

SVG Cookbook For 4D

Presented by: **Keisuke Miyako**



INTRODUCTION

A cookbook is a rich source of creative problem solving as well a great learning device. Going through each recipe allows you to get familiar with different methods and techniques useful to accomplish specific tasks. In time, you begin to appreciate the variety of tools and ingredients at your disposal.

The goal of this session is to provide you with practical coding recipes you can use in your projects right away. Each informative recipe includes sample code and detailed discussion of why and how the solution works.

1. BASIC SYNTAX AND SEMANTICS

This section is designed to get you up and running with SVG features in 4D v15.

1.1 Creating an SVG

Problem

You need to dynamically create an SVG image based on some arbitrary data.

Solution

Use [PROCESS 4D TAGS](#) to fill dynamic elements of a template document.

First, **parameterise** the variables of a static SVG document:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!--4deval $x:=OB Get:C1224($2;"x";Is_longint:K8:6)-->
<!--4deval $y:=OB Get:C1224($2;"y";Is_longint:K8:6)-->
<!--4deval $fontFamily:=OB Get:C1224($2;"font-family";Is_text:K8:3)-->
<!--4deval $fontSize:=OB Get:C1224($2;"font-size";Is_text:K8:3)-->
<!--4deval $fill:=OB Get:C1224($2;"fill";Is_text:K8:3)-->

<svg
  xmlns="http://www.w3.org/2000/svg"
  version="1.1">
```

```

<text
  x="<!--4dtext $x-->"
  y="<!--4dtext $y-->"
  font-family="<!--4dhtml $fontFamily-->"
  font-size="<!--4dtext $fontSize-->"
  fill="<!--4dtext $fill-->"><!--4dtext $1--></text>

</svg>

```

Next, **process** the template and convert it to SVG:

```

C_OBJECT($params)

OB SET($params;\
  "x";20;\
  "y";20;\
  "font-family";"sans-serif";\
  "font-size";"20";\
  "fill";"red")

$path:=Get 4D folder(Current resources folder)+"SVG"+Folder separator+"text.4dsvg"
DOCUMENT TO BLOB($path;"utf-8";$template)
PROCESS 4D TAGS($template;$template;"Hello World!";$params)
BLOB TO PICTURE($template;$picture;".svg")

```

Discussion

SVG is based on XML, which is essentially a **declarative** language. For that reason, unless the structure of the expected data-driven image document is totally unpredictable, it makes more sense to work with a pre-defined template document than to build the output **procedurally** from scratch using [SVG Component](#) commands or [DOM/XML commands](#).

The use of a template document comes with several advantages:

- You can start from an existing static SVG image in the public domain.
- You can freely format the SVG and check for its structural integrity with any text editor.
- You can easily extend the code by adding more parameters.
- You can decide whether or not to create a DOM tree. (more on trees later)

Note: To prevent 4D from parsing the file and throwing errors about nested tags, it is better not to use the file extension ".svg" for SVG templates stored inside the Resources folder.

For comparison, here is an equivalent piece of code that uses SVG Component commands:

```
$x:=20
$y:=20
$fontFamily:="sans-serif"
$fontSize:=20
$fill:="red"

$svg:=SVG_New
$text:=SVG_New_text ($svg;"Hello World!";\
    $x;\
    $y;\
    $fontFamily;\
    $fontSize;\
    Plain;\
    Align left;\
    $fill)

$picture:=SVG_Export_to_picture ($svg)
SVG_CLEAR ($svg)

SET PICTURE TO PASTEBOARD ($picture)
```

The problem here is that the type and order of parameters for the command [SVG New text](#) is fixed. For that reason, you are required to pass optional values (style and alignment in the above example) that you could have otherwise skipped. 4D numeric constants such as "Plain", "Align left" are mapped to their SVG counterparts ("normal", "start"), which could lead to confusion. The font size parameter is numeric, which does not allow for an optional [unit identifier](#) (em, ex, px, pt, pc, cm, mm, in, or percentage).

It is also worth baring in mind that the component is written in a highly defensive manner (as compiled components should be), which makes it somewhat slower than calling DOM /XML commands directly.

For comparison, here is an equivalent piece of code that uses DOM/XML commands directly:

```
$x:=20
$y:=20
$fontFamily:="sans-serif"
$fontSize:=20
$fill:="red"
```

```

$svg:=DOM Create XML Ref("svg";"http://www.w3.org/2000/svg")
$text:=DOM Create XML element($svg;"text";\
"x";$x;\
"y";$y;\
"font-family";$fontFamily;\
"font-size";$fontSize;\
"fill";$fill)
DOM SET XML ELEMENT VALUE($text;"Hello World!")

