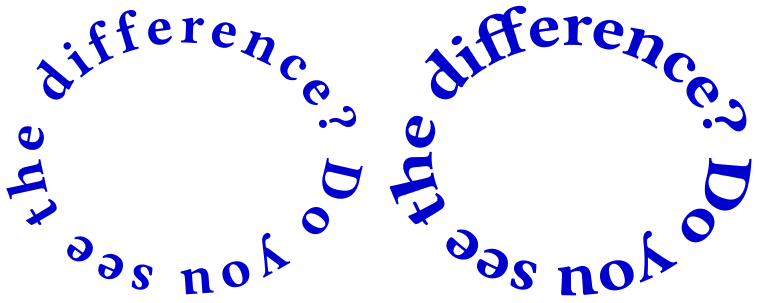
Integration

When you put text on a curve there are (at least) two variants to consider:

- You typeset the text and put glyphs along the shape, rotating the glyph following the curve. Rotation happens around the the center of the baseline of the glyph.
- You typeset the text and transform the outline according to the position on the shape. So, when the shape is a circle the glyphs becomes wider at its top.

The first variant uses the original glyph path and therefore we can use the regular font resource, while the second variant has to fetch the outline from the font resource and work with the path. Here we discuss the second case.



first variant second variant

Figure 1.

Because we gradually translate points we cannot simply take the existing points that define the glyph. For instance, the topmost line of the T normally has two points and just translating these two ignores the fact that in the middle we move up. As demonstrated in figure 2 we have to add points.

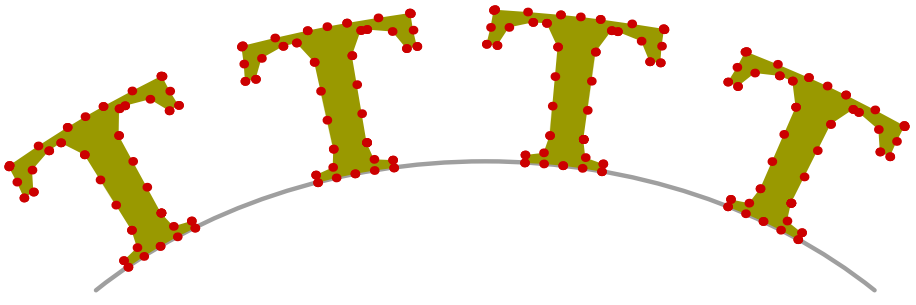


Figure 2.

In ConTEXt speak, this example is generated with:

```
\startMPcode
    path shape ;
    shape := reverse subpath(1,3) of fullcircle
        xyscaled (500,400)
```

```
    ;
    draw shape
        withpen pencircle scaled 2
        withcolor "middlegray"
    ;
    path p ; p := lmt_oscillated [
        text       = "\bf T T T T ",
        path       = shape,
        resolution = 5,
    ] ;
    fill       p withcolor "middleyellow" ;
    drawpoints p withcolor "middlered" ;
\stopMPcode
```

This feature is a good example of the way we integrate TEX, Lua and MetaPost so let's see what happens here. We start at the TEX end, where we ask for some MetaPost graphic. The code between `\startMPcode` and `\stopMPcode` is picked up at the Lua end and passed to a MetaPost instance. When processed the result is picked up by Lua, and the graphic operations become pdf code.

When we are in MetaPost, we need the outline of the text. So there MetaPost picks up the text and (via Lua) passes it to TEX where it gets typeset and ends up in a box. Processing the text uses the (set) font and font features are applied, think of ligature building and kerning. The resulting node list contains references to glyphs in a font as well as kerns (and maybe rules). Because we want the outlines, we trigger a dedicated so called 'shipout' routine, one that assembles a picture from those outlines. That picture is then passed back to MetaPost, where successive paths get positioned along the shape whereby points are added and relocated. Some glyphs are made from multiple paths. for instance an `O` has two paths and a `B` has three.

Getting the outline from the font means that we need to collect the so called cff or ttf glyph path descriptions and translate them to MetaPost operations. Reading a font and parsing these path descriptions is again a task for Lua. We actually need these (original) path descriptions when we subset a font for embedding and have to manipulate them when we are dealing with a variable font. The export to a MetaPost variant was written at the time we wrote the Lua code, if only because we love tracing such properties and features.

It will be clear that we call Lua from TEX as well as MetaPost various times. In this case `lmt_oscillated` is rather simple, most code involves calculating the (additional) points from given points, the rest was just there. This macro actually calls out to Lua where we pick up the parameters. Only when we are done typesetting and collecting glyph paths we return to MetaPost. We use scanners to get data and injectors to push data at the TEX and MetaPost end. These mechanism are quite efficient. We basically extend both languages with new features. We can summarize the process as:

$$\text{TEX}_1 \rightarrow \text{Lua}_1 \rightarrow \text{MetaPost}_1 \rightarrow \text{Lua}_2 \rightarrow \text{TEX}_2 \rightarrow \text{Lua}_3 \rightarrow \text{Lua}_4 \rightarrow \text{MetaPost}_2 \rightarrow \text{Lua}_5 \rightarrow \text{pdf}_1 \rightarrow \text{TEX}_3$$

