Bijlage 31

The Calculator Demo

Integrating TEX, MetaPost, JavaScript and PDF

Hans Hagen
pragma@pi.net

Keywords
MetaPost, JavaScript, PDF, CONTEXT, pdfTEX

abstract

Due to its open character, TEX can act as an authoring tool. This article demonstrates that by integrating TEX, METAPOST, JAVASCRIPT and PDF, one can build pretty advanced documents. More and more documents will get the characteristics of programs, and TEX will be our main tool for producing them. The example described here can be produced with PDFTEX as well as traditional TEX.

Introduction

When Acrobat Forms were discussed at the PDFTEX mailing list, Phillip Taylor confessed: "... they're one of the nicest features of PDF". Sebastian Ratz told us that he was "... convinced that people are waiting for forms.". A few mails later he reported: "I just found I can embed JAVASCRIPT in forms, I can see the world is my oyster" after which in a personal mail he challenged me to pick up the Acrobat Forms plugin and wishing me "Happy JavaScripting".

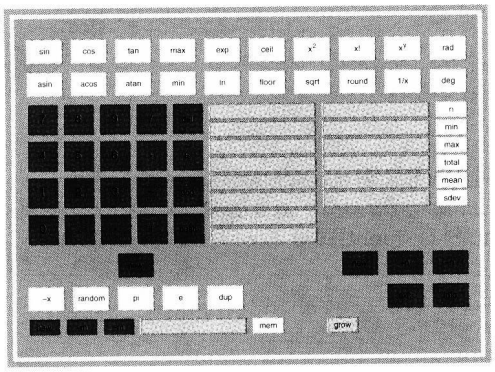


Figure 1 The calculator demo.

At the moment that these opinions were shared, I already had form support ready in CONTEXT, so picking up the challenge was a sort of natural behaviour. In this article I'll describe some of the experiences I had when building a demo document that shows how forms and JAVASCRIPT can be used from within TEX. I also take the opportunity to introduce some of the potentials of PDFTEX, so let's start with introducing this extension to TEX.

Where do we stand

While ε-TEX extends TEX's programming and typographic capabilities, PDFTEX primarily acts at the back end of the TEX processor. Traditionally, TEX was (and is) used in the production chain:

ASCII → TEX → DVI → whatever

The most versatile process probably is:

ASCII → TEX → DVI → POSTSCRIPT

or even:

ASCII → TEX → DVI → POSTSCRIPT → PDF

All functionality that TEX lacks, is to be taken care of by the DVI postprocessing program, and that's why TEX can do color and graphic inclusion. Especially when producing huge files or files with huge graphics, the POSTSCRIPT → PDF steps can become a nuisance, if only in terms of time and disk space.

With PDF becoming more and more popular, it will be no surprise that Han The Thanh's PDFTEX becomes more and more popular too among the TEX users. With PDFTEX we can reduce the chain to:

ASCII → TEX → PDF

The lack of the postprocessing stage, forces PDFTEX (i.e. TEX) to take care of font inclusion, graphic inserts, color and more. One can imagine that this leads to lively discussions on the PDFTEX mailing list and thereby puts an extra burden on the developer(s). Take only the fact that PDFTEX is already used in real life situations while PDF is not stable yet.

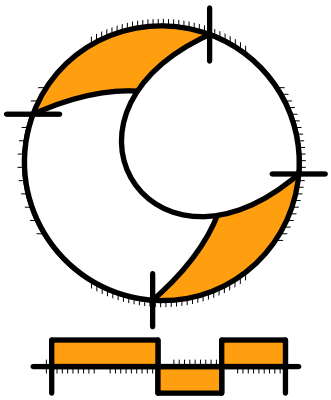
To those who know PDF, it will be no surprise that PDFTEX also supports all kind of hyper referencing. The version[1] I

Integrating TEX, MetaPost, JavaScript and PDF

Hans Hagen

Spring 1998

Due to it's open character, TEX can act as an authoring tool. This article demonstrates that by integrating TEX, METAPOST, JAVASCRIPT and PDF, one can build pretty advanced documents. More and more documents will get the characteristics of programs, and TEX will be our main tool for producing them. The example described here can be produced with PDFTEX as well as traditional TEX.