SVG EXPORT TO PICTURE($svg;$picture;Own XML data source)
SET PICTURE TO PASTEBOARD($picture)

```

The code is similar to the previous example, but you have more control over which attributes to set (or skip) and in which format (string, number or boolean). You also have the ability to set the "id" of the element together with all the other attributes in one call (with the component you need to call [SVG SET ID](#), which can be slow with large documents). Still, the code is procedural, which makes it harder to predict and correct the final structure as the document gets more sophisticated.

Note: It is essential that you assign a unique "id" attribute to any SVG element you wish to reference later.

Whether you use DOM commands directly or the SVG component, procedurally building the SVG means you create a **DOM tree** as well as a **rendering tree** (more discussion on the two trees later), which you may or may not need. To simply create and display the SVG, you just need the rendering tree, which doesn't require parsing the source text. You just have to call [BLOB TO PICTURE](#) with the option ".svg".

1.2 Updating an SVG

Problem

You need to change parts of an SVG image programmatically.

Solution

Use [SVG SET ATTRIBUTE](#) to change the appearance of an SVG element.

Discussion

Sometimes it is absolutely necessary to recreate the entire SVG, which requires a considerable amount of computing. As a general rule, it is good practice to take some time to explore whether a less drastic measure is applicable before you go down that road. [SVG SET ATTRIBUTE](#) should always be first on your list. This command allows you to change the value of one or more SVG attributes, which sounds pretty straightforward, but its actual behaviour depends on several factors, such as how the SVG was created, whether it is already rendered on screen, whether you specify a variable or an object, or whether you pass the optional asterisk. If you haven't already, take a minute to read documentation for [SVG SET ATTRIBUTE](#).

The documentation repeatedly refers to an "internal DOM tree" as well as a "rendering tree". It also implies that how the SVG was created somehow has an impact on the command. A picture variable/field does not have to be bound to a form object, but the documentation says at the end that the command may fail if used "outside of a form execution context". Confusing? I agree.

You could think of an SVG image as having 4 distinct layers:

level	layer	description
1	bitmap	the actual image rendered on screen
2	rendering tree	the structured data used to create the bitmap image
3	DOM tree	structured representation of the parsed source code
4	XML source	text representation of the source code in memory

An SVG displayed on a form will always have the 1st layer. Its size is proportional to the canvas, that is, the dimensions of the object used to draw the image. By contrast, it does not depend on how complicated the SVG document is designed.

An SVG displayed on a form will also have the 2nd layer. Anytime the bitmap needs to be updated, for example as a result of scaling or resizing, this layer is consulted to create a newly rendered version of the bitmap layer. The engine can access any node of this in-memory tree and instantly change its properties. If all it had was the XML source code, it would have had to locate the relevant portion of text and rewrite its content, which is not an easy task given that SVG elements inherit some properties from its ancestors.

These two layers are created the moment an SVG is **rendered**, that is, displayed on screen, printed, copied to the pasteboard, converted to another image format, etc.

When you call [SVG SET ATTRIBUTE](#) **without the optional second asterisk**, you instruct the command to work on the **rendering tree layer**. The command won't work if the image has not been rendered yet. This includes the "On Load" phase. Here is an excerpt from the documentation that explains this concept:

*"The SVG SET ATTRIBUTE command is used to modify the value of an existing attribute in the SVG **rendering tree**...If you pass the optional * parameter, you indicate that the pictureObject parameter is an object name (string)...In this case, the command applies to the parameters of the **rendered image attached to the object** (note that the parameters and therefore the **rendered image of the object** are only created if the SVG SET ATTRIBUTE command is called at least once)...By default, modifications made by this command apply only to the **rendered images**."*

Even if the SVG has never been rendered, that is, if it only exists as a picture field or variable in memory, it may still have the 3rd layer, the internal DOM tree. A picture created with [SVG EXPORT TO PICTURE](#) has an internal DOM tree, unless you pass the "Get XML data source" option, in which case the DOM tree will remain "detached" from the picture.

Note: The documentation is misleading in that it mentions "DOM EXPORT TO VAR" as the definitive factor, and states that the DOM tree is not created if the SVG "was created from a file". A more accurate description would be that you need to call a DOM "parse" command, in which case the source can be a text, BLOB or indeed a file. What will not work is the use of [READ PICTURE FILE](#) to load the file directly. Likewise, [BLOB TO PICTURE](#) will not create a DOM tree either, even if the source is not a file. Finally, the appropriate "export" command is [SVG EXPORT TO PICTURE](#), not DOM EXPORT TO VAR.

WRONG: *"To be able to transfer modifications, the SVG variable must have been created from a DOM document (with **DOM EXPORT TO VAR**). If the SVG variable was **created from a file**, when you pass the second * parameter, the command does nothing."*

When you call [SVG SET ATTRIBUTE](#) **with the optional second asterisk**, you instruct the command to work on the **DOM tree layer**. Altering the 3rd layer also affects the 4th layer, but the change at that level is only visible if you actually retrieve the source code or assign the picture to another field or variable.

Here is an excerpt from the documentation that explains this concept:

*"You can transfer these changes into the **internal DOM tree** of the image when the pictureObject parameter references a variable: you just need to pass the second * as the last parameter. This lets you keep the modifications made on the fly...Transferring modifications into the **internal DOM tree** is not possible when the pictureObject parameter references an object."*

Note: There was a bug (ACI0094002) where the 4th layer was not correctly updated when [SVG SET ATTRIBUTE](#) was called with the optional second asterisk. The problem has been corrected in v15.1 HF1.

The DOM tree layer is only accessible if you apply the command to a field or variable. If you pass an object name, the second asterisk is simply ignored.

Note: The documentation wrongly tells that an error is generated if you pass an object name.

WRONG: *"If the SVG variable was created from a file, when you pass the second * parameter, the command does nothing and **an error is generated** because, in this case, the data source does not contain a modifiable DOM document."*

A valid SVG will also have the 4th layer. Its size is equivalent to the document/text size you supplied to [DOM Parse XML source](#), [DOM Parse XML variable](#), [READ PICTURE FILE](#), or [BLOB TO PICTURE](#). It is not related to the canvas size. You can retrieve the content of this layer by calling [PICTURE TO BLOB](#) on the picture, or [DOM EXPORT TO VAR](#), if the DOM tree is "detached" from the picture.

In summary, there are four factors that affect the behaviour of the command [SVG SET ATTRIBUTE](#):

- Whether the source was **parsed** or not.
- Whether the image has been **rendered** or not.
- Whether the **target** is a picture variable/field or a picture object.
- Whether the second **asterisk** option was passed or not.





The result will be one of the following:

- **Both** the rendering and DOM trees are updated (OK with *)
- **Only** the rendering tree is updated (OK without *, or the * being ignored)
- **Nothing** happens (NG)

Looking at it from the opposite end, you could ask yourself two questions:

1. Do I need to process the SVG in memory? (this includes the On Load phase of a form)
2. Do I want my updates to the SVG be persistent?

If your answer you either question is "yes", it means you are going to need a **DOM tree**, which means you will have to **parse** the source text or file. If your answers to both questions are "no", it means you just need a **rendering tree**, you can simply **read** the source text or file.

		variable		object	
		with *		with *	
READ SOURCE	on screen		OK		OK
	in memory		NG	NG	NG
PARSE SOURCE	on screen	OK	OK		OK
	in memory	OK	NG	NG	NG

Reading the source means you use [READ PICTURE FILE](#) or [BLOB TO PICTURE](#) to create the SVG. You can update the rendering tree with [SVG SET ATTRIBUTE](#), but the picture will be missing a DOM tree to store the changes. If you call [WRITE PICTURE FILE](#) or [PICTURE TO BLOB](#), you simply get the **original** source.

Parsing the source means you use [SVG EXPORT TO PICTURE](#) with "Copy XML data source" or "Own XML data source" to create the SVG. The picture will have an internal DOM tree which you can update using [SVG SET ATTRIBUTE](#) with the asterisk option. If you call [PICTURE TO BLOB](#) you get the **current** source.

1.3 Modifying an SVG

Problem

You need to modify an SVG image programmatically.

Solution

Modify the DOM tree directly and re-export the picture with [SVG EXPORT TO PICTURE](#).

Discussion

Some kind of modification are simply beyond the scope of what is possible with [SVG SET ATTRIBUTE](#). In particular, the command is not capable of doing the following:

- Add, remove, or move an element in the DOM tree
- Remove an attribute
- Modify locked attributes; more specifically, the [id](#) or [class](#) attribute
- Modify attributes of defined "xlink:href" elements; filters, styles, markers, etc.
- Modify the width or the height of the SVG document

Note: The documentation states that *"the command is used to modify the value of an **existing** attribute"*, but [SVG SET ATTRIBUTE](#) can also be used to **add** new attributes to an element.

If the modification requires any of the operations listed above, you need to update the DOM tree and re-export the picture. To do that, you need a **DOM reference**.

When you call [SVG EXPORT TO PICTURE](#) with "Copy XML data source" or "Own XML data source", the picture will have an internal, or **embedded** DOM tree. You don't have direct access to that tree. You need to get a fresh copy of the XML source ([PICTURE TO BLOB](#)) and then parse it to get a new DOM reference.

If you frequently find yourself updating the SVG from its source, perhaps you could consider using an external, or **detached DOM tree** instead. You can do so by passing "Get XML data source" when you call [SVG EXPORT TO PICTURE](#). The resulting picture will have **no size** and keep a reference to the original DOM as its backend DOM tree. You can work with the original DOM and call [SVG EXPORT TO PICTURE](#) with the "Get" option whenever you need to update the rendering tree. Finally, when you need to store the picture in a field or a file, call [SVG EXPORT TO PICTURE](#) again but with the "Copy" option to create a picture with substance. The "Own" option would also create a picture with an internal DOM tree, but the difference is that unlike "Copy", you no longer have access to the original DOM reference. You would normally use the "Own" option when you are not interested in keeping a reference to the DOM tree.

	DOM tree	original DOM reference
get	detached from picture	valid
copy	embedded in picture	valid but unrelated to the picture
own	embedded in picture	invalid

Strategies for modifying an SVG picture using the original DOM reference.

	start	update	save	end
SVG EXPORT	get	get	copy	-
	copy	copy	-	close

If you need to save the SVG frequently, you might want to consider the "copy" option. If you only need to save the SVG sporadically, using the "get" option is more efficient.

Another way to look at the problem is to explore if the modification could be substituted by updating an attribute with [SVG SET ATTRIBUTE](#) which doesn't require a DOM reference. For example:

- instead of changing the width and height, change the viewBox
- instead of removing an attribute, simply revert to a default value (auto, inherit, etc)
- instead of changing the class, update the style attributes directly

1.4 Scaling an SVG

Problem

You need to resize an SVG.

Solution

Use [TRANSFORM PICTURE](#) with the "scale" option.

Discussion

The command operates on the rendering tree, which means that the original picture is unaffected. You can restore the original image by passing the "reset" option.

Example of scaling the image to 50% of its original size:

