MAPS

NUMMER 50 • VOORJAAR 2020

NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

Voorzitter
Hans Hagen
voorzitter@ntg.nl

Secretaris
Taco Hoekwater
secretaris@ntg.nl

Penningmeester
Ferdy Hanssen
penningmeester@ntg.nl

Bestuursleden
Frans Goddijn
Pieter van Oostrum

Postadres
Nederlandstalige TEX Gebruikersgroep
Baarsjesweg 268-I
1058 AD Amsterdam

ING bankrekening
IBAN: NL53INGB0001306238
BIC: INGBNL2A

E-mail bestuur
ntg@ntg.nl

E-mail MAPS redactie
maps@ntg.nl

WWW
www.ntg.nl

De **Nederlandstalige TEX Gebruikersgroep (NTG)** is een vereniging die tot doel heeft de kennis en het gebruik van TEX te bevorderen. De NTG fungeert als een forum voor nieuwe ontwikkelingen met betrekking tot computergebaseerde documentopmaak in het algemeen en de ontwikkeling van 'TEX and friends' in het bijzonder. De doelstellingen probeert de NTG te realiseren door onder meer het uitwisselen van informatie, het organiseren van conferenties en symposia met betrekking tot TEX en daarmee verwante programmatuur.
De NTG biedt haar leden ondermeer:

- Tweemaal per jaar een NTG-bijeenkomst.
- Het NTG-tijdschrift MAPS.
- De 'TEX Live'-distributie op DVD/CDROM inclusief de complete CTAN software-archieven.
- Verschillende discussielijsten (mailing lists) over TEX-gerelateerde onderwerpen, zowel voor beginners als gevorderden, algemeen en specialistisch.
- De FTP server `ftp.ntg.nl` waarop vele honderden megabytes aan algemeen te gebruiken 'TEX-producten' staan.
- De WWW server `www.ntg.nl` waarop algemene informatie staat over de NTG, bijeenkomsten, publicaties en links naar andere TEX sites.
- Korting op (buitenlandse) TEX-conferenties en -cursussen en op het lidmaatschap van andere TEX-gebruikersgroepen.

Lid worden kan door overmaking van de verschuldigde contributie naar de NTG-giro (zie links); vermeld IBAN zowel als SWIFT/BIC en selecteer shared cost. Daarnaast dient via `www.ntg.nl` een informatieformulier te worden ingevuld. Zonodig kan ook een papieren formulier bij het secretariaat worden opgevraagd.
De contributie bedraagt € 35. Voor studenten geldt een tarief van € 18. Dit geeft alle lidmaatschapsvoordelen maar *geen stemrecht.* Een bewijs van inschrijving is vereist. Een gecombineerd NTG/TUG-lidmaatschap levert een korting van 10% op beide contributies op. De prijs in euro's wordt bepaald door de dollarkoers aan het begin van het jaar. De ongekorte TUG-contributie is momenteel $105.

Afmelding kan met ingang van het volgende kalenderjaar door opzegging per e-mail aan de penningmeester.

MAPS bijdragen kunt u opsturen naar `maps@ntg.nl`, bij voorkeur in LATEX- of ConTEXt formaat. Bijdragen op alle niveaus van expertise zijn welkom.

Productie. De Maps wordt gezet met behulp van een LATEX class file en een ConTEXt module. Het pdf bestand voor de drukker wordt aangemaakt met behulp van pdftex 1.40.20 en luatex 1.11.1 draaiend onder MacOS X 10.15. De gebruikte fonts zijn Linux Libertine, het niet-proportionele font Inconsolata, schreefloze fonts uit de Latin Modern collectie, en de Euler wiskunde fonts, alle vrij beschikbaar.

TEX is een door professor Donald E. Knuth ontwikkelde 'opmaaktaal' voor het letterzetten van documenten, een documentopmaaksysteem. Met TEX is het mogelijk om kwalitatief hoogstaand drukwerk te vervaardigen. Het is eveneens zeer geschikt voor formules in mathematische teksten.
Er is een aantal op TEX gebaseerde producten, waarmee ook de logische structuur van een document beschreven kan worden, met behoud van de letterzet-mogelijkheden van TEX. Voorbeelden zijn LATEX van Leslie Lamport, $\mathcal{AMS}$-TEX van Michael Spivak, en ConTEXt van Hans Hagen.

Contents

Redactioneel

Door middel van de Maps willen we u op de hoogte houden van ontwikkelingen, ook om daarmee onze leden te danken voor hun trouwe steun aan de TEX ontwikkelaars. Verder bieden we ruimte aan lezers die anderen laten delen in hun ervaringen met TEX, MetaPost, fonts en aanverwanten. Aarzel dus niet ons artikelen te sturen. Een halve pagina is al heel leuk, meer mag ook, graag zelfs. Het hoeft geen ‚zware kost' te zijn want het is voor lezers bijvoorbeeld al heel interessant te lezen hoe anderen TEX gebruiken. Dus een artikeltje als „dit doe ik met TEX, zo doe ik dat en nu kun jij het ook" is zeer welkom!

Hoewel het internet tegenwoordig een belangrijke bron van informatie is, blijft papier een functie vervullen binnen de vereniging. Dat past immers bij TEX!

Deze Maps bevat van alles wat, en daar kunnen we met zijn allen best trots op zijn. Niet alleen hebben we de bijkans verplichte ‚vooral voor ontwikkelaars' artikelen van Hans Hagen en updates van buitenlandse auteurs – waaronder een tweetal Duitstalige artikelen – maar we hebben ook diverse Nederlandstalige artikelen van de hand van onze eigen (ex)leden.

Onze welgemeende excuses naar Dennis van Dok: zijn artikel verscheen eerder onvolledig in Maps 49. Deze keer hebben we *wél* je volledige artikel opgenomen!

Veel leesplezier,

Uw redactie

Waarom is een NTG lidmaatschap belangrijk?

TEX bestaat bijna 40 jaar en het ziet er niet naar uit dat het systeem snel zal verdwijnen. Oorspronkelijk was het alleen beschikbaar op universiteiten, later kon men het ook thuis installeren. Verschillende ‚journals' worden vormgegeven in TEX en op universiteiten is het nog steeds populair.

Het begon met TEX, geschreven door Donald Knuth. Later kwamen er wat extensies onder de vlag ε-TEX. De succesvolle variant met ingebouwd backend PDFTEX werd na verloop van tijd de standaard. Toen UNICODE opgang maakte kwam XETEX erbij, en wat later begon de ontwikkeling van LUATEX. Parallel aan deze ontwikkeling zijn fonts ontwikkeld, is MetaPost uitgekristalliseerd en zijn macro-pakketten als LATEX en ConTEXt ontstaan die een groot aantal talen en scripts ondersteunen.

De hele TEX beweging kan worden getypeerd met woorden als *originaliteit*, *solidariteit*, *continuïteit*, *ondersteuning* en *ontwikkeling*. Dat alles vinden we als gebruikers heel normaal. Vanzelfsprekend is het feit dat we al zo lang aanwezig zijn mede het gevolg van de sympathieke Donald Knuth. Rond hem ontstond de eerste gebruikersgroep TUG, maar al snel verenigden gebruikers zich in groepen, in de regel per taalgebied, met als doel ondersteuning van het taalgebied en de kenmerkende (lokale) typografie. De gebruikersgroepen organiseren bijeenkomsten, publiceren tijdschriften en hosten websites en mailing lijsten.

De verschillende programma's (ook wel ‚engines' genoemd) worden al sinds jaren verspreid door de gebruikersgroepen, de laatste decennia op CD's and DVD's. De Nederlandstalige Gebruikersgroep heeft daarbij een pioniersrol gespeeld. De programma's worden vergezeld van een uitgebreide collectie fonts en macros.

Tegenwoordig vindt de communicatie vooral plaats via internet, maar in het verleden (en met enige regelmaat nog steeds) kwamen ontwikkelaars samen op (internationale) bijeenkomsten. Regelmatig zijn door gebruikersgroepen projecten geïnitieerd en ondersteund. Op dit moment draagt de NTG substantieel bij aan de ontwikkeling van moderne fonts.

Wat gebeurt er als de gebruikersgroepen wegvallen? Dat is niet met zekerheid te zeggen, maar het is een feit dat er heel veel gebruikers zijn die hun hele leven lang het TEX ecosysteem gebruiken. Zij kunnen dankzij de gebruikersgroepen vooralsnog verzekerd zijn van continuïteit en stabiliteit. En dat zonder de tegenwoordig alom tegenwoordige reclame (zoals in apps), zonder lock-in op een bepaald hardware platform, onafhankelijk van een besturingssysteem, en natuurlijk zonder dringende uitnodiging om een betaalde pro-versie te gebruiken of deel te nemen aan een of ander ‚plan'.

Zal de rol van de vereniging veranderen? Natuurlijk! Publicaties worden bijvoorbeeld wat zeldzamer en concentreren zich op ontwikkelingen. Bijeenkomsten worden kleinschaliger en trekken vooral actieve TEXers wat weer bijdraagt aan de kwaliteit, volledigheid en continuïteit van distributies. Mailing lijsten en websites worden tegenwoordig aangevuld met forums en blogs. Er zijn nog steeds internationale bijeenkomsten.

De financiële positie van de NTG is solide, ondanks het feit dan het ledental wat afneemt en vergrijst. Vooralsnog wordt er gebruik gemaakt van (door leden) gesponsorde sites voor het hosten van de distributies en archieven. Als de font projecten zijn afgerond, is er weinig reden om geld te reserveren voor ontwikkeling. Om die reden zijn de kosten niet gestegen en kan het lidmaatschap laagdrempelig worden gehouden.

Lid zijn van de vereniging is afgezien van een garantie voor de beschikbaarheid van distributies ook een signaal aan gebruikers dat die garantie er vooralsnog is. Niet alles kan via internet, soms moeten we elkaar gewoon ontmoeten. En het helpt als ontwikkelaars weten dat er gebruikers zijn die zich willen verbinden aan het lot van TEX. Kortom, als de verenigingen zouden verdwijnen dan ontstaat denk ik snel een versnipperde en instabiele situatie. En dat willen we niet, toch?

Hans Hagen, voorzitter NTG

Koffiedik kijken

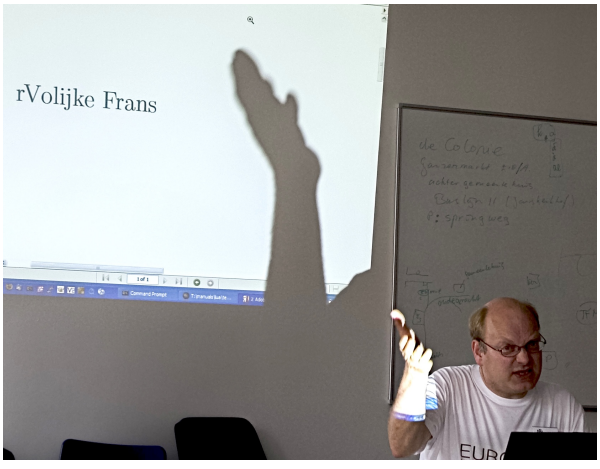
Rond TEX bijeenkomsten wordt weleens opgemerkt dat de kring steeds kleiner wordt. Minder leden dan begin jaren negentig, afnemend aantal deelnemers aan bijeenkomsten, geen uitgeverijen meer die stafleden afvaardigen om te komen vertellen over wat ze met TEX doen en komen afkijken hoe anderen hun typesetting aanpakken.

Komt het doordat steeds minder mensen wat met TEX doen? Nog steeds is TEX noodzakelijk voor veel gebruikers, zoals studenten informatica waarvan er juist méér zijn dan in de vroege jaren negentig. Maar voor hen is TEX een noodzakelijk kwaad, iets dat wordt gebruikt maar dat nou ook weer niet zó interessant is dat men zich er nu eens echt in gaat verdiepen. Zoals de meeste autobezitters geen behoefte hebben om te leren over wat er zich onder de motorkap bevindt.

Tijdens BachoTEX2019 zei een Tsjechische deelnemer me hoe leuk hij het vindt dat sprekers en luisteraars onderling vrij goed begrijpen waar de anderen het over hebben. Bij conferenties van economen is dat niet zo, vertelde hij, want daar houdt de een zijn voordracht over invloed van import en export op de lokale economie terwijl de ander het heeft over de hogere wiskunde van valutawisselkoersen. Geen van beiden steken ze wat van de ander op en daar omheen is ook weinig gezelligheid.

Nu moet ik bekennen dat ik zelf op de bijeenkomsten de meeste voordrachten aan me voorbij laat gaan. Ik ga weleens kijken maar doorgaans ben ik na de eerste zinnen de draad kwijt.

Toch is het wel fijn om erbij te zijn. Ik kijk bijvoorbeeld naar de schaduw van de spreker die ijsberend af en toe in de lichtstraal van de beamer stapt. De schaduw op de geprojecteerde pagina tekst en symbolen is dan vaak interessant, vooral als de spreker een brildrager is want de lichtbreking door de lens van de bril levert een flits van een zandloper op en de handgebaren van de spreker geven met hun schaduw op het doek als het ware commentaar op wat naar voren wordt gebracht. Schaduwvingers wijzen onbedoeld passages aan die ik dan ga lezen en zo kan men toch veel inspiratie opdoen.



Van de eerste jaren dat ik TEX bijeenkomsten bezocht herinner ik me dat ik thuiskwam met meer in mijn mars dan ik erheen ging. Ik begreep hoe ik een nieuw lettertype kon invoegen, had geleerd een zin in kleur te zetten, wist wat de nieuwe manier was om accenten op letters te plaatsen.

In de pauzes sprak ik dan weleens met NTG-leden die speciaal voor zulke weetjes kwamen. Een vrouw die voor haar werk drukkerijen adviseerde, vertelde me dat die ‚hints en tips' geld waard zijn in haar werk als consulent en zo hadden de bijeenkomsten aantrekkingskracht voor allerlei mensen.

Hoe zou dat vandaag kunnen? Het kan leuk zijn als je als aanwezige bij een NTG-dag op je USB-stick de bronbestanden van een presentatie krijgt zodat je daarna thuis ook zoiets kunt door de brontekst te vervangen.

Tijdens de afgelopen BachoTEX conferentie kwamen bezoekers op de koffie in het houten huisje in het bos op nummer 10 en ditmaal maakte ik van veel van hen een portretfoto en ook een foto van het ‚koffiedik' in hun kopje, zodat daaruit de toekomst van TEX zou zijn af te lezen.

Ton Otten had een van de meest uitgebreide interpretaties van hetgeen hij kon aflezen tijdens zijn oefening ‚koffiedik-kijken' en ik vat deze hier kort samen:

What I saw in my coffee grounds
by Ton Otten

In an attempt to Make America Great Again, Donald starts a trade war with Brazil, halting import of beef and McDonald's suffers severely. Other industries are targeted next with NSA declaring the Lua programming language to be a security liability. NATO partners are forced to stop using Lua and the gaming industry is also impacted, as is the TEX community. All LuaTEX development comes to a standstill as TEX users are advised to revert to the old pdfTEX. In response, development goes underground and because of this, large numbers of young tech students are attracted to the typesetting system and it is widely used as a form of protest. Development soars. When eventually the ban on Lua is lifted, TEX is well established as the de facto standard of typography tools.

Een mooie voorspelling.

De ommekeer van het lot waardoor TEX alsnog door een dynamische nieuwe generatie wordt opgepikt doet me denken aan de wendingen in het leven van Charles K. Blitz (1897–1985). Bij geboorte in de Oostenrijks-Hongaarse stad Czernowitz (het huidige Chernivtsi in Oekraïne) heet hij Karl Kasiel Blitz. Na een studie chemie aan de TU van Wenen werkt hij als onderzoeker bij een bedrijf in elektronica tot hij, een Jood, in 1938 wordt opgepakt en naar de concentratiekampen van Dachau en Buchenwald gebracht. Zijn Duitse vrouw Claire weet hem vrij te krijgen en ze vinden als vluchtelingen onderdak bij familie in het getto van Sjanghai.

Geïnspireerd door de Chinese karakters werkt Bliss (die zijn naam wijzigt omdat ‚Blitz' teveel mensen doet denken aan ‚Blitzkrieg') daar, en later in Sydney, aan een zelfverzonnen taal van symbolen omdat hij ervan is overtuigd dat oorlog wordt uitgebannen als ieder dezelfde ‚beeldtaal' begrijpt.

Hij zet zijn systeem uiteen in een boek (*Semantography*, 1949). Later verandert hij de naam van de symbolentaal in *Bliss* en hij noemt het ook wel *Blissymbolics* met het oog op de snelle opkomst van nieuwe standaarden van symbolen die nu verschijnen langs wegen en op stations en luchthavens waar een explosief groeiende massa mensen van verschillende nationaliteiten de weg moet worden gewezen.

Zakelijk succes blijft echter uit en het in eigen beheer uitgegeven boek wordt weinig gelezen tot begin jaren 70 Shirley McNaughton in Ontario zich realiseert dat de *Bliss* symbolen bij uitstek geschikt zijn om gehandicapte kinderen die niet in staat zijn te praten de kans te geven zich te uiten. Dit werkt verrassend goed en kinderen van wie men dacht dat ze mentaal zwakbegaafd waren blijken te kunnen vertellen wat er in hen omgaat, wat ze willen en met wie.

Bliss zelf intussen is enerzijds blij met de erkenning maar kan niet verdragen dat in verschillende taalgebieden en in verschillende toepassingen mensen zijn taal gaan voorzien van uitbreidingen en lokale zegswijzen. Ook blijken de *Bliss* symbolen een geweldig goede aanloop naar het leren van taal in letters, woorden en zinnen, geheel tegen de intentie van Bliss die juist de geschreven taal wil vervangen door zijn symbolen.

Uiteindelijk wordt het conflict met de bedenker in 1982 afgekocht voor $ 160.000 en sindsdien is de symbolentaal vrij, al rust het copyright bij *Blissymbolic Communication International.*

Misschien wordt TEX binnenkort door een nu nog onbekende groep mensen ‚ontdekt' die er vervolgens iets radicaal anders mee gaat doen, waardoor er een actieve gebruikersgroep ontstaat waar het bruist als nooit tevoren.

Als dat gebeurt, laten we het dan toejuichen!

Zou het intussen geen mooi idee zijn om de *Bliss* symbolen als font op te nemen in TEX?

Bronnen:

https://www.wnycstudios.org/story/257194-man-became-bliss
https://en.wikipedia.org/wiki/Blissymbols
http://www.blissymbolics.org/
http://www.blissymbolics.org/index.php/resources

Frans Goddijn

Dagboek van een Informaticus

Mijn eerste ervaring—of aanvaring?—met TEX moet ergens rond 1995 zijn geweest, als jonge twintiger, student informatica aan de Universiteit van Amsterdam. U moet zich de situatie trachten voor te stellen: ik woonde nog thuis bij mijn ouders, in een dorpje ver van Amsterdam. De keuze voor informatica was ingegeven door een fascinatie met computers die begon toen in mijn lagere-schooljaren een Commodore 64 het huis binnenkwam en die nooit meer voorbij is gegaan. De Commodore werd verdrongen door een opeenvolgende reeks van *IBM compatible* PC's die vader via het PC-Privé programma had aangeschaft. Eerst uitgerust met MS-DOS en later Microsoft Windows 3.1.

Het contrast met de computers op de universiteit was groot. Als tweedejaars student kregen we een UNIX account op het SunOS systeem van de faculteit, wat ook meteen het portaal naar de rest van de wereld was. Ik kreeg mijn eerste e-mailaccount en bijdehantere medestudenten lieten me zien welke FTP sites de beste spullen hadden.

De grafische Sun werkstations hadden een heel ander uiterlijk, gaven een heel ander *gevoel* dan de PC van thuis. Het oogde allemaal wat professioneler, voelde wat stabieler. Mettertijd leek het thuis-PC'tje steeds meer te krimpen, steeds onbehaaglijker te voelen, alsof het te heet werd gewassen. Naarmate ik behendiger werd op de UNIX commandoregel, kreeg mijn interactie met de DOS-prompt steeds meer het karakter van een gesprek met de dorpsgek: korte zinnen, geen moeilijke woorden gebruiken.

In één opzicht was de PC thuis echter superieur: deze was uitgerust met Ami Pro, een grafisch *desktop-publishing*-programma wat zijn gelijke niet kende op de Sun machines. Het programma kon moeiteloos overweg met een keur aan TrueType lettertypes en maakte het een fluitje van een cent om stijlen te definiëren voor koppen, broodtekst, opsommingen, etc. Alle typische elementen waren aanwezig: uitlijning, werken in kolommen, witruimtes, bladspiegel, inhoudsopgave, nummering, referenties, invoegen van afbeeldingen; in die tijd had het naar mijn smaak weinig tekortkomingen. Uiteraard is dit product van de markt gedrukt door het onontkoombare Microsoft Word, maar dit terzijde.

Ik herinner me dat ik na wat vruchteloos rondzoeken op het Internet van vóór het wereldwijde web bij de systeembeheerder op de faculteit binnenstapte en vroeg wat men eigenlijk gebruikte voor het maken van drukwerk. Het antwoord was nogal kortaf: dat was TEX en/of LATEX en daarmee kon ik het doen. Hierop volgde dus mijn eerste aanvaring met dat systeem en dat ging ongeveer als volgt.

Ik logde in op één van de werkstations in de practicumlokalen, opende een Xterm en tikte:

```
tex
```

waarop het programma mij onverwijld antwoordde:

```
This is TeX, Version 3.14159
**
```

met de cursor achter de beide sterretjes. Er werd dus iets van mij verwacht. Ik had ook wel iets verwacht en zeker meer dan alleen dit. Ik weet niet meer wat ik hierna ingetikt heb, maar het leidde nergens toe. De geheimen van TEX bleven voor mij vooralsnog verborgen.

Ik vroeg een oudere medestudent om hulp, en die legde mij eerst uit dat ik ten eerste geen TEX maar LATEX moest gebruiken (maar dat commando gaf precies hetzelfde resultaat toen ik het probeerde) en ten tweede dat ik eerst een bestand moest schrijven met een zekere structuur en een groot aantal mysterieuze commando's en ten derde dat ik eigenlijk het boek van Lamport zou moeten aanschaffen.

Dit maakte mij erg boos. Hoe kon je arme studenten nu dwingen om zulke dure boeken te kopen voor zoiets essentieels! Maar een informaticastudent laat zich niet zo gemakkelijk uit het veld slaan. Voor een veteraan van vakken zoals complexiteitstheorie is geen uitdaging te groot.

Aangezien het wereldwijde web nog even op zich liet wachten moest ik het hebben van FTP en Usenet, maar uiteindelijk ontdekte ik de schatkamer van CTAN. Alles wat ik ooit zou willen weten was daar te vinden. Na wat gesnuffel ontdekte ik de broncode van het TEXbook, de officiële handleiding van het systeem, en ik besloot dat dit het boek moest zijn wat ik zou gaan lezen om het systeem te leren kennen. Ik leg hier een bekentenis af, namelijk dat ik als arme student de ingebouwde beveiliging uit de broncode heb verwijderd om het boek in zijn volledigheid (behalve de illustraties) te kunnen afdrukken op de nagelnieuwe laserprinter van de faculteit. Ik heb het later (toen ik wat salaris had) wel met Knuth goedgemaakt door verschillende boeken van zijn hand te kopen.

Voor wie het TEXbook niet kent: behalve dat het een uitputtende handleiding is van TEX zelf, is het ook een leerzaam en leesbaar werk. Vooral de eerste paar hoofdstukken vormen een heldere introductie van de achterliggende bedoeling van het programma. Knuth geeft bij sommige paragrafen met verkeersborden aan dat er een gevaarlijke bocht volgt in de lijn der gedachten, en hij raadt aan om bij eerste lezing deze paragrafen over te slaan. Ik kan deze raad van harte onderschrijven, zeker omdat ik slachtoffer werd van mijn eigenwijsheid en zelfoverschatting toen ik toch alles wilde weten. Het boek wordt bij eerste lezing veel korter en verzandt niet in details.

Wat ik ook leerde van dit boek was dat ik er toch verstandiger aan zou doen om me verder toe te leggen op LATEX, want dat *format* had meer hulpstukken ingebouwd voor het schrijven van kant en klare documenten. Uiteindelijk heb ik de bijbehorende boeken ook maar gekocht (en er geen spijt van gekregen).

Als informaticus sprak LATEX me enorm aan. Wat kon er nou leuker zijn dan een document schrijven als een computerprogramma? De resultaten waren oogverblindend mooi, hier kon de tekstverwerker thuis niet aan tippen. Wat me er uiteraard toe bracht om thuis ook met TEX aan de slag te willen. Aangezien de thuiscomputer geen internet had, zat er niets anders op dan de MikTeX installatie te downloaden op de universiteit en deze op paar dozijn 3.5" floppy's te transporteren naar huis. (Later zou ik hetzelfde doen met mijn eerste Linux distributie, maar dan met een paar honderd floppy's. De thuis-PC werd dual-boot en ik zou voor altijd afscheid nemen van MS-DOS en MS-Windows.) Dit luidde een nieuwe fase in van experimenteren met LATEX.

Beheersing van dit obscure systeem gaf me een machtig gevoel. De beheersing en controle over elk aspect van de drukkunst (maar dan in digitale vorm) wakkerde bij mij het gevoel aan in de voetsporen van Gütenberg te zijn getreden. De kunst van goede typografie, zo heb ik later geleerd, is dat het eigenlijk helemaal niet mag opvallen. Ik was toen nog niet zo wijs, maar ik begon al wel een zintuig te ontwikkelen voor slechte typografie. Dit zintuig is zowel een zegen als een vloek, want wie het eenmaal heeft ziet het overal waar hij kijkt; zolang mensen met Word en Powerpoint blijven werken zal daar geen verandering in komen.

Een van de toepassingen had te maken met het vak calculus. Hoewel gezegend met een redelijke wiskundeknobbel heb ik een aantal vaardigheden zoals integraalrekenen op school danig verwaarloosd, en daar in mijn studie flink spijt van gekregen. Ik moest serieus aan de bak voor dat vak, en na een paar mislukte pogingen besefte ik dat het maken van het huiswerk en de daarmee gepaard gaande oefening onontbeerlijk was. Dat het huiswerk echter stomvervelend was maakte het allemaal niet leuker. Wat het natuurlijk wel leuker maakte was om al het huiswerk uit te werken in LATEX. De docent was er in ieder geval van onder de indruk.

Het echte werk begon bij mijn schaatsclub. De secretaris van de club was een echte duizendpoot, die naast al het geregel ook het mededelingenblad verzorgde. Dit was een maandelijks boekwerkje vol met wedstrijduitslagen, overzichten en dergelijke, met veel zorg en liefde bijeengeknipt en geplakt en gekopieerd in honderdvoud op de kopieermachine op zijn werk. Het schrijfwerk leek afkomstig van een ouderwetse schrijfmachine. Hier lag een taak voor iemand met typografische en redactionele vaardigheden, die bovendien ingewijd was in een geheimzinnig systeem voor het produceren van ongeëvenaarde kwaliteit drukwerk. Ik bood aan het samenstellen van het clubblad voor mijn rekening te nemen en nam me voor dit naar een hoger niveau te tillen.

Dit was voor mij een zeer leerzame periode. De magie van LATEX die ervoor leek te zorgen dat alles wat erin ging er als een prachtig document uitkwam bleek wel te werken voor wetenschappelijke artikelen en afstudeerscripties, maar minder voor iets wat eruit moest komen te zien als een tijdschrift of krant. Als ik toentertijd had kunnen kennismaken met ConTEXt dan had ik waarschijnlijk heel andere keuzes kunnen maken en was het allemaal iets anders gelopen. Ook hier geldt de algemene regel dat een redelijke bekwaamheid met één hulpmiddel ertoe leidt dat dat hulpmiddel voor bijna alle passende—en niet passende—klussen wordt ingezet.