PRAGMA

Advanced Document Engineering | Ridderstraat 27 | 8061GH Hasselt NL
tel: +31 (0)38 477 53 69 | e-mail: pragma@wxs.nl | ConTEXt: www.pragma-ade.nl

This article was first published in the *Minutes and Appendices* of the NTG (Nederlandstalige TEX Gebruikersgroep), Issue 98.1. It was presented as paper at the 1998 annual meeting of the (international) TEX User Group that took place in Toruń (Poland).

Introduction

When Acrobat Forms were discussed at the pdfTeX mailing list, Phillip Taylor confessed: "... they're one of the nicest features of PDF". Sebastian Ratz told us that he was "... convinced that people are waiting for forms.". A few mails later he reported: "I just found I can embed JavaScript in forms, I can see the world is my oyster" after which in a personal mail he challenged me to pick up the Acrobat Forms plugin and wishing me "Happy JavaScripting".

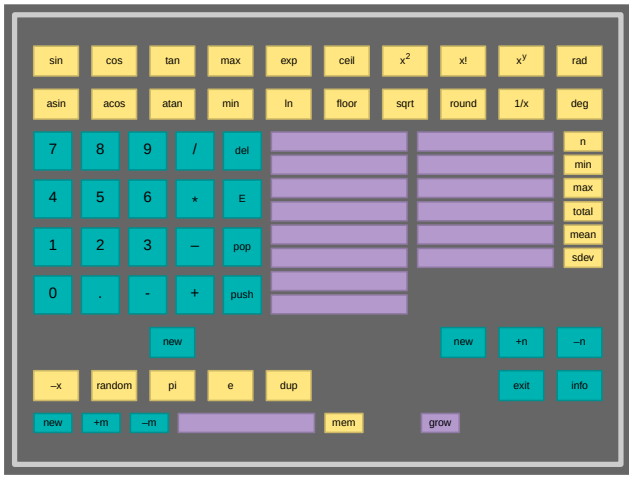


Figure 1 The calculator demo.

At the moment that these opinions were shared, I already had form support ready in ConTeXt, so picking up the challenge was a sort of natural behaviour. In this article I'll describe some of the experiences I had when building a demo document that shows how forms and JavaScript can be used from within TeX. I also take the opportunity to introduce some of the potentials of pdfTeX, so let's start with introducing this extension to TeX.

Where do we stand

While ε-TeX extends TeX's programming and typographic capabilities, pdfTeX primarily acts at the back end of the TeX processor. Traditionally, TeX was (and is) used in the production chain:

ASCII → TeX → DVI → whatever

The most versatile process probably is:

ASCII → TeX → DVI → PostScript

or even:

ASCII → TeX → DVI → PostScript → PDF

All functionality that TeX lacks, is to be taken care of by the DVI postprocessing program, and that's why TeX can do color and graphic inclusion. Especially when producing huge files or files with huge graphics, the PostScript → PDF steps can become a nuisance, if only in terms of time and disk space.

With PDF becoming more and more popular, it will be no surprise that Hàn Thế Thành's pdfTeX becomes more and more popular too among the TeX users. With pdfTeX we can reduce the chain to:

ASCII → TeX → PDF

The lack of the postprocessing stage, forces pdfTeX (i.e. TeX) to take care of font inclusion, graphic inserts, color and more. One can imagine that this leads to lively discussions on the pdfTeX mailing list and thereby puts an extra burden on the developer(s). Take only the fact that pdfTeX is already used in real life situations while PDF is not stable yet.

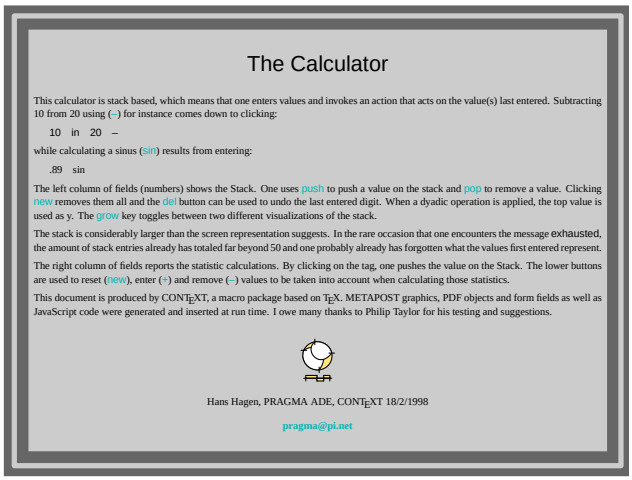


Figure 2 The help information screen.