```
TRANSFORM PICTURE ($Image->Scale;0.5;0.5)
```

Example of scaling the image to 200% of its original size:

```
TRANSFORM PICTURE ($Image->Scale;2;2)
```

You can also scale the picture at the rendering level using [picture operators](#).

Example of scaling the image to 50% of its original size using a picture operator:

```
$Image->:=$Image->*0.5
```

Example of scaling the image to 200% of its original size using a picture operator:

```
$Image->:=$Image->*2
```

Another way to resize an SVG is to create a top-level group ([<g>](#)) element and apply a scale factor using the [transform](#) attribute.

Example code of a ruler object that changes the scale factor:

```
$sx:=Self->/10
$sy:=Self->/10

$scale:=String($sx;"&xml")+",""+String($sy;"&xml")
$transform:="scale ("+$scale+") "

SVG SET ATTRIBUTE (*;"Image";"container";"transform";$transform)
```

Note: When you pass numeric values to the [String](#) function, be careful it doesn't inject invalid values to the XML, which will raise an SVG parser error at runtime. [ON ERR CALL](#) will not silence such errors.

For example:

```
$value:=String(Pi;"&xml")
```

The function will use the current locale's **decimal separator** by default, which might be the comma character. To make sure the period is used regardless of regional settings, specify the "&xml" format.

Another example:

```
$INF:=1/0
$value:=String(Num(String($INF)));"&xml")
```

The INF (infinity) constant is not a valid value in SVG. If division by zero is a logical possibility, make sure you filter out any NaNs (not-a-number).

IN summary, scaling by any of these methods only affect the rendering tree:

- [TRANSFORM PICTURE](#) with the "scale" option
- [Picture operator](#) for scaling
- [SVG SET ATTRIBUTE](#) without the asterisk on the [transform](#) attribute

You can always revert to the original picture:

```
TRANSFORM PICTURE ($Image->;Reset)
```

In other words, the "reset" option will rebuilt the rendering tree based on the original XML source.

Note: Apparently there is a technical ceiling as to how large the image can be. Here is an extreme example of a large view port. Notice that an [exponent](#) can be used to describe a large numeric value in SVG.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg">
  <rect height="4.2e+9" width="4.2e+9" x="0" y="0" />
</svg>
```

Yet another way to scale an image is to alter the [viewBox](#) attribute.

Example of scaling the image to 50% of its original size by altering the viewBox:

```
SVG GET ATTRIBUTE (*;"Image";"svg";"width";$width)
SVG GET ATTRIBUTE (*;"Image";"svg";"height";$height)

SVG SET ATTRIBUTE (*;"Image";"svg";"viewBox";\
"0 0 "+String($width*2;"&xml")+" "+String($height*2;"&xml"))
```

Example of scaling the image to 200% of its original size by altering the viewBox:

```
SVG GET ATTRIBUTE (*;"Image";"svg";"width";$width)
SVG GET ATTRIBUTE (*;"Image";"svg";"height";$height)

SVG SET ATTRIBUTE (*;"Image";"svg";"viewBox";\
"0 0 "+String($width*0.5;"&xml")+" "+String($height*0.5;"&xml"))
```

Notice the scale factor is the **inverse** of the zoom factor (i.e. 0.5 for 200%, 2 for 50%).

1.5 Cropping an SVG

Problem

You need to crop an SVG.

Solution

Translate the image using the [transform](#) attribute.

Discussion

Unlike scaling, cropping an SVG with the [TRANSFORM PICTURE](#) command or a [picture operator](#) will **permanently** transform the picture. In fact, the image will be **rasterised** as a result, which means that the image will no longer have a DOM tree or a rendering tree, it will not even be an SVG anymore.

In SVG, you can shift a group of elements by translating their coordinates. The offset is passed as a string in the [transform](#) attribute, which means you have to check for its existence and parse it to retrieve the current value. To make each value more accessible by code, you could store them in a private attribute, one you prefix with a custom **namespace**.