Er waren veel uitdagingen op typografisch gebied, daarover later meer, maar er waren ook andere problemen. Omdat dit het tijdperk was waarin de *personal computer* tot ieders leven was doorgedrongen verlangde ik van iedereen die materiaal voor het clubblad aanleverde om dit zoveel mogelijk in elektronische vorm te doen. In de begintijd was e-mail nog niet helemaal gemeengoed dus werd er veel uitgewisseld via het *floppynet*. Dit bracht mij de eerste echte ervaring op het gebied van redactioneel werk. Het blijkt dat niet iedereen een volmaakte beheersing heeft van het geschreven woord en het correcte gebruik van leestekens is een zeldzaamheid. Natuurlijk maakt iedereen spelfouten en ik vond het niet meer dan vanzelfsprekend dat een redacteur die verbetert, maar omdat het clubblad wel een zekere provinciale uitstraling mocht hebben liet ik de meeste stijlfouten intact, tenzij het begrip van de zin er teveel onder te lijden had. Ook corrigeerde ik de plaatsing van leestekens zodat het afbreken van de regels op de juiste plaats zou gebeuren. Het betekende dat ik minstens één keer door alle tekst moest die werd aangeleverd, maar ik hoefde het tenminste niet over te tikken. Het recht-toe-recht-aan schrijfwerk van verhaaltjes van allerlei was het minste werk.

Ik moet de kanttekening plaatsen dat ik veel stukjes kreeg aangeleverd die geschreven waren in Microsoft Word. Door de jaren heen heb ik steeds meer een hartgrondige hekel ontwikkeld aan dit verfoeide product, dit gedrocht van een stuk schrijfmachineautomatisering, helaas al even onontkoombaar als onbruikbaar. In de beginjaren heb ik geprobeerd mensen op betere gedachten te brengen en geduldig uit te leggen dat het programma een mogelijkheid biedt om platte tekst te produceren, maar later zag ik in hoe zinloos mijn pogingen waren. Het was niet louter donquichotterie; zeker in de beginjaren had ik werkelijk problemen om deze documenten in te lezen omdat er onder Linux geen software voorhanden was die dit zonder meer kon. Er waren wel verschillende programma's die pretendeerden het te kunnen maar het succes was wisselend, om het maar mild uit te drukken.

Het lastigste was het verwerken van informatie in tabelvorm. Hoe krijg je tekstelementen netjes onder elkaar? We kunnen een aantal basismethoden onderscheiden die door verschillende categorieën mensen worden gebruik. Mensen uit de eerste categorie hebben ooit met een mechanische schrijfmachine leren werken. Het vinden van de juiste horizontale positie is een kwestie van het voortdurend gebruik van de spatiebalk. In een tekstverwerker met een proportioneel lettertype kun je vervolgens je lol op. Importeer de tekst in een zinnige tekstverwerker en niets is meer wat het ooit was. In de tweede categorie heeft men ontdekt dat er zoiets is als een tabstop, maar nooit beseft dat je de breedte van de tabs zelf kunt instellen. De tekst eindigt dan daar waar toevalligerwijze de volgende tabstop zit, en met de variabele inhoud in de kolommen heeft men soms aan één tab genoeg, dan weer twee of drie. In de derde categorie treffen we dan het hogereschoolwerk aan waar men werkelijk een tabel heeft gebruikt, hoewel hier nog twee variaties mogelijk zijn: sommigen hanteren voor het noteren van tijden met minuten, seconden en honderdsten van seconden verschillende kolommen, anderen slechts één.

De omzetting naar iets wat LaTeX lust houdt in dat alle kolommen gescheiden dienen te worden door ampersands (&) en elke regel eindigt met een dubbele backslash (\\). Ik heb verwoede pogingen gedaan de bovenstaande categorieën op een soort halfautomatische manier om te zetten, maar er resteerde me altijd heel veel handmatig gecorrigeer.

Ik heb ooit een keer een oude prijsopgave gezien voor professioneel drukwerk. Geheel niet tot mijn verbazing was de prijs voor materiaal in tabelvorm het hoogst.

Hoewel dit allemaal enorm ergerniswekkend lijkt had ik er wel plezier in. Ik schets weliswaar het beeld een zure muggenzifter te zijn, maar ik was altijd heel blij als ik kopij kreeg aangeleverd. Als ik me er al aan ergerde om het werk te doen was dat vooral omdat ik zelf te laat begonnen was met het zetwerk, wat daardoor niet zelden tot in de kleine uurtjes duurde.

Terug naar de typografie. Bij de allereerste poging om het blad te zetten worstelde ik met de opmaak van de tabellen met schaatsuitslagen. De standaard tabelvorm zag er achteraf gezien niet zo heel mooi uit. Ik was scheutig met het gebruik van lijnen tussen de kolommen en ook veel lijnen in de rijen na elk kopje. Ik zag wel dat het niet goed was maar ik wist in het begin geen raad. Later hanteerde ik deze richtlijnen:

- ☐ gebruik in een lange tabel een klein beetje meer wit na elke vijfde regel
- ☐ gebruik een vulmiddel in de lege cellen van een schaars gevulde tabel, bijvoorbeeld een em-streep (—).

De belangrijke les is dat onze ogen behoefte hebben aan enig houvast als ze moeten zoeken in een tabel; een klein beetje witruimte is daarvoor minstens zo effectief als een lijn, en minder storend.

Uiteindelijk zou ik alle lijnen uit mijn opmaak elimineren.

Het gebruik van lettertypes was natuurlijk een belangrijk punt. Ik vond Computer Modern geen geschikt type voor het clubblad, en veel keuze had ik niet (als zuinige Hollander wilde ik er ook geen geld aan uitgeven) dus werd het Palatino. Ik maakte de broodtekst wat kleiner en de regelafstand iets groter voor meer lucht in de tekst.

De standaard bladspiegel werd een indeling in drie kolommen op A4 formaat. Dat laatste was uiteraard ingegeven door de simpele beperking dat het enigszins goedkoop gereproduceerd moest kunnen worden. Hoewel A4 erg standaard is was het niet een erg goede match voor het soort materiaal. Veel tabellen en uitslagen waren net iets breder dan de helft van de pagina, waardoor de bladvulling ongunstig werd. In de eindfase was het altijd de kunst om goed met de pagina's uit te komen en ik heb heel wat keren zitten zuchten omdat er net een paar regels overliepen naar een nieuwe pagina. Geschuif met marges, gedraai aan knoppen van lettergrootte en witruimte volgden tot ik eindelijk mijn zin had.

Terugkijkend naar wat vroege pogingen valt me op dat tekst in meerdere kolommen niet verticaal gelijkloopt. Dit is te wijten aan de ingebouwde flexibiliteit in witruimte tussen de regels die bedoeld is om het materiaal mooi passend op een pagina te krijgen. Zeker de verticale ruimte bij opsommingen is veel te ruim. Hier restte niets anders dan rigoureus het standaard stijlblad aan te passen en alle variabele witruimte te elimineren. Dit leidde weliswaar tot klachten van TeX over *underfull vboxen* maar het oogde beter.

Het typische inspringen van de eerste regel van

een paragraaf werkte ook niet in dit medium; dat is trouwens toch iets wat je heden ten dage niet meer ziet op websites. De truc hier was om het wit aan het begin van de paragraaf te vervangen voor een portie wit aan het *einde* van de paragraaf, zodat er een herkenbare overgang overblijft.

Het lettertype voor tabellen is mettertijd nog wat kleiner geworden zodat deze meestal niet meer dan tweederde van een pagina besloegen; de resterende ruimte kon mooi gebruikt worden voor een column.

Een interessante aanpassing die veel ruimte won was het colofon. Hierin staan ieder blad weer de contactgegevens van diverse functionarissen van de club. Eerst besloeg dit een flinke portie van de eerste pagina, mede doordat dit in tabelvorm gezet werd. Ik verhuisde dit naar een smalle kolom aan de linkerzijde, onder de inhoudsopgave in Helvetica Narrow in één stroom lopende tekst. Tot mijn verrassing werd het geen zoekplaatje; iedere functionaris werd voorzien van een duidelijke stip en een dikgedrukt eerste woord, zodat het oog makkelijk zijn weg erin kan vinden.

Ik heb deze methode later nog eens toegepast op een heel ander format. In de tijd dat CD-ROMs nog gangbaar waren heb ik veel van mijn muziekcollectie omgezet naar MP3 formaat en op CD gebrand. Al doende passen er wel tien albums op één CD, en ik schreef een paar macro's om de nummers in kolommen te zetten op het formaat van een CD-hoesje. Als experiment heb ik al deze informatie ook eens als één enkele paragraaf gezet, wat wonderbaarlijk goed werkte. Hierover schreef ik een artikel in de MAPS in 2005. (Klopt dat? Ja, MAPS 33, 2005, paginas 4-13 – red.)

Ik heb het opmaken van het clubblad tien jaar volgehouden en het toen aan een opvolger overgedragen. Maar de opmaak is nooit meer met TEX gemaakt.

Intussen was ik klaar met studeren en begonnen aan een carrière als software engineer. Ook hier was volop gelegenheid tot het schrijven van artikelen en technische rapporten met LATEX. Echt veel bijzonders valt er echter niet over te vermelden, behalve één project wat we deden voor de makelaardij. Onze opdrachtgever wilde een softwarepakket voor object- en relatiebeheer wat makelaars op hun eigen kantoor moesten kunnen gebruiken. Hier was behoefte om met een paar simpele invulvelden een kant-en-klare folder te kunnen drukken van een huis. Ik zag wel mogelijkheden om dit te doen met een enigszins uitgeklede Windows distributie van TEX. Hier was vooral de uitdaging om speciale tekens te interpreteren als gewone karakters, want het zou teveel gevraagd zijn om van makelaars te verlangen dat ze LATEX commando's zouden moeten gebruiken. Hoewel het systeem redelijk voldeed is het project als geheel nooit uit de prototypefase gekomen.

Ik moet bekennen dat ik me de laatste jaren niet meer specifiek toeleg op TEX; ik schrijf nog altijd stukjes, maar de bron is nu org-mode in Emacs wat een prima basis is om te exporteren naar LATEX en met `pdflatex` te verwerken tot een PDF document. Als ik een presentatie moet geven doe ik dat met `beamer.cls` of (tegenwoordig vaker) met `reveal.js`. TEX is voor mij steeds meer een motor die ergens diep vanbinnen aan het werk is, maar ik hoef er niet meer zo nodig aan te sleutelen.

Het concept van de motor brengt me eigenlijk op een idee; waarom zou deze motor niet de drijvende kracht kunnen zijn van veel meer typografie in ons leven? Het centrale algoritme van Knuth is het balanceren van paragrafen van de eerste letter tot de laatste. Waar andere software probeert regel voor regel af te handelen, overziet TEX de hele paragraaf in één keer en zoekt de optimale balans tussen alle regels tegelijk. Het resultaat is een veel rustiger beeld zonder opvallende gaten tussen de woorden. Waarom kunnen we dit niet toepassen op veel meer software? Veel open-source pakketten zouden hier direct bij gebaat zijn. Tekstverwerkers als LibreOffice, browsers als Chrome en FireFox, e-readers voor e-books, de mogelijkheden zijn er volop. De uitdaging is natuurlijk wel om het aloude programma, gemaakt om een tekst van begin tot eind te behandelen, te veranderen in een soort plug-in voor andere software.

Ik heb het gevoel dat er nog heel veel in het vat zit voor ons geliefde programma. Het is in deze snelle wereld van *apps*, *cloud* en *github* wel een beetje een deftige oude dame geworden die niet helemaal met haar tijd is meegegaan, maar de basis is solide en de gemeenschap is nog altijd hecht. Zit er nog een verjongingskuur in? Zijn we in staat om een nieuwe generatie enthousiast te maken voor de briljante nukken van dit systeem? Ik weet het niet. Maar ik wet wel dat ik nog niet ben uitgeleerd. Mocht zich een typografische uitdaging aankondigen, dan zal ik die handschoen zeker weer oppakken.

Dennis van Dok
2018-07-11

Verslag van de bezorger

Verreweg de meeste MAPSen zijn ingeleverd bij de lokale boekhandel tevens postagentschap maar leden in Amsterdam en Amstelveen die geen postbus hebben opgegeven zijn bijna allen persoonlijk bediend om zodoende voor de vereniging wat verzendkosten te besparen maar ook om een indruk te krijgen van de spreiding van onze leden in deze omgeving.

De website van https://www.routexl.nl/ helpt mee om een route te verzinnen zodat men niet onnodige afstanden af hoeft te leggen.

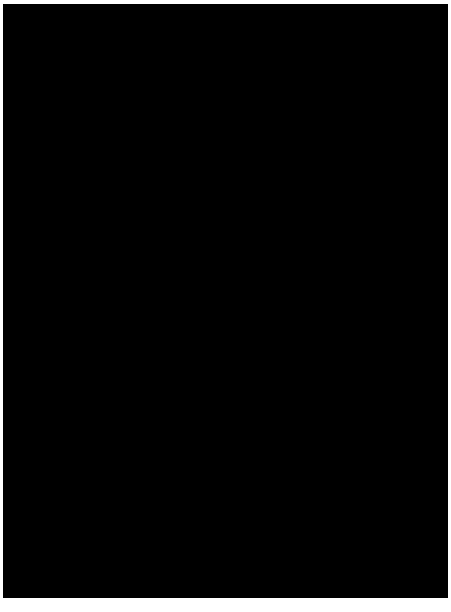
Een export-functie naar mijn Garmin GPS apparaatjes zit er zo te zien niet bij dus ik heb de voorgestelde volgorde van adressen overgenomen in een routeplanner op mijn kleine Garmin GPSMAP 66st.

Wanneer ik de TOPO-Active kaart heb geladen, die geschikt is voor fietsverkeer, blijken sommige adressen in de Bijlmer onvindbaar dus ik heb de meer algemene City Navigator kaart geladen. Er is ook net een nieuwe Open Fietsmap kaart uitgekomen maar zo kort voor vertrek lijkt het geen goed idee nieuwigheden uit te proberen.

Het eerste bezorgadres is vlakbij huis maar de GPS heeft niet door dat ik er ben aangekomen en schakelt niet door naar het eerstvolgende adres dus ik heb daarna maar gewoon telkens het volgende adres van de enveloppen als doel gekozen.

Bij het waterleidingbedrijf tegenover het VUmc kijk ik altijd even naar het standbeeld van een vrouw die er immer tevreden bij staat. Op zomerse dagen plast ze uit beide broekspijpen, een gutsende vrolijke stroom die haar enorm oplucht. Dat straalt ze uit. Waterleiding is een nutsbedrijf maar ze hebben ook die andere kant, dat wordt weleens vergeten.

Ergens in Amstelveen kwam ik op een kruispunt een in het wegdek geplette fietsbeldop tegen en die kon ik goed fotograferen voor mijn verzameling van wegdek-fietsbeldoppen (https://woordenpraat.blogspot.com/2018/05/beldopspotting.html). De meeste mensen letten daar niet op, maar als je er eenmaal eentje in de smiezen hebt, kom je ze vaker tegen en het is zeer de moeite waard deze goed te bekijken aangezien je daarbij toch weer op allerlei gedachten komt.



Ook in Amstelveen kwam de GPS op zeker moment met de opdracht mij naar het "westelijk halfrond" te begeven. Geen idee of dat geestig was bedoeld van de mensen van Garmin.

Ook in Amstelveen, pal om de hoek bij een prominent NTG-lid, is er een deurbel defect op nummer 13. Dat zie je vaker. Terwijl het heel gemakkelijk is tijdelijk een draadloze deurbelknop te monteren kiest men er toch vaak voor om een geïmproviseerd bordje op te hangen en mede te delen dat de bel buiten werking is, terwijl men met dezelfde moeite een extra bel had kunnen ophangen of voor reparatie zorgdragen.

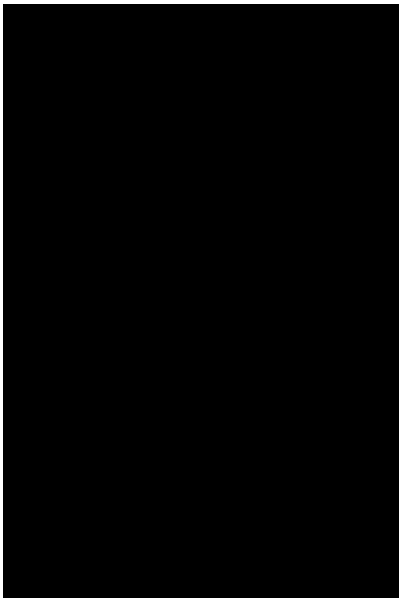
In de Kerkstraat woont een NTG lid in een historisch pand, waarschijnlijk een Rijksmonument uit de tijd dat publicaties als de MAPS op aanmerkelijk veel kleiner formaat verschenen en de enveloppe past daar ook dit jaar niet zomaar door de brievenbus dus ik heb de MAPS uit de enveloppe gehaald, dubbelgevouwen, naar binnen geduwd en daarna ook de nu lege enveloppe naar binnen geschoven.

Een laatste opmerkelijkheid is dat de Allard Piersonstraat in Amsterdam precies ophoudt waar het huisnummer van ons lid zou moeten wonen, en bij nader inzien blijkt dit nummer in een gelijknamige straat in Amstelveen te zijn. Ik zou eens moeten zien of die straat in Amstelveen een andere straat is met dezelfde naam of dat de straat uit Amsterdam daar precies verder gaat waar die in Amsterdam was gebleven, wat eigenlijk het meeste voor de hand ligt natuurlijk.

Nog enkele gegevens voor wie overweegt deze route ook eens te rijden: 52km te doen in 3 uur en 33 minuten met snelheden tot 30 km/u (slechts 15 km/u gemiddeld) waarbij rekening moet worden gehouden met een hartritme tussen de 110 en 140 slagen per minuut dus enkele bananen zijn wel prettig om in de fietstas te hebben en ook voor fris water moet worden gezorgd.

Frans Goddijn
16 april 2019
Video: https://www.relive.cc/view/2291765840

NTG oudheden



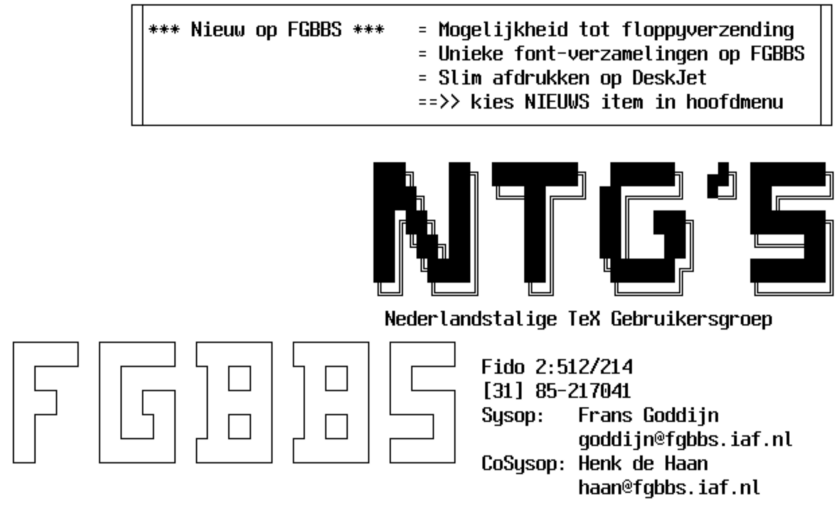
Van Willi Egger, die jarenlang NTG secretaris is geweest, kreeg ik materiaal van de NTG ‚postkamer' dat waarschijnlijk ooit door Gerard van Nes is aangeschaft. Een mooi vintage weegschaaltje, een „Port betaald" stempel en een paar rollen plakkers voor post waar haast bij was. Het stempelkussen doet het nog, al liggen er wat kristallen van inkt aan de randen van het kussen.

Naar aanleiding van deze vondst is onze penningmeester Ferdy in de papieren gedoken en we bleken nog steeds een contract te hebben met de posterijen voor het verzenden van „Port betaald" stukken. Omdat we daar al lang geen gebruik van maken is dit contract nu opgezegd en er bleek zelfs nog geld van de NTG in depot te rusten.

Frans Goddijn
15 mei 2019

The sound of data

Electronic TEX promotion in the nineties



Today's wifi, cellular and cable information moves in silence and the speed has become a non-issue since high resolution Netflix video is streamed while we receive email in the background and laptops maintain backups in the cloud.

Elderly persons like me remember vividly the sound data transmissions used to make. In the eighties, a journalist with a Tandy100 word processor used a set of acoustic couplers on the horn of a phone and the screeching of modem negotiations was very audible.

In the early nineties I had a tower pc and one could hear the hard disc's tock-tock while information was read and written, a big rumbling fan in the back trying to cool the tower and a nervous mosquito-like buzz from the tiny fan clamped on the CPU. The backup tape streamer produced a loud and urgent whining sound as if a dentist was drilling into a molar. And the CD reader would occasionally start spinning with a soft whirr, or suddenly break to a hickup halt to eject the CD.

When the Internet was almost exclusively available for universities, an alternative network had been built for the others out there and this FIDO net worked so very well that its top maintainers assumed the internet would just be a temporary thing, way too complicated for ordinary people.

My PC was a BBS (Bulletin board System) node in that Fido network and callers who logged in by calling its dedicated phone number with their modem could exchange mail worldwide with anyone listed in the huge 'phonebook' that was updated daily. International mail could be sent 'on the cheap' using local tariff communication, as the message was handed over to a local hub sharing it with another hub higher up in the tree, to another continent's hub, and onwards to the BBS closest to the recipient. Alternatively, the message would be handed to my system with the 'crash bit' set and then, if the user had permissions, the BBS would directly call the destination and the message would be delivered in minutes. My users could pick up and deliver mail automatically if they had a compatible 'point' system or they could dial in and use an ASCII menu screen to browse around the options.

In the early nineties new TEX users without Internet access needed to find someone with the right stack of 15 diskettes and my Fidonet BBS system, called FGBBS 2:512/214 (phone number +31 26 3217041) provided an alternative. Visitors could still request floppies to be sent but one could also select parts of the file collection and download these.

Henk de Haan, a young man at Delft University preparing his doctorate in nuclear science about muon catalized nuclear fusion, did most of the complex work assembling the many software packages that were required to make the modular FGBBS environment work

smoothly. Soon, a 'waffle' was made to merge internet email into the Fidonet message base. We were not the only ones offering such services and Henk sometimes branstormed with others like the small team that founded XS4ALL, now one of the major internet poviders in The Netherlands.

Jason Fesler, an American programmer who is currently with Apple, was pioneering in this field and we were among the first users of his GIGO program which could convert UUCP data to our format and message-base. He also wrote WME, a Windowed Modem Environment which gave our FGBBS the screen presentation and 'look and feel' much like MS Windows would have later on.

The tower PC would be buzzing and whirring 24/7 in our little house in Arnhem and my two daughters in their adjoining bedroom got used to the sounds the modem would make whenever a caller made a connection at all times of night. Two rings, followed by a high pitched scream of the modem, a lower step pitch, another lower pitch and then silence as the modems agreed on speed and protocol.

All that hardware was frightfully expensive compared to what we use today. A good modem cost 1500 guilders (today's € 1200) and the backup tape streamer with 4 tapes of 400 MB each was 621 guilders (today's € 500). A CD drive was subsidized by NTG.

Henk de Haan provided a steady stream of file updates using his university account and he actively expanded the offering of messages so callers also had access to TeX related mailing lists like TEX-NL and newsgroups like comp.text.tex.

To save telephone costs of 'online' time, standalone ulitities like Silver Express were offered to allow callers to quickly get all they wanted and then unpack news offline.

The latest updates of TEX modules like Babel and LATEX2e were there to be picked up and also a 612 kilobyte (!) "TinyTeX" package could be fetched so one would have functional TeX setup so small that it's hard to imagine today.

In a series of updates in NTG's MAPS publication the sysops reported about new developments, also listing the FGBBS file contents. In 1994, the contents of the 4allTEX CD was added and the number of (zipped) file collections was a little over 1500 and 170 MB in size which was considered a lot of files.

In today's perspective the number of users was not very high. In 6 months of 1994, almost 40 individuals contacted FGBBS nearly 1400 times to pick up almost 3500 files.

The phone company, anticipating more intense growth, had by then installed a box with 16 telephone lines so FGBBS could, if needed, be greatly expanded.

In 1995, 72% of callers used 14k4 speed modems, 22% still used 2400 baud and just 1.5% had the state of the art 28k8 speed connection.

In 1996 FGBBS had on average 3 callers per day, a year later a call would come a little less than once a day.

In 1998 the system upgraded to the high speeds of ISDN and users could get internet email facilities for free. Once every two days a caller would stop by to use the facilities.

In 1999 the system was finally closed down and once again silence was to be heard in the little family house. Deleting the entire file library hardly made a sound. Just a soft crackle of the hard disk.

Sources:

http://stuff.gigo.com/resume.html
http://software.bbsdocumentary.com/IBM/DOS/WME/
http://www.ntg.nl/maps/index.html

Frans Goddijn

De duizend-dagen-klok

elke dag telt

Tellen & Doorgaan

Verjaardagen. Wie wordt er niet oud mee.

Maar de beleving van jaren verandert met diezelfde jaren. Tellen we bij de geboorte van een kind de leeftijd in dagen, al snel worden het weken en maanden. En omdat het kind door blijft groeien gaan de maanden over in jaren.

Maar dat hoeft natuurlijk niet. Er is weinig voor nodig om de tijd wat overzichtelijker te beleven. Een leeftijd in dagen. Immers, elke dag telt.

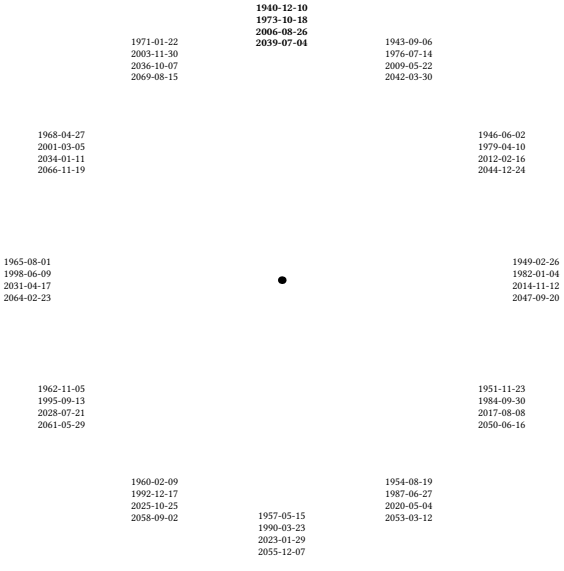
De ontwikkeling van ConTEXt begon naar zeggen ergens in 1990, laten we zeggen op 1 januari precies. Op 1 januari 2020 is dat precies dertig jaar. Dat is mooi, maar nog mooier is 11000 dagen (op 2020/02/13). En wat te denken van de palindroom-dag 11011 (op 2020-02-24) of de all-in-one 11111 (op 2020-06-03)?

Kortom het is altijd handig om het verloop van de tijd op zowel micro als macro nivo een beetje in de gaten te houden.

Duizend Dagen Klok

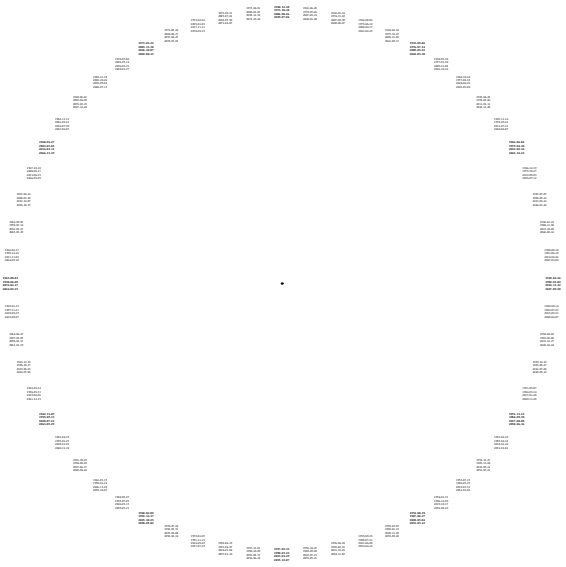
De *duizend-dagen-klok* geeft dat heldere overzicht. Want naast de uren en minuten binnen een dag, staan bij elk uur ook de datum van het duizendtal in dagen: 1000, 2000, 3000, ..., 12000.

En net als dat de klok telkens twaalf uren van een hele dag van vierentwintig toont, zo telt de duizend-dagen-klok door in groepen van 12k: 0 – 12 – 24 – 36 – 48.



200 dagen / minuut

Voor de ongeduldigen bestaat er natuurlijk ook nog de *200 dagen per minuut* variant. Voor extra feestelijke dagen.