To those who know PDF, it will be no surprise that pdfTeX also supports all kind of hyper referencing. The version[1] I used when writing this article supports:

1. link annotations
2. screen handling
3. arbitrary annotations

[1] Currently I'm using β-version 1.12g.

where especially the last one is accompanied by:

4. form objects
5. direct objects

and of course there is also:

6. extensive font support

Be prepared: PDFTEX's font support probably goes (and certainly will go) beyond everything DVI drivers as well as ACROBAT supports!

TEX stands in the typographic tradition and therefore has unsurpassed qualities. For many thousands of years people have trusted their ideas to paper and used glyphs for communication. The last decades however there has been a shift towards media like video, animations and interactive programs and currently these means of communication meet in hyper documents.

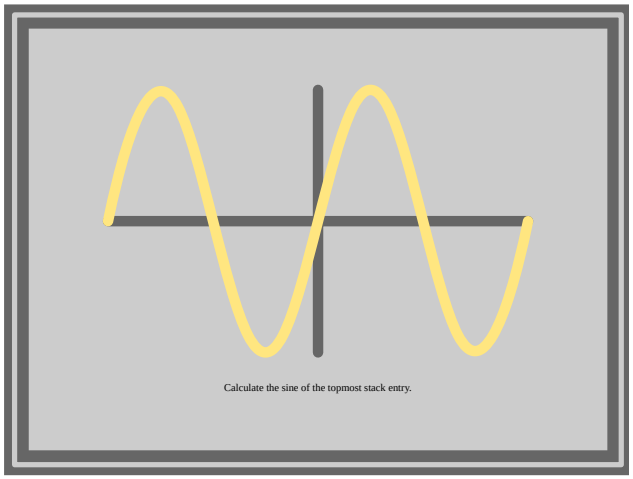


Figure 3 The $\sin(x)$ screen.

Now what has this to do with PDFTEX. Recently this program started to support the PDF annotations other than the familiar hyperlink ones. As we will see later on, this enables users of TEX to enhance their documents with features that until now had to be programmed with dedicated tools, which could not even touch TEX's typographic quality. This means that currently TEX has become a tool for producing rather advanced documents within the typographic and (largely paper based) communication traditions. Even better, by using PDF as medium, one can produce very sophisticated interactive documents that are not bound to ill documented standards and programs and thereby stand a better chance to be accessible for future generations.

The calculator demo

The document described here is produced with CONTEXT. This document represents a full featured calculator which took me about two weeks to design and build. Most of the time was spend on defining METAPOST graphics that could explain the functionality of the buttons.[2] Extending CONTEXT for supporting JAVASCRIPT took me a few days and the rest of the time was spend on learning JAVASCRIPT itself.

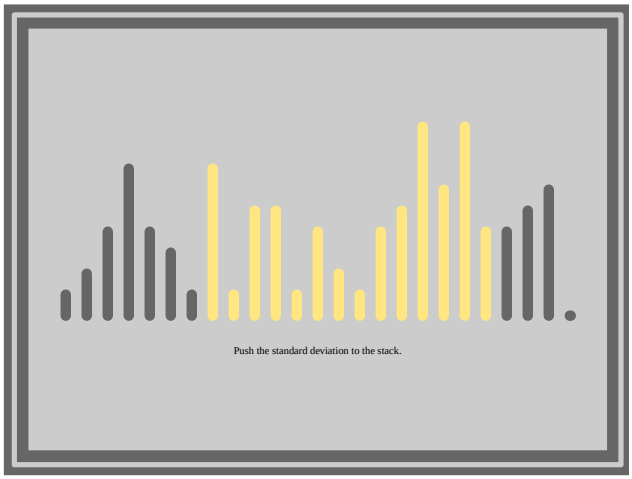


Figure 4 The standard deviation screen.

The calculator demo was first developed using DVIPSONE and ACROBAT. At that moment, PDFTEX did not yet provide the hooks needed, and the demo thereby served as a source of inspiration of what additional functionality was needed to let PDFTEX produce similar documents.