Example of shifting the SVG to the left:

```
C_REAL($tx,$ty)
SVG GET ATTRIBUTE (*;"Image";"container";"translate:x";$tx)
SVG GET ATTRIBUTE (*;"Image";"container";"translate:y";$ty)

$tx:=$tx-10

$translate:=String($tx;"&xml")+",""+String($ty;"&xml")
$transform="translate("+ $translate+)"

SVG SET ATTRIBUTE (*;"Image";"container";"transform";$transform;\
"translate:x";$tx;"translate:y";$ty)
```

The above code assumes that a custom namespace called "translate" is defined in the SVG.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:translate="http://www.4d.com/namespaces/translate">
  ...
</svg>
```

Note: [Namespaces](#) provide a way to use non-standard or duplicate attribute names inside an SVG.

You can also work with the viewBox to effectively crop an SVG.

For instance, you could procedurally shift the view box of an SVG to print it across multiple pages. As mentioned earlier, the [TRANSFORM PICTURE](#) command will rasterise the image, which means that the quality after cropping would not be good for printing (unless you have sufficiently scaled it beforehand).

1.6 Converting an SVG

Problem

You need to convert an SVG to an alternative image format.

Solution

Use [CONVERT PICTURE](#).

Discussion

SVG images are generally concise and their quality do not deteriorate by scaling, so normally you would want to keep them in their original format. However, there are some cases where it makes sense to convert an SVG to an alternative image format, even if it means you permanently rasterise the image.

You can convert SVG to most of the formats listed by [PICTURE CODEC LIST](#):

	size in kilobytes	equal pictures	alpha channel
svg	3727.60	-	-
4pct	3727.60	YES	YES
jpg	48.48	NO	NO
png	41.85	NO	YES
bmp	156.30	NO	NO
gif	18.20	NO	NO
tif	156.44	NO	YES
pict	64.19	NO	NO
pdf	308.30	NO	YES
jp2	33.06	NO	NO
icns	0	-	-
psd	136.33	NO	YES
ico	0	-	-
tga	70.30	NO	YES
sgi	0	-	-
exr	70.20	NO	YES

Notice how all formats except the internal ".4pct" fails the "equals" text. Conversion to ".pdf" seems to preserve the vectorial properties of the original image, but it is a feature only available on Mac. Normally, ".png" should be the best option in terms of quality, size, and portability.

Sometimes the document structure could be so complicated that the size of XML source code exceeds that of the actual bitmap. This could happen as a result of an extensive use of paths or embedded images. If you are only using SVG as a way to create data-driven dynamic images programmatically, and you don't need to change or update it once you have the rendered result, you might want to rasterise the SVG to save the engine from re-rendering it each time the user scrolls or resizes the object with no scaling. It is much easier for the application to shift and crop an existing bitmap, no matter how large and complex the image is, than to re-create a new bitmap from its rendering tree.

You might also be forced to convert an image to another format if you have to transfer it to an application that does not support SVG, or a platform where the implementation could be significantly different.

In summary, you might want to rasterise an SVG for the following reasons:

- The image needs to be browsed by a client application that does not support SVG
- The image must look exactly the same (not just semantically) everywhere
- The image is large and complex, but non-interactive and sparsely updated

2. TEXT

2.1 Inserting text in SVG

Problem

You need to display some text at a specific location in SVG.

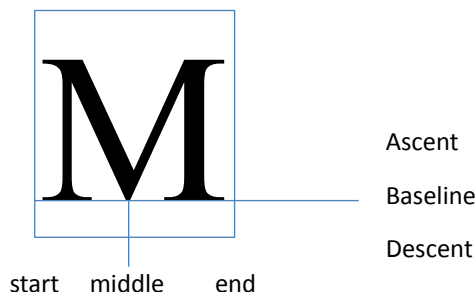
Solution

Use `<textArea>` if the SVG only needs to be displayed in 4D, otherwise `<text>` if it also needs to be displayed outside of 4D (in a web browser, for example).

Discussion

The `<text>` element defines a graphics element consisting of text in SVG. You can specify the font, style, etc., you can even use multi-styled text using the `<tspan>` element. As far as positioning is concerned, you only have control over a single point, specified by the `x` and `y` coordinates, against which the text baseline is aligned.

To give an example:



If the above text is positioned at "0, 0", you will see nothing on the canvas, as there are no stroke under the baseline. As such, it can be quite a challenge to vertically position a [<text>](#) element in SVG.

There are no automatic word-wrap features in [<text>](#), so you have to break lines programmatically and create as many elements as needed, vertically shifting them by a computed offset value.

In essence, [<text>](#) is useful for "labels", or short pieces of static text that do not require word-wrapping, whose size and font are fixed so that you do not have to strictly calculate the [x](#) and [y](#) positions.

The [<textArea>](#) element, on the other hand, is much easier to position in that you provide not only the [x](#) and [y](#) coordinates but also the [width](#) and [height](#). The positioning is not relative to the baseline, but rather, the coordinate system inside the object itself. There is automatic word-wrapping support, and you can align the text both horizontally and vertically.

However, there is a major downside to the [<textArea>](#) SVG element, which is that most web browsers do not support it. If you plan to export the SVG for browsing outside of 4D, you might have to convert the picture to another format, or use [<text>](#) instead.

2.2 Updating text in SVG

Problem

You need to update the text displayed in SVG.

Solution

Set the pseudo-attribute "4D-text" with [SVG SET ATTRIBUTE](#).

Discussion

The content of an SVG [<text>](#) or [<textArea>](#) is its element value, so you could use DOM commands to update the text if you have a valid DOM reference (sample code below). However, it is much easier to use the pseudo-attribute "4D-text", which incidentally will also take care of line breaks.

```
SVG SET ATTRIBUTE (*;"Image";"text";"4D-text";Get edited text)
```

For comparison, here is an equivalent piece of code that uses DOM/XML commands directly:

```
C_TEXT($1;$2)

$textArea:=$1
$textValue:=$2

C_LONGINT($i;$j;$count)
$count:=DOM Count XML elements($textArea;"tbreak")

ARRAY TEXT($tbreaks;0)
$tbreak:=DOM Find XML element($textArea;"textArea/tbreak";$tbreaks)
For ($i;1;Size of array($tbreaks))
    DOM REMOVE XML ELEMENT($tbreaks{$i})
End for
```

```

$i:=1

ARRAY LONGINT($len;0)
ARRAY LONGINT($pos;0)