De Implementatie

De basis voor de klok is afgeleid van een label-voorbeeld in de metafun manual. Hans Hagen wees op de mogelijkheid hoe ook meerdere regels als een enkel label te kunnen gebruiken door die regels in een `framed` box the plaatsen. De datums die nodig zijn om de klok voor verschillende individuen of events te vullen genereerde ik met een paar regels python code. Dat script genereerde dan ook meteen het ConTEXt bestand met de overige metafun code regels. Python is voor mijn werk de dagelijkse kost. Het maakt het voor mij eenvoudig om invoer bestanden voor uiteenlopende applicaties te genereren. Nadat de klok werkte, bleef het toch knagen. Hoe lastig zou het zijn om ook die paar regels python naar lua om te zetten? Dan kan de klok immers direct met een enkel ConTEXt bestand worden gegenereerd. Probleem, ik had nog nooit lua code geschreven.

Het is even wennen aan de verschillende perspectieven van de twee talen. Gelukkig biedt de *ConTEXt garden* een rijke bron aan voorbeelden om te leren begrijpen hoe het werkt. En zie hier het resultaat van de implementatie van de `duizenddagenklok` opdracht in twee smaken:

```
\duizenddagenklok[year=1940, month=12, day=10, unit=hour]
\duizenddagenklok[year=1940, month=12, day=10, unit=minute]
```

De code:

```
\startluacode
userdata = userdata or {}
function userdata.klok(keyvals)
  local kv = utilities.parsers.settings_to_hash(keyvals)
  local steps    = 12
  local diameter = "83mm"
  local bold     = 12
```

```
  local day      = 24 * 3600
  local bf
  local birthday = {
    year  = kv.year,
    month = kv.month,
    day   = kv.day,
  }
  if kv.unit == "minute" then
      steps    = 60
      diameter = "300mm"
      bold     = 5
  end
  local t    = os.time(birthday)
  local step = 12000 / steps

  function datum(i)
    return os.date("%Y-%m-%d", t + i * step * day)
  end

  context.startMPcode()
  for i = 0, steps-1 do
    if (i % bold) == 0 then
      bf = "\\bf "
    else
      bf = ""
    end
    local hoek = 450 - (i * (360 / steps))
    local blok = table.concat({
    datum(i), datum(i+steps), datum(i+steps*2), datum(i+steps*3)
    }, "\\\\")
    context.metafun(
      "draw textext(\"" ..
      "\\framed[align=flushleft,frame=off]{" .. bf .. blok .. "}\") " ..
      "shifted (dir(" .. hoek .. ") * " .. diameter .. ");"
    )
  end
  context.metafun("pickup pencircle scaled 0.25cm; drawdot origin;")
  context.stopMPcode()
end
\stopluacode

\def\duizenddagenklok[#1]{\ctxlua{userdata.klok('#1')}}
```

Ook Handig

Als we het begin van het project op 1 januari 1990 stellen, dan zijn dit naast de 200-dagen ook de eerstkomende ConTEXt palindroom-dagen momenten:

11000: 2020-02-13	11400: 2021-03-19	11800: 2022-04-23
11011: 2020-02-24	11411: 2021-03-30	11811: 2022-05-04
11111: 2020-06-03	11511: 2021-07-08	11911: 2022-08-12
11200: 2020-08-31	11600: 2021-10-05	12000: 2022-11-09
11211: 2020-09-11	11611: 2021-10-16	12021: 2022-11-30
11311: 2020-12-20	11711: 2022-01-24	12121: 2023-03-10

Floris van Manen
2019-10-16

fancyhdr und scrlayer in trauter Zweisamkeit

Abstract
Auf CTAN gibt es diverse Pakete, mit denen man den Seitenstil eines Dokuments verändern kann. Eines der beliebtesten dürfte dabei fancyhdr sein. Ein anderes, eher grundlegendes Paket ist das KOMA-Script-Paket scrlayer. Dieses Paket geht mit seinem Ebenenmodell weit über das hinaus, was ein Seitenstilpaket üblicherweise leistet. Daher wurde schon bald nach Veröffentlichung von scrlayer der Wunsch geäußert, die Ebenen von scrlayer zusammen mit den Seitenstilen von fancyhdr verwenden zu können. Mit dem experimentellen Paket scrlayer-fancyhdr ist dies nun möglich.

Was ist fancyhdr?

Das Paket fancyhdr stellt im Wesentlichen den neuen Seitenstil fancy bereit. Dieser besteht in Kopf und Fuß aus jeweils einem Element, das linksbündig gesetzt wird, einem Element, das zentriert gesetzt wird, und einem Element, das rechtsbündig gesetzt wird. Im doppelseitigen Satz sind auf linken Seiten das linksbündige und das rechtsbündige Element vertauscht. Diese sechs Elemente des Seitenstils können über die Anweisungen \fancyhead und \fancyfoot oder alternativ mit \lhead, \chead, \rhead, \lfoot, \cfoot und \rfoot mit Inhalt versehen werden. Darüber hinaus, gibt es Befehle, um die Dicke einer Linie unter dem Kopf und über dem Fuß dieses Seitenstils einzustellen.

Zusätzlich gibt es auch noch eine Anweisung \fancypagestyle, über die man weitere Seitenstile definieren kann. Intern handelt es sich allerdings bei all diesen zusätzlichen Stilen ebenfalls um fancy. Bei der Aktivierung der neuen Seitenstile werden lediglich zunächst die bei der Definition mit \fancypagestyle als zweites Argument angegebenen Anweisungen ausgeführt.

Was die meisten Anwender nicht wissen ist, dass fancy intern wiederum auf dem Seitenstil @fancy basiert. Außerdem stellt fancyhdr auch noch einen nicht dokumentierten Seitenstil fancyplain bereit. Dieser aktiviert nicht nur den Stil fancy, sondern ändert zusätzlich den Standard-Seitenstil plain in den ebenfalls internen Seitenstil plain@fancy, der auch auf @fancy basiert. Verwendet man diesen nicht dokumentierten Seitenstil fancyplain, so kann man Unterschiede zwischen den Seitenstilen fancy und plain bei den Einstellungen für fancy über die Anweisung \fancyplain festlegen. In der Voreinstellung verwendet fancyhdr beispielsweise

```
\fancyhead[l]{\fancyplain{}{\slshape\rightmark}}
```

um im linken Element des Kopfes für den plain-Seitenstil (erstes Argument von \fancyplain) keinen Kolumnentitel zu setzen, während für fancy (zweites Argument von \fancyplain) die rechte Marke in schräger Schrift ausgegeben wird.

Anzumerken wäre außerdem noch, dass der Seitenstil @fancy und damit alle von fancyhdr definierten und verwendeten Seitenstile genau wie headings automatische Kolumnentitel aktiviert. Die Verwendung eines mit fancyhdr definierten Seitenstils mit rein manuellen Kolumnentiteln ist daher nur möglich, wenn man in der Definition der Seitenstile direkt den gewünschten Kolumnentitel angibt, statt mit

\leftmark und \rightmark Platzhalter zu setzen und diese im Dokument dann mit \markboth und \markright zu befüllen. Erfahrene Anwender wissen, dass dieses direkte Umdefinieren des Seitenstils in einigen Fällen zu falschen Kolumnentiteln führen kann.

Zusammenfassend kann man also sagen, dass fancyhdr es dem Anwender ermöglicht den Inhalt von Kopf und Fuß der Seite über jeweils drei Elemente einzustellen. Die meisten Anwender schätzen die einfach gehaltene Schnittstelle und kommen mit den dokumentierten Möglichkeiten zur Definition eines Seitenkopfes und Seitenfußes sehr gut aus. Näheres zu den dokumentierten Möglichkeiten des Pakets ist [6] zu entnehmen.

Was ist scrlayer?

Das Paket scrlayer geht weit darüber hinaus nur den Kopf und Fuß einer Seite konfigurierbar zu machen. Es führt dazu ein Ebenenmodell ein, bei dem im Hintergrund und im Vordergrund der Seite eine ganze Reihe an Darstellungsebenen übereinander gestapelt werden können. Ebenen können sogar mehrfach verwendet werden. Seitenstile dienen dabei einerseits als Anker für die Ebenen, andererseits werden Seitenstile durch diese Ebenen selbst definiert. Auf diesem Weg ist es über einen Seitenstil nicht nur möglich, den Inhalt des Kopfes und des Fußes zu bestimmen. Es kann auch Material an beliebigen Stellen einer Seite platziert werden.

Eine neue Ebene wird über die Anweisung \DeclareNewLayer deklariert. Bei dieser Deklaration können über Optionen eine ganze Reihe von Eigenschaften der Ebene bestimmt werden. Der Inhalt der Ebene wird beispielsweise als Wert von contents angegeben. Über weitere Optionen kann die Position dieses Inhalts auf der Seite, die Ausgabe der Ebene im Hinter- oder Vordergrund der Seite und vieles mehr eingestellt werden. Derartige Ebenen können dann über \DeclarePageStyleByLayers zu einem Seitenstil kombiniert werden. Es ist auch möglich, Ebenen nachträglich zu ändern oder zu Seitenstilen hinzuzufügen oder von diesen zu entfernen. Grundvoraussetzung ist, dass der Seitenstil irgendwann per \DeclarePageStyleByLayers definiert wurde.

Das Paket scrlayer erhebt auf der anderen Seite nicht den Anspruch, eine einfache Benutzerschnittstelle zur Bestimmung des Inhalts von Kopf und Fuß der Seite bereit zu stellen. Stattdessen überlässt es diese Aufgabe spezialisierten Schnittstellenpaketen. Mit scrlayer-scrpage existiert in KOMA-Script beispielsweise so ein Paket, das unter anderem ähnlich fancyhdr Seitenstile mit dreigeteiltem Kopf und Fuß bereitstellt. Näheres zu scrlayer-scrpage ist [4] zu entnehmen. Darüber hinaus wurden in »Die TEXnische Komödie« 3/2017 in den Artikeln [1], [2] und [3] weitere Anwendungsmöglichkeiten für scrlayer beispielhaft vorgeführt.

Wie mächtig die Möglichkeiten von scrlayer sind, zeigt das ebenfalls in [4] dokumentierte Schnittstellenpaket scrlayer-notecolumn, das basierend auf scrlayer die Möglichkeit schafft, beliebige Notizspalten auf Seiten zu platzieren, wobei die Notizen auch automatisch über mehrere Seiten umbrochen werden können. Die Pakete scrlayer-scrpage und scrlayer-notecolumn stehen dabei nicht in Konkurrenz zueinander, sondern können auch zusammen verwendet werden.

Zusammenfassend kann man also sagen, dass scrlayer LaTEX mit Hilfe von Seitenstilen sehr grundlegend um die Fähigkeit von Ebenen erweitert. Die Bereitstellung einer darauf basierenden Schnittstelle, beispielsweise zur einfachen Einstellung von Kopf und Fuß einer Seite, wird dagegen spezialisierten Schnittstellenpaketen überlassen. Dokumentiert sind die Möglichkeiten von scrlayer daher im Expertenteil der KOMA-Script-Anleitung, [4].

Kombination von fancyhdr und scrlayer

Bereits während des Vortrags zu scrlayer bei der Jubiläumstagung von DANTE e.V. in Heidelberg, auf dem der Artikel [2] basiert, wurde die Frage geäußert, ob man die weitreichenden Möglichkeiten von scrlayer auch zusammen mit fancyhdr verwenden könne. Damals wurde die Frage aus dem Auditorium heraus damit beantwortet, dass dies nicht notwendig sei, da bereits scrlayer selbst die Definition von Seitenstilen erlaube und dies mit scrlayer-scrpage genauso einfach sei wie mit fancyhdr. Grundsätzlich ist dies richtig. Allerdings gibt es Anwender, die fancyhdr bevorzugen. Damit ist die Frage durchaus berechtigt.

Wie im letzten Abschnitt erwähnt, ist die Verwendung der Ebenen von scrlayer daran gebunden, dass der aktuelle Seitenstil ein mit scrlayer definierter Ebenenseitenstil ist. Dies ist bei den Seitenstilen von fancyhdr nicht der Fall. Damit ist zwar eine parallele Verwendung der beiden Pakete möglich, gleichzeitig auf derselben Seite beispielsweise den Seitenstil fancy und Ebenen von scrlayer zu verwenden, ist dagegen nicht möglich.

Erinnern wir uns jedoch kurz, dass scrlayer als Grundlagenpaket konzipiert ist, auf dem aufbauend Benutzerschnittstellen bereitgestellt werden können. Erinnern wir uns weiter, dass scrlayer-scrpage darauf aufbauend eine Schnittstelle anbietet, die weitgehend kompatibel zu scrpage2 ist, um eben dieses Paket zu ersetzen.

Es drängt sich daher die Frage auf, ob man nicht ebenso eine zu fancyhdr kompatible Schnittstelle bereitstellen kann, unter deren Oberfläche scrlayer arbeitet. Bereits beim Design von scrlayer war ich mir durchaus bewusst, dass ein solcher Wunsch entstehen könnte, und hatte von vornherein eingeplant, dass dieses Paket alles bereitstellt, um eine mit fancyhdr kompatible Schnittstelle bereitzustellen.[1]

Beim Design und der Implementierung sind zwei sehr unterschiedliche Ansätze denkbar. Zum einen könnte man, genau wie bei scrlayer-scrpage, ein komplett neues Paket schreiben, das sämtliche, dokumentierte Anwenderbefehle von fancyhdr auf Grundlage von scrlayer bereitstellt. Dieser Ansatz hätte den Vorteil, dass er sauber aufgebaut ein von der Implementierung von fancyhdr unabhängiges Paket bereitstellen würde. Die Kompatibilität mit scrlayer wäre dabei sehr hoch und man könnte optional leicht Abweichungen von fancyhdr implementieren. Änderungen an fancyhdr würden zwar möglicherweise die Kompatibilität einer solchen Implementierung gefährden, nicht jedoch deren generelle Funktion. Nachteil wäre, dass die Kompatibilität zu fancyhdr nur im Rahmen der aktuell dokumentierten Möglichkeiten gegeben wäre und etwaige Änderungen an fancyhdr gegebenenfalls nachgeführt werden müssten.

Zum anderen könnte man, wie in solchen Fällen häufig üblich, ein sogenanntes *Wrapper*-Paket schreiben. Ein solches Paket würde intern sowohl scrlayer als auch fancyhdr laden und dann nur die notwendigen Anpassungen an den (internen) Makros von fancyhdr vornehmen, um dieses auf die Beine von scrlayer zu stellen. Vorteil dieses Ansatzes wäre, dass er automatisch alle Befehle von fancyhdr bereitstellen würde. Die Kompatibilität auf Befehlsebene wäre also gegeben. Die Kompatibilität in der Funktion würde jedoch sowohl von der Implementierung des neuen Pakets als auch von fancyhdr abhängen. Als Nachteil bestünde also jederzeit die Gefahr, dass bei Änderungen an fancyhdr die Funktion des neuen Pakets nicht mehr gewährleistet wäre.

In Anbetracht der Tatsache, dass fancyhdr ein sehr stabiles Paket ist, tiefgreifende Änderungen also nicht erwartet werden, habe ich mich Mitte 2018 entschlossen, den zweiten Ansatz zu verfolgen. Nachdem ich mir die Implementierung näher angeschaut habe und dabei auf die im ersten Abschnitt dieses Artikels erwähnte Tatsache gestoßen bin, dass alle Seitenstile von fancyhdr letztlich auf demselben, internen Seitenstil @fancy basieren und alle Anweisungen von fancyhdr darauf abzielen, genau diesen Seitenstil zu konfigurieren, drängte sich die Idee auf, dass nur eben dieser durch einen auf scrlayer basierenden Ebenenseitenstil ersetzt werden

müsste, um die Funktion von fancyhdr zu gewährleisten und gleichzeitig die Ebenen von scrlayer verwenden zu können.

Da fancyhdr sehr deutlich zwischen dem Kopf und dem Fuß, jedoch nicht sehr stark zwischen den drei Elementen jeweils von Kopf und Fuß unterscheidet, habe ich beschlossen, für Kopf und Fuß eigene Ebenen zu definieren. Darüber hinaus unterscheidet fancyhdr zwischen linken und rechten Seiten, wie das auch schon der LaTeX-Kern tut. Daraus ergeben sich dann insgesamt vier Ebenen:

```
\DeclareNewLayer[%
  background,oddpage,
  head,
  contents={\hb@xt@ \layerwidth{%
      \f@nch@head\f@nch@Oolh\f@nch@olh
      \f@nch@och\f@nch@orh\f@nch@Oorh}}
]{fancy.head.odd}
\DeclareNewLayer[%
  background,evenpage,
  head,
  contents={\hb@xt@ \layerwidth{%
      \f@nch@head\f@nch@Oelh\f@nch@elh
      \f@nch@ech\f@nch@erh\f@nch@Oerh}}
]{fancy.head.even}
\DeclareNewLayer[%
  foreground,oddpage,
  foot,
  contents={\hb@xt@ \layerwidth{%
      \f@nch@foot\f@nch@Oolf\f@nch@olf
      \f@nch@ocf\f@nch@orf\f@nch@Oorf}}
]{fancy.foot.odd}
\DeclareNewLayer[%
  foreground,evenpage,
  foot,
  contents={\hb@xt@ \layerwidth{%
      \f@nch@foot\f@nch@Oelf\f@nch@elf
      \f@nch@ecf\f@nch@erf\f@nch@Oerf}}
]{fancy.foot.even}
```

Die verwendeten internen Befehle, die mit \f@nch@ beginnen, sind dabei in fancyhdr definiert. Bis auf das zusätzlich eingefügte \hb@xt@ \layerwidth, das einer \makebox mit linksbündigem Inhalt in Ebenenbreite entspricht, kopiert die Definition der Inhalte der Ebenen genau die Definition von Kopf und Fuß von ungeraden und geraden Seiten im Seitenstil @fancy.

Basierend auf den vier Ebenen wird dann der Seitenstil \@fancy mit

```
\DeclarePageStyleByLayers[
  onselect={\def\@mkboth{\protect\markboth}},
]{@fancy}{%
  fancy.head.odd,fancy.head.even,
  fancy.foot.odd,fancy.foot.even
}
```

umdefiniert. Da alle Seitenstile von fancyhdr letztlich diesen einen Seitenstil verwenden, sind bereits damit alle von fancyhdr definierten Seitenstile auf die Verwendung von scrlayer umgestellt. Für den Seitenstil plain gilt dies jedoch nur, wenn man entweder mit

```
\pagestyle{fancyplain}
```

den in der fancyhdr-Anleitung nicht dokumentierten Seitenstil verwendet, der

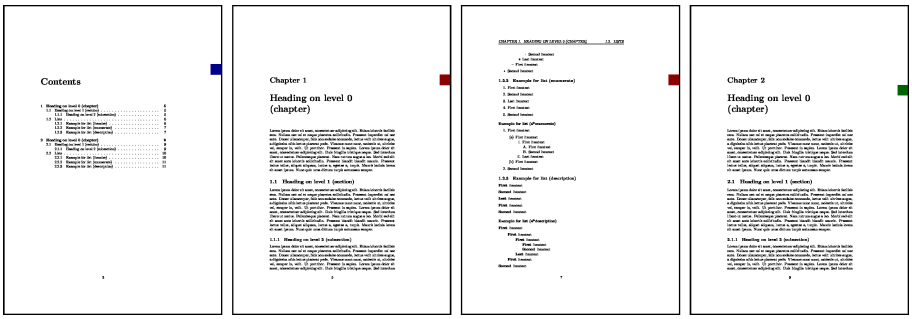


Abbildung 1. Kombination von farbigen Kapitelmarken mit `scrlayer` mit einem Seitenstil von `fancyhdr`

einerseits ebenfalls den Seitenstil `fancy` aktiviert und andererseits den Seitenstil `plain` umdefiniert, oder aber den Seitenstil `plain` selbst mit `\fancypagestyle` oder `\DeclarePageStyleByLayers` umdefiniert.

Wozu das Ganze?

Mit dem neuen Paket `scrlayer-fancyhdr` ist es nun möglich, Ebenen wie in [2] mit einem Seitenstil von `fancyhdr` zu kombinieren. Man erhält dann beispielsweise Seiten wie in Abbildung 1 gezeigt. Gegenüber dem Originalquelltext aus [2] wurde lediglich `scrlayer-fancyhdr` anstelle von `scrlayer-scrpage` geladen und ein `\pagestyle{fancyplain}` eingefügt.

Aber auch unmittelbare Erweiterungen der Fähigkeiten von `fancyhdr` sind so möglich. Beispielsweise könnte man einfach den Seitenkopf farbig hinterlegen:

```
\documentclass[a4paper]{article}
\usepackage{scrlayer-fancyhdr}
\usepackage[svgnames]{xcolor}
\usepackage{blindtext}
\pagestyle{fancy}
\DeclareNewLayer[%
  background,head,addheight=.5ex,
  contents={\color{Coral}\rule{\layerwidth}{\layerheight}}
]{Background}
\AddLayersAtBeginOfPageStyle{@fancy}{Background}
\begin{document}
\tableofcontents
\blinddocument
```

Abbildung 2 zeigt das Ergebnis.

Beschränkungen

Das aktuelle verfügbare Paket ist noch nicht für den produktiven Einsatz gedacht. Es dient lediglich ersten Experimenten, um weitere, mögliche Probleme zu ermitteln und deren Relevanz beurteilen zu können.

Wer sich die Deklaration des Ebenseitenstils `@fancy` genauer angeschaut hat, dem wird vielleicht aufgefallen sein, dass dabei `\@mkboth` umdefiniert wird. Das führt dazu, dass dieser Seitenstil automatische Kolumnentitel an den `scrlayer`-Befehlen `\automark` und `\manualmark` vorbei aktiviert. Dies ist beabsichtigt und entspricht dem Verhalten von `@fancy` bei `fancyhdr`.

Vergleicht man in Abbildung 3 das Ergebnis eines einfachen Dokuments mit `fancyhdr` mit dem Ergebnis eines einfachen Dokuments mit `scrlayer-fancyhdr`,

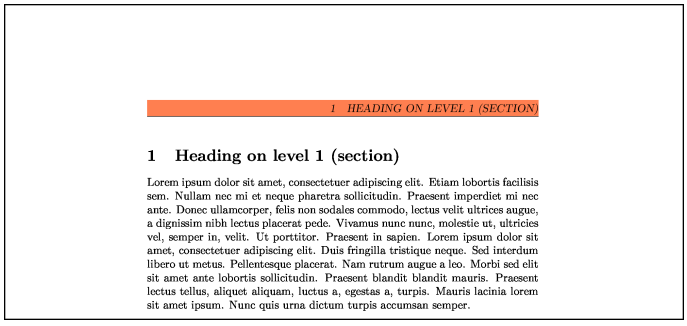


Abbildung 2. Farbig hinterlegte Kopfzeile durch Kombination von `fancyhdr` und `scrlayer`

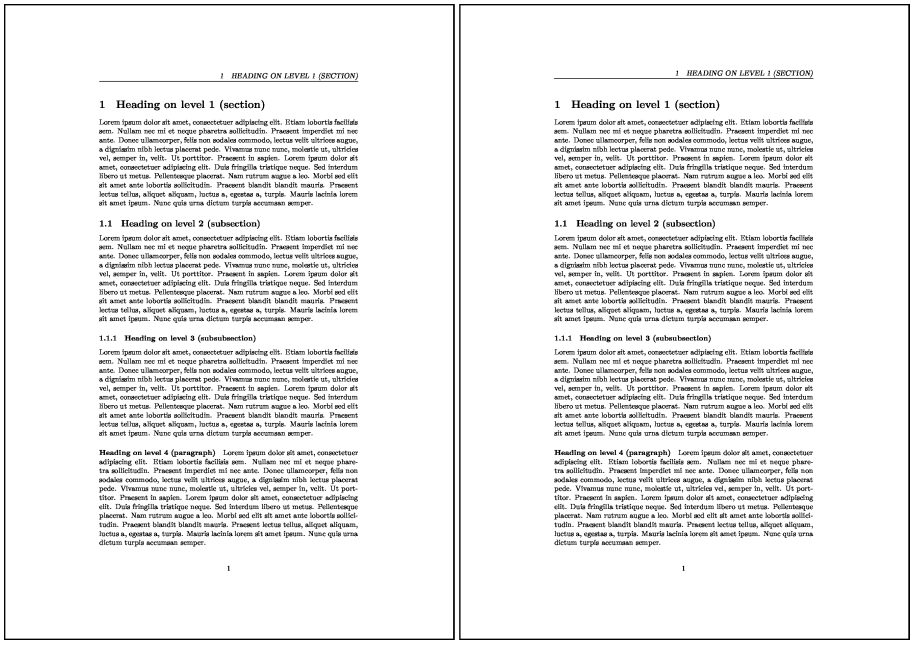


Abbildung 3. Gegenüberstellung eines einfachen Dokument mit `fancyhdr` (links) und mit `scrlayer-fancyhdr` (rechts)

so fällt eventuell auf, dass die vertikale Position des Kopfes nicht ganz identisch ist. Hier muss gegebenfalls noch nachgebessert werden. In vielen Fällen wird dieser Unterschied jedoch auch gar keine Rolle spielen.

Aktuell wirken sich Änderungen an den Ebenen von `@fancy` auf alle mit `fancyhdr` definierten Seitenstile gleichermaßen aus. Will man bestimmte Ebenen nur für einzelne Seitenstile, so muss man diese gegebenenfalls in der Definition des Seitenstils hinzufügen und auch wieder entfernen. Ob das Resultat dann immer den Erwartungen entspricht, ist noch unklar.

Ebenso wird die Kompatibilität zwischen den KOMA-Script-Klassen und `fancyhdr` durch die Verwendung von `scrlayer-fancyhdr` nicht verbessert. Dies ist auch nicht Ziel des Experiments und würde sich umgekehrt auf die Kompatibilität von `scrlayer-fancyhdr` mit `fancyhdr` auswirken.

Wie geht es weiter?

Trotz der im vorherigen Abschnitt genannten Einschränkungen kann das Experiment als Erfolg betrachtet werden. Dennoch wird von einem produktiven Einsatz derzeit noch abgeraten. Wer möchte kann aber bereits jetzt `scrlayer-fancyhdr` testen. Zum Zeitpunkt des Verfassens dieses Artikels wird dazu die aktuelle Pre-Release von [5] benötigt. Darin findet sich auch eine eigene Anleitung zu dem Paket. Es ist geplant, die aktuelle Testversion von `scrlayer-fancyhdr` auch mit der nächsten offiziellen Release von KOMA-Script zu verteilen. Von den Rückmeldungen durch die Anwender hängt es dann ab, inwiefern dieses Paket weiterentwickelt und fester Bestandteil von KOMA-Script wird.

Literatur und Software

[1] Markus Kohm: »Dokumentversion mit `scrlayer`«, DTK, 27.3 (2015), 20–24.
[2] — »Farbige, kleine Kapitelmarken am Rand mit `scrlayer`«, DTK, 27.3 (2015), 24–30.
[3] — »Firmenlogo mit scrlayer«, DTK, 27.3 (2015), 14–19.
[4] — KOMA-Script, ein wandelbares LaTeX 2_ε-Paket, 2018, `CTAN:/macros/latex/contrib/koma-script/doc/scrguide.pdf` (besucht am 26. 9. 2018).
[5] — KOMA-Script, The `scrlayer` interface `scrlayer-fancyhdr`, 2018, `https://komascript.de/current` (besucht am 26. 9. 2018).
[6] Piet van Oostrum: Page layout in LaTeX, 2017, `CTAN:http://mirrors.ctan.org/macros/latex/contrib/fancyhdr/fancyhdr.pdf` (besucht am 26. 9. 2018).

