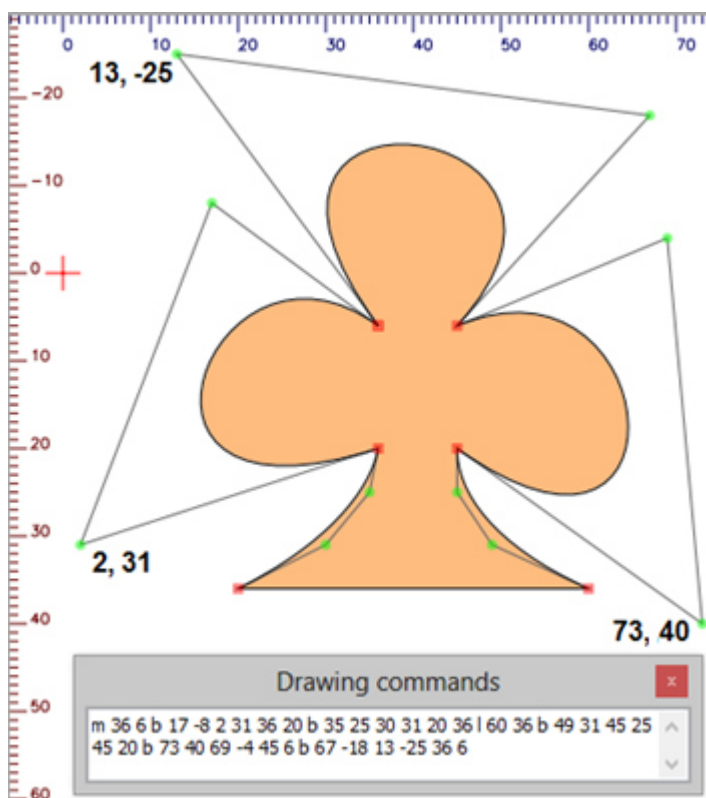


La ventaja de **shape.animation2** es poder hacer modificar fácilmente el tamaño de la misma ya que está hecha con Shapes, y éstas se modifican con los tags **\fscx** y **\fscy**.

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:

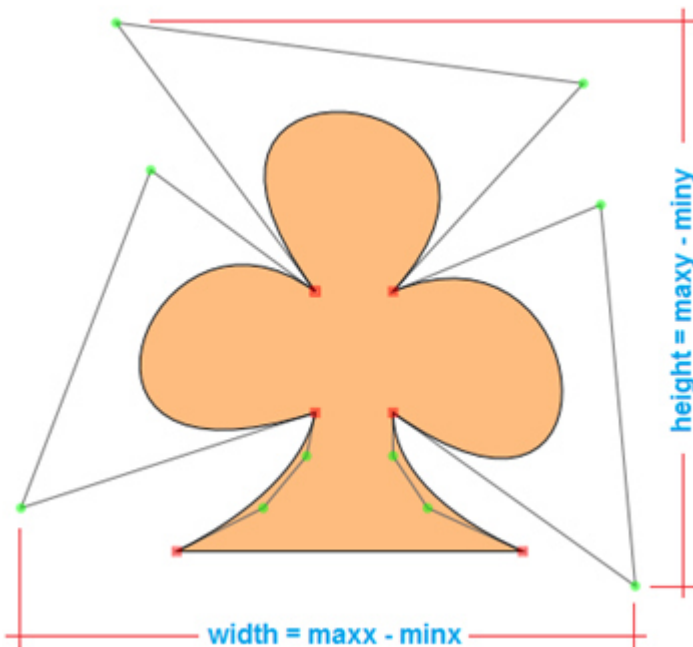


Usaremos la anterior **shape** para los siguientes ejemplos, y como siempre (aunque no es obligatorio), la declararé a modo de variable en la celda de texto “**Variables**” para que sea un poco más simple su uso:

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:

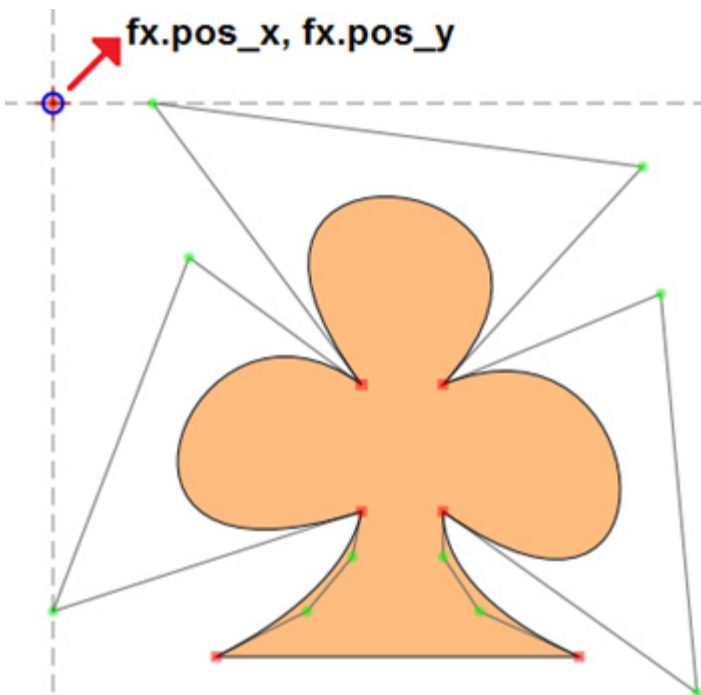
Add Tags:
Add Tags Language:
Lua

```

shape.config( mi_shape, "move" )

