low level TEX

accuracy

Contents

1 Introduction

When you look at TEX and MetaPost output the accuracy of the rendering stands out, unless of course you do a sloppy job on design and interfere badly with the system. Much has to do with the fact that calculations are very precise, especially given the time when TEX was written. Because TEX doesn't rely on (at that time non-portable) floating point calculations, it does all with integers, except in the backend where glue calculations are used for finalizing the glue values. It all changed a bit when we added Lua because there we mix integers and doubles but in practice it works out okay.

When looking at floating point (and posits) one can end up in discussions about which one is better, what the flaws fo each are, etc. Here we're only interested in the fact that posits are more accurate in the ranges where TEX and MetaPost operate, as well as the fact that we only have 32 bits for floats in TEX, unless we patch more heavily. So, it is also very much about storage.

When you work with dimensions like points, they get converted to an integer number (the `sp` unit) and from that it's just integer calculations. The maximum dimension is 16383.99998pt, which already shows a rounding issue. Of course when one goes precise for sure there is some loss, but on the average we're okay. So, in the next example the two last rows are equivalent:

```
       .1pt  0.1pt   6554sp
       .2pt  0.2pt  13107sp
       .3pt  0.3pt  19661sp
.1pt + .2pt  0.3pt  19661sp
```

When we're at the Lua end things are different, there numbers are mapped onto floating point variables (doubles) and not all numbers map well. This is what we get when we work with doubles in Lua:

```
     .1  0.1
     .2  0.2
     .3  0.3
.1 + .2  0.3
```

The serialization looks as if all is okay but when we test for equality there is a problem:

```
     .3 == .3  true
.1 + .2 == .3  false
```

This means that a test like this can give false positives or negatives unless one tests the difference against the accuracy (in MetaPost we have the eps variable for that). In TEX clipping of the decimal fraction influences equality.

```
     \iflua{ .3 == .3 } Y\else N\fi  equal
\iflua{ .1 + .2 == .3 } Y\else N\fi  different
```

The serialization above misguides us because the number of digits displayed is limited. Actually, when we would compare serialized strings the equality holds, definitely within the accuracy of TEX. But here is reality:

	.3	.1 + .2
%0.10g	0.3	0.3
%0.17g	0.29999999999999999	0.30000000000000004
%0.20g	0.2999999999999999889	0.30000000000000004441
%0.25g	0.2999999999999999888977698	0.3000000000000000044408921

The above examples use 0.1, 0.2 and 0.3 and on a 32 bit float that actually works out okay, but LuaMetaTEX is 64 bit. Is this really important in practice? There are indeed cases where we are bitten by this. At the Lua end we seldom test for equality on calculated values but it might impact check for less or greater then. At the TEX end there are a few cases where we have issues but these also relate to the limited precision. It is not uncommon to loose a few scaled points so that has to be taken into account then. So how can we deal with this? In the next section(s) an alternative approach is discussed. It is not so much the solution for all problems but who knows.

2 Posits

The next table shows the same as what we started with but with a different serialization.

```
     .1          0.1
     .2          0.2
     .3  0.300000001
.1 + .2  0.300000001
```

And here we get equality in both cases:

```
     .3 == .3  true
.1 + .2 == .3  true
```

The next table shows what we actually are dealing with. The `\if`-test is not a primitive but provided by ConTEXt.

```
     \ifpositunum{ .3 == .3 } Y\else N\fi  equal
\ifpositunum{ .1 + .2 == .3 } Y\else N\fi  equal
```

And what happens when we do more complex calculations:

```
math .sin(0.1 + 0.2) == math .sin(0.3)  false
posit.sin(0.1 + 0.2) == posit.sin(0.3)  true
```

Of course other numbers might work out differently! I just took the simple tests that came to mind.

So what are these posits? Here it's enough to know that they are a different way to store numbers with fractions. They still can loose precision but a bit less on smaller values and often we have relative small values in TEX. Here are some links:

```
https://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf
https://posithub.org/conga/2019/docs/14/1130-FlorentDeDinechin.pdf
```

There are better explanations out there than I can provide (if at all). When I first read about these unums (a review of the 2015 book “The End of Error Unum Computing”) I was intrigued and when in 2023 I read something about it in relation to RISCV I decided to just add this playground for the users. After all we also have decimal support. And interval based solutions might actually be good for MetaPost, so that is why we have it as extra number model. There we need to keep in mind that MetaPost in non scaled models also apply some of the range checking and clipping that happens in scaled (these magick 4096 tricks).

For now it is enough to know that it's an alternative for floats that *could* work better in some cases but not all. The presentation mentioned above gives some examples of physics constants where 32 posits are not good enough for encoding the extremely large or small constants, but for π it's all fine.[1] In double mode we actually have quite good precision compared to 32 bit posits but with 32 bit floats we gain some. Very small numbers and very large numbers are less precise, but around 1 we gain: the next value after 1 is 1.0000001 for a float and 1.000000008 for a posit (both 32 bit). So, currently

[1] Are 64 bit posits actually being worked on in softposit? There are some commented sections. We also need to patch some unions to make it compile as C.

for MetaPost there is no real gain but if we'd add posits to TEX we could gain some because there a halfword (used for storing data) is 32 bit.