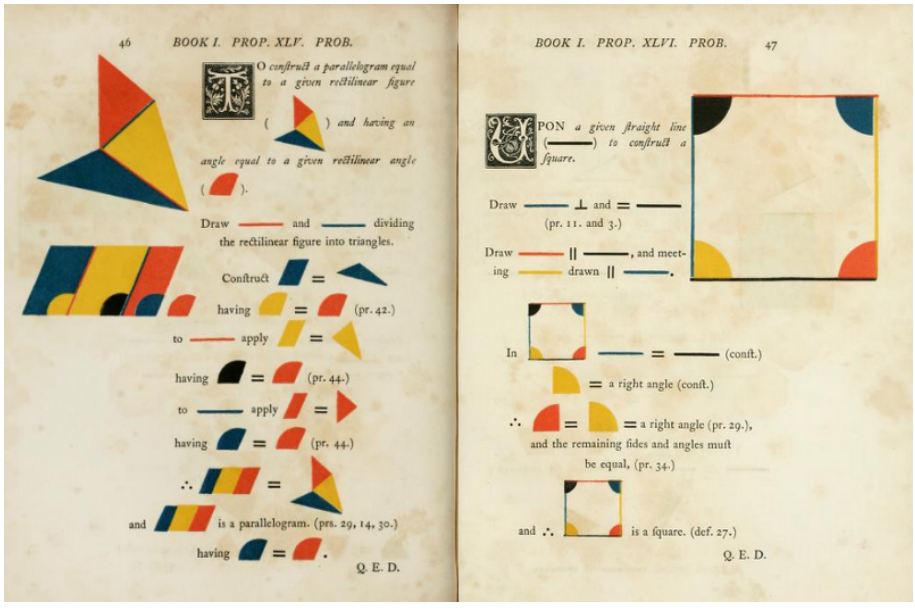
Notes

1. Allerdings hatte ich gehofft, dass mir die damit verbundene Arbeit jemand abnehmen würde.

Markus Kohm
komascript@gmx.info

Oliver Byrne's "The first six books of the Elements of Euclid" in ConTEXt and MetaPost

The most famous book of an Irish mathematician Oliver Byrne is his 1847 rendition of the first books of Euclid's "Elements." What makes this book stand out is that instead of ordinary letter designations such as "triangle ABC," miniature pictures directly in the text are used, like this  . As fancy as it looks, it's actually quite a nice way to deal with referencing diagram features without overwhelming the reader with a lot of mediatory letters. On the writer side, however, this approach presents more challenges than the traditional one, since typesetting tools tend not to be suited to be used in such a manner.



GUI tools aren't especially easy to extend, but TEX and its extensions such as LATEX or ConTEXt combined with MetaPost or a similar software provide an excellent environment to add the tools needed to recreate such a book. To prove that this is indeed the case I chose ConTEXt and MetaPost combination to recreate Byrne's book and to make appropriate tools to potentially apply Byrne's approach to different situations.

The overall structure

Byrne has made only 6 of 13 books of the "Elements." Each book mostly consists of "propositions"—theorems and problems. Most propositions have diagrams (usually one per proposition) which are referenced in proposition texts.

For diagrams, I made a ConTEXt macro that creates a new MetaPost instance. And in MetaPost, some functions to create these constructions. Their use looks somewhat like this:

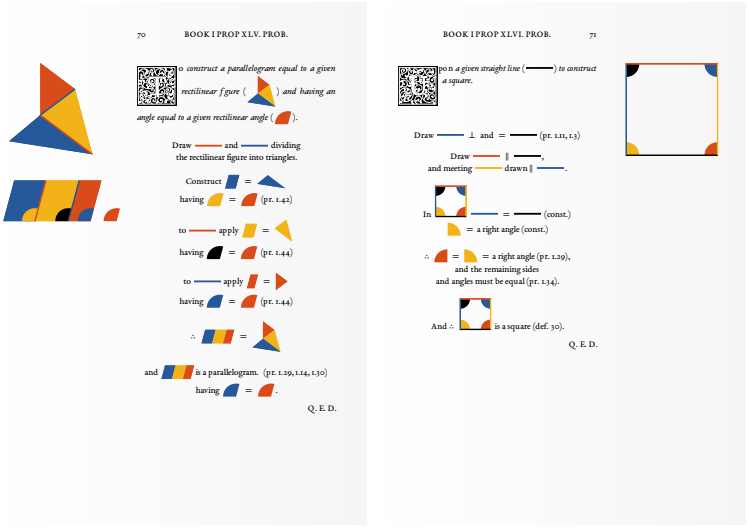
```
\defineNewPicture{ % Inside this thing you put the construction
    pair A, B, C, D; % MetaPost has a special type of variables
                     % for coordinates
    numeric d;
    d := 2u;
    A := (0, 0);                %
    B := A shifted (d, 0);      % Point coordinates are set here
    C := A shifted (0, -d);     %
    D := A shifted (d, -d);     %
    byAngleDefine(B, A, C, byblack, 0);     % These define angles:
    byAngleDefine(D, B, A, byblue, 0);      % firstangle points,
    byAngleDefine(C, D, B, byred, 0);       % then color,
    byAngleDefine(A, C, D, byyellow, 0);    % then style.
    draw byNamedAngleResized();     % This thing draws all the angles.
    byLineDefine(A, B, byred, 0, 0);        % These define line segments:
    byLineDefine(B, D, byyellow, 0, 0);     % firstends,
    byLineDefine(D, C, byblack, 0, 0);      % then color and style,
    byLineDefine(C, A, byblue, 0, 0);       % then thickness.
    draw byNamedLineSeq(0)(AB,BD,DC,CA);    % This thing draws a sequence
                                            % of lines.
}
\drawCurrentPicture % And this thing draws the whole picture.
```

For proposition texts, I made a series of macros that draw pictures in the same MetaPost instance and based on the existing diagram. In general, they execute arbitrary MetaPost code, but most of the time, they take object names as arguments. Like this:

```
% Names are automatically assigned to line segments but can be assigned
% manually.
Draw $\drawUnitLine{CA} \perp \mbox{ and } = \drawUnitLine{DC}$.\\
Draw $\drawUnitLine{AB} \parallel \drawUnitLine{DC}$,\\
and meeting \drawUnitLine{BD} drawn $\parallel \drawUnitLine{CA}$.
```

Draw ——— ⊥ and = ———.
Draw ——— ∥ ———,
and meeting ——— drawn ∥ ———.

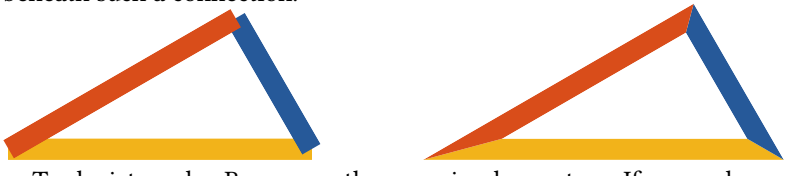
This is how these parts work together:



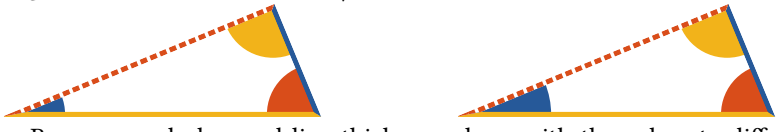
Some features

Pictures in the book look simplistic, but not to the point when you can get away with just using the default drawing commands.

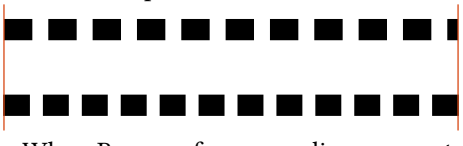
The touching points of line segments for example need some nice representation. For now, connection of only two lines is supported, but other lines can simply be put beneath such a connection.



To depict angles Byrne mostly uses circular sectors. If an angle is small enough, the sector with the same radius may look tiny so it makes sense to enlarge it. Currently, the radius stays the same for angles above 60 degrees, and for smaller angles, this formula is used: $r/\sqrt{a/60}$



Byrne uses dashes and line thickness along with the colors to differentiate the lines. Dashed lines to look nice should begin and end with full dashes. To achieve this, the dash pattern is scaled a bit to fit a line length.



When Byrne references a line segment in text, he draws it always horizontally retaining the color (or colors) and style of the original line on a diagram. With respect to the length he either makes all the lines in the text the same length, or tries to keep relative lengths to some degree. To achieve this I use this formula: $L_{text} = (L_{diagram}S)^a L_{desired}^{1-a}$ where $0 \leq a \leq 1$ and S is the scale factor for in-text graphics.

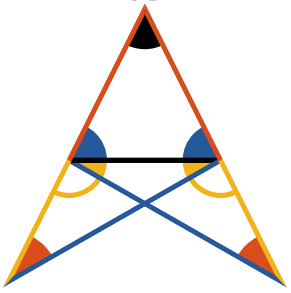
When $a = 1$ the length of the segment in text L_{text} is equal to that on the diagram $L_{diagram}$ times S. When $a = 0$, L_{text} is always equal to some desired length $L_{desired}$. And for values of a in between L_{text} retains the relative sizes of the lines on the diagram, but makes shorter than $L_{desired}$ lines a bit longer and vice versa. You may want this if you are about to retain relative line lengths while avoiding 2 mm lines next to 2 cm ones.

$a = 0$:

$a = \frac{1}{2}$:

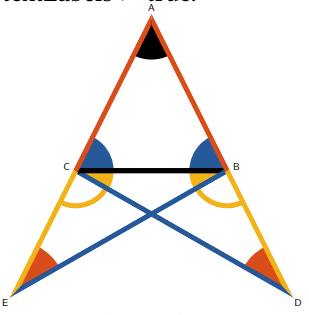
$a = 1$:

Byrne didn't use letter designations for points except for in the Introduction. But even in his own book he couldn't avoid using the same colors and styles for different lines causing potential confusion. Take this, for example:



Produce ,
take ,
draw .

Edward Tufte in his "Envisioning Information" has pointed out that letters wouldn't do any harm to Byrne's method and on the contrary would be very useful. To check if this is true, I added optional text labels, which you can turn on by setting textLabels := true.



Produce ,
take ,
draw .

Text labels aren't fully automatic on the main diagram and you have to specify their placement like so:

```
% labels will be placed at the vertices of ECABDCB polygon
% note that labels on C and B are placed only once
 draw byLabelsOnPolygon(E, C, A, B, D, C, B)(0, 0);
```

On the inline pictures labels are placed automatically in most cases.

Initials

One thing Byrne's book shares with many other books of the same period is the extensive use of initials. Not being an overly important feature, initials are still fun and I decided to keep them. There are plenty of fonts with initials out there including the one which comes with EB Garamond used in this book, but not many of them contain letters outside the English alphabet, which is an issue, because the Russian translation requires Cyrillic letters. So instead of looking for an appropriate font or drawing one myself I chose to try and make some kind of procedurally generated initials substitute in a way that would allow me to get any letter without drawing it manually and would make all the letters decorated with slightly different ornaments.

The algorithm is pretty straightforward: curls are placed on the parts of a letter and on the frame, as large as they can grow. This is done several times. All the curls are included in the subsequent iterations. After that, some "leafs" are grown in the same manner, filling the remaining gaps. It's also quite slow, so initials generator is located in a separate MetaPost file and it processes a text file with the list of all the initials and vignettes created when the book is processed.

The results aren't fully satisfying, so it's possible to put PDF or SVG files with any initials instead of generating ones.

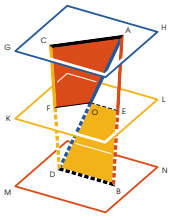
Translation

Having in mind the flexibility of both the diagrams and the initials, translating this book to Russian was not more difficult than any other book would be. At first, I used the well-known Murduhai-Boltovskoi's translation as a reference but soon found that Byrne changed too much in Euclid's theorems for a reference translation to be very useful. The process of translation helped catching bugs, both mine and Byrne's. For instance, the diagram for proposition 9 in the sixth book doesn't match the text, making the proof incorrect, so I had to change it.

Marcin Ciura from Poland started to make a Polish translation of the book. He have already found even more typos and bugs and made many extremely valuable contributions on the way (scaling angles, for instance, was his idea).

Plans

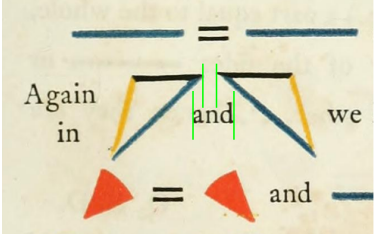
Byrne's book doesn't contain any solid geometry (save an image of a parallelepiped in the Introduction). Neither did I make any tools for it, but at some point, I decided to add some, so I started to "byrnify" books 11–13 to have something to use the new tools on. For now, only a bit more than half of the 11th book is roughly made, along with some functions to make solid constructions and project them onto the screen plane. Solid constructions, though, on the average are substantially more complex than plane ones, and I'm not yet sure if Byrne's approach would fit them well and if MetaPost is a reasonably convenient tool for the job.

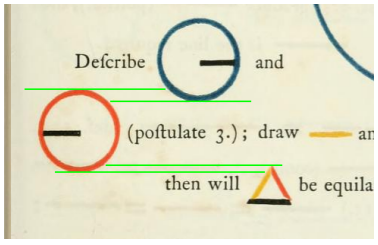


If two straight line be cut by parallel plane, they will be cut in the same ratios.

For let the two straight lines A–B, C–D be cut by the parallel planes G–H, K–L, M–N at the points A, E, B and C, F, D. I say that, A–E : E–B :: C–F : F–D. For let A–C, B–D, A–D be joined, let A–D meet the K–L at the point O, and let E–O, O–F be joined.

Pictures in text ofttimes need some kerning. Similarly, yet trickier, taller pictures on adjacent lines spread the lines very wide, which is appropriate only if these pictures collide. I don't yet know how and if I can make an automated solution for these issues, but I definitely want to because fixing them by hand is really tedious.





MetaPost can also be used from within LaTeX and as a standalone program. In the future, I plan to make LaTeX macros similar to ConTeXt ones, to allow using them there too.

The code as of now is poorly documented and this issue is high on my priority list.

Last but not least, I want to apply the tools to something more modern and practical than "the Elements."

The code is licensed under GNU GPLv3 or later and the book and its translation are CC-BY-SA 4.0. Both can be found here: github.com/jemmybutton/byrne-euclid

Sergey Slyusarev

XML to PDF with ConTEXt

Background

One of the DocWolves product lines is an on-line production environment for documents related to decision workflows. From the users' input, we create published HTML pages and PDF documents.

Our clients input text fragments into a web form built around a customized installation of CKEditor. CKEditor is a free collection of javascript components for WYSIWYG editing of HTML that runs inside the web browser, without needing any client-side plugins (see http://ckeditor.com for more details). We save these text fragments along with various bits of meta-information in a MySQL database. When the editing cycle is done and the client decides to publish a document we combine these various text fragments, meta-information and any needed images into either an HTML page or an XML file. In the case of the XML file, that is then converted into PDF using ConTEXt.

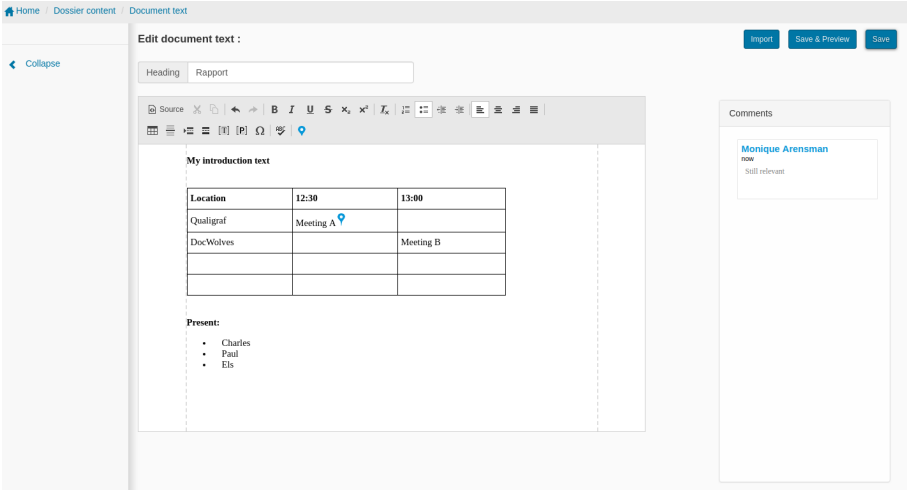


Figure 1. Example of the web-based input screen

Various Perl scripts control and monitor all stages of the document creation process. Before being stored, the raw textual (HTML) input is passed through a parser that removes anything that cannot be handled properly by both the HTML and the XML-to-PDF backends. But the monitoring system starts even earlier: cut and paste within the CKEditor is modified to prevent weird HTML (for example from word processing software) from entering the input stream. Also, various options have been disabled in the CKEditor.

To make sure that we know exactly what is in our records, the input parser also converts the angle brackets of the acceptable HTML tags into square brackets in the stored data, and it removes any unsupported HTML attributes. At the same time square brackets in the input are converted to HTML character entities.

Using this intermediate format for storage means that at the output stage, the backend file generator can convert any and all angle brackets it encounters into character entities, thus ensuring that the generated output is always valid XML/HTML.

Besides blocking weird (and potentially damaging) input, there is a another reason for all the filtering. The generated published documents often have to adhere to a specific client house style, with (for example) predefined margins, font settings, and a page background. Thus, removing visual markup that conflicts with the house style settings is a secondary but quite important function of the filtering system. By the way, those style settings are also stored in our system, somewhere separate from the actual text fragments. I do not have to explain here why that is handy, TEX users are quite familiar with separating content from presentation, but for many outside the TEX community this still a foreign concept.

Here is a small part of a text fragment as it is actually stored inside the database:

```
[p align="left" class="western" lang="fr-FR"][br  /][/p][table cellpadding="7"
cellspacing="0" width="615"][tr ][td style="border: 1px solid #000000;
padding: 0in 0.08in" valign="top" width="599"][h3 class="western"
lang="fr-FR"]RAPPORT DE PRESENTATION[/h3][h3 class="western" lang="fr-FR"]BUDGET
PRIMITIF 2018 - EAU[/h3][/td ][/tr ][/table ][p align="justify" class="western"
lang="fr-FR"][br  /][/p][p align="justify" class="western" lang="fr-FR"]Le
budget du service de l'Eau s'équilibre en Dépenses et Recettes, aussi
bien en Fonctionnement qu'en Investissement à hauteur de [b ]11 459
610,00 €, [/b]soit :[/p][p align="left" class="western" lang="fr-FR"][br
/][/p]
....
[p align="justify" class="western" lang="en-US"][img
data-image-id="2279" height="305" src="
....
Y2UyODA1hnDS2gAAAABJRU5ErkJggg==" width="675"][/p][p align="justify"
class="western" lang="fr-FR"][br  /][/p]
```

There are normally no line breaks in the tag portions of the database record, the line breaks are added here just for this example. In the actual input text, we preserve the whitespace characters from the input.

As you can see, there is still a fair bit of HTML and CSS supported in the system.

ConTEXt input files

The main ConTEXt input is the generated XML file. This contains not only the collected text fragments, but also a block of house style settings, and some meta-information about the document.

The XML processing takes place via an environment file called `dw-workflow`. Most of the content of this file is Lua code, but there is also a bit of plain TEX code and a few ConTEXt patches included in there.

There is only one other include file, and that contains our typescript definitions.

In the future `dw-workflow` may be split into a few more generic modules so that code may be reused in other DocWolves products, but for now everything is in one (fairly long) file.

XML structure

The listing below shows the general structure of the generated XML file.

```
<?xml version="1.0"?>
<root>
   <settings>
      <setting name="papersize" value="A4"/>
      <setting name="textwidth" value="170"/>
      <setting name="..." value="..."/>
   </settings>
   <documentinfo>
     <meta name="DOSSIER_REF" value="1819"/>
     <meta name="..." value="..."/>
```

```
   <document_content>
      <content format="html">
         <fragment id="19804" dos_id="2157" dsd_id="5562">
            <h2>asdf</h2> ...
         </fragment>
         <fragment>...</fragment>
      </content>
   </document_content>
</root>
```

XML settings and setup

The <meta> information is directly copied to the PDF XMP info, and otherwise ignored.

The <setting> tag is where the style options are communicated to ConTEXt. Currently, there are about two dozen settings. As is to be expected, most of them deal with typical page setup. There are settings that are converted into arguments for \setuppapersize and \setuplayout, settings for \setupfootertexts and \setupheadertexts, and settings for \switchtobodyfont.

A few of the settings are more interesting. In particular, there is a setting for the font size and a small group of settings for setting up the optional background image for each page.

As is customary in CSS, the document font size does not only set up the size of the main text font, but it also sets up the relative sizes of the headings and footnotes, the white space before and after headings, et cetera, using a multiplier of the base font size value. And since these multiplications can produce odd body font sizes, there is a Lua function that not only sets up the actual ConTEXt commands, but also executes the needed \definebodyfontenvironment commands. The start of this function looks like this:

```
function userdata.setupheads(argsize)
   local thesize = string.gsub(argsize,"pt",'')
   local size = tonumber(thesize)
   local e  = math.floor(size*cssparser.typemaps.font_size['xx-large'] + 0.5)
   local d  = math.floor(size*cssparser.typemaps.font_size['x-large'] + 0.5)
   local c  = math.floor(size*cssparser.typemaps.font_size['large'] + 0.5)
   local x  = math.floor(size*cssparser.typemaps.font_size['small'] + 0.5)
   local xx = math.floor(size*cssparser.typemaps.font_size['x-small'] + 0.5)
   context.definebodyfontenvironment({e ..'pt'})
   context.definebodyfontenvironment({d ..'pt'})
   context.definebodyfontenvironment({c ..'pt'})
   context.definebodyfontenvironment({x ..'pt'})
   context.definebodyfontenvironment({xx ..'pt'})
   -- h1
   context.setuphead({'part'},
     {align="flushleft", page='no',placehead='yes',number='no',
      style='\\switchtobodyfont['..e ..'pt]\\bf ',
      before='{\\blank['..xx ..'pt]}',
      after='{\\blank['..xx ..'pt]}',
      aligntitle='yes',
      })
   ...
```

Per-client page background images are used so that we do not have to write settings for each and every client with a special logo in the page top or bottom. We have set up the system such that for each client, dedicated single-page PDF files are searched for. If found, these are then added to the page on their own layer. For each combination of document type and paper size, there are two possible PDFs: one for the first page, and another for all following pages (often client logos are larger on the first page of a document). Such backgrounds are searched for in four locations: the client folder with and without paper size, and the global locations for the same (the global locations contain single-page empty files).

The code that deals with this is:

```
local paths = {}
paths[#paths+1] = valueordefault(settings.clientbackgrounddirectory .. '/'
                                 .. settings.papersize,nil)
paths[#paths+1] = valueordefault(settings.clientbackgrounddirectory,nil)
paths[#paths+1] = valueordefault(settings.backgrounddirectory .. '/'
                                 .. settings.papersize,nil)
paths[#paths+1] = valueordefault(settings.backgrounddirectory,nil)
context.setupexternalfigures({directory = table.concat(paths,',') })
if settings.clientbackgroundname and #settings.clientbackgroundname>0 then
   context.resetbackgroundfigure(settings.clientbackgroundname,"1")
end
```

where `\resetbackgroundfigure` is a separate macro that ensures the PDF image is included:

```
\def\pagebackgroundfigure{}

\def\resetbackgroundfigure#1#2%
  {\gdef\outputpagen{#2}%
   \gdef\pagebackgroundfigure
       {\externalfigure[#1-page-\outputpagen.pdf]
          [width=\the\paperwidth,height=\the\paperheight,page=1]}}

\defineoverlay
  [pagebackground]
  [{\pagebackgroundfigure \gdef\outputpagen{n}}]

\setupbackgrounds
  [page]
  [state=repeat,background={pagebackground}]
```

Now, if the above looks a bit sneaky to you? ... yeah, I know!
This setup is inherited from an older (`mkii`) project, where continuous redefining of the `\pagebackgroundfigure` was necessary. And it works fine, so I saw no reason to implement something else.

Before getting into the actual processing of the XML, let me introduce some of the `dw-workflow` code that is needed for almost all XML, and especially HTML processing. First, there is the setup that connects XML tags to Lua functions.

```
\startxmlsetups xml:oursetups
      \xmlsetfunction {\xmldocument}{*}                {xml.functions.panic}
      \xmlsetfunction {\xmldocument}{root}             {xml.functions.flush}
      \xmlsetfunction {\xmldocument}{settings}         {xml.functions.settings}
      \xmlsetfunction {\xmldocument}{setting}          {xml.functions.setting}
      \xmlsetfunction {\xmldocument}{document_content} {xml.functions.document}
      \xmlsetfunction {\xmldocument}{content}          {xml.functions.flush}
      \xmlsetfunction {\xmldocument}{fragment}         {xml.functions.fragment}
      \xmlsetfunction {\xmldocument}{h1}               {xml.functions.h1}
      ....
\stopxmlsetups

\xmlregistersetup{xml:oursetups}
```

At the dotted line are all the functions for the separate HTML tags we support, which are skipped here for brevity.

The `\xmlsetfunction` for `*` is a visualization trick. The `panic` function typesets all child data in a bold, red, and ugly way. Even with all the precautions we take for making sure the input is predictable, it is still possible that something sneaks through. One example of that happening was when we updated the whole workflow subsystem in development to support some extra tags, but we had forgotten to update the `dw-workflow` accordingly. The `panic` function's output made that mistake clearly visible during testing.

Next up in the dw-workflow is a set of definitions like this:

```
\def\htmlentity#1#2#3#4{\xmlsetentity{#2}{#1}}
\def\htmltexentity#1#2#3#4{\xmltexentity{#2}{#1}}

% latin chars
\htmlentity{À}{Agrave}{192}{Capital a with grave accent}
\htmlentity{Á}{Aacute}{193}{Capital a with acute accent}
\htmltexentity{~}{nbsp}{160}{Non-breaking space}
...
```

There are some 250 lines of these, adding support for all of the predefined HTML entities. ConTEXt converts numeric entities automatically, but the named HTML versions need explicit definitions. There are four arguments because this information is converted to TEX macros from a HTML table listing all the entities. In an earlier stage of development, the third and fourth arguments were used to typeset a ConTEXt table for comparison to that HTML table.

XML text fragments

The text fragments are written out as <fragment> tags in the XML file, and the content of each of those is basically HTML with a bit of optional inline CSS. Let's start with a bit of example listing:

```
<fragment id="18202" dos_id="3279" dsd_id="6980" title="no">
  <p style="text-align:center;"><strong>COMMISSION &test; PERMANENTE</strong></p>
  <p style="text-align:center;">Séance du </p>
  <p style="text-align:center;"><strong>DOSSIER N°</strong> <strong></strong></p>
</fragment>
<fragment id="18203" dos_id="3279" dsd_id="6980" title="no">
  <table border="1" cellpadding="1" cellspacing="1" style="width:639px;">
    <tbody>
      <tr>
        <td style="height:22px; width:626px;"><br/>
          <b>Politique : </b>COPY<br/><br/>
          <b>Programma: </b><placeholder>Arensman Monique</placeholder><br/><br/>
           Opération : SOME TEXT<br/><br/><br/>
         </td>
      </tr>
   </tbody>
  </table>
 </fragment>
```

Most of the attributes of the fragment tag in the XML example above are for debugging purposes only and are ignored during typesetting. The one processed attribute is title. As you can see in figure 1, each text fragment can have a system-supplied heading. If that heading is not given, we add an extra blank to give some visual separation between adjacent text fragments.

The fragment also supports an is_framed argument, which is not used in the above example. This creates a border around the whole fragment using \starttextbackground. The system makes use of text backgrounds to make sure that the fragment can still break across pages, which is an absolute requirement.

```
function xml.functions.fragment(t)
   local framed = false
   if t.at.is_framed and t.at.is_framed == 'true' then
      cssparser.prependstyle(t,"border: 1px solid black; padding: 10px;")
      framed = true
   end
   context.flushsidefloats() -- clear left/right divs
   if t.at.title and t.at.title:lower() == "no" then
     context.blank({'line'})
   end
   if framed then
     local args = textbackgroundarguments(t)
     context.definetextbackground({'fragmentbackground'.. tonumber(t.at.id)},args)
```

```
    context.starttextbackground({'fragmentbackground'.. tonumber(t.at.id)})
  end
  lxml.flush(t)
  if framed then
    context.stoptextbackground()
  end
end
```

Rather than try to process the various background options right in the above function, the requested frame is converted into CSS statements. The function `textbackgroundarguments()` converts that CSS specification into arguments for `\definetextbackground`.