While (Match regex("(.)";$textValue;$i;$pos;$len))
    $line:=Substring($textValue;$pos{1};$len{1})
    If ($i=1)
        DOM SET XML ELEMENT VALUE($textArea;$line)
    Else
        $breaks:=Substring($textValue;$i;$pos{1}-$i)
        $count:=Length(Replace string($breaks;"\r\n";"\n";*))
        For ($j;1;$count)
            $tbreak:=DOM Create XML element($textArea;"tbreak")
        End for
        $append:=DOM Append XML child node($textArea;XML DATA;$line)
    End if
    $i:=$pos{1}+$len{1}
End while

```

Note that you can't break a line with a `
` element in an SVG [<textArea>](#), as you would typically do in HTML. Instead, you need to insert a [<tbreak />](#), which is also part of the SVG Tiny 1.2 specification.

IMPORTANT: The following elements and attributes are not supported in 4D:

- `<tref>`, `<textPath>`
- [<text>](#): "rotate", "textLength", "lengthAdjust", "font-size-adjust", "font-stretch", "font-variant", "word-spacing", "glyph-orientation-horizontal", "glyph-orientation-vertical", "dominant-baseline", "direction", "unicode-bidi"
- [<textArea>](#): "editable", "focusable"

2.3 Horizontal alignment

Problem

You need to align the text horizontally.

Solution

Use the [text-align](#) attribute for [<textArea>](#), or the [text-anchor](#) attribute for [<text>](#).

Discussion

A different attribute is used to align the text horizontally, depending on the element type.

Note that the [text-anchor](#) attribute expects values such as "start", "**middle**" or "end". whereas the [text-align](#) attribute expects "start", "**center**" or "end". In any case, the horizontal alignment is **never** described using expressions such as "left" or "right", as text may start from the right or left, depending on the script. For [<textArea>](#) elements, you can also use "justify".

2.4 Vertical alignment

Problem

You need to align the text vertically.

Solution

Use the [display-align](#) attribute for [<textArea>](#), or the [y](#) or [dy](#) attribute for [<text>](#).

Discussion

Note that the [display-align](#) attribute expects values such as "**before**", "center" or "**after**". whereas the [text-align](#) attribute expects "**start**", "center" or "**end**". This is because the alignment is described relative to the text's baseline. In any case, the vertical alignment is **never** described using expressions such as "top" or "bottom".

Note: The SVG specification defines the [dominant-baseline](#) and [alignment-baseline](#) attributes, which tells the engine how to vertically align the baseline against the "[y](#)" coordinate. These attributes are useful for vertically positioning a [<text>](#) or [<tspan>](#) element, but unfortunately they are **not supported** in 4D v15.

Here is a summary of text alignment options:

	vertical		horizontal	
	text	textArea	text	textArea
		display-align	text-anchor	text-align
default	N/A	before	start	start
medium		center	middle	center
opposite		after	end	end

2.5 Text direction

Problem

You need to display right-to-left or top-to-bottom text in SVG.

Solution

Use the [writing-mode](#) attribute.

Discussion

The SVG engine supports "lr-tb" (English), "rl-tb" (Arabic, Hebrew) and "tb-rl" (Chinese, Japanese) scripts.

Example of SVG vertical text:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg">
  <g
    writing-mode="tb"
    text-align="center"
    display-align="before"
    text-decoration="none"
    font-family="sans-serif"
    font-size="20"
    font-style="normal"
    font-weight="normal">
    <textArea
      x="0"
      y="0"
      height="400"
      width="40"> 『セールス・エンジニア』 </textArea>
    </g>
  </svg>
```



The result is different to that of [SVG New vertical text](#), which splits the text and places a letter on each line. Fonts that support vertical writing mode typically have different glyph settings for different modes.

3. SHAPES

3.1 Creating shapes

Problem

You need to put a shape object at a specific location in SVG.

Solution

Use the [SVG Component](#), [DOM/XML commands](#) or [PROCESS 4D TAGS](#) to create an SVG document that contain basic shape elements.

Discussion

The SVG engine supports all of the basic shape elements defined in the SVG 1.1 specification, namely [<rect>](#), [<circle>](#), [<ellipse>](#), [<line>](#), [<polyline>](#), [<polygon>](#) and [<path>](#).

Note: The [SVG Component](#) includes several "helper" APIs to create typical variants of certain basic shapes:

- [SVG New regular polygon](#)
- [SVG New ellipse bounded](#)
- [SVG New arc](#)

3.2 Drawing hair lines

Problem

You need to **draw a thin straight line on the screen** in SVG.

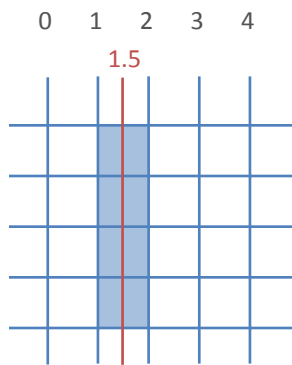
Solution

Offset the coordinates by 0.5 if the stroke width corresponds to an odd number of pixels on screen.

Discussion

The horizontal and vertical axis on an SVG coordinate system are logical lines that has no width. On a normal (i.e. not high-resolution) screen, the axis will be running through pixels, not between them. If you specify an integer value as the [x](#) or [y](#) position of a straight single-pixel line, the rendering engine will have to use half a pixel on each side of the line, which is accomplished by using a full pixel but reducing its paint opacity by 50% (anti-aliasing). The result is that the line displays blurred.

On the other hand, a straight single-pixel line shifted by 0.5 will use exactly one pixel.



Another way to ensure that straight line are rendered without anti-aliasing is to specify a rendering mode.

The following values are support for the [shape-rendering](#) attribute:

- `crispEdges` (turn off anti-aliasing)
- `optimizeSpeed` (turn off anti-aliasing)
- `geometricPrecision`
- `auto`

When anti-aliasing is turned off, the engine may also adjust the position and width of a straight line that is almost vertical or horizontal, in order to align its edges with device pixels.

Note: 4D does not differentiate between "`crispEdges`" and "`optimizeSpeed`". The same principle applies to anti-aliasing of [<text>](#) and [<textArea>](#) elements which can be disabled with the [text-rendering](#) attribute.

3.3 Moving shapes

Problem

You need to move around a shape in SVG.

Solution

Center the object coordinates at "0, 0" and use the [transform](#) attribute to move its position.

Discussion

A "translate" transformation will shift the actual position of an object along the current coordinate system. For example, an object located at "0, 0" and translated by "100, 100" will be positioned at "100, 100", if no other factors (scaling, skewing by viewBox, etc.) are involved.

It often makes sense to position **all** SVG objects at "0, 0" on their local coordinate system and translate.

Rotating: You can simply rotate an object around the "0, 0" centre point. Normally you would have to use trigonometric functions to determine the centre of rotation.

Moving: You can move an object by simply adding or subtracting the translate value. Normally you would have to re-compute all the coordinates of a shape and all the points of a [polyline](#) or [polygon](#) element.

4. IMAGES

4.1 Creating images

Problem

You need to put an image inside an SVG.

Solution

Use the [SVG Component](#), [DOM /XML commands](#) or [PROCESS 4D TAGS](#) to create an SVG document that contain [image](#) elements.

Discussion

An image can be a [file URI](#), a [data URI](#) or a local URI (see section 5.2).

Note: The engine doesn't support nested [svg](#) elements, but it is possible to include an SVG inside an SVG by using an [image](#) element.

The [image](#) element accepts a "file:" URI in its "[xlink:href](#)" attribute. The "xlink" namespace must be defined in the document to use a [file URI](#) or [data URI](#).

Example of using a [file URI](#):

```
<svg
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  id="svg">
  <image x="0" y="0" width="100" height="100" xlink:href="<!--4dhtml $url-->" />
</svg>
```

Example of converting a system file path to a [file URI](#):

```
$url:="file://" + Convert path system to POSIX($imageFilePath;*)
```

Note: You must pass the "file://" protocol specifier and use an absolute path if you create the SVG from text data. You can use relative paths and omit the protocol if the SVG is a file. Special characters should be escaped by passing the optional asterisk to [Convert path system to POSIX](#).

Alternatively, you can embed the image file by passing a base64 encoded [data URI](#). Though it will increase the document size significantly, it has the advantage of the SVG being self-contained.