```

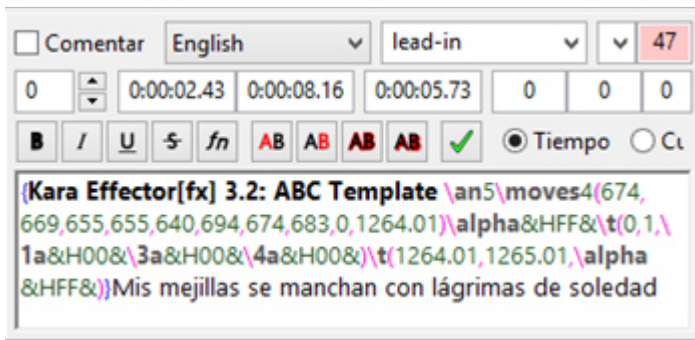
Lo que internamente hace la función es desplazar a la shape a un origen relativo, que ya no es el punto (0, 0) sino el centro en el vídeo del objeto karaoke:



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 $6 \times 3 = 18$, 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”)**

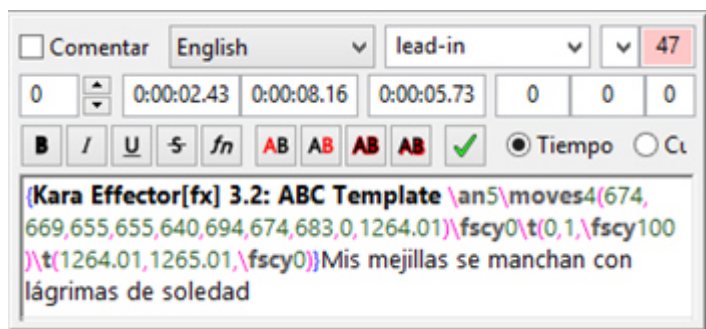
Este último ejemplo haría que el objeto karaoke se mueva siguiendo como trayectoria al perímetro de **shape.circle**, que tiene 100 px de diámetro, que es la medida de esta **shape predeterminada** del **Kara Effector**. Si quisiéramos que un objeto karaoke se moviera siguiendo la trayectoria de un círculo 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.

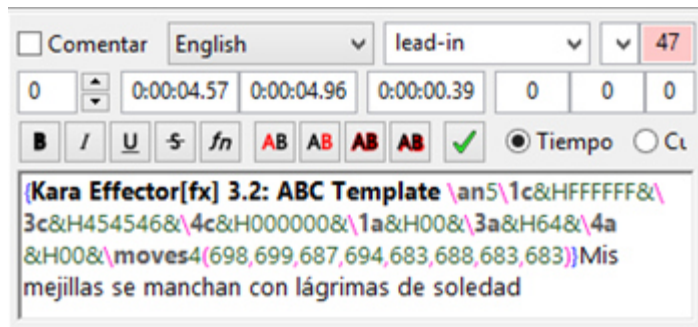
- **shape.config(mi_shape, “move3”)**: este modo es similar a los modos “**move**” y “**move2**“, genera el mismo efecto de movimiento, pero sin usar los tags de transparencias ni tampoco el tag **\fscy** en sus transformaciones. El modo “**move3**” genera líneas independientes respecto al tiempo, o 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

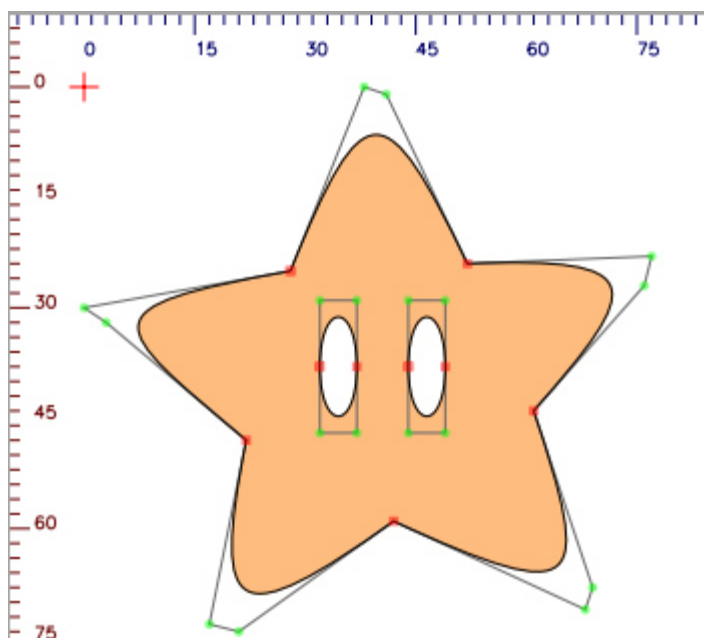
El modo “**move3**” estará disponible a partir de la **Versión**