Interpreting CSS specifications

Parsing CSS specifications is actually quite easy, especially for the only specification format we support right now: in-line style elements. A smallish Lua function does all of the initial work:

```
local P, S, C = lpeg.P, lpeg.S, lpeg.C

function cssparser.parse(t)
  local result = {}
  local found = {}
  if t.at.style then
    local function store(a,b)
      found[a] = b
    end
    local skipspace    = S(" \t")^0
    local colon        = P(":")
    local semicolon    = P(";")
    local eos          = P(-1)
    local somevalue    = (1 - (skipspace * (semicolon + eos)))^1
    local somekey      = (1 - (skipspace * (colon + eos)))^1
    local cssmatch = ((C(somekey) * skipspace * colon * skipspace * C(somevalue))
                     /store * skipspace * (semicolon + eos) * skipspace)^1 + eos

    lpeg.match(cssmatch,t.at.style)

    for i,v in ipairs(cssparser.registered) do
      local k = v[1]
      if found[k] then
        local f = v[2]
        f(t,result,k,found[k])
        if cssparser.inherited_trait[k] then
          if found[k] ~= 'inherit' then
            cssparser.inherit(t,k,found[k])
          end
        end
        found[k]=nil
      end
    end
    for i,v in pairs(found) do
      cssparser.report('unknown css property: '..i)
    end
    for k,v in pairs(result) do
      if v == 'inherit' then
        result[k] = cssparser.inherited(t,k, cssparser.inherited_trait[k])
      elseif v == 'initial' then
        result[k] = cssparser.inherited_trait[k]
      end
    end
  end
  return result
end
```

The first half of the above function is the LPEG match needed to split the string into a key–value table. The second half takes care of interpreting the registered traits. There are two separate tables that are used in this process:

`cssparser.registered`
: is a table of CSS traits that are known to the system. Each of the array values is a further array with two items: the name of the trait, and a processing function for that trait. These functions are then called to interpret the trait's CSS specification. They take care of things like converting special color names and oddball length specifications to something ConTEXt and MetaPost understand. The main table is an array instead of a dictionary because ordering is important in the case of shortcut traits, as we will see later.

`cssparser.inherited_trait`
: contains the initial values for inheritable traits. The processing taking place here is essentially a callback, since normally inheritance is handled by the processing functions in the previous loop.

Both arrays are set up by calls to `cssparser.register`:

```
function cssparser.register (k,f,inherited)
   cssparser.registered[#cssparser.registered+1] = {k, f}
   if inherited then
      cssparser.inherited_trait[k] = inherited
   end
end
```

Throughout the rest of `dw-workflow`, there are dozens of calls like this:

```
cssparser.register('font-size',
   function (t,result,key,value) result[key] = value end, '11pt')
```

The simplest of those calls use an inline function as seen above. The more complicated ones define the function separately, just because that produces nicer formatting of the source. 'Complicated' is perhaps too big a word: the CSS parser callbacks do not do all that much work besides verifying the input syntax and resolving CSS shortcuts.

```
local function parse_padding_shortcut(t,result,key,value)
   local function process(a,b,c,d)
      local t,r,l,b = trlb(a,b,c,d)
      result[key ..'-top']    = cssparser.htmldimension(t)
      result[key ..'-bottom'] = cssparser.htmldimension(b)
      result[key ..'-right']  = cssparser.htmldimension(r)
      result[key ..'-left']   = cssparser.htmldimension(l)
   end
   local pattern = (cssparser.matches.width^-4/process)
   pattern:match(value)
end

local function parse_one_padding(t,result,key,value)
   local function process(a)
      result[key] = cssparser.htmldimension(a)
   end
   local pattern = (cssparser.matches.width^1/process)
   pattern:match(value)
end

cssparser.register("padding",        parse_padding_shortcut)
cssparser.register("padding-left",   parse_one_padding)
```

As you can see, there are some other helpers in the `cssparser` table. For example, `htmldimension` converts the CSS width keywords (`thin`, `medium`, `thick`) into discrete HTML lengths. There is also `htmlcolor`, which converts named colors into HEX values. The goal of these functions is not to produce the final trait value that will be used for typesetting, but just to get rid of some of the idiosyncrasies of CSS.

The `cssparser.matches` table contains a small set of predefined LPEG matches for common CSS data types. Besides `width`, there are definitions for `length`, `color`, and `border_style`.

The previous takes care of parsing CSS specifications. But how to use them?

When inside one of the XML tag processing functions, it is normally enough to call either `cssparser.style()` or `cssparser.styled()`. The latter takes care of implicit inheritance, the former just returns the locally specified CSS trait, if there is any.

```
function cssparser.styled(t,name)
  if not t.__style then
     t.__style = cssparser.parse(t)
  end
  if t.__style[name] then
     return t.__style[name]
  -- the next elseif is not done in cssparser.style
  elseif cssparser.inherited_trait[name] then
     return cssparser.inherited(t,name)
  end
  return nil
end
```

It is worth noting that `cssparser.styled()` is not just used for explicit `inherit`. What it actually returns is either a local value, or the value you would get *if* explicit `inherit` *was* present, even if it is not actually there. This turned out to be quite useful, for example to query the current font size and text. Both of these are quite often needed during processing, even in tags that do not actually inherit the value.

The functions `cssparser.inherited` and its companion `cssparser.inherit` are used to query and set inherited traits. They actually allow arbitrary keywords, so they can also be used to set up inheritance for ad-hoc values, as that turned out be quite useful. For example when dealing with the implicit CSS state, as in whether the current XML subtree is part of a floating object or not.

We have already seen `cssparser.prependstyle()`. In that example, it was used with a literal CSS string. But the most prevalent use of that function is to convert HTML attributes into CSS traits. We will see a usage example further down in this article.

Attribute values

Some CSS attribute values can be used directly in the Lua code, like for the various keyword-valued traits. These are commonly used for decisions in the processing step, and do not actually need to be passed to ConTEXt. Most of the ones that *do* need to be passed on can be handled by a simple hash. The table `cssparser.typemaps` contains a small set of tables that map CSS keywords to ConTEXt keywords for this purpose.

For example:

```
cssparser.typemaps.text_align = {
   left    = 'flushleft,verytolerant,extremestretch',
   right   = 'flushright,verytolerant,extremestretch',
   center  = 'center,verytolerant,extremestretch',
   justify = 'verytolerant,extremestretch'
}
```

At the moment, there are typemaps for `float`, `font_size`, `font_weight`, `list_style_type`, `text_align` and `vertical_align`.

Very handy, but this does not work for all attribute values. In particular, dimensions and colors require more attention.

CSS dimensions can have a fairly elaborate list of units, and only the basic ones actually match up with TEX dimensions. Our current system does not support all of the possible relative CSS dimensions like 'X percent of the viewport' and 'X percent of the width of the zero', but it does handle the absolute dimensions, `em` / `ex`, bare

numbers, and \%. We use a helper function from ConTEXt itself to convert the value from its CSS format to a number of TEX points.

```
function styledimension (val, full)
  if not full then
     full = userdata.settings.actualwidth
  else
     full = string.gsub(full,"pt",'') -- just in case
     full = tonumber(full)*65536
  end
  ret = (xml.css.dimension(val,72/\pixelsperinch*65536,full/100)/65536)
  return ret
end
```

CSS colors are even more flexible. Not only are there predefined named colors as mentioned above and the special keyword cases `transparent`, `inherit`, `initial`, and `currentcolor`; but colors can also be specified using RGB values, HEX values, HSL values, RGBA values, and HSLA values. The CSS parser has already converted the named colors and resolved inheritance into HEX colors when the actual color interpretation starts, but there is still quite a bit of processing needed.

Here is an example of some of the possibilities borrowed from `w3schools.com`:

```
<h1 style="color:Tomato;">Hello World</h1>
<h1 style="background-color:rgb(255, 99, 71);">...</h1>
<h1 style="background-color:#ff6347;">...</h1>
<h1 style="background-color:rgba(255, 99, 71, 0.5);">...</h1>

<h1 style="background-color:hsl(9, 100%, 64%);">...</h1>
<h1 style="background-color:hsla(9, 100%, 64%, 0.5);">...</h1>
```

HSL values and HSLA values are not supported currently. We have not encountered any HTML generating software yet that actually uses HSL, and as we do not allow the users to key in raw HTML code, we are very unlikely to encounter such definitions. Until we actually need these, they are not worth the hassle.

We *do* support transparency, both the `transparent` keyword and the `rgba()` format. There are two separate Lua functions, `texcolor()` and `mpcolor()` to convert the value into either `\definecolor` / `\directcolored` ConTEXt style (`r=1,b=0.45,g=0.4`), or to a format that our MetaPost macros understand. The latter is used for all arguments to `\framed` and `\definetextbackgrounds`, so it is used much more often.

For MetaPost, HEX values are converted into CSS `rgb()` or `rgba()` syntax. This is the most straightforward way to pass around the color values in the Lua processing code.

But it means using transparency in our MetaPost macros requires a little trick, because the MetaFun macro for transparent colors is based on a different (and more flexible) syntax:

```
def rgb(expr a,b,c) = (a/255,b/255,c/255) enddef;
def rgba(expr a,b,c,d) = (a/255,b/255,c/255,d) enddef;
def checkedcolor(expr a)=
  if cmykcolor a:
    (cyanpart a, magentapart a, yellowpart a) withtransparency(1, blackpart a)
  else:
     a
  fi
enddef;
```

Actual XML processing

There are some two dozen of XML tag processing functions. Some are shorter, some are longer, but most all of them are easy to understand. Showing all of the processing

functions seems overkill, but let's look at some of the more interesting ones in a bit of detail.

Images

```
function xml.functions.img(t)
   local function style(a)  return cssparser.style(t,a) end
   if t.at.width  then cssparser.prependstyle(t, 'width:'..  t.at.width)      end
   if t.at.height then cssparser.prependstyle(t, 'height:' .. t.at.height)    end
   if t.at.align  then cssparser.prependstyle(t, 'text-align:' .. t.at.align) end
   local args = {}
   if style('width')  then args.width  = texdimension(style('width')) .. 'pt'  end
   if style('height') then args.height = texdimension(style('height')) .. 'pt' end
   if not userdata.patchimagesource(t) then
     t.at.src = userdata.settings.externalfigures .. '/' .. t.at.src
   end
   local textalignmap = cssparser.typemaps.text_align
   if style('align') then context.startalign({textalignmap[style('align')]}) end
   if cssparser.inherited(t,'infloat', 0) ~= 1 then context.dontleavehmode() end
   context.externalfigure({t.at.src}, args)
   if style('align') then context.stopalign() end
end
```

Most noteworthy here is the call to `userdata.patchimagesource()`. That function checks the HTML `src` attribute for the existence of inline base64 JPEG or PNG images.

```
<img data-image-id="2590" height="164" src="..."/>
```

If it finds one of these, it writes the binary data to a disk file and returns `true`. Since those images will always be in the local directory, there is no need to prepend the client's image directory to the `src` value.

The check for the virtual `infloat` trait is there because if the image is not inside of a `\placefigure`, then it should be handled in TEX's horizontal mode. The `\dontleavehmode` forces the start of a paragraph in that case.

Inline font switches

Some of the processing functions make use of dedicated subroutines, like the ones for inline font switches:

```
function xml.functions.b(t) -- also strong
   cssparser.prependstyle(t, 'font-weight:bold;font-style:inherit;')
   context.start()
   context.dontleavehmode()
   handlefontspan(t)
   lxml.flush(t)
   context.stop()
   context("{}")
end
```

The `handlefontspan()` routine takes care of all inline font switches and typical `<span>` settings like color changes and text decoration options. It is used throughout `dw-workflow` where `<span>`-style traits need to be set.

Text blocks and structure

Similarly, there is the `textbackgroundarguments()` function (that we saw earlier) to take care of typical block level traits like vertical whitespace, margins, and frames. This makes processing `<div>` quite simple.

The various `<h1..6>` tags are simply mapped onto the ConTEXt sectioning commands.

Lists

Itemization lists are a bit problematic because CSS essentially treats every item in a list separately. The net result of that is that ConTEXt needs to start and stop itemization lists regularly, and while that works ok, it is quite suboptimal.

```
function xml.functions.li(t)
   if t.at.type then
      cssparser.prependstyle(t,'list-style-type:' .. t.at.type)
   end
   cssparser.inherit(t,'inlist', true)
   local typemap = cssparser.typemaps.list_style_type
   local itemstyle = cssparser.style(t,'list-style-type')
   if itemstyle then
      context.stopitemize()
      if t.at.value then
         context.setupitemize({start = t.at.value})
      end
      context.startitemize({typemap[itemstyle]})
   elseif t.at.value then
      context.stopitemize()
      context.setupitemize({start = t.at.value})
      context.startitemize({typemap[cssparser.styled(t,'list-style-type')]})
   end
   style  = cssparser.styled(t,'text-align') or 'left'
   context.testpage({'1'})
   context.startitem()
    -- \vadjust to fix vertical spacing for p
   context('\\vadjust{\\kern -\\baselineskip}\\nobreak')
   context.startalign({cssparser.typemaps.text_align[style]})
   lxml.flush(t)
   context.par()
   context.stopalign()
   context.stopitem()
end
```

The `\vadjust` here is particularly ugly, but it is needed because the textual input can have item content both with and without <p> tags.

At some time in the future, we should move to a completely new list model that is closer to how CSS thinks about list items. The CSS model for list items is much closer to plain TEX's way of having separate `\item` commands than to the more structured ConTEXt way of having an enclosing environment. On first sight, you would think that the ConTEXt way is very close to the HTML structure for lists. But on closer examination, CSS allows so many low-level traits on the actual list items that it will probably work better if we switch to something more low-level than the `\startitem` ...`\stopitem` environment. For now, these low-level CSS traits are on the 'unsupported' list.

Tables

Tables are problematic as well. Not so much because of the cell formatting itself (`\bTABLE` generally does a fine job of that), but because all of the possible border styles and spacing variations around those cells.

Individual `\bTD` cells inside a `\bTABLE` are actually disguised `\framed` calls. This is great in that it allows various border and background settings. But `\framed` by itself is not quite powerful enough to do everything that is possible in CSS. As a result of that, quite a large section of `dw-workflow` consists of small extensions to `\framed` and a rather long list of MetaPost graphic definitions.

The normal `\framed` already has four detail values for `frame`: `leftframe=on`, etc. Our version has a similar splits for `margin`, `rulethickness`, and `framecolor`. All of these variables can be set independently. Also, the `..frame` keys like `leftframe` take named versions for all of the CSS border styles instead of just `on` or `off`. The options are: `dotted`, `solid`, `double`, `dashed`, `none`, `hidden`, `groove`, `ridge`, `inset`, and `outset`. And the color settings are a bit different as well: instead of an actual ConTEXt color definition, they take a triplet of `(r,g,b)` or a quarted of `(r,g,b,a)`.

Most of this data is passed on to MetaPost macros that take care of the actual typesetting of the border segments. A little section of that part of `dw-workflow` looks like this:

```
def border_left_dotdash(expr wid, col, w, h, pre, post, dist, dotted,
                            left, top, bottom) =
  if wid>0:
     pickup pencircle scaled wid;
     n := floor((h-pre-post-top-bottom)/(dist));
     if not odd n: n:=n-1; fi
     nw := (h-pre-post-top-bottom)/n ;
     linecap := butt;
     draw ((wid/2+left,post+bottom)--(wid/2+left,h-pre-top))
       dashed dashpattern(if dotted: off nw on nw else: on nw off nw fi)
         withcolor checkedcolor(col);
     fill (wid+left,post+bottom)--(left,post+bottom)--(left,bottom)--cycle
       withcolor checkedcolor(col);
     fill (wid+left,h-pre-top)--(left,h-top)--(left,h-pre-top)--cycle
       withcolor checkedcolor(col);
  fi
enddef;
def border_left_dotted(expr wid, col, w, h, pre, post, left, top, bottom) =
  border_left_dotdash(wid,col,w,h,pre,post,wid,true, left, top, bottom)
enddef;
def border_left_dashed(expr wid, col, w, h, pre, post, left, top, bottom) =
  border_left_dotdash(wid,col,w,h,pre,post,3*wid,false, left, top, bottom)
enddef;
```

The actual connection between `\framed` and these MetaPost definitions is done by

```
\startuseMPgraphic{cellbackground}
   pickup pencircle scaled 0.0001;
   drawdot(0,0) withcolor transparent(1,0,(1,1,1));
   drawdot(\overlaywidth,\overlayheight) withcolor transparent(1,0,(1,1,1));
   border_left_\framedparameter{leftframe}
     (\framedparameter{leftrulethickness},  \framedparameter{leftframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{toprulethickness},\framedparameter{bottomrulethickness},
      \framedparameter{leftmargin}, \framedparameter{topmargin},
      \framedparameter{bottommargin});
   border_right_\framedparameter{rightframe}
     (\framedparameter{rightrulethickness}, \framedparameter{rightframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{bottomrulethickness},\framedparameter{toprulethickness},
      \framedparameter{rightmargin}, \framedparameter{bottommargin},
      \framedparameter{topmargin});
   border_top_\framedparameter{topframe}
     (\framedparameter{toprulethickness},   \framedparameter{topframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{rightrulethickness},\framedparameter{leftrulethickness},
      \framedparameter{topmargin}, \framedparameter{rightmargin},
      \framedparameter{leftmargin});
   border_bottom_\framedparameter{bottomframe}
     (\framedparameter{bottomrulethickness},\framedparameter{bottomframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{leftrulethickness},\framedparameter{rightrulethickness},
      \framedparameter{bottommargin}, \framedparameter{leftmargin},
      \framedparameter{rightmargin});
\stopuseMPgraphic
```

with the aid of a simple overlay that contains the `cellbackground` graphic, this is used as the `background` for `\framed`.

In case you are wondering: the two `drawdots` are needed to 'anchor' the graphic inside of the overlay in cases where not all sides are actually drawn.

The hardest part of table processing is support for the CSS property `border-collapse`. Our current version is not quite perfect, but it comes close. Close enough for our

clients. The remaining fault is that when two borders are collapsed into one, the 'winning' border should be placed in the center of the space between the two cells. Our code does not do this recentering. In most tables, this is fine. But the flaw is noticeable in tables where some of the rows (of a single table) have different left and right border widths, or where some of the columns have different top and bottom border widths. Considering the complexity of the collapsed border model, this is a limitation that we can live with. For the moment, at least.

Here is a cleaned up example of the kind of table input we have to process:

```
<table cellspacing="3" style="width:210mm;border-collapse: separate;">
  <tbody>
     <tr>
        <td style="height:60px;border: 6px solid orange;padding: 4px;">
           6px solid orange</td>
        <td style="height:60px;border: 3px solid orange;padding: 4px;">
           3px solid orange</td>
     </tr>
     <tr>
        <td style="height:60px;border: 5px inset green;padding: 4px;">
           inset green</td>
        <td style="height:60px;border: 5px outset green;padding: 4px;">
           outset green</td>
     </tr>
  </tbody>
</table>
```

Figure 2 shows what comes out of our system.

Figure 2. Example output of a table with and without the `border-collapse: collapse` setting.

Summary of the current project state

This project has been in development for about a year now. In that time, we solved all of the acute problems so that we can correctly process the current input to PDF.

To that end, we:

- ☐ wrote a (partial) CSS parser
- ☐ with the help of the ntg-context mailing list we added ConTEXt support for the CSS `minheight` attribute in `\framed`
- ☐ wrote enough code that we are supporting nearly all of the CSS table border features
- ☐ can handle in-line images in base64 encoding
- ☐ figured out how to support transparent colors in borders.

But that does not mean that we are done. In the future:

- ☐ we will have to support more CSS properties as they become requested by our clients
- ☐ we should implement a better solution for CSS inheritance than the current brute-force method
- ☐ we will probably need to implement an itemization model that is closer to the CSS approach
- ☐ and likely more stuff will pop up as we go along.

Taco Hoekwater
DocWolves B.V.

Schriften für mehrsprachige Texte

Die von Google unterstützte Schrift Noto (https://www.google.com/get/noto/ gibt es mittlerweile in zahlreichen Varianten, sodass die komprimierte Datei über ein Giga-Byte groß ist. Mit einer Vollinstallation von TeXLive oder MiKTeX hat man bereits die wesentlichen Schriften für die europäischen Sprechen im Format OpenType auf seinem Rechner.

Andere Sprachen kann man bei Verwendung von XeLaTeX oder LuaLaTeX leicht verfügbar machen. Das folgende Beispiel verwendet Beispieltexte, die ohne Rücksicht auf die eigentliche Schreibrichtung ausgegeben werden. Die Definition der Schriften lautet:

```
\setmainfont {Noto Serif}\defaultfontfeatures{Renderer=Harfbuzz}
\newfontfamily\Thai[Script=Thai]{Noto Serif Thai}
\newfontfamily\Bengali[Script=Bengali]{Noto Serif Bengali}
\newfontfamily\Marathi[Language=Marathi,Script=Devanagari]{Noto Sans Devanagari}
\newfontfamily\Nepali[Language=Nepali,Script=Devanagari]{Noto Sans Devanagari}
\newfontfamily\Japanese[Language=Japanese,Script=Hangul]{Noto Sans CJK JP}
\newfontfamily\Korean[Language=Korean, Script=Hangul] {Noto Sans CJK KR}
\newfontfamily\Mandarin [Language=Chinese Traditional,Script=Hangul]{Noto Sans CJK TC}
\newfontfamily\Tibetan[Script=Tibetan]{Noto Sans Tibetan}
\newfontfamily\Arabic[Script=Arabic]{Noto Naskh Arabic}
\newfontfamily\ArabicII[Script=Arabic]{Noto Kufi Arabic}
\newfontfamily\Urdu[Script=Arabic,Language=Urdu]{Noto Nastaliq Urdu}
\newfontfamily\Hebrew[Script=Hebrew]{Noto Sans Hebrew}
\newfontfamily\Armenian[Script=Armenian]{Noto Sans Armenian}
```

Noto Serif	Some text without any meaning
Thai	ข้อความบางส่วนไม่มีความหมายใด ๆ
Bengali	কোন অর্থ ছাড়া কিছু টেক্সট
Marathi	काही मजकूर विना अर्थ
Nepali	कुनै पाठ बिना कुनै पाठ
Japanese	意味のないテキスト
Korean	의미없는 일부 텍스트
Mandarin	有些文字沒有任何意義
Tibetan	འཁྲི་བ་མེད་རིགས་རྒྱུད་ཡོངས་ལ་
Kufi Arabic	بنعم يأ نود صنلا ضعب
Nashk Arabic	بنعم يأ نود صنلا ضعب
Urdu	ے ک ینعم یسک ریغب نتم ھچک
Hebrew	תועמשמ לכ אלל והשלכ טסקט
Armenian	Որոշ տեքստ առանց որևէ իմաստի

Herbert Voß
herbert@dante.de

A thing of beauty is a joy for ever

29 jaar NTG lid, dat doet iets met een mens ...

Abstract
Na net geen drie decennia heb ik uiteindelijk besloten mijn lidmaatschap niet langer te vernieuwen. Frans Goddijn vroeg me om een verslagje over mijn ervaringen met LaTeX/TeX sinds 'de tijd van toen' begin de jaren '90 tot nu... Een lang verhaal natuurlijk, maar ook leuk voor mij om even naar het verleden terug te blikken.

In den beginne was er niets (of toch niet veel)

December 1987 werd ik als belgisch marineofficier bij het CAWCS[1] geplaatst te Den Helder om als systeemontwerper aan de slag te gaan, dit in het kader van de Belgisch-Nederlandse (marine)samenwerking. Eerst diende ik de geheimen van de aldaar gebruikte VAX/VMS systemen en de programmeertaal ADA onder de knie te krijgen. Mede dankzij de goede begeleiding van mijn burgercollega's werd deze klus geklaard en kon ik het project MISS[2] dat pas in de steigers stond verder uitwerken en implementeren. Ik blij... tot mij tussen neus en lippen werd gemeld dat de (nogal wiskundige) documentatie voor dit project door mijn voorganger in LaTeX geschreven was en verder diende uitgewerkt te worden. Swahili, daar had deze Belg nog van gehoord, maar LaTeX?

De wiskundige kern van de MISS systeem was gebaseerd op de methode van de kleinste kwadraten welke op een vernuftige wijze door mijn voorganger (een Belgische polytechnisch ingenieur, type 'confirmed math wizzard') in ADA geprogrammeerd was. Een noemenswaardige overgave/ overname was er niet maar met de nagelaten 3 boeken met formules van de *London Polytechnic School* aangaande de *Least Square Calculus* het boek *LaTeX A Document Preparation System* en twee boeken over PostScript zou alles duidelijk worden. Toch?

In zee met de NTG

Eind 1989 namen we met een drietal collega's van het CAWCS deel aan een NTG bijeenkomst en werd ik lid van de vereniging. Ik herinner me nog heel goed de reactie van Kees van der Laan, toenmalig voorzitter, nadat mijn collega Leo van Geest hem toelichtte waarom LaTeX ons van nut was: *'Oh... en u gebruikt het ook nog...?'* Daar stonden we wel even van versteld eerlijk gezegd...[3]

Inmiddels was ik al behoorlijk geïntrigeerd (eerder geïnfecteerd) door LaTeX en via Gerard van Nes bleek een DOS distributie voorhanden (EmTeX) zodat ik ook thuis aan de slag kon! Om de versie aan de praat te krijgen moesten er wel een sloot diskettes ingelezen worden en enige bijkomende PC know how was geen luxe. Niettemin, met mijn Atari 286 AT en Citizen 24 naalds printer kon ik mij verder verdiepen in de eindeloos lijkende mogelijkheden van LaTeX zoals het gebruik van diverse fonts, het invoegen en scalen van figuren, PostScript, enz. Genoeg om talrijke avonden en week-ends zoet te zijn en het zaakje steeds beter en beter in de vingers te krijgen. De verslaving was inmiddels een feit...

Voor zij dit het zich nog kunnen herinneren: het credo was indertijd 'What You See is What You Get' (LaTEX) en 'What You see is All You've Got (WP 5.0 en Word). Een waarheid als een koe toendertijd.

Franse adelborsten

Na het CAWCS werd ik in 1992 in België teruggeplaatst. Het was net alsof ik op IT vlak van Star Trek teruggeflitst werd naar de Flinstones…Eerst kreeg ik een oud stalen bureau toegewezen met een zwarte bakelieten telefoon (overigens niet aangesloten!) zonder PC. Wat later werden weliswaar de eerste PC's geleverd met Windows als OS maar zonder bijbehorende muis, want (en dit is géén grap) de hoofdverantwoordelijke voor de aankoopdienst IT binnen de marine 'was tegen muizen' (sic). 't Kan verkeren zei Bredero…

Thuis had ik wel het nodige materiaal en ik heb me toen een tijd onledig gehouden met de opmaak van een maandelijks schaakblad en de opmaak van een paar uitgaven van de uitgeverij ACCO voor de Universiteit Leuven.

Vanaf '94 werd ik verbindingofficier en leraar 'Maritime English' bij de Franse Ecole navale te Lanvéoc in Bretagne. Geografische vrijgezel zijnde en slechts beschikkende over een 20-tal oude slides als lesmateriaal heb ik daar heel wat ervaring opgedaan in het opmaken van documenten met LaTEX. Na een jaar zwoegen had ik tweetalig (Frans/Engels) handboek van ruim 250 blz klaar met aangepaste teksten, tekeningen en woordenlijst, aangemaakt met LaTEX2ϵ en afgedrukt op een HP 600dpi PS printer[4]. Dit bleek achteraf een goede voorbereiding voor latere opdrachten.

Foute belgenmop…

Terug in belgenland medio '97 kon ik me, na het commando over een fregat, terug wat actiever met de NTG bezighouden. Een aantal jaren was ik o.a. de 'belgische' secretaris, kwestie van de lidgelden aldaar te innen en die dan in haar geheel over te boeken naar Nederland en zodoende de bankkosten hiervoor minimaal te houden. Hiervoor had ik een rekening geopend bij de Postbank in België (achteraf bekeken niet de beste keuze overigens) voor de *Nederlandstalige TEX vereniging.* Alle documenten keurig ingevuld bij de kantoorverantwoordelijk en enkele dagen later kreeg ik alle documenten (overschrijvingen e.a.) in de bus op naam van de…*Nederlandstalige ReX vereniging.*[5]

Niet echt een vereniging waar je als hoger marineofficier mee wil geassocieerd wil worden dus, tenzij je je absoluut bij de nationale veiligheidsdienst in de kijker wil werken en – op zijn minst – je veiligheidscertificaten tot confetti wenst te zien versnipperen. Het heeft me 2 weken gekost om dit recht te zetten en gelukkig zijn er in Brussel geen knipperlichten afgegaan over de korstondige 'revival' van ReX in België!