Example of creating a [data URI](#):

```
C_BLOB($imageData)
PICTURE TO BLOB($image;$imageData;".png")
C_TEXT($data)
BASE64 ENCODE($imageData;$data)
$data:="data:image/png;base64,"+$data
```

Note: You can omit the MIME type ("image/png" in the above example). The engine will deduct the actual image format from the first couple of bytes, which serves as a signature.

Decoding a large data URI can take a toll on the rendering engine and deteriorate performance. One way to circumvent such issues is to create and reference temporary files during edit and embed the data immediately before the document is saved. In addition, you could check the hash with [Generate digest](#) to avoid creating redundant copies of the same image reference.

Note: Regardless of how you passed the URI, the image will be stored in its native format in the rendering tree. If you use [SVG SET ATTRIBUTE](#) to update some parts of the SVG, the engine will simply refresh the invalidated logical portion which is a relatively fast operation. On the other hand, [SVG EXPORT TO PICTURE](#) will force the rendering engine to create a new rendering tree from the DOM, including any nested SVG images. For best performance, you should consider using raster formats for images that you need to scale or rotate at short intervals.

4.2 Scaling images

Problem

You need to scale an image to fit inside a rectangle.

Solution

Use the [preserveAspectRatio](#) attribute.

Discussion

In addition to the [width](#) and [height](#) attributes, you can set the [preserveAspectRatio](#) attribute to instruct how the image should fit inside its frame. For instance, the value "xMidYMid" corresponds to "scale proportionally and centred" in 4D picture display format terminology.

Reducing the size of a raster image proportionally on a high-definition display can increase its perceived quality. In other words, to take full advantage of the screen resolution, you would need a larger picture.

5. ADVANCED SYNTAX AND SEMANTICS

5.1 Reusing objects

Problem

You need to avoid duplicate definitions of the same object in SVG.

Solution

Reference an object defined earlier in the document structure by its [id](#).

Discussion

You can pass the [id](#) of an object to the "[xlink:href](#)" attribute of a [<use>](#) element. The "[xlink](#)" namespace must be defined beforehand in the document.

You can manage a stock of "sample" objects by placing them inside the [<defs>](#) element. Graphical elements defined in a [<defs>](#) element are not directly rendered. A [<use>](#) element can itself be a reference to another [<use>](#) element.

Example of using a local URI with a double layer of [<use>](#) elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  width="320" height="3840">
<g>
  <g id="loader">
    <line id="line0" x1="15" y1="160" x2="65" y2="160" stroke="#111"
      stroke-width="30"
      style="stroke-linecap:round" />
    <use xlink:href="#line0" transform="rotate(30,160,160)" opacity=".0833" />
    <use xlink:href="#line0" transform="rotate(60,160,160)" opacity=".166" />
    <use xlink:href="#line0" transform="rotate(90,160,160)" opacity=".25" />
    <use xlink:href="#line0" transform="rotate(120,160,160)" opacity=".3333" />
    <use xlink:href="#line0" transform="rotate(150,160,160)" opacity=".4166" />
    <use xlink:href="#line0" transform="rotate(180,160,160)" opacity=".5" />
    <use xlink:href="#line0" transform="rotate(210,160,160)" opacity=".5833" />
    <use xlink:href="#line0" transform="rotate(240,160,160)" opacity=".6666" />
    <use xlink:href="#line0" transform="rotate(270,160,160)" opacity=".75" />
    <use xlink:href="#line0" transform="rotate(300,160,160)" opacity=".8333" />
    <use xlink:href="#line0" transform="rotate(330,160,160)" opacity=".9166" />
  </g>
</g>
```

```

<use xlink:href="#loader" transform="translate(0,320) rotate(30,160,160)" />
<use xlink:href="#loader" transform="translate(0,640) rotate(60,160,160)" />
<use xlink:href="#loader" transform="translate(0,960) rotate(90,160,160)" />
<use xlink:href="#loader" transform="translate(0,1280) rotate(120,160,160)" />
<use xlink:href="#loader" transform="translate(0,1600) rotate(150,160,160)" />
<use xlink:href="#loader" transform="translate(0,1920) rotate(180,160,160)" />
<use xlink:href="#loader" transform="translate(0,2240) rotate(210,160,160)" />
<use xlink:href="#loader" transform="translate(0,2560) rotate(240,160,160)" />
<use xlink:href="#loader" transform="translate(0,2880) rotate(270,160,160)" />
<use xlink:href="#loader" transform="translate(0,3200) rotate(300,160,160)" />
<use xlink:href="#loader" transform="translate(0,3520) rotate(330,160,160)" />
</g>
</svg>

```

5.2 Gradients

Problem

You need to paint an object with linear or radial gradient colours.

Solution

Define a [<linearGradient>](#) or [<radialGradient>](#) element in [<defs>](#) and use a local URI in the [fill](#) attribute of a graphical element.

Discussion

The SVG rendering engines supports all types of [gradients](#), including multiple stop elements and all three types of wrapping modes; "pad", "reflect", and "repeat".

Note: The "xlink" namespace must be defined in the document to use a local URI.

5.3 Styles

Problem

You need to apply the same a set of style attributes in multiple instances.

Solution

Define a [<style>](#) element in [<defs>](#) and apply a [class](#) attribute to a graphical element.

Discussion

You can either include an external ".css" file, or define an inline stylesheet definition.

Example of referencing an external ".css" file:

```

<style media="screen" type="text/css">@import "screen.css";</style>
<style media="print" type="text/css">@import "print.css";</style>

```

Example of an inline stylesheet definition:

```
<style media="screen" type="text/css">
.thin{
    stroke:black;
    stroke-width:1px;
}
</style>
<style media="print" type="text/css">
.thin{
    stroke:black;
    stroke-width:0.5pt;
}
</style>
```

Note: For document integrity, you might want to enclose the CSS in a CDATA section.

The SVG rendering engines supports basic CSS2 syntax, including the following features:

- The "[:first-child](#)" pseudo-class
- The "[:lang\(\)](#)" pseudo-class
- The "[!important](#)" exception
- [Comment](#): /* ... */
- [Type](#) selector: rect {...}
- [ID](#) selector: #rect1 {...}
- [Class](#) selector: .className {...}
- [Child](#) selector: g > rect {...}
- [Descendant](#) selector: g rect {...}
- [Adjacent sibling](#) selector: rect + text {...}
- [Universal](#) selector

5.4 Clipping

Problem

You need to clip a region out of an SVG.

Solution

Define a [clipPath](#) and use it with a [clip-path](#) attribute.

Discussion

The rendering engine supports the following elements as a clip path: [<rect>](#), [<line>](#), [<polyline>](#), [<polygon>](#), [<circle>](#), [<ellipse>](#) and [<path>](#). However, only a single element can make a [<clipPath>](#). Moreover, the [clip-rule](#) attribute is not supported, which means, for instance, that you can clip the interior of a shape but you can't clip the exterior of a shape using a single clip path. Another minor inconvenience is that [transform](#) ("translate" in particular) is not supported in a [<clipPath>](#). You have to use absolute coordinates. Finally, you can't crop and rasterise a clipped SVG with [TRANSFORM PICTURE](#). You must alter the [viewBox](#) instead.

5.5 Filter effects

Problem

You need to apply a filter effect on a graphic object in SVG.

Solution

Define a [<filter>](#) element and use it with a [filter](#) attribute.

Discussion

The rendering engine supports six of the basic filter effects defined in the [SVG 1.1 specification](#).

effect	availability
feBlend	OK
feColorMatrix	OK
feComponentTransfer	
feComposite	OK
feConvolveMatrix	
feDiffuseLighting	
feDisplacementMap	
feFlood	
feGaussianBlur	OK
feImage	
feMerge	
feMorphology	
feOffset	OK
feSpecularLighting	
feTile	
feTurbulence	
feDropShadow	

The [SVG component](#) includes APIs to quickly define common filters:

- [SVG Filter Blend](#)
- [SVG Filter Blur](#)
- [SVG Filter Offset](#)

In addition, the following commands internally uses a [ColorMatrix](#) filter:

- [SET_HUE](#)
- [SET_SATURATION](#)
- [SET_BRIGHTNESS](#)

Here is an example of using both a filter effect and a clip path:

```
//base64 data URI from file
C_TEXT($data)
$path:=Get 4D folder(Current resources folder)+"SVG"+Folder separator+"logo.png"
READ PICTURE FILE($path;$image)
PICTURE TO BLOB($image;$imageData;".png")
PICTURE PROPERTIES($image;$width;$height)
BASE64 ENCODE($imageData;$data)

//SVG document root
$svg:=DOM Create XML Ref("svg";"http://www.w3.org/2000/svg";\
"xmlns:xlink";"http://www.w3.org/1999/xlink")
DOM SET XML ATTRIBUTE($svg;\
"width";$width;"height";$height)
$defs:=DOM Create XML element($svg;"defs";"id";"defs")

//filter
$stdDeviation:=8
$filter:=DOM Create XML element($defs;"filter";"id";"filter-blur")
$feGaussianBlur:=DOM Create XML element($filter;\
"feGaussianBlur";"stdDeviation";$stdDeviation)

//clip
$clipX:=125
$clipY:=125
$clipWidth:=200
$clipHeight:=250
$clipPath:=DOM Create XML element($defs;"clipPath";"id";"clip-path-ellipse")
$ellipse:=DOM Create XML element($clipPath;"ellipse";\
"cx";$clipX+($clipWidth/2);"cy";$clipY+($clipHeight/2);\
"rx";$clipWidth/2;"ry";$clipHeight/2;\
"shape-rendering";"geometricPrecision")
```

```

//local URI
$clipImage:=DOM Create XML element($defs;"image";\
"id";"clip-image";"width";$width;"height";$height;\
"xlink:href";"data:image/png;base64,"+$data)

//original image
$use:=DOM Create XML element($svg;"use";\
"xlink:href";"#clip-image";\
"x";0;"y";0;"width";$width;"height";$height)

//filter the image, crop the image
$g:=DOM Create XML element($svg;"g";"clip-path";"url(#clip-path-ellipse)")
$use:=DOM Create XML element($g;"use";\
"xlink:href";"#clip-image";\
"filter";"url(#filter-blur)";\
"x";0;"y";0;"width";$width;"height";$height)

C_PICTURE($filteredImage)
SVG EXPORT TO PICTURE($svg;$filteredImage)
DOM CLOSE XML($svg)

WRITE PICTURE FILE(System folder(Desktop)+"sample.svg";$filteredImage)

```

The result image:



6. INTERACTION

6.1 Handling mouse events

Problem

You want to process a mouse event on an SVG.

Solution

Use [SVG Find element ID by coordinates](#) to identify the [id](#) of the clicked element.

The command works as long as the picture has a rendering tree. It reports the topmost visible element that has an [id](#) and an opacity of 10% or higher. Transparent objects are considered invisible.

You can pass the system variables MOUSEX and MOUSEY directly during the following events:

- On Clicked
- On Double Clicked
- On Mouse Enter
- On Mouse Move

By contrast, the system variables are not synchronised during an "On Timer" event, and their values are set to "-1" during an "On Mouse Leave" event. You can work out the local mouse position using the following formula:

```
OBJECT GET COORDINATES (*; "Image"; $l; $t; $r; $b)
```

```
GET MOUSE ($x; $y; $b)
```

```
$MOUSEX:=$x-$l
```

```
$MOUSEY:=$y-$t
```

Note: The command takes care of the current scroll position or zoom factor.

The "On Clicked" event is generated when the mouse button is pressed, not when it is released. The picture object does not have to be enterable, but the "Draggable" property should be unselected. When a picture object is draggable, the "On Clicked" event will only fire if the mouse button is released without any movement.

If you intend to move or resize objects during a drag event, it might help to use [SVG SET ATTRIBUTE](#) without the asterisk, so that only the rendering tree is updated. It allows you to consult the original attribute values from the DOM, based on which you can calculate the moved or resized coordinates.

6.2 Handling selections

Problem

You want to draw a selection rectangle over an SVG.

Solution

Draw and update the selection rectangle using an "On Timer" event.

You can add a new `<rect>` element in response to a click event, start a timer, and update the width and size of the selection while the mouse button is down. This way you can update the rectangle even if the mouse moves beyond the bounds of the picture object.

If no objects are selected, you can simply update the [width](#) and [height](#) of the selection. Make sure you always pass a positive [width](#) or [height](#), by flipping the coordinates if the mouse has moved to the left or above the starting point.

If the image is larger than the picture object, you must also take into account the scroll position to draw the selection rectangle correctly. The formula to work out the local mouse position will be as follows:

```
OBJECT GET COORDINATES (*;"Image";$l;$t;$r;$b)
OBJECT GET SCROLL POSITION (*;"Image";$scrollV;$scrollH)
GET MOUSE ($x;$y;$b)
$MOUSEX:=$x-$l+$scrollH
$MOUSEY:=$y-$t+$scrollV
```

Objects touched by the selection rectangle can be found with the [SVG Find element IDs by rect](#) command. Obviously you need a scheme to exclude the selection rectangle and handles from the list of object IDs.

6.3 Drawing resize handles

Problem

You want to draw resizing handles around objects in SVG.

Solution

Implement a naming scheme with which you can identify the handle kind: top-left, top-middle, top-right, middle-left, middle-right, bottom-left, bottom-middle and bottom right. The amount by which you grow the object in response to moving a handle is determined by several factors:

- The handle kind, i.e. on which side of the object it is placed (without rotation)
- The scroll position multiplied by the global zoom factor, if any
- The object's rotation
- The distance by which the mouse moved relative to where the drag has started

First, you need to know the logical travel distance of the mouse pointer (movement + scroll / zoom):