3.2.7 del Kara Effector, para versiones anteriores solo es posible usar los modos “**move**” y “**move2**”.

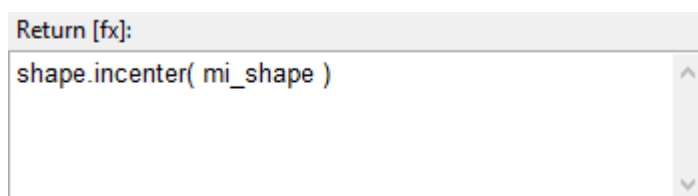
Al ver en detalle una de las líneas de fx generadas en este modo, notamos que el único tag que retorna es el de movimiento, **\move** para las restas de la **shape** y **\moves4** para las curvas **Bezier**, como en este ejemplo:



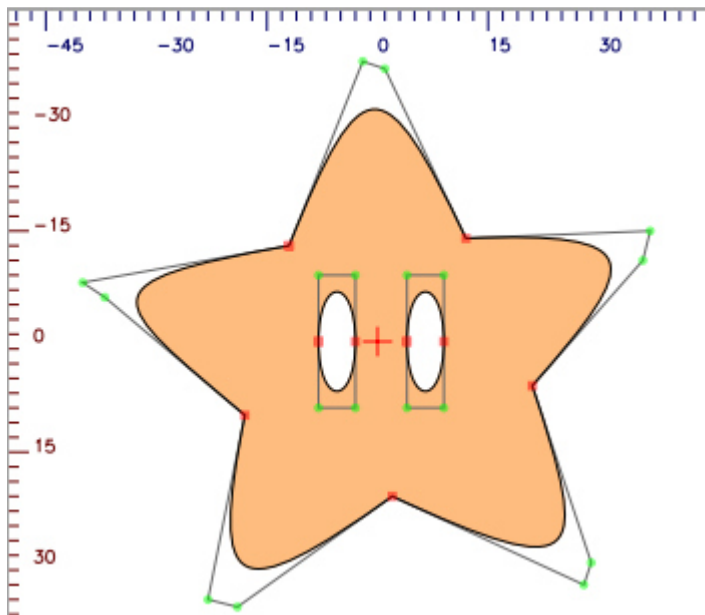
shape.incenter(Shape): esta función desplaza a la **shape** ingresada en el centro de las coordenadas del **AssDraw3**, con referencia al punto P = (0, 0). Ejemplo:



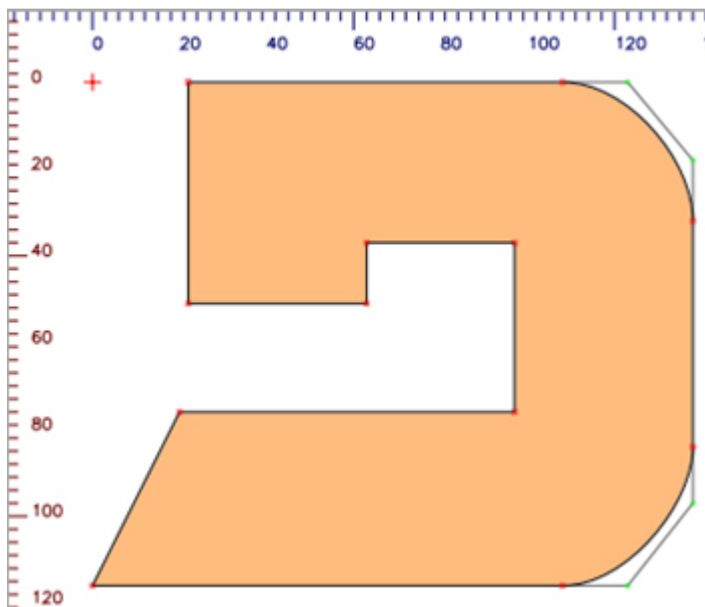
La anterior **shape** está ubicada en el **Cuadrante IV** del **AssDraw3**. Usamos la función con esta **shape**, por ejemplo en la celda de texto **Return [fx]**:



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):