Gedurende enkele jaren heb ik dan gependeld naar de Staf Defensie te Brussel. Geen oord waar aan typografische kwaliteit van documenten enig belang werd gegeven. Sinds 2002 dienden alle interne documenten overigens verplicht opgemaakt te worden met het font... Comic sans serif! Inderdaad u leest dit goed: *Comic sans serif…* zoals beslist door een bende, duidelijk ver over houdbaarheidsdatum zijnde, kolonels. Dit font, weliswaar redelijk geschikt voor de aankondiging van een bingo-avond of een fuif voor 12-jarigen is, tot op heden, het standaard font voor alle documenten binnen de belgische defensie. Ik heb nog andere horrorverhalen inzake typografie uit die periode maar om mezelf (en u) te sparen zal ik er hier niet verder op ingaan.

Goede herinneringen heb ik wel aan de deelnames aan de NTG dagen, steevast in aangenaam gezelschap van Luc Deconinck († 2007). Vaak was de trip naar Nederland moeilijk haalbaar vermits de bijeenkomst steevast op een donderdag gepland werd wat niet echt praktisch was, overnachten achteraf was niet mogelijk gezien we vrijdag weer werken moesten ofwel twee dagen verlof dienden op te offeren. Waarom telkens

het telkens op een donderdag moest plaatsvinden is mij overigens nooit duidelijk geworden…

Verder heb ik mij tot eind 2004 intens bezig gehouden met het zetten van boeken en tijdschriften allerhande met – meestal – wiskundige achtergrond en dit voor diverse opdrachtgevers. Hoe moeilijker hoe liever ik het had omdat dit telkens een nieuwe uitdaging was om het met LaTeX toch voor mekaar te krijgen. Leuke tijd maar op de duur toch iets te intens om goed te zijn, een beetje vrije tijd heeft ook zo zijn charmes uiteindelijk.

Epiloog

Begin 2005 werd ik bij de Franse marine geplaatst te Toulon, niet dat ik continu van de Franse rivièra kon genieten want deze plaatsing impliceerde frequent langere periodes op zee als stafofficier met de *Charles de Gaule, Mistral, Tonnerre* e.a. Nadien, medio 2010 verkaste ik naar het NATO hoofdkwartier te Londen. Drukke tijden met weinig ruimte voor enige typografische recreatie Inmiddels had ik ook een huis gekocht in ZW Frankrijk zodat we regelmatig pendelden tussen beide locaties. In 2013 heb ik de marine verlaten en ben dan definitief met mijn eega in Frankrijk gaan wonen, even ten zuiden van Toulouse. Eventuele deelname aan NTG dagen werd door de te overbruggen afstand en bijbehorende kost hierbij nog minder voor de hand liggend natuurlijk. Sindsdien gebruik ik TeXworks voor mijn huis- tuin- en keuken correspondentie en ja, het blijft nog steeds leuk.

Tot besluit

Als lid nr 150 van de NTG ben ik altijd een 'gebruiker' gebleven, geen 'ontwikkelaar' (waar ik overigens veel respect en veel aan te danken heb). De gestage evolutie binnen de NTG van de 'artisanale' LaTeX gebruiker naar de meer gesofistikeerde ConTeXt aanhanger heeft me echter nooit echt kunnen boeien. Niettemin ben ik tot en met 2018 lid gebleven, niet langer puur uit interesse voor verdere ontwikkelingen, maar enkel en alleen als (kleine) financiële steun voor de NTG.

Tevens is een der initiële doelstellingen van de NTG ruimschoots bereikt; daar waar in 1989 gevreesd werd dat het gebruik van TeX en LaTeX wegdeemsterde en op sterven na dood was (o.a. met de opkomst van Word en WordPerfect) is het aantal gebruikers sindsdien exponentieel gestegen op universiteiten, hogescholen e.a., zodoende…

Ah, scheiden doet altijd een beetje lijden, maar de link die ons al die jaren verbonden heeft die blijft natuurlijk; net als de talrijke handboeken inzake LaTeX, PostScript, typografie e.a. die hier binnen handbereik blijven, zoveel is zeker!

Rest mij hier dus nog enkel aan de NTG en haar leden van harte *'fair winds and following seas'* te wensen!

Notes

1. Centrum voor Automatisering van Wapen- en CommandoSystemen, KM.
2. Mine Data and Information System - Positiebepaling en databeheer van onderwaterobjecten t.b.v. de Mijnendienst.
3. Zie ook MAPS 6, 6de bijeenkomst 20 nov 1990
4. Monnikenwerk was het maar uiteindelijk is het tweemaal in herdruk gegaan in de Ecole navale en toen ik ettelijke jaren later in 2010 ingescheept was aan boord van de FS Mistral ontmoette ik iemand die een exemplaar gebruikte ter voorbereiding van zijn examen Engels.
5. Rex (van Christus Rex, *Christus is koning*) was een Belgische fascistische politieke beweging(1930 – 1945) onder leiding van Léon Degrelle (tevens notoir lid van de Waffen-SS). De beweging werd in 1945 van staatswege verboden.

Philippe Vanoverbeke

Performance again

Introduction

In a MAPS article of 2019 I tried to answer the question 'Is TEX really slow?'. A while after it was published on the Dutch TEX mailing list a user posted a comment stating that in his experience the LUATEX engine in combination with LATEX was terribly slow: one page per second for a Japanese text. It was also slower than PDFTEX with English, but for Japanese it was close to unusable. The alternative, using a Japanese TEX engine was no option due to lack of support for certain images.

In order to check this claim I ran a test in CONTEXT. Even on my 8 year old laptop I could get 45 pages per second for full page Japanese texts (6 paragraphs with each 300 characters per page): 167 pages took just less than 4 seconds. Typesetting Japanese involves specific spacing and line break handling. So, naturally the question arises: why the difference. Frans Goddijn wondered if I could explain a bit more about that, so here we go.

In the mentioned article I already have explained what factors play a role and the macro package is one of them. It is hard to say to what extent inefficient macros or a complex layout influence the runtime, but my experience is that it is pretty hard to get speeds as low as 1 page per second. On an average complex document like the LUATEX manual (lots of verbatim and tables, but nothing else demanding apart from color being used and a unique MetaPost graphic per page) I get at least a comfortable 20 pages per second.

I can imagine that for a TEX user who sees other programs on a computer do complex things fast, the performance of TEX is puzzling. But, where for instance rendering videos can benefit from specific features of (video) processors, multiple cores, or just aggressive optimization by compilers of (nested) loops and manipulation of arrays of bytes, this is not the case for TEX. This program processes all in sequence, there is not much repetition that can be optimized, it cannot exploit the processor in special ways and the compiler can not do that many optimizations.

I can't answer why a LATEX run is slower than a CONTEXT run. Actually, one persistent story has always been that CONTEXT was slow in comparison. But maybe it helps to know a bit what happens deep down in TEX and how macro code can play a role in performance. When doing that I will simplify things a bit.

Text and nodes

The TEX machinery takes input and turns that into some representation that can be turned into a visual representation ending up as PDF. So say that we have this:

```
hello
```

In a regular programming language this is a string with five characters. When the string is manipulated it is basically still a sequence of bytes in memory. In TEX, if this is meant as text, at some point the internal representation is a so called node list:

```
[h] -> [e] -> [l] -> [l] -> [o]
```

In traditional TEX these are actually character nodes. They have a few properties, like what font the character is from and what the character code is (0 up to 255). At some point TEX will turn that list into a glyph list. Say that we have this:

```
efficient
```

This will eventually become seven nodes:

```
[e] -> [ffi] -> [c] -> [i] -> [e] -> [n] -> [t]
```

The ffi ligature is a glyph node which actually also keeps information about this one character being made from three.

In LUATEX it is different, and this is one of the reasons for it being slower. We stick to the first example:

```
[h] <-> [e] <-> [l] <-> [l] <-> [o]
```

So, instead of pointing to the next node, we also point back to the previous: we have a double linked list. This means that all over the program we need to maintain these extra links too. They are not used by TEX itself, but handy at the LUA end. But, instead of only having the font as property there is much more. The TEX program can deal with multiple languages at the same time and this relates to hyphenation. In traditional TEX there are language nodes that indicate a switch to another language. But in LUATEX that property is kept with each glyph node. Actually, even specific language properties like the hyphen min, hyphen max and the choice if

uppercase should be hyphenated are kept with these nodes. Spaces are turned into glue nodes, and these nodes are also larger than in regular TEX engines.

So, in LuaTEX, when a character goes from the input into a node, a more complex data structure has to be set up and the larger data structure also takes more memory. That in turn means that caching (close to the CPU) gets influenced. Add to that the fact that we operate on 32 bit character values, which also comes with higher memory demands.

We mentioned that a traditional engine goes from one state of node list into another (the ligature building). Actually this is an integrated process: a lot happens on the fly. If something is put into a \hbox no hyphenation takes place, only ligature building and kerning. When a paragraph is typeset, hyphenation happens on demand, in places where it makes sense.

In LuaTEX these stages are split. A node list is *always* hyphenated. This step as well as ligature building and kerning are *three* separate steps. So, there's always more hyphenation going on than in a traditional TEX engine: we get more discretionary nodes and again these take more memory than before; also the more nodes we have, the more it will impact performance down the line. The reason for this is that each step can be intercepted and replaced by a Lua driven one. In practice, with modern OpenType fonts that is what happens: these are dealt with (or at least managed in) Lua. For Japanese for sure the built-in ligature and kerning doesn't apply: the work is delegated and this comes at a price. Japanese needs no hyphenation but instead characters are treated with respect to their neighbors and glue nodes are injected when needed. This is something that Lua code is used for so here performance is determined by how well the plugged in code behaves. It can be inefficient but it can also be so clever that it just takes a bit of time to complete.

I didn't mention another property of nodes: attributes. Each node that has some meaning in the node list (glyphs, kerns, glue, penalties, discretionary, ..., these terms should ring bells for a TEX user) have a pointer to an attribute list. Often these are the same for neighboring nodes, but they can be different. If a macro package sets 10 attributes, then there will be lists of ten attributes nodes (plus some overhead) active. When values change, copies are made with the change applied. Grouping even complicates this a little more. This has an impact on performance. Not only need these lists be managed, when they are consulted at the Lua end (as they are meant as communication with that bit of the engine) these lists are interpreted. It all adds up to more runtime. There is nothing like that in traditional TEX, but there some more macro juggling to achieve the same effects can cause a performance hit.

Macros and tokens

When you define macro like this:

```
\def\MyMacro#1{\hbox{here: #1!}}
```

the TEX engine will parse this as follows (we keep it simple):

\def	primitive token
\MyMacro	user macro pointing to:
#1	argument list of length 1 and no delimiters
{	openbrace token
\hbox	hbox primitive token
h	letter token h
e	letter token e
r	letter token r
e	letter token e
:	other token :
	space token
#1	reference to argument
!	other token !
}	close brace token

The \def is eventually lost, and the meaning of the macro is stored as a linked list of tokens that get bound to the user macro \MyMacro. The details about how this list is stored internally can differ a bit per engine but the idea remains. If you compare tokens of a traditional TEX engine with LuaTEX, the main difference is in the size: those in LuaTEX take more memory and again that impacts performance.

Processing

Now, for a moment we step aside and look at a regular programming language, like Pascal, the language TEX is written in, or C that is used for LuaTEX. The high level definitions, using the syntax of the language, gets compiled into low level machine code: a sequence of instructions for the CPU. When doing so the compiler can try to optimize the code. When the program is executed all the CPU has to do is fetch the instructions, and execute them, which in turn can lead to fetching data from memory. Successive versions of CPU's have become more clever in handling this, predicting what might happen, (pre) fetching data from memory etc.

When you look at scripting languages, again a high level syntax is used but after interpretation it becomes compact so called byte-code: a sequence of instructions for a virtual machine that itself is a compiled program. The virtual machine fetches the bytes and acts upon them. It also deals with managing memory and variables. There is not much optimization going on there, certainly not when the language permits dynamically changing function calls and such. Here performance is not only influenced by the virtual machine but also by the quality of the original code (the scripts). In LuaTEX

we're talking Lua here, a scripting language that is actually considered to be pretty fast.

Sometimes byte-code can be compiled Just In Time into low level machine code but for LuaTEX that doesn't work out well. Much Lua code is executed only once or a few times so it simply doesn't pay off. Apart from that there are other limitations with this (in itself impressive) technology so I will not go into more detail.

So how does TEX work? It is important to realize that we have a mix of input and macros. The engine interprets that on the fly. A character enters the input and TEX has to look at it in the perspective of what it what it expects. Is is just a character? Is it part of a control sequence that started (normally) with a backslash? Does it have a special meaning, like triggering math mode? When a macro is defined, it gets stored as a linked list of tokens and when it gets called the engine has to expand that meaning. In the process some actions themselves kind of generate input. When that happens a new level of input is entered and further expansion takes place. Sometimes TEX looks ahead and when not satisfied, pushes something back into the input which again introduces a new level. A lot can happen when a macro gets expanded. If you want to see this, just add `\tracingall` at the top of your file: you will be surprised! You will not see how often tokens get pushed and popped but you can see how much got expanded and how often local changes get restored. By the way, here is something to think about:

```
\count4=123
\advance \count4 by 123
```

If this is in your running text, the scanner sees `\count` and then triggers the code that handles it. That code expects a register number, here that is the `4`. Then it checks if there is an optional `=` which means that it has to look ahead. In the second line it checks for the optional keyword `by`. This optional scanning has a side effect: when the next token is *not* an equal or keyword, it has to push back what it just read (we enter a new input level) and go forward. It then scans a number. That number ends with a space or `\relax` or something not being a number. Again, some push back onto the input can happen. In fact, say that instead of `4` we have a macro indicating the register number, intermediate expansion takes place. So, even these simple lines already involve a lot of action! Now, say that we have this

```
\scratchcounter 123
\scratchcounter =123
\advance\scratchcounter by 123
\advance\scratchcounter 123
```

Can you predict what is more efficient? If this doesn't happen a lot performance wise there is no real difference because TEX is pretty fast in doing this, but given what we said before, adding the equal sign and `by` *could* actually be faster because there is no pushing back onto the input stack involved. It probably makes no sense to take this into account when writing macros but just keep in mind that performance is in the details.

This model of expansion is very different from compiled code or byte-code. To some extent you can consider a list of tokens that make up a macro to be byte-code, but instead of a sequence of bytes it is a linked list. That itself has a penalty in performance. Depending on how macros expand, the engine can be hopping all over the token memory following that list. That means that quite likely the data that gets accessed is not in your CPU cache and as a result performance cannot benefit from it apart of course from the expanding machinery itself, but that one is not a simple loop messing around with variables: it accesses code all over the place! Text gets hyphenated, fonts get applied, material gets boxed, paragraphs constructed, pages built. We're not moving a blob of bits around (as in a video) but we're constantly manipulating small amounts of memory scattered around memory space.

Now, where a traditional TEX engine works on 8 bit characters and smaller tokens, the 32 bit LuaTEX works on larger chunks. Although macro names are stored as single symbolic units, there are times when its real name is used, for instance when the `\csname` primitive is used. At that time, the real name is used and there are plenty cases where temporary string variables are allocated and filled. Compare:

```
\def\foo{\hello}
```

Here the macro `\foo` has just a one token reference to `\hello` because that's how a macro reference gets stored. But in

```
\def\foo{\csname hello\endcsname}
```

we have two plus five tokens to access what effectively is `\hello`. Each character token has to be converted to a byte into the assembled string. Now it must be said that in practice this is still pretty fast but when we have longer names and especially when we have UTF8 characters in there it can come at a price. It really depends on how your macro package works and sometimes you just pay the price of progress. Buying a faster machine is then the solution because often we're not talking of extreme performance loss here. And modern CPU's can juggle bytes quite efficiently. Actually, when we go to 64 bit architectures, LuaTEX's data structures fit quite well to that. As a side note: when you run a 32 bit binary on a 64 bit architecture there can even be a price being paid for that when you use LuaTEX. Just move on!

Management

Before we can even reach the point that some content becomes typeset, much can happen: the engine has to start up. It is quite common that a macro package uses a memory dump so that macros are not to be parsed each run. In traditional engines hyphenation patterns are stored in the memory dump as well. And some macro packages can put fonts in it. All kind of details, like upper- and lowercase codes can get stored too. In LuaTEX fonts and patterns are normally kept out of the dump. That dump itself is much larger already because we have 32 bit characters instead of 8 bit so more memory is used. There are also new concepts, like catcode tables that take space. Math is 32 bit too, so more codes related to math are stored. Actually the format is so much larger that LuaTEX compresses it. Anyway, it has an impact on startup time. It is not that much, but when you measure differences on a one page document the overhead in getting LuaTEX up and running will definitely impact the measurement.

The same is true for the backend. A traditional engine uses (normally) Type1 fonts and LuaTEX relies on OpenType. So, the backend has to do more work. The impact is normally only visible when the document is finalized. There can be a slightly larger hickup after the last page. So, when you measure one page performance, it again pollutes the page per second performance.

Summary

So, to come back to the observation that LuaTEX is slower that pdfTEX. At least for ConTEXt we can safely conclude that indeed pdfTEX is faster when we talk about a standard English document, with TEX ascii input, where we can do with traditional small fonts, with only some kerning and simple ligatures. But as soon as we deal with for instance xml, have different languages and scripts, have more demanding layouts, use color and images, and maybe even features that we were not aware of and therefore didn't require in former times the LuaTEX engine (and for ConTEXt it's LuaMetaTEX follow up) performs way better than pdfTEX. So, there is no simple answer and explanation for the fact that the observed slow LATEX run on Japanese text, apart from that we can say: look at the whole picture: we have more complex tokens, nodes, scripts and languages, fonts, macros, demands on the machinery, etc. Maybe it is just the price you are paying for that.

Hans Hagen
Hasselt NL
Februari 2020

All those TEX's

This is about TEX, the program that is used as part of the large suite of resources that make up what we call a 'TEX distribution', which is used to typeset documents. There are many flavors of this program and all end with tex. But not everything in a distribution that ends with these three characters is a typesetting program. For instance, latex launches the a macro package LATEX, code that feeds the program tex to do something useful. Other formats are Plain (no tex appended) or ConTEXt (tex in the middle. Just take a look at the binary path of the TEX distribution to get an idea. When you see pdftex it is the program, when you see pdflatex it is the macro package LATEX using the pdfTEX program. You won't find this for ConTEXt as we don't use that model of mixing program names and macro package names.

Here I will discuss the programs, not the macro packages that use them. When you look at a complete TEXLive installation, you will see many TEX binaries. (I will use the verbatim names to indicate that we're talking of programs). Of course there is the original tex. Then there is its also official extended version etex, which is mostly known for adding some more primitives and more registers. There can be aleph, which is a stable variant of omega meant for handling more complex scripts. When pdf became popular the pdftex program popped up: this was the first TEX engine that has a backend built in. Before that you always had to run an additional program to convert the native dvi output of TEX into for instance PostScript. Much later, xetex showed up, that, like Omega, dealt with more complex scripts, but using recent font technologies. Eventually we saw luatex enter the landscape, an engine that opened up the internals with the Lua script subsystem; it was basically a follow up on pdftex and aleph.

The previous paragraph mentions a lot of variants and there are plenty more. For cjk and especially Japanese there are ptex, eptex, uptex, euptex. Parallel to luatex we have luajittex and luahbtex. As a follow up on the (presumed stable) luatex the ConTEXt community now develops luametatex. A not yet mentioned side track is nts (New TEX system), a rewrite of good old TEX in Java, which in the end didn't take off and was never really used.

There are even more TEX's and they came and went. There was enctex which added encoding support, there were emtex and hugeemtex that didn't add functionality but made more possible by removing some limits on memory and such; these were quite important. Then there were vendors of TEX systems that came up with variants (some had extra capabilities), like microtex, pctex, yandytex and vtex but they never became part of the public effort.

For sure there are more, and I know this because not so long ago, when I cleaned up some of my archives, I found eetex (extended ε-TEX), and suddenly remembered that Taco Hoekwater and I indeed had experimented with some extensions that we had in mind but that never made it into ε-TEX. I had completely forgotten about it, probably because we moved on to LuaTEX. It is the reason why I wrap this up here.

In parallel there have been some developments in the graphic counterparts. Knuts metafont program got a Lua enhanced cousin mflua while metapost (aka mpost or mp) became a library that is embedded in LuaTEX (and gets a follow up in LuaMetaTEX). I will not discuss these here.

If we look back at all this, we need to keep in mind that originally TEX was made by Don Knuth for typesetting his books. These are in English (although over time due to references he needed to handle different scripts than Latin, be it just snippets and not whole paragraphs). Much development of successors was the result of demands with respect to scripts other than Latin and languages other than English. Given the fact that (at least in my country) English seems to become more dominant (kids use it, universities switch to it) one can wonder if at some point the traditional engine can just serve us as well.

The original tex program was actually extended once: support for mixed usage of multiple languages became possible. But apart from that, the standard program has been pretty stable in terms of functionality. Of course, the parts that made the extension interface have seen changes but that was foreseeable. For instance, the file system hooks into the kpse library and one can execute programs via the \write command. Virtual font technology was also an extension but that didn't require a change in the program but involved postprocessing the dvi files.

The first major 'upgrade' was ε-TEX. For quite a while extensions were discussed but at some point the first version became available. For me, once pdfTEX incorporated these extensions, it became the default. So what did it bring? First of all we got more than 256 registers (counters, dimensions, etc.). Then there are some extra primitives, for instance `\protected` that permits the definition of unexpandable macros (although before that one could simulate it at the cost of some overhead) and convenient ways to test the existence of a macro with `\ifdefined` and `\ifcsname`. Although not strictly needed, one could use `\dimexpr` for expressions. A probably seldom used extension was the (paragraph bound) right to left typesetting. That actually is a less large extension than one might imagine: we just signal where the direction changes and the backend deals with the reverse flushing. It was mostly about convenience.

The Omega project (later followed up by Aleph) didn't provide the additional programming related primitives but made the use of wide fonts possible. It did extend the number of registers, just by bumping the limits. As a consequence it was much more demanding with respect to memory. The first time I heard of ε-TEX and Omega was at the 1995 euroTEX meeting organized by the ntg and I was sort of surprised by the sometimes emotional clash between the supporters of these two variants. Actually it was the first time I became aware of TEX politics in general, but that is another story. It was also the time that I realized that practical discussions could be obscured by nitpicking about speaking the right terminology (token, node, primitive, expansion, gut, stomach, etc.) and that one could best keep silent about some issues.

The pdfTEX follow up had quite some impact: as mentioned it had a backend built in, but it also permitted hyperlinks and such by means of additional primitives. It added a couple more, for instance for generating random numbers. But it actually was a research project: the frontend was extended with so called character protrusion (which lets glyphs hang into the margin) and expansion (a way to make the output look better by scaling shapes horizontally). Both these extensions were integrated in the paragraph builder and are thereby extending core code. Adding some primitives to the macro processor is one thing, adapting a very fundamental property of the typesetting machinery is something else. Users could get excited: TEX renders a text even better (of course hardly anyone notices this, even TEX users, as experiments proved).

In the end Omega never took off, probably because there was never a really stable version and because at some time XETEX showed up. This variant was first only available on Apple computers because it depends on third party libraries. Later, ports to other systems showed up. Using libraries is not specific for XETEX. For instance pdfTEX uses them for embedding images. But, as that is actually a (backend) extension it is not critical. Using libraries in the frontend is more tricky as it adds a dependency and the whole idea about TEX was that is is independent. The fact that after a while XETEX switched libraries is an indication of this dependency. But, if a user can live with that, it's okay. The same is true for (possibly changing) fonts provided by the operating system. Not all users care too strongly about long term compatibility. In fact, most users work on a document, and once finished store some pdf copy some place and then move on and forget about it.

It must be noted that where ε-TEX has some limited right to left support, Omega supports more. That has some more impact on all kinds of calculations in the machinery because when one goes vertical the width is swapped with the height/depth and therefore the progression is calculated differently.

Naturally, in order to deal with scripts other than Latin, XETEX did add some primitives. I must admit that I never looked into those, as ConTEXt only added support for wide fonts. Maybe these extensions were natural for LATEX, but I never saw a reason to adapt the ConTEXt machinery to it, also because some pdfTEX features were lacking in XETEX that ConTEXt assumed to be present (for the kind of usage it is meant for). But we can safely say that the impact of XETEX was that the TEX community became aware that there were new font technologies that were taking over the existing ones used till now. One thing that is worth noticing is that XETEX is still pretty much a traditional TEX engine: it does for instance OpenType math in a traditional TEX way. This is understandable as one realizes that the OpenType math standard was kind of fuzzy for quite a while. A consequence is that for instance the OpenType math fonts produced by the Gust foundation are a kind of hybrid. Later versions adopted some more pdfTEX features like expansion and protrusion.

I skip the Japanese TEX engines because they serve a very specific audience and provide features for scripts that don't hyphenate but use specific spacing and line breaks by injecting glues and penalties. One should keep in mind that before Unicode all kinds of encodings were used for these scripts and the 256 limitations of traditional TEX were not suited for that. Add to that demands for vertical typesetting and it will be clear that a specialized engine makes sense. It actually fits perfectly in the original idea that one could extend TEX for any purpose. It is a typical example of where one can argue that users should switch to for instance XETEX or LuaTEX but these were not available and therefore there is no reason to ditch a good working system just because some new (yet unproven) alternative shows up a while later.

We now arrive at LuaTEX. It started as an experiment in 2005 where a Lua interpreter was added to pdfTEX. One could pipe data into the TEX machinery and query some properties, like the values of registers. At some point the project sped up because Idris Hamid got involved. He was one of the few ConTEXt users who used Omega (which it actually did support to some extent) but he was not satisfied with the results. His oriental TEX project helped pushing the LuaTEX project forward. The idea was that by opening up the internals of TEX we could do things with fonts and paragraph building that were not possible before. The alternative, XƎTEX was not suitable for him as it was too bound to what the libraries provides (rendering then depends on what library gets used and what is possible at what time). But, dealing with scripts and fonts is just one aspect of LuaTEX. For instance more primitives were added and the math machinery got an additional OpenType code path. Memory constraints were lifted and all became Unicode internally. Each stage in the typesetting process can be intercepted, overloaded, extended.

Where the ε-TEX and Omega extensions were the result of many years of discussion, the pdfTEX, XƎTEX and LuaTEX originate in practical demands. Very small development teams that made fast decisions made that possible.

Let's give some more examples of extensions in LuaTEX. Because pdfTEX is the starting point there is protrusion and expansion, but these mechanisms have been promoted to core functionality. The same is true for embedding images and content reuse: these are now core features. This makes it possible to implement them more naturally and efficiently. All the backend related functionality (literal pdf, hyperlinks, etc) is now collected in a few extension primitives and the code is better isolated. This took a bit of effort but is in my opinion better. Support for directions comes from Omega and after consulting with its authors it was decided that only four made sense. Here we also promoted the directionality to core features instead of extensions. Because we wanted to serve Omega users too extended tfm fonts can be read, not that there are many of them, which fits nicely into the whole machinery going 32 instead of 8 bits. Instead of the ε-TEX register model, where register numbers larger than 255 were implemented differently, we adopted the Omega model of just bumping 256 to 65536 (and of course, 16K would have been sufficient too but the additional memory it uses can be neglected compared to what other programs use and/or what resources users carry on their machines).

The modus operandi for extending TEX is to take the original literate web sources and define change files. The pdfTEX program already deviated from that by using a monolithic source. But still Pascal is used for the body of core code. It gets translated to C before being compiled. In the LuaTEX project Taco Hoekwater took that converted code and laid the foundation for what became the original LuaTEX code base.

Some extensions relate to the fact that we have Lua and have access to TEX's internal node lists for manipulations. An example is the concept of attributes. By setting an attribute to a value, the current nodes (glyphs, kerns, glue, penalties, boxes, etc) get these as properties and one can query them at the Lua end. This basically permits variables to travel with nodes and act accordingly. One can for instance implement color support this way. Instead of injecting literal or special nodes that themselves can interfere we now can have information that does not interfere at all (apart from maybe some performance hit). I think that conceptually this is pretty nice.

At the Lua one has access to the TEX internals but one can also use specific token scanners to fetch information from the input streams. In principle one can create new primitives this way. It is always a chicken-egg question what works better but the possibility is there. There are many such conceptual additions in LuaTEX, which for sure makes it the most 'aggressive' extension of TEX so far. One reason for these experiments and extensions is that Lua is such a nice and suitable language for this purpose.

Of course a fundamental part of LuaTEX is the embedded MetaPost library. For sure the fact that ConTEXt integrates MetaPost has been the main reason for that.