But how about TEX? Per April 2023 the LuaMetaTEX engine has native support for floats (this in addition to Lua based floats that we already had in ConTEXt). How that works can be demonstrated with some examples. The float related commands are similar to those for numbers and dimensions: `\floatdef`, `\float`, `\floatexpr`, `\iffloat`, `\ifzerofloat` and `\ifintervalfloat`. That means that we also have them as registers. The `\positdef` primitive is similar to `\dimensiondef`. When a float (posit) is seen in a dimension context it will be interpreted as points, and in an integer context it will be a rounded number. As with other registers we have a `\newfloat` macro. The `\advance`, `\multiply` and `\divide` primitives accept floats.

```
\scratchdimen=1.23456pt
\scratchfloat=1.23456
```

We now use these two variables in an example:

```
\setbox0\hbox to \scratchdimen {x}\the\wd0
\scratchdimen \dimexpr \scratchdimen * 2\relax
\setbox0\hbox to \scratchdimen {x}\the\wd0
\advance \scratchdimen \scratchdimen
\setbox0\hbox to \scratchdimen {x}\the\wd0
\multiply\scratchdimen by 2
\setbox0\hbox to \scratchdimen {x}\the\wd0
```

```
1.23456pt
2.46912pt
4.93823pt
9.87646pt
```

When we use floats we get this:

```
\setbox0\hbox to \scratchfloat {x}\the\wd0
\scratchfloat \floatexpr \scratchfloat * 2\relax
\setbox0\hbox to \scratchfloat {x}\the\wd0
\advance \scratchfloat \scratchfloat
\setbox0\hbox to \scratchfloat {x}\the\wd0
\multiply\scratchfloat by 2
\setbox0\hbox to \scratchfloat {x}\the\wd0
```

```
1.23456pt
2.46912pt
```

```
4.93823pt
9.87648pt
```

So which approach is more accurate? At first sight you might think that the dimensions are better because in the last two lines they indeed duplicate. However, the next example shows that with dimensions we lost some between steps.

```
                                                \the\scratchfloat
\scratchfloat \floatexpr \scratchfloat * 2\relax \the\scratchfloat
\advance \scratchfloat \scratchfloat             \the\scratchfloat
\multiply\scratchfloat by 2                      \the\scratchfloat
```

```
1.2345600128173828
2.4691200256347656
4.9382400512695313
9.8764801025390625
```

One problem with accuracy is that it can build up. So when one eventually does some comparison the expectations can be wrong.

```
\dimen0=1.2345pt
\dimen2=1.2345pt

\ifdim           \dimen0=\dimen2 S\else D\fi \space +0sp: [dim]
\ifintervaldim0sp\dimen0 \dimen2 O\else D\fi \space +0sp: [0sp]

\advance\dimen2 1sp

\ifdim             \dimen0=\dimen2 S\else D\fi \space +1sp: [dim]
\ifintervaldim 1sp \dimen0 \dimen2 O\else D\fi \space +1sp: [1sp]
\ifintervaldim 1sp \dimen2 \dimen0 O\else D\fi \space +1sp: [1sp]
\ifintervaldim 2sp \dimen0 \dimen2 O\else D\fi \space +1sp: [2sp]
\ifintervaldim 2sp \dimen2 \dimen0 O\else D\fi \space +1sp: [2sp]

\advance\dimen2 1sp

\ifintervaldim 1sp \dimen0\dimen2 O\else D\fi \space +2sp: [1sp]
\ifintervaldim 1sp \dimen2\dimen0 O\else D\fi \space +2sp: [1sp]
\ifintervaldim 5sp \dimen0\dimen2 O\else D\fi \space +2sp: [5sp]
\ifintervaldim 5sp \dimen2\dimen0 O\else D\fi \space +2sp: [5sp]
```

Here we show a test for overlap in values, the same can be done with integer numbers (counts) and floats. This interval checking is an experiment and we'll see it if gets used.

```
S +0sp: [dim] O +0sp: [0sp]
D +1sp: [dim] O +1sp: [1sp] O +1sp: [1sp] O +1sp: [2sp] O +1sp: [2sp]
D +2sp: [1sp] D +2sp: [1sp] O +2sp: [5sp] O +2sp: [5sp]
```

Just to be clear, we use 32 bit posits and not 32 bit floats, which we could have but that way we gain some accuracy because less bits are used by default for the exponential.

In ConTEXt we also provide a bunch of pseudo primitives. These take one float: `\pfsin`, `\pfcos`, `\pftan`, `\pfasin`, `\pfacos`, `\pfatan`, `\pfsinh`, `\pfcosh`, `\pftanh`, `\pfasinh`, `\pfacosh`, `\pfatanh`, `\pfsqrt`, `\pflog`, `\pfexp`, `\pfceil`, `\pffloor`, `\pfround`, `\pfabs`, `\pfrad` and `\pfdeg`, whiel these expect two floats: `\pfatantwo`, `\pfpow`, `\pfmod` and `\pfrem`.

3 MetaPost

In addition to the instances `metafun` (double in LMTX), `scaledfun`, `doublefun`, `decimalfun` we now also have `positfun`. Because we currently use 32 bit posits in the new number system there is no real gain over the already present 64 bit doubles. When 64 bit posits show up we might move on to that.

3 Colofon

Author	Hans Hagen
ConTEXt	2023.05.08 17:39
LuaMetaTEX	2.1009
Support	www.pragma-ade.com
	contextgarden.net