```
$moveX:=$mouseX-$clickX+($scrollH/$zoomH)
$moveY:=$mouseY-$clickY+($scrollV/$zoomV)
```

Next, you calibrate the distance according to the object's rotation:

```
$offsetX:=(Cos((($r)*Degree))*$moveX*$sx)+(Sin((($r)*Degree))*$moveY*$sy)
$offsetY:=(Cos((($r)*Degree))*$moveY*$sy)+(-Sin((($r)*Degree))*$moveX*$sx)
```

Note: Translating the object coordinates so that the center position is always "0, 0" makes these kinds of calculation so much easier, compared to using absolute coordinates.

Finally, you calculate the offset you need to add depending on the handle kind:

		offset
tl	x	$-\left(\frac{\text{offsetY}}{2}\right) * \sin((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \cos((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetY}}{2}\right) * \cos((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \sin((r) * \text{Degree})$
	width	-offsetX
	height	-offsetY
tm	x	$-\left(\frac{\text{offsetY}}{2}\right) * \sin((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetY}}{2}\right) * \cos((r) * \text{Degree})$
	width	±0
	height	-offsetY
tr	x	$-\left(\frac{\text{offsetY}}{2}\right) * \sin((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \cos((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetY}}{2}\right) * \cos((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \sin((r) * \text{Degree})$
	width	+offsetX
	height	-offsetY
ml	x	$+\left(\frac{\text{offsetX}}{2}\right) * \cos((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetX}}{2}\right) * \sin((r) * \text{Degree})$
	width	-offsetX
	height	±0
mr	x	$+\left(\frac{\text{offsetX}}{2}\right) * \cos((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetX}}{2}\right) * \sin((r) * \text{Degree})$
	width	+offsetX
	height	±0
bl	x	$-\left(\frac{\text{offsetY}}{2}\right) * \sin((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \cos((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetY}}{2}\right) * \cos((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \sin((r) * \text{Degree})$
	width	-offsetX
	height	+offsetY
bm	x	$-\left(\frac{\text{offsetY}}{2}\right) * \sin((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetY}}{2}\right) * \cos((r) * \text{Degree})$
	width	±0
	height	+offsetY
br	x	$-\left(\frac{\text{offsetY}}{2}\right) * \sin((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \cos((r) * \text{Degree})$
	y	$+\left(\frac{\text{offsetY}}{2}\right) * \cos((r) * \text{Degree}) + \left(\frac{\text{offsetX}}{2}\right) * \sin((r) * \text{Degree})$
	width	+offsetX
	height	+offsetY

7. MISCELLANEOUS

7.1 Activating hardware acceleration (Windows)

Problem

You want to take advantage of hardware acceleration for rendering SVG.

Solution

Set "Direct2D Status" to "Direct2D hardware SVG and editors" with [SET DATABASE PARAMETER](#).

Discussion

The Windows version of 4D has 3 graphic contexts; **GDI+**, **WARP** (a.k.a. software Direct2D) and hardware **Direct2D**. By default, the SVG rendering engine uses [WARP](#), which means that hardware acceleration by the graphic card is disabled. You can change this by switching the database parameter.

```
SET DATABASE PARAMETER(Direct2D status;Direct2D hardware SVG and editors)
```

7.2 Activating legacy font rendering (Windows)

Problem

You want to follow the font rendering system in 4D versions prior to v13.

Solution

Set the private attribute "4D-enableD2D" to "false" with [SVG SET ATTRIBUTE](#).

Discussion

In general, Direct2D is more performant and preferable over GDI+. GDI+ mode is mainly kept for compatibility, as there are known differences in drawing especially related to rendering font. If you prefer the legacy GDI+ rendering, you can **turn off Direct2D** with [SET DATABASE PARAMETER](#), as explained the [SVG component](#) documentation.

```
SET DATABASE PARAMETER(Direct2D status;Direct2D disabled)
```

Note: GDI+ was selected by default on Windows XP (v13) or Vista (v14), as software Direct2D can be too stressful on systems with limited computing capacity.

Doing this, however, means that hardware Direct2D is disabled across the board, including text, primitives (shapes) and all other form objects. If you prefer to have hardware acceleration for generic form objects, yet use GDI+ for the SVG engine, you can specifically opt out of Direct2D by setting the private attribute "4D-enableD2D" to "false" with [SVG SET ATTRIBUTE](#). Since the scope of the setting is global, you only need to call the command once and you can pass an empty string as the element ID. However the variable must be a valid SVG picture.

```
SVG SET ATTRIBUTE($mySVG;"";"4d-enableD2D";"false")
```

Note: The list of attributes prefixed by "4D-" can be found in the documentation for [SVG GET ATTRIBUTE](#), even the ones that can also be pass to the "SET" command.

There used to be another reason for using GDI+ mode, which was that SVG filters were not support in Direct2D mode. This limitation has been [removed in v15](#) (14R5). Now, basic SVG filters (Gaussian Blur, Offset, Blend, and Colour Matrix) are also supported in WARP (software Direct2D) mode.

7.3 Changing the background colour

Problem

You want to change the colour and/or opacity of the view port.

Solution

Set the [viewport-fill](#) and [viewport-fill-opacity](#).

Discussion

The SVG engine supports **most** of the features defined in SVG 1.1, as well as **some** of the features defined in [SVG Tiny 1.2](#). Such include the ability to define the fill properties of the view port. Note that most browsers and graphic viewers do not support these properties yet.

```
SVG SET ATTRIBUTE (*;"Image";"svg";\
"viewport-fill";"#0000FF";\
"viewport-fill-opacity";0.5)
```

If the picture object does not have a transparent background, regions outside the SVG will be painted with the background colour. You can always make the background transparent by code:

```
OBJECT SET RGB COLORS (*;"Image";Foreground color;Background color none)
```

7.4 Changing the DPI

Problem

You want to change the DPI (dot-per-inch) of the SVG.

Solution

Set the private attribute "ns4d:DPI".

Discussion

For historical reasons, the standard DPI of 4D form objects are set to 72, though most browsers display SVG in 96DPI. As a result, an SVG with **mixed** units may be rendered disproportionately. Since v13, it is possible to change the SVG engine's DPI using a private attribute.

<http://forums.4d.fr/Post/FR/7887199/1/7887828>

Example of setting the DPI to "96":

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<svg xmlns="http://www.w3.org/2000/svg" xmlns:ns4d="http://www.4d.com" ns4d:DPI="96">
  <rect x="0.5" y="0.5" height="1in" stroke-width="1" width="96" />
</svg>
```

REMARKS

There is another way to use SVG in 4D, which is to use it as part of an HTML document rendered inside a WebArea. Although technically part of 4D, the WebArea (WebKit or native) is effectively a third party rendering engine and its vast feature set (HTML, CSS and JavaScript) is too broad to cover in this session. You should look for external resources on the subject of using SVG in a modern web browser.