The ConTEXt macro package is well adapted to LuaTEX and the fact that its users are always willing to update made the development of LuaTEX possible. However, we are now in a stage that other macro packages use it so LuaTEX has entered a state where nothing more gets added. The LATEX macro package now also supports LuaTEX, although it uses a variant that falls back on a library to deal with fonts (like XƎTEX does).

With LuaTEX being frozen (of course bugs will be fixed), further exploration and development is now moved to LuaMetaTEX, again in the perspective of ConTEXt. I will not go into details apart from saying that is is a lightweight version of LuaTEX. More is delegated to Lua, which already happened in ConTEXt anyway, but also some extra primitives were added, mostly to enable writing nicer looking code. However, a major aspect is that this program uses a lean and mean code base, is supposed to compile out of the box, and that sources will be an integral part of the ConTEXt code base, so that users are always in sync.

So, to summarize: we started with `tex` and moved on to `etex` and `pdftex`. At some point `omega` and `xetex` filled the Unicode and script gaps, but it now looks

like `luatex` is becoming popular. Although `luatex` is the reference implementation, LaTeX exclusively uses `luahbtex`, while ConTeXt has a version that targets at `luametatex`. In parallel, the `[e][u][p]tex` engines fill the specific needs for Japanese users. In most cases, good old `tex` and less old `etex` are just shortcuts to `pdftex` which is compatible but has the PDF backend on board. That 8 bit engine is not only faster than the more recent engines, but also suits quite well for a large audience, simply because for articles, thesis, etc. (written in a Latin script, most often English) it fits the bill well.

I deliberately didn't mention names and years as well as detailed pros and cons. A user should have the freedom to choose what suits best. I'm not sure how well TeX would have evolved or will evolve in these days of polarized views on operating systems, changing popularity of languages, many (also open source) projects being set up to eventually be monetized. We live in a time where so called influencers play a role, where experience and age often matters less than being fancy or able to target audiences. Where something called a standard today is replaced quickly by a new one tomorrow. Where stability and long term usage of a program is only a valid argument for a few. Where one can read claims that one should use this or that because it is todays fashion instead of the older thing that was the actually the only way to achieve something at all a while ago. Where a presence on facebook, twitter, instagram, whatsapp, stack exchange is also an indication of being around at all. Where hits, likes, badges, bounties all play a role in competing and self promotion. Where today's standards are tomorrow's drawbacks. Where even in the TeX community politics seem to creep in. Maybe you can best not tell what is your favorite TeX engine because what is hip today makes you look out of place tomorrow.

Hans Hagen
February 2020

Hidden treasures

At ConTEXt meetings we always find our moments to reflect on the interesting things that relate to TEX that we have run into. Among those we discussed were some of the historic treasures one can run into when one looks at source files. I will show examples from several domains in the ecosystem and we hereby invite the reader to come up with other interesting observations, not so much in order to criticize the fantastic open source efforts related to TEX, but just to indicate how decades of development and usage are reflected in the code base and usage, if only to make it part of the history of computing.

I start with the plain TEX format. At the top of that file we run into this:

```
% The following changes define internal codes as recommended
% in Appendix C of The TeXbook:

\mathcode`\^^@="2201 % \cdot
\mathcode`\^^A="3223 % \downarrow
\mathcode`\^^B="010B % \alpha
\mathcode`\^^C="010C % \beta
\mathcode`\^^D="225E % \land
...
\mathcode`\^^Y="3221 % \rightarrow
\mathcode`\^^Z="8000 % \ne
\mathcode`\^^[="2205 % \diamond
\mathcode`\^^\="3214 % \le
\mathcode`\^^]="3215 % \ge
\mathcode`\^^^="3211 % \equiv
\mathcode`\^^_="225F % \lor
```

This means that when you manage to key in one of these recommended character codes that in ASCII sits below the space slot, you will get some math symbol, given that you are in math mode. Now, if you also consider that the plain TEX format is pretty compact and that no bytes are wasted,[1] you might wonder what these lines do there. The answer is simple: there were keyboards out there that had these symbols. But, by the time TEX became popular, the dominance of the IBM keyboard let those memories fade away. This is just Don's personal touch I guess. Of course the question remains if the sources of TAOCP contain these characters.

1. Such definitions don't take additional space in the format file.

There is another interesting hack in the plain TEX file, one that actually, when I first looked at the file, didn't immediately made sense to me.

```
\font\preloaded=cmti9
\font\preloaded=cmti8
\font\preloaded=cmti7

\let\preloaded=\undefined
```

What happens here is that a bunch of fonts get defined and they all use the same name. Then eventually that name gets nilled. The reason that these definitions are there is that when TEX dumps a format file, the information that comes from those fonts is embedded to (dimensions, ligatures, kerns, parameters and math related) data. It is an indication that in those days it was more efficient to have them preloaded (that is why they use that name) than loading them at runtime. The fonts are loaded but you can only access them when you define them again! Of course nowadays that makes less sense, especially because storage is fast and operating systems do a nice job at caching files in memory so that successive runs have font files available already.

Talking of fonts, one of the things a new TEX user will notice and also one of the things users love to brag about is ligatures. If you run the `tftopl` program on a file like `cmr10.tfm` you will get a verbose representation of the font. Here are some lines:

```
(LABEL C f)  (LIG C i O 14) (LIG C f O 13) (LIG C l O 15)
(LABEL O 13) (LIG C i O 16) (LIG C l O 17)
(LABEL C `)  (LIG C ` C \)
(LABEL C ')  (LIG C ' C ")
(LABEL C -)  (LIG C - C {)
(LABEL C {)  (LIG C - C |)
(LABEL C !)  (LIG C ` C <)
(LABEL C ?)  (LIG C ` C >)
```

The `C` is followed by an ASCII representation and the `)` by the position in the font `O` (a number) or `C` (a character). So, consider the first two lines to be a puzzle: they define the fi, ff, fl ligatures as well as the ffi and ffl ones. Do you see how ligatures are chained?

But anyway, what do these other lines do there? It looks like `` ` ` `` becomes the character in the backslash slot and `''` the one in the double quote. Keep in mind that TEX treats the backslash special and when you want it, it will be taken from elsewhere. But still, these two liga-

tures look familiar: they point to slots that have the left and right double quotes.[2] They are not really ligatures but abuse the ligature mechanism to achieve a similar effect. The last four lines are the most interesting: these are ligatures that (probably) no TEX user ever uses or encounters. They are again something from the past. Also, changes are low that you mistakenly enter these sequences and the follow up Latin Modern fonts don't have them anyway.

Actually, if you look at the Metafontand MetaPost sources you can find lines like these (here we took from `mp.w` in the LuaTEX repository):

```
@ @<Put each...@>=
mp_primitive (mp, "=:", mp_lig_kern_token, 0);
@:=:_}{\.{=:} primitive@>;
mp_primitive (mp, "=:|", mp_lig_kern_token, 1);
@:=:/_}{\.{=:\char'174} primitive@>;
mp_primitive (mp, "=:|>", mp_lig_kern_token, 5);
@:=:/>_}{\.{=:\char'174>} primitive@>;
mp_primitive (mp, "|=:", mp_lig_kern_token, 2);
@:=:/_}{\.{\char'174=:} primitive@>;
mp_primitive (mp, "|=:>", mp_lig_kern_token, 6);
@:=:/>_}{\.{\char'174=:>} primitive@>;
mp_primitive (mp, "|=:|", mp_lig_kern_token, 3);
@:=:/_}{\.{\char'174=:\char'174} primitive@>;
mp_primitive (mp, "|=:|>", mp_lig_kern_token, 7);
@:=:/>_}{\.{\char'174=:\char'174>} primitive@>;
mp_primitive (mp, "|=:|>>", mp_lig_kern_token, 11);
@:=:/>_}{\.{\char'174=:\char'174>>} primitive@>;
```

I won't explain what happens there (as I would have to reread the relevant sections of TEX The Program) but the magic is in the special sequences: `=: =:| =:|> |=: |=:> |=:| |=:|> |=:|>>`. Similar sequences are used in some font related files. I bet that most MetaPost users never entered these as they relate to defining ligatures for fonts. Most users know that combining a `f` and `i` gives a `fi` but there are other ways to combine too. One can praise today's capabilities of OpenType ligature building but TEX was not stupid either! But these options were never really used and this treasure will stay hidden. Actually, to come back to a previous remark about abusing the ligature mechanism: OpenType fonts are just as sloppy as TEX with the quotes: there a ligature is just a name for a multiple-to-one mapping which is not always the same as a ligature.

But there are even more surprises with fonts. When Alan Braslau and I redid the bibliography subsystem of ConTEXt with help from Lua, I wrote a converter in that language. I actually did that the way I normally do: look at a file (in this case a bibTEX file) and write a parser from scratch. However, at some point we wondered how exactly strings got concatenated so I decided to locate the source and look at it there. When I scrolled down I noticed a peculiar section:

```
@^character set dependencies@>
@^system dependencies@>
Now we initialize the system-dependent |char_width| array,
for which |space| is the only |white_space| character given
a nonzero printing width.  The widths here are taken from
Stanford's June~'87 $cmr10$~font and represent hundredths
of a point (rounded), but since they're used only for
relative comparisons, the units have no meaning.

@d ss_width = 500 {character |@'31|'s width in the $cmr10$ font}
@d ae_width = 722 {character |@'32|'s width in the $cmr10$ font}
@d oe_width = 778 {character |@'33|'s width in the $cmr10$ font}
@d upper_ae_width = 903 {character |@'35|'s width in the $cmr10$
font}
@d upper_oe_width = 1014 {character |@'36|'s width in the $cmr10$
font}

@<Set initial values of key variables@>=
for i:=0 to @'177 do char_width[i] := 0;
@#
char_width[@'40] := 278;
char_width[@'41] := 278;
char_width[@'42] := 500;
char_width[@'43] := 833;
char_width[@'44] := 500;
char_width[@'45] := 833;
```

Do you see what happens here? There are hard coded font metrics in there! As far as I can tell, these are used in order to guess the width of the margin for references. Of course that won't work well in practice, simply because fonts differ. But given that the majority of documents that need references are using Computer Modern fonts, it actually might work well, especially with Plain TEX because that is also hardwired for 10pt fonts. Personally I'd go for a multipass analysis (or maybe would have had bibTEX produce a list of those labels for the purpose of analysis but for sure at that time any extra pass was costly in terms of performance). That code stays around of course. It makes for some nice deduction by historians in the future.

I bet that one can also find weird or unexpected code in ConTEXt, and definitely on the machines of TEX users all around the world. For instance, now that most people use utf8 all those encoding related hacks have become history. On the other hand, as history tends to cycle, bitmap symbolic fonts suddenly can look modern in a time when emoji are often bitmaps. We should guard our treasures.

Hans Hagen,
February 2020

2. ConTEXt never assumed this and encourages users to use the quotation macros. Those ``quotes'' look horrible in a source anyway.

Knuth en Schuh

Donald Knuth is niet te stoppen. Men zou denken dat hij zo langzamerhand op zijn lauweren zou rusten, thuis zijn grote huiskamerorgel zou bespelen en slechts de deur uit zou gaan om nog ergens een eredoctoraat op te halen, maar niets is minder waar.

Afgelopen jaar woonde hij in Tsjechië de première bij van zijn multimediale orgelwerk Fantasia Apocalyptica, gebaseerd op het het laatste boek van het Nieuwe Testament *De openbaring van Johannes* en voorzien van illustraties door Duane Bibby. Ik vermoed dat het werk bij de doorsnee liefhebber van klassieke muziek even veel (of even weinig) weerklank zal vinden als Frank Zappa's uitstapjes naar het componeren van klassieke muziek (*The Yellow Shark*) maar het past wel in het idee dat computers programmeren een kunstvorm is als vele andere en zo'n uitvoering van Knuth's muziek is een geweldig mooie aanleiding voor gelijkgestemden om samen te komen en te genieten van elkaars gezelschap en goede luim.

Bovendien schrijft Knuth op volle kracht verder aan *The Art of Computer Programming*, een boek in vele delen als een gestaag uitdijend zonnestelsel, bestaande uit zijn denkwijze en leerstof. Het is een ‚work in progress' waarvan de voltooide delen ingebonden beschikbaar zijn terwijl kleinere deeltjes die vergevorderd zijn maar nog niet 'af' in paperback verschijnen, 'fascicle' genaamd en voorafgaande aan hun fysieke verschijning als voor-deeltje (pre-fascicle) in PostScript zijn te bekijken.

Een van die kleinste deeltjes in het Knuth universum is *The Art of Computer Programming, Volume 4, Pre-Fascicle 9B, A Potpourri of Puzzles* dat begin mei 2019 zichtbaar werd. Deel 4 gaat over ‚Combinatorial Algorithms', een onderdeel van computerwetenschap waar één goed idee een programma een miljoen keer sneller kan maken, aldus Knuth. De voorpublicatie van het 'puzzelboekje' is bedoeld als speels voorbeeld van de manier waarop deze aanpak in de praktijk kan werken.

Knuth opent zijn voorwoord met fraai geformuleerde bescheidenheid in klassieke traditie van welsprekendheid door de zeggen dat het niet zozeer zijn bedoeling is de lezer te imponeren met dit werk maar integendeel een kleine kring van lezers de mogelijkheid te bieden om fouten in zijn manuscript aan te wijzen voordat al te veel anderen deze zien ‚in de hoop dat ik mij, op een zekere dag, niet langer hoef te verontschuldigen voor wat nu nog slechts een hoopje schetsen is.'

7.2.2.8 A POTPOURRI OF PUZZLES 1

7.2.2.8. A potpourri of puzzles. Blah blah de blah blah blah. We'll discuss some of the most interesting time-wasters that have captured the attention of computer programmers over the years ... The "obvious" ways to solve them can often be greatly improved by using what we've learned in previous sections ...

* * *

Who knows what I might eventually say here?

* * *

Perfect digital invariants. In 1923, the great puzzlist Henry E. Dudeney observed that

$$370 = 3^3 + 7^3 + 0^3 \qquad \text{and} \qquad 407 = 4^3 + 0^3 + 7^3,$$

and asked his readers to find a similar example that doesn't have a zero in its decimal representation. A month later he gave the solution, $153 = 1^3 + 5^3 + 3^3$ [*Strand* **65** (1923), 103, 208] — curiously saying nothing about the obvious answer $371 = 3^3 + 7^3 + 1^3$. These examples were rediscovered independently by several other people, and eventually extended to mth powers of the digits for $m > 3$, and to radix-b numbers for $b \neq 10$. Max Runney [*Recreational Math. Magazine* #12 (December 1962), 6–8] mentioned $8208 = 8^4 + 2^4 + 0^4 + 8^4$, $(491)_{11} = 794 = 4^3 + 9^3 + 1^3$, ..., and named such numbers *perfect digital invariants* of order m.

Let $\pi_m x$ be the sum of the mth powers of the decimal digits of x. With this notation, the number x is a perfect digital invariant of order m in radix 10 if and only if $\pi_m x = x$. In particular, every order $m > 0$ has at least two perfect digital invariants, since the numbers 0 and 1 always qualify. And it turns out that most orders have *more* than two (see exercise 34), because of more or less random coincidences. For example, when $m = 100$ there's a unique third solution,

$$x = 26\,561\,629\,619\,330\,109\,806\,364\,160\,710\,032\,970\,207\,878\,434\,854\,147 \\ 717\,669\,387\,628\,693\,204\,788\,451\,137\,448\,014\,798\,509\,429\,68 = \pi_{100} x, \quad (20)$$

discovered in 2009 by Joseph Myers.

How can such a humongous number be found in a reasonable time? In the first place, we can always write $x = (x_m \ldots x_1 x_0)_{10}$, because exercise 30 shows that every mth-order solution has at most $m + 1$ digits. In the second place, we can see that π_m depends only on the multiset $M_m(x) = \{x_m, \ldots, x_1, x_0\}$ of x's digits, not on the actual order of those digits. All we have to do, therefore, is look at each multiset, and see if $M_m(x_m^m + \cdots + x_1^m + x_0^m) = \{x_m, \ldots, x_1, x_0\}$.

A multiset of $m + 1$ digits is what Section 7.2.1.3 calls a "multicombination," also known as a *combination of the ten objects* $\{0, 1, \ldots, 9\}$ *taken* $m+1$ *at a time with repetitions allowed*. If we renumber the subscripts by sorting the digits into order, such a multicombination is nothing more nor less than a solution to

$$9 \geq x_m \geq \cdots \geq x_1 \geq x_0 \geq 0. \quad (21)$$

May 1, 2019

PDE: A perfect digital invariant
Perfect digital invariants
Dudeney
powers of the digits
radix-b numbers
Runney
perfect digital invariants
narcissistic numbers
coincidences
Myers
multiset
multicombination
combination
sorting

The Art of Computer Programming, Volume 4, Pre-Fascicle 9B, A Potpourri of Puzzles, pagina 5

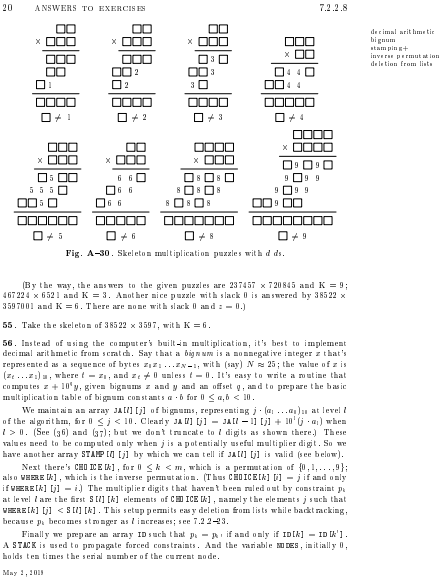
Zulk een bescheidenheid vergt een zuiver gevoel voor balans. Het moet oprecht gemeend zijn en geen vluchtig vermomde verwaandheid maar men moet ook weer niet te ver gaan in het afwaarderen van het eigen werk want waarom zou de lezer eraan beginnen als de auteur het zelf niks vindt? In de Nederlandse literatuur is de ontstaansgeschiedenis van P.C. Boutens' *Strofen uit de nalatenschap van Andries de Hoghe* een aangrijpend voorbeeld hiervan. Een jonge dichter, Jan S. van Drooge, zond Boutens wat werk ter beoordeling en schreef er overdreven bescheiden bij dat ‚de betere gedichten van zijn hand niet mede waren ingesloten, daar hij de tijd voor dat werk nog niet gekomen achtte.' Boutens, in een korzelige bui, zond alles per kerende post terug aangezien ‚het zinloos was verzen te lezen, aan welke de schrijver zelf al voorbij was.' Korte tijd later toen Boutens vernam dat de jonge man een einde aan zijn leven had gemaakt, voelde hij zich daaraan schuldig en jaren daarna eerde hij de jongeling door een verzenbundel min of meer aan deze toe te schrijven. (Zie ook Johan Polak, *Het voorbeeldige boek*, uitg. Balans 2006, blz 79, – overigens met behulp van ConTEXt getypeset door Willi Egger).

Knuth intussen is sympathiek, wars van arrogantie en hoffelijk naar zijn lezers: ‚het zou mij verbazen als het 'tagging algorithm' in antwoord 39 niet al eens ergens eerder is gepubliceerd, al denk ik niet dat ik het ooit eerder heb gezien.' Voorts biedt hij 32 dollarcent voor als iemand een waardevolle suggestie heeft voor tekstverbetering en zelfs 'onsterfelijke glorie' komt binnen bereik indien je naam daarna in het boek zal worden opgenomen.

Het boekje zelf is duizelingwekkend doordat je, als je de tekst leest, aanvankelijk de indruk kijgt dat het allemaal heel goed is te volgen en zelfs anekdotisch is, zoals het voorbeeld van 'mottige vermenigvuldiging' waarbij een veel van de cijfers onzichtbaar is. Het leest vlot weg tot je je realiseert dat je geen idee hebt wat er gebeurt. Er zijn kennelijk mensen die zo slim zijn dat zelfs onzichtbare getallen herkend worden.

In het hoofdstuk 'Discrete dissections' wordt iets besproken dat ik me vaag herinner uit mijn Montessori lagere schooltijd. Een uit kleurige vlakjes opgebouwd vierkant moet worden opgedeeld in twee kleinere vierkanten waarbij de keurpatronen gelijk blijven. Op school was het idee dat je door het maken van dergelijke 'taakjes' spelenderwijs leerde rekenen. Het rekenen heb ik er niet van geleerd maar het spelen des te beter. Knuth laat zien hoe je dergelijke vraagstukken op eenvoudige wijze wiskundig kan oplossen. Voor deze puzzles ontwierp Knuth een toepasselijk font, FONT36.

Knuth verwijst ook naar Fred Schuh (1875-1966), een Nederlandse wiskundige waar ik nog nooit van had gehoord. Online lees ik dat Schuh als kleurrijke radiopersoonlijkheid bekend stond om zijn voordrachten ‚Hoe leert men denken?' In Delft waren naar verluidt zijn mechanica-colleges van een niet te overtreffen helderheid en zijn meesterwerk *Wonderlijke problemen, Leerzaam tijdverdrijf door puzzle en spel* vormde voor Knuth wellicht de inspiratie voor juist deze fascicle.



20 ANSWERS TO EXERCISES 7.2.2.8

decimal arithmetic
bignum
stamping
inverse permutation
deletion from lists

Fig. A–30. Skeleton multiplication puzzles with d ds.

(By the way, the answers to the given puzzles are 237457×720845 and $\mathrm{K} = 9$; 467224×6521 and $\mathrm{K} = 3$. Another nice puzzle with slack 0 is answered by 38522×3597001 and $\mathrm{K} = 6$. There are none with slack 0 and $z = 0$.)

55. Take the skeleton of 38522×3597, with $\mathrm{K} = 6$.

56. Instead of using the computer's built-in multiplication, it's best to implement decimal arithmetic from scratch. Say that a *bignum* is a nonnegative integer x that's represented as a sequence of bytes $x_0 x_1 \ldots x_{N-1}$, with (say) $N \approx 25$; the value of x is $(x_t \ldots x_1)_{10}$, where $t = x_0$, and $x_t \neq 0$ unless $t = 0$. It's easy to write a routine that computes $x + 10^k y$, given bignums x and y and an offset k, and to prepare the basic multiplication table of bignum constants $a \cdot b$ for $0 \le a, b < 10$.

We maintain an array JAM[j] of bignums, representing $j \cdot (a_l \ldots a_1)_{10}$ at level l of the algorithm, for $0 \le j < 10$. Clearly JAM[j] $=$ JAM[-1][j] $+ 10^l (j \cdot a_l)$ when $l > 0$. (See (36) and (37); but we don't truncate to l digits as shown there.) These values need to be computed only when j is a potentially useful multiplier digit. So we have another array STAMP[l][j] by which we can tell if JAM[l][j] is valid (see below).

Next there's CHOICE[k], for $0 \le k < m$, which is a permutation of $\{0, 1, \ldots, 9\}$; also WHERE[k], which is the inverse permutation. (Thus CHOICE[k][j] $= j$ if and only if WHERE[k][j] $= i$.) The multiplier digits that haven't been ruled out by constraint p_k at level l are the first SIZ[l][k] elements of CHOICE[k], namely the elements j such that WHERE[k][j] $<$ SIZ[l][k]. This setup permits easy deletion from lists while backtracking, because p_k becomes stronger as l increases; see 7.2.2–23.

Finally we prepare an array ID such that $p_k = p_{k'}$ if and only if ID[k] $=$ ID[k']. A STACK is used to propagate forced constraints. And the variable NODES, initially 0, holds ten times the serial number of the current node.

May 2, 2013

‚Knuth's keyboard' zoals door hem geschonken aan het Computer History Museum https://www.computerhistory.org/collections/catalog/102667297

Frans Goddijn

Mijn leven met TEX als student

Hoi! Ik ben Dennis Holierhoek en ik ben student aan de Christelijke Hogeschool Windesheim te Zwolle. Onlangs kreeg ik een uitnodiging van Frans Goddijn om eens een stukje te schrijven voor de MAPS over het werken met TEX als student, en ik voelde mij enorm vereerd! Met toch wel enig gevoel van trots presenteer ik u hier mijn artikel en wens ik u veel leesplezier!

Voor dit artikel zal ik een aantal termen gebruiken uit de wereld van de applicatieontwikkeling en het moderne onderwijs. Ik snap dat niet iedereen met deze termen bekend is, daarom zal ik een woordenlijst bijhouden. Het kan misschien wel zo zijn dat ik op sommige punten iets teveel woorden omschrijf en dingen uitleg die vrijwel iedereen weet, en op andere punten dingen niet weet uit te leggen (voornamelijk de stukken die gaan over ICT) voor iemand die niet uit „mijn" hoek komt.

Ik wil vooraf ook zeggen dat ik al mijn kennis over LaTEX en vooral over goede documenten schrijven zelf heb opgedaan, en ik dus misschien een paar onhandige fouten maak waar ik mij niet van bewust ben. Wees dus alstublieft niet te kritisch bij het lezen van dit artikel.

1 Ik en TEX door de jaren heen

1.1 Eerste stapjes

De eerste keer dat ik de term LaTEX heb horen vallen, was in ongeveer 2012. Ik zat toen in de derde of vierde klas van de havo en was een jaar of 16. Een klasgenoot van mij was enorm fan van Linux-besturingssystemen en stak mij met zijn enthousiasme aan. Windows ging van mijn computer af en ging vrolijk aan de knutsel met Linux.

Tijdens mijn uitpluiswerk over de mogelijkheden van Linux, zag ik veelvuldig LaTEX langskomen. Ik deed er destijd nog niet zoveel mee, ik zag de noodzaak er niet van in. Maar ik onthield wel de naam.

In 2017 begon ik met een opleiding hbo-ICT op hogeschool Windesheim te Zwolle. Het ging goed, ik haalde goede cijfers en was altijd op zoek naar dé perfecte manier om iets te doen. Zo kwam ik in 2018 op het idee om TEX te gebruiken. En op 21 mei 2018 om 19.50 was het een feit: ik heb mijn eerste LaTEX-document opgeleverd. Een simpele klachtenbrief voor het vak Nederlands (ja, dat is een verplicht vak op het hbo), dat wel, maar het was een begin.

1.2 Ervaring opdoen op Honoursprogramma

In 2018 meldde ik mij aan voor een Honoursprogramma op een dochterinstelling van Windesheim: Windesheim Flevoland te Almere. Dit Honoursprogramma, New Towns heette het, was een extracurriculair traject dat ging over zelfontwikkeling, dingen leren die niet binnen je opleiding ter sprake komen en het aanpakken van zogenaamde „wicked problems", problemen die erg lastig zijn op te lossen omdat er zoveel betrokken partijen zijn met elk hun eigen belang. Een pittig maar leerzaam traject. Tijdens dit programma moest ik een Individueel Leertraject bijhouden, een soort dagboek over al mijn denkprocessen. Ik ben helemaal losgegaan en een bestand van 1400 pagina's en een „gewicht" van 900 MB geschreven. Uiteindelijk heb ik om de leesbaarheid te garanderen, het bestand herschreven zodat er maar 13(!) pagina's overbleven. Toch heb ik er veel aan gehad. Ik ben aardig vaardig geworden en ik begrijp het hele TEX-ecosysteem nu beter. Veel code die ik vind op Internet is mij nog steeds onduidelijk, maar ik heb nu wel het idee dat ik alles kan maken wat ik zou willen maken.

1.3 Tweedejaarsstage: TEX automatiseren

Aan het eind van schooljaar 2018-2019 moest ik een stage doen over webontwikkeling. Ik deed hem samen met een vriend van de opleiding voor Trespa. Ik denk dat dit bedrijf bij (bijna) iedereen wel een lampje doet branden. Dat is ook niet zo gek: het is een enorme speler in sierbeplating voor gevels van huizen. Zij wilden een webapplicatie waarmee je voor één van hun producten (namelijk Trespa Pura NFC) kon berekenen hoeveel platen je nodig had op een huis. Aan het einde van het programma kon je een bestelbon downloaden en – u raadt het al – dit heb ik gemaakt met TEX.

Door XeLaTEX aan de populaire programmeertaal PHP te koppelen, kon ik het genereren van bestelbonnen automatiseren. Ik had niet genoeg tijd om het systeem schaalbaar en efficiënt te maken, maar toch was ik erg trots! Het smaakte naar meer en ik zie mezelf later graag werken bij een baan waar TEX niet (alleen) als

codeertaal wordt gebruikt om documentatie met de hand mee te schrijven, maar het (ook) als programmeertaal geïntegreerd zit in automatische systemen.