The various steps shown here are:

TEX_1	ask for graphic with macro
Lua_1	prepare graphic data and pass to instance
MetaPost_1	process graphic and call macro
Lua_2	pick up arguments
TEX_2	typeset text
Lua_3	pickup outlines
Lua_4	convert result to MetaPost shape
MetaPost_2	position transformed glyphs
Lua_5	pickup result from run and convert

pdf_1	graphic description
TEX_3	position and scale result in document

It will be clear that writing this solution from scratch without any code being present takes some effort. However, as were already using LuaTEX and LuaMetaTEX for a while in the end we only had to spend time on step $MetaPost_2$. Mikael Sundqvist and I started our explorations with a more complex approach, which actually exposed some complications so eventually we decided on a slightly more demanding variant in terms of runtime, and the final code actually has less lines. Figure 3 demonstrates that ligatures are handled well.

Figure 3.

Of course we can combine this with other integrated features, for instance with so called 'Perlin Noise' which generates patterns that looks random but where distortions are related (think for instance clouds):

```
\definecharacterkerning
  [mine]
  [factor=0.05]

\startMPcode
    draw lmt_noise [
        bytemap     = 4,
        nx          = 1000,
        ny          = 1000,
        nz          = 1,
        iterations  = 2,
        frequency   = 0.1,
        amplitude   = 1,
        persistence = 0.5,
        lacunarity  = 2.0,
        minimum     = 0,
        maximum     = 255,
        method      = "angle",
    ] xysized (4cm,4cm) shifted (-2cm,-2cm) ;
    fill lmt_oscillated [
        text = "\setcharacterkerning[mine1]\bf\CONTEXT\ LMTX is FUN! "
        path = reverse fullcircle xyscaled (25mm,25mm) ,
    ] withtransparency (1,.75) withcolor white ;
\stopMPcode
```

Here we first create a background: again we parse the specification from Lua using MetaPost scanning routines, then we generate a byte map with noise using a small C library that we access from Lua. When done we tell MetaPost the dimensions so that it can make a picture that refers to the byte map (in this case 4) which it can scale, rotate, whatever. The backend gets told that it has to include that byte map instead of the picture (path) so again Lua kicks in and we get figure 4.

Figure 4.

Especially in the LuaMetaFun manual there are various examples of the tight integration of the three languages, with occasional help from a C library. We wrap up with two examples where we use a small library that generates a QRcode.

The first variant used a Lua helper in the ConTEXt graphic library. Here MetaPost is not involved:

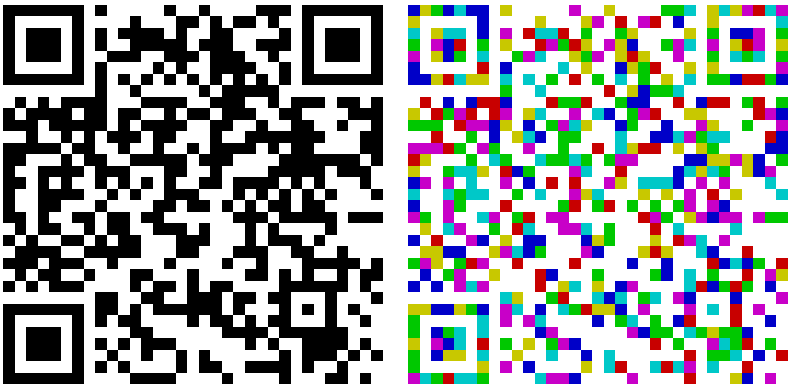
```
\startluacode
    context.dontleavehmode()
    figures.qrcode(
        "To use LUA or METAPOST, that's the question.",
        tex.sp("5cm")
    )
\stopluacode
```

The second example calls a Lua function from the MetaPost end where we replace the black rectangles with colorful ones. My phone can handle it.

```
\startluacode

local colors = {
    string.char(200,  0,  0),
    string.char(  0,200,  0),
    string.char(  0,  0,200),
    string.char(  0,200,200),
    string.char(200,  0,200),
    string.char(200,200,  0),
}

local white = string.char(255,255,255)
local black = string.char(  0)

function MP.Test(index,str)
     local data, size = qrcodegen.generate(str)
     data = string.gsub(data,"(.)",function(c)
        if c == black then
            return colors[math.random(6)]
        else
            return white
        end
     end)
```

```
    mp.newbytemap(index,size,size,3,data)
end
\stopluacode

\startMPcode
    lua.MP.Test(3, "To use LUA or METAPOST, that's the question.") ;
    draw bytemap 3 xsized 5cm ;
\stopMPcode
```

The reason why we added this library is that you can configure ConTEXt to pop up an error message in QRcode speak. We also used these codes on the ConTEXt 2025 meeting poster and for ear tags with familiar TEX error messages at BachoTEX. Anyway, we show both examples side by side in figure 5. I hope that we made clear that the possibilities of combining techniques are endless.



Just Lua. Lua and MetaPost.

Figure 5.

Hans Hagen