Ook bij de documentatie van dit project kon ik uitvoerig gebruik maken van TEX. Zo moest ik mijn voortgangsrapportages bundelen en voorzien van een voorblad. Met TEX kan dit zo elegant dat dit haast magisch wordt, je gebruikt een simpele PGF-`\for`-loop die één voor één alle pdfjes inlaadt en je bent klaar.

1.4 De toekomst

Ik moet toegeven dat ik nog steeds minder weet over TEX dan ik eigenlijk zou willen. Veel dingen vind ik nog steeds erg lastig, zoals begrijpen wat de ingewikkelde macro's doen die je wel eens op TEX Answers tegenkomt. Ook ben ik nog steeds benieuwd naar wat voor mogelijkheden LuaLATEX heeft omtrent het schrijven van automatische scripts in mijn documenten. Dit alles wil ik in de toekomst nog een keer oppakken. Mijn uiteindelijke einddoel is dat ik zelf TEX-packages kan schrijven en steeds meer van de dingen die ik nu met binaire bestanden doe (zoals afbeeldingen) oplos met code. De reden hiervoor staat beschreven in subsectie 2.1. Dit gaat nu al erg goed: ik upload mijn UML-diagrammen niet meer als afbeelding, maar als code van speciale diagramcodeertalen.

2 Mijn tools naar keuze

2.1 Toolchains

Frans vroeg mij ook om een beschrijving van mijn gebruikte tools. Welnu, het interessante deel van mijn *setup* is niet wat ik ín TEX gebruik, maar meer wat er aan de buitenkant zit.

Aan de basis gebruik ik Windows. Hierop draai ik Chocolatey, een zogeheten package manager voor Windows. Hiermee kan je in een commandoprompt programma's installen en updaten. Het voordeel is dat je met één commando alle programma's op je computer kan updaten. Elke ochtend zit ik dus op de meest recente versies van alle software die ik op mijn computer heb.

Ik gebruik MikTEX als TEX-distributie en TEXstudio om mijn code in te schrijven. Ik ben niet helemaal fan van MikTEX, het is naar mijn mening nogal traag als je meer packages op je systeem hebt staan. Ik moet ooit nog eens TEX Live proberen, omdat dat op Linuxmachines altijd best wel snel draaide, ook met veel packages erop. Ik moet alleen even kijken of het niet vreselijk misgaat als ik twee TEX-distributies tegelijk installeer voordat ik het ga proberen.

Een veelgehoord argument tegenwoordig om TEX te blijven gebruiken, ook al is Word door de jaren heen sterk verbeterd, is dat TEX onder versiebeheer kan worden geplaatst. Deze krachtige tools zijn voor software-ontwikkelaars gemeengoed, voor anderen is er misschien iets meer uitleg nodig. Ik heb geprobeerd het op een korte en bondige manier uit te leggen, maar dat is helaas niet gelukt. Maar probeer het maar te zien als een soort Dropbox of Google Drive, maar dan op zo'n manier gedaan dat je heel nauwkeurig kan sturen hoe het project wordt gesynchroniseerd. Deze geavanceerde systemen werken alleen echt goed met platte tekst. Daarom gebruik ik ook PlantUML in mijn code. Dit is het hierboven in subsectie 1.4 al genoemde systeem om door middel van code softwarediagrammen te maken. Doordat ik dit in code doe en niet met een grafische editor, kan ik dit ook onder versiebeheer zetten. In mijn versiebeheer maak ik ook gebruik van een CI/CD-straat. Dit is een systeem dat ervoor zorgt dat de bestanden op de versiebeheerserver automatisch worden gecompileerd.

2.2 Classfiles en fonts

Ik gebruik eigenlijk alleen maar standaard classfiles en lettertypes. Standaard is goed, is mijn mening, anders hebben ze het wel anders bedacht. Soms gebruik ik KOMA-Script voor functies als `\part`. Als ik Unicode-karakters moet schrijven in XELATEX en ik het bestand ook op een Linux-machine moet kunnen gebruiken, gebruik ik FreeSans. En als ik echt een mooie romantische brief moet schrijven, gebruik ik het Uwebodingbatfont. Ziet er fantastisch uit!

3 Werkwijze

Veel van mijn vrienden op de universiteit krijgen les in TEX. Ik niet, ik heb mijzelf alles aangeleerd. Wikibooks en Overleaf bieden hiervoor heel goede documentatie en de leercurve is ook zeker niet te steil. Als ik iets niet weet, dan kopieer ik het vaak van TEX - LaTEX Stack Exchange. Dit zorgt ervoor dat ik vrij veel kan, maar jammer genoeg zonder echt te weten hoe het allemaal werkt.

4 TEX in mijn omgeving

Zoals hiervoor al gezegd, heb ik TEX mijzelf aan moeten leren. Op het HBO op „Windowsheim" zijn Microsoft Word en Google Docs erg populair. In theorie integreert het perfect met het grote intranet en kan je naadloos samenwerken, maar in de praktijk is het erg brak. Enkele studenten en docenten zijn erg enthousiast over TEX. Soms leer ik mijn docenten ook nog eens wat! Andere mensen vinden het maar een gedoe om een extra taal te leren. En een enkeling vindt het wel oké om samen te werken, maar is er niet superenthousiast over.

5 Mijn visie op TEX

TEX is een erg leuk systeem en het is ook leuk dat er zo'n trotse community voor beschikbaar is. Vooral de artikelen in de TUGboat over onderzoek in typografie vind ik erg interessant. En het compileren van een document heeft altijd wat moois, ook al is het jammer dat het vaak zo lang duurt. En het eerder genoemde voordeel, dat je een heel project in code kan schrijven, is een prachtig fenomeen.

Maar TEX voelt wel primitief. Het is duidelijk niet gebouwd om er echt stevig in te programmeren. Veel codevoorbeelden op TEX - LaTEX Stack Exchange geven me nog steeds hoofdpijn. Er zijn alternatieven, zoals Markdown en `wkhtmltopdf`, die beide de uitgebreide programmeertaal Javascript ondersteunen. Toegegeven, Javascript heeft ook zijn eigenaardigheden, maar simpele dingen zoals arrays zitten er wel gewoon in. Maar als ik die gebruik, is er geen uitdaging meer om dingen op te lossen in echte (LA)TEX. En juist die uitdaging vind ik leuk. Bovendien krijg ik dan ook niet de prachtige output van TEX. Misschien is Lua(LA)TEX een oplossing?

Verklarende Woordenlijst

applicatie een stuk software. Hoeft niet per se een app te zijn voor de smartphone of tablet, maar kan ook een computerprogramma of website zijn.

array lijst variabelen.

codeertaal een taal die lijkt op een programmeertaal, maar niet aan alle voorwaarden voldoet om een programmeertaal te zijn. Een voorbeeld is HTML.

Linux familie van veelal gratis alternatieven op het Windows-besturingssysteem van Microsoft.

UML Unified Modeling Language, een diagramstandaard voor het beschrijven van software.

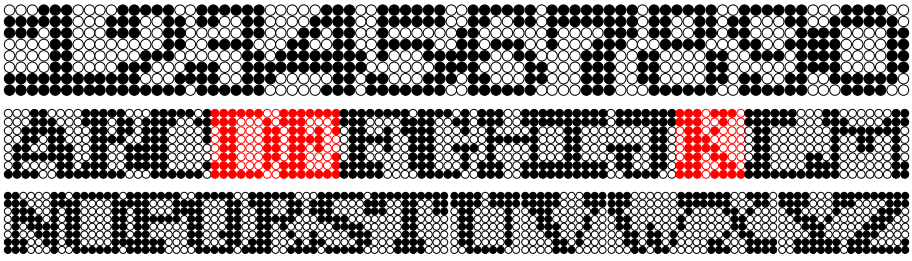
Dennis Holierhoek
24 februari 2020

ThreeSix

Don Knuths first colorfont?

In the process of reaching completion and perfection Don Knuth occasionally posts links to upcoming parts of the TAOCP series on his web pages. Now, I admit that much is way beyond me but I do understand (and like) the graphics and I know that Don uses MetaPost. The next example code is just a proof of concept but might eventually become a decent module (with helpers) for making (runtime) fonts. After all, we need to adapt to current developments and TEXies are always willing to adapt and experiment. This chapter was written at the same time as the previous one on TYPE3 fonts so you might want to read that first.

The font explored here is FONT36, used in "A potpourri of puzzles" and flagged as "a special font designed for dissection puzzles" (in fascicle 9b for Volume 4). Playing with and visualizing for me often works better than formulas, which then distracts me from the original purpose, but let's have a closer look anyway.



The font has a fixed maximum height of 8 quantities. There is no depth in the characters. Some characters are wider. In this example we use a tight bounding box. In ConTEXt speak this font is just a regular font but with a special feature enabled.

```
\definefontfeature
  [fontthreesix]
  [default]
  [metapost=fontthreesix]

\definefont[DEKFontA][Serif*fontthreesix]
```

After this the \DEKFontA command will set this font as current font. The definition mentions Serif as font name. In ConTEXt this name will resolve in the currently defined Serif, so when your document uses Latin Modern that will be the one. The fontthreesix will make this instance use that feature set, and the feature definition has the defaults as parent (so we get kerning, ligatures, etc.) but as extra feature also metapost. This means that the new glyphs that are about to be defined will actually be injected in the Serif! We will replace characters in that instance. So, the following:

```
This font is used in \quotation {The Art Of Computer Programming} by
Don Knuth, not in volume~1, 2 or~3, but in number~4!
```

comes out as:

> This font is used in “The ▀rt ▀f ▀omputer ▀rogramming” by ▀on ▀nuth, not in volume ▀, ▀ or ▀, but in number ▀!

But that doesn’t look too good, so we will tweak the font a bit:

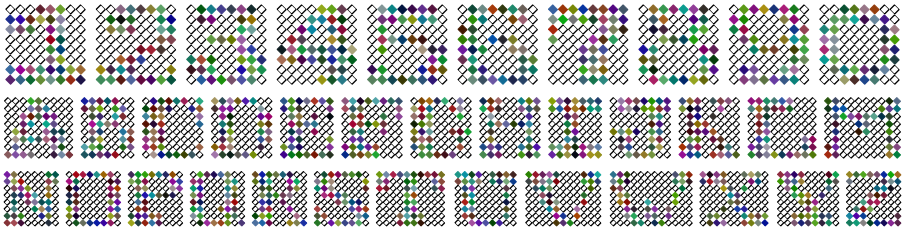
```
\definefontfeature
  [fontthreesix-color]
  [default]
  [metapost={category=fontthreesix,spread=.1}]

\definefont[DEKFontD][Serif*fontthreesix]
```

The `spread` (multiplied by the font unit, which is 12 basepoints here) will add a bit more spacing around the blob:

> This font is used in “The ▀rt ▀f ▀omputer ▀rogramming” by ▀on ▀nuth, not in volume ▀, ▀ or ▀, but in number ▀!

Now, keep in mind that we’re talking of a real font here. You can cut and paste these characters. It’s just the default uppercase Latin alphabet plus digits.

Before we go and look at some of the code needed to render this, a few more examples will be given.



In the above example we not only use color, but also a different shape and random colors (that is: random per TEX job). The feature definition for this is:

```
\definefontfeature
  [fontthreesix-color]
  [default]
  [metapost={%
    category=fontthreesix,shape=diamond,%
    color=random,pen=fancy,spread=.1%
  }]
```

Possible shapes are `circle`, `diamond` and `square` and instead of a random color one can give a known color name. Using transparency makes no sense in this font.

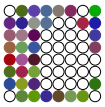
A nice usage for this font are initials:

```
\setupinitial[font=Serif*fontthreesix-initial sa 5]
{\DEKFontB \placeinitial \input zapf\par}
```

The initial feature is defined as:

```
\definefontfeature
  [fontthreesix-initial]
  [metapost={category=fontthreesix,color=random,shape=circle}]
```

We use this in quoting Hermann Zapf, one that for sure is very applicable in a case like this:


oming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a  or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design.  any people are just fascinated by their  's tricks, and think that a widely–praised program, called up on the screen, will make everything automatic from now on.

Some combinations of sub-features are shown in figure 1. We blow up the diamond with fancy pen example in figure 2. Alas, the TEX logo doesn't look that good in such a font. Using it for acronyms is not a good idea anyway, but maybe you can figure out figure 3.

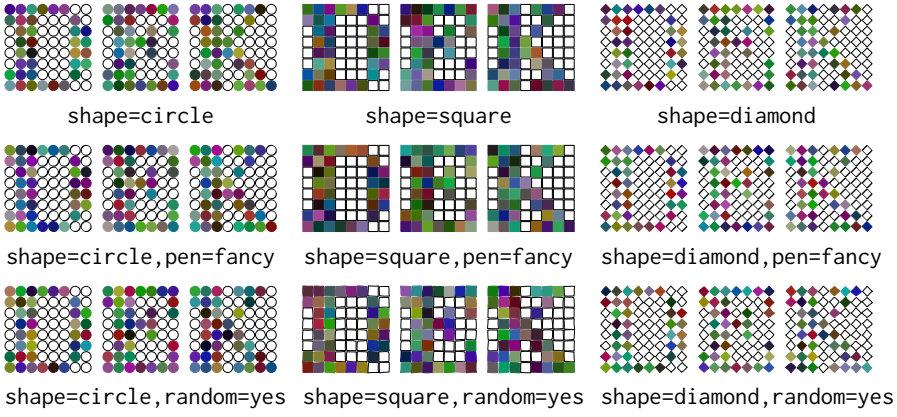


Figure 1.

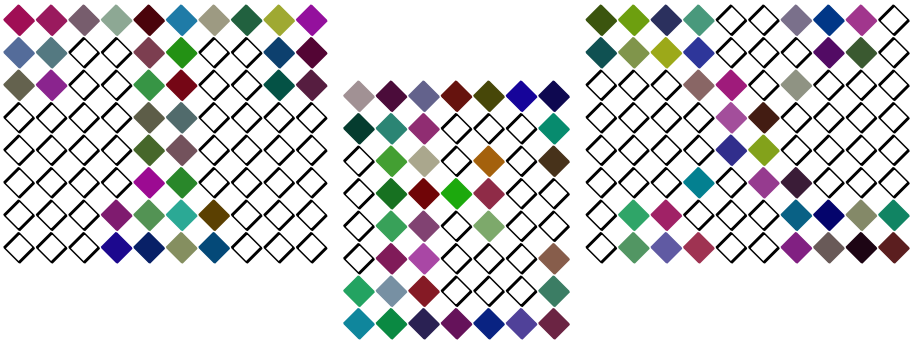


Figure 2.

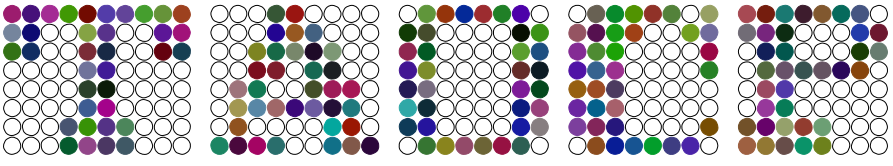


Figure 3.

You can quit reading now or expose yourself to how this is coded. We use a combination of Lua and MetaPost, but different solutions are possible. The shapes are entered (or course) with zeros and ones.

```
\startluacode
local font36 = {
    ["0"] = "00111100 01111110 11000011 11000011 11000011 ...",
    ["1"] = "00011100 11111100 11101100 00001100 00001100 ...",
    .....
    ["D"] = "11111100 11100010 01100011 01100011 01100011 ...",
    ["E"] = "1111111 1110001 0110101 0111100 0110100 0110001 ...",
    .....
    ["K"] = "11101110 11100100 01101000 01110000 01111000 ...",
    .....
}
\stopluacode
```

We also use Lua to register this font. The actual code looks slightly different because it uses some helpers from the ConTEXt Lua libraries. We remap the bits pattern onto MetaPost macro calls.

```
\startluacode
local replace = {
    ["0"] = "N;",
    ["1"] = "Y;",
    [" "] = "L;",
}

function MP.registerthreesix(name)
    fonts.dropins.registerglyphs {
        name     = name,
        units    = 12,
        usecolor = true,
    }
    for u, v in table.sortedhash(font36) do
        local ny     = 8
        local nx     = (#v - ny + 1) // ny
        local height = ny * 1.1 - 0.1
        local width  = nx * 1.1 - 0.1
        local code   = string.gsub(v,".",replace)
        fonts.dropins.registerglyph {
            category = name,
            unicode  = utf.byte(u),
            width    = width,
            height   = height,
            code     = string.format("ThreeSix(%s);",code),
        }
    end
end

MP.registerthreesix("fontthreesix")
\stopluacode
```

So, after this the font `fontthreesix` is known to the system but we still need to provide MetaPost code to generate it. The glyphs themselves are now just sequences of `N`, `Y` and `L` with some wrapper code around it. The definitions are put in the `MP` namespace simply because a first version initialized in MetaPost, and there could create variants, but in the end I settled on the parameter interface at the TEX end.

The next definition looks a bit complex but normally such a macro is stepwise constructed. Notice how we can query the sub features. In order to make that possible

the regular MetaFun parameter handling code is used. We just push the sub-features into to mpsfont namespace.

```
\startMPcalculation{simplefun}

def InitializeThreeSix =
    save Y, N, L, S ; save dx, dy, nx, ny ; save currentpen ;
    save shape, fillcolor, mypen, random, spread, hoffset ;
    string shape, fillcolor, mypen ; boolean random ;
    pen currentpen ;
    dx :=   11/10 ;
    dy := - 11/10 ;
    nx := - dx ;
    ny :=   0 ;
    shape      := getparameterdefault "mpsfont" "shape" "circle" ;
    random     := hasoption           "mpsfont" "random" "true" ;
    fillcolor  := getparameterdefault "mpsfont" "color" "" ;
    mypen      := getparameterdefault "mpsfont" "pen" "" ;
    spread     := getparameterdefault "mpsfont" "spread" 0 ;
    hoffset    := 12 * spread / 2 ;
    currentpen := pencircle
        if mypen = "fancy" :
            xscaled 1/20 yscaled 2/20 rotated 45
        else :
            scaled 1/20
        fi ;
    if shape == "square" :
        def S =
            unitsquare if random : randomized 1/10 fi
            shifted (nx,ny)
        enddef ;
    elseif shape = "diamond" :
        def S =
            unitdiamond if random : randomized 1/10 fi
            shifted (nx,ny)
        enddef ;
    else :
        def S =
            unitcircle if random : randomizedcontrols 1/20 fi
            shifted (nx,ny)
        enddef ;
    fi ;
    def N =
        nx := nx + dx ;
        draw S ;
    enddef ;
    if fillcolor = "random" :
        def Y =
            nx := nx + dx ;
            fillup S withcolor white randomized (2/3,2/3,2/3) ;
        enddef ;
    elseif fillcolor = "" :
        def Y =
            nx := nx + dx ;
            fillup S ;
```

```
        enddef ;
    else :
        def Y =
            nx := nx + dx ;
            fillup S withcolor fillcolor ;
        enddef ;
    fi ;
    def L =
        nx := - dx ;
        ny := ny + dy ;
    enddef ;
enddef ;

vardef ThreeSix (text code) =
    InitializeThreeSix ; % todo: once per instance run
    draw image (code) shifted (hoffset,-ny) ;
enddef ;

\stopMPcalculation
```

This code is not that efficient in the sense that there's quite some MetaPost-Lua-MetaPost traffic going on, for instance each parameter check involves this, but in practice performance is quite okay, certainly for such a small font. There will be an initializer option some day soon. The `simplefun` is a reference to an MPLIB instance that does load MetaFun but only the modules that make no sense for this kind of usage. It also enforces double mode. The calculations wrapper just executes the code and does not place some (otherwise empty) graphic.

Those who have seen (and/or read) "Concrete Mathematics" will have noticed the use of inline images, like dice. Dice are also used in "pre-fascicle 5a" (I need a few more lives to grasp that, so I stick to the images for now!). So, to compensate the somewhat complex code above, I will show how to accomplish that. This time we do all in MetaPost:

This is not that hard to follow. We define some shapes first. These could have been assigned to the `code` parameter directly but this is nicer. Next we register the font itself and after that we set glyphs. We also set the official Unicode slots. So, copying a dice can either result in a digit or in a Unicode slot for a dice. In the example below we switch to a color which demonstrates that our dice can be colored at the TEX end. It's either that or coloring at the MetaPost end as both demand a different kind of Type3 embedding trickery.

We actually predefine three features. The digits one will map regular digit in the input to dice. We accomplish that via a font feature:

```
\startluacode
fonts.handlers.otf.addfeature("dice:digits", {
    type      = "substitution",
    order     = { "dice:digits" },
    nocheck   = true,
    data      = {
        [0x30] = "invaliddice",
        [0x31] = 0x2680,
        [0x32] = 0x2681,
        [0x33] = 0x2682,
        [0x34] = 0x2683,
        [0x35] = 0x2684,
```

```
        [0x36] = 0x2685,
        [0x37] = "invaliddice",
        [0x38] = "invaliddice",
        [0x39] = "invaliddice",
    },
} )
\stopluacode
```

This kind of trickery is part of the font machinery used in ConTEXt and permits runtime adaption of fonts, so we just use the same mechanism. The nocheck is needed to avoid this feature not kicking in due to lack of (at the time of checking) yet undefined dice.

```
\definefontfeature
  [dice:normal]
  [default]
  [metapost={category=dice}]
\definefontfeature
  [dice:reverse]
  [default]
  [metapost={category=dice,option=reverse}]
\definefontfeature
  [dice:digits]
  [dice:digits=yes]

\definefont[DiceN] [Serif*dice:normal]
\definefont[DiceD] [Serif*dice:normal,dice:digits]
\definefont[DiceR] [Serif*dice:reverse,dice:digits]

{\DiceD Does 1 it 4 work? And {\darkgreen 3} too?} {\DiceR And how about
{\darkred 3} then? But 8 should sort of fail!}
```

Does ⚀ it ⚃ work? And ⚂ too? And how about ⚄ then? But ☒ should sort of fail!

The six digits and Unicode characters come out the same:

```
\red  \DiceD \dostepwiserecurse   {`1}   {`6}{1}{\char#1\quad}%
\blue \DiceN \dostepwiserecurse{"2680}{"2685}{1}{\char#1\quad}%
```

It is tempting to implement for instance 7 as two dice (a one to multi mapping in OpenType speak) but then one has to decide what combination is taken. One can also implement ligatures so that for instance 12 results in two six dice. But I think that's over the top and only showing TEX muscles. It is anyway not that hard to do as we have an interface for that already.

So why not do the dominos as well? Because there are so many dominos we predefine the shapes and then register the lot in a loop. We have horizontal and vertical variants. Being lazy I just made two helpers while one could have done but with some rotation and shifting of the horizontal one. The loop could be a macro but we don't save much code that way.

```
\startMPcalculation{simplefun}

picture Dominos[] ;

Dominos[0] := image() ;
Dominos[1] := image(draw(4,4);) ;
```

```
Dominos[2] := image(draw(2,6);draw(6,2););
Dominos[3] := image(draw(2,6);draw(4,4);draw(6,2););
Dominos[4] := image(draw(2,6);draw(6,6);draw(2,2);draw(6,2););
Dominos[5] := image(draw(2,6);draw(6,6);draw(4,4);draw(2,2);draw(6,2););
Dominos[6] := image(draw(2,6);draw(4,6);draw(6,6);draw(2,2);draw(4,2);draw(6,2););

lmt_registerglyphs [
    name     = "dominos",
    units    = 12,
    width    = 16,
    height   = 8,
    depth    = 0,
    usecolor = true,
] ;

def DrawDominoH(expr a, b) =
    draw image (
        pickup pencircle scaled 1/2 ;
        if (getparameterdefault "mpsfont" "color" "") = "black" :
            fillup unitsquare xyscaled (16,8) ;
            draw (8,.5) -- (8,7.5) withcolor white ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a]
                withpen currentpen
                withcolor white ;
            draw Dominos[b] shifted (8,0)
                withpen currentpen
                withcolor white ;
        else :
            draw unitsquare xyscaled (16,8) ;
            draw (8,0) -- (8,8) ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a]
                withpen currentpen ;
            draw Dominos[b] shifted (8,0)
                withpen currentpen ;
        fi ;
    ) ;
enddef ;

def DrawDominoV(expr a, b) = % is H rotated and shifted
    draw image (
        pickup pencircle scaled 1/2 ;
        if (getparameterdefault "mpsfont" "color" "") = "black" :
            fillup unitsquare xyscaled (8,16) ;
            draw (.5,8) -- (7.5,8) withcolor white ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a] rotatedaround(center Dominos[a],90)
                withpen currentpen
                withcolor white  ;
            draw Dominos[b] rotatedaround(center Dominos[b],90) shifted (0,8)
                withpen currentpen
                withcolor white  ;
        else :
            draw unitsquare xyscaled (8,16) ;
            draw (0,8) -- (8,8) ;
```

```
            pickup pencircle scaled 3/2 ;
            draw Dominos[a] rotatedaround(center Dominos[a],90)
                withpen currentpen ;
            draw Dominos[b] rotatedaround(center Dominos[b],90) shifted (0,8)
                withpen currentpen ;
        fi ;
    ) ;
enddef ;

begingroup
    save unicode ; numeric unicode ; unicode := 127025 ; % 1F031

    for i=0 upto 6 :
        for j=0 upto 6 :
            lmt_registerglyph [
                category = "dominos",
                unicode  = unicode,
                code     = "DrawDominoH(" & decimal i & "," & decimal j & ");",
                width    = 16,
                height   = 8,
            ] ;
            unicode := unicode + 1 ;
        endfor ;
    endfor ;

    save unicode ; numeric unicode ; unicode := 127075 ;

    for i=0 upto 6 :
        for j=0 upto 6 :
            lmt_registerglyph [
                category = "dominos",
                unicode  = unicode,
                code     = "DrawDominoV(" & decimal i & "," & decimal j & ");",
                width    = 8,
                height   = 16,
            ] ;
            unicode := unicode + 1 ;
        endfor ;
    endfor ;
endgroup ;

\stopMPcalculation
```

Again we predefine a couple of features:

```
\definefontfeature
  [dominos:white]
  [default]
  [metapost={category=dominos}]

\definefontfeature
  [dominos:black]
  [default]
  [metapost={category=dominos,color=black}]

\definefontfeature
  [dominos:digits]
  [dominos:digits=yes]
```

This last feature is yet to be defined. We could deal with the invalid dominos with some substitution trickery but let's keep it simple.

```
\startluacode
local ligatures = { }
local unicode   = 127025

for i=0x30,0x36 do
    for j=0x30,0x36 do
        ligatures[unicode] = { i, j }
        unicode = unicode + 1 ;
    end
end

fonts.handlers.otf.addfeature("dominos:digits", {
    type      = "ligature",
    order     = { "dominos:digits" },
    nocheck   = true,
    data      = ligatures,
} )
\stopluacode
```

That leaves showing an example. We define a few fonts and again we just extend the Serif:

```
\definefont[DominoW][Serif*dominos:white]
\definefont[DominoB][Serif*dominos:black]
\definefont[DominoD][Serif*dominos:white,dominos:digits]
```

The example is:

```
\DominoW
    \char"1F043\quad \quad
    \char"1F052\quad \quad
    \char"1F038\quad \quad
    \darkgreen\char"1F049\quad \char"1F07B\quad
\DominoB
    \char"1F087\quad
    \char"1F088\quad
    \char"1F089\quad
\DominoD
    \darkred 12\quad56\quad64
```

Watch the ligatures in action:

To what extent the usage of symbols like dice and dominos as characters in the mentioned book are responsible for them being in Unicode, I don't know. Now with all these emoji showing up one can wonder about graphics in such a standard anyway. But for sure, even after more than three decades, Don still makes nice fonts.

A treasure of tiny graphics can be found in "pre-fascicle 5c" and many are in color! In fact, it has dominos too. It must have been a lot of fun writing this! I'm thinking of turning the pentominoes into a font where a GPOS feature can deal with the inter-pentomino kerning (which mighty work out okay for example 36. The windmill dominos also make a nice example for a font where ligatures will boil down to the base form and the (one or more) blades are laid over. It's definitely an inspiring read.

Hans Hagen