Throughout this article I show some of the screens that make up the calculator demo. These graphics are no screen dumps but just POSTSCRIPT inclusions. Just keep in mind that when using TEX, one does not need bitmap screen dumps, but can use snapshots from the real document. A screen, although looking as one graphic, consist of a background with frame, a centered graphic, some additional text and an invisible active area the size of the gray center.

The demo implements a stack based calculator. The stack can optionally grow in two directions, de-

[2] This included writing some auxiliary general purpose METAPOST macros.

pending on the taste of the user. Only the topmost entries of about 50 are visible.

The calculator demo, called `calculator.pdf`, itself can be fetched from the PDFTEX related site:

```
http://www.tug.org/applications/pdftex
```

or from the CONTEXT repository at:

```
http://www.ntg.nl/context
```

The calculator is defined in one document source file, which not only holds the TEX code, but also contains the definitions of the METAPOST graphics and the JAVASCRIPT's. I considered including a movie (video) showing an animation of our company logo programmed in METAPOST and prepared in Adobe Premiere, but the mere fact that movies are (still) stored outside the PDF file made me remove this feature.

Now keep in mind that, when viewing the calculator PDF file, you're actually working with a document, not a program. A rather intelligent document for that matter, but still a document.

Forms and annotations

Before I go into details, I'll spend some words on forms and annotations in PDF. To start with the latter, annotations are elements in a PDF file that are not related to (typo)graphic issues, like movies and sound, hyper things, navigation and fill–in–forms. Formally annotations are dealt with by drivers plugged into the graphic engine, but in practice some annotations are handled by the viewer itself.

Forms in PDF are more or less the same as in HTML and once filled in can be send over the net to be processed. When filling in form fields, run time error checking on the input can prevent problems later on. Instead of building all kind of validation options into the form editor, such validations are handled by either a dedicated plugin, or better: by means of JAVASCRIPT. Therefore, one can attach such scripts to all kind of events related to form editing and one can launch scripts by associating them to active, that is clickable, areas on the screen.

So we've got fields, which can be used to let users provide input other than mere clicks on hyper

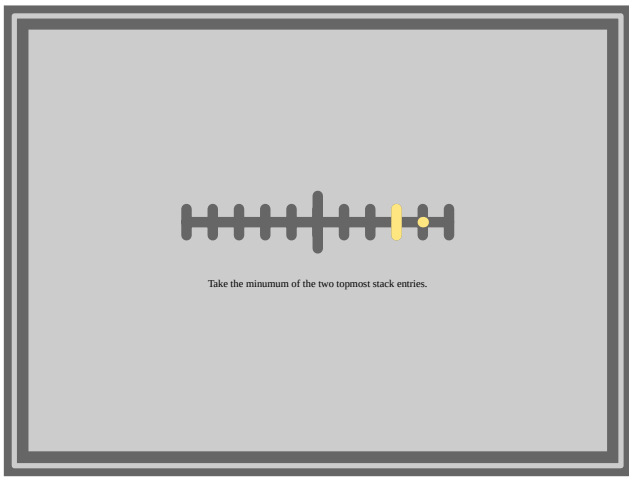


Figure 5 The $\min(x, y)$ screen.

links, we've got run time access to those fields using JAVASCRIPT, and we can let users launch such scripts by mouse events or keystrokes, either when entering data or by explicit request.

Currently entering data by using the keyboard is prohibited in the calculator. The main reason for this is that field allocation and access are yet sort of asynchronic and therefore lead to confusion.[3]

So, what actually happens in the calculator, is that a user clicks on a visualized key, thereby launching a JAVASCRIPT that in turn does something to field data (like adding a digit or calculating a sine), after which the field data is updated.

JAVASCRIPT

Writing this demo at least learned me that in fact support for JAVASCRIPT is just another sort of referencing and therefore needed incorporation in the general cross referencing scheme. The main reason is that for instance navigational tools like menus and buttons must have access to all cross reference mechanisms.

Consider for instance buttons. We already supported:

```
\button{...}[the chapter on whatever]
\button{...}[otherdoc::some topic]
\button{...}[previouspage]
\button{...}[PreviousJump]
```

[3] Initializing a field from within JAVASCRIPT is not possible unless the viewer has (at some dubious moment) decided that the field indeed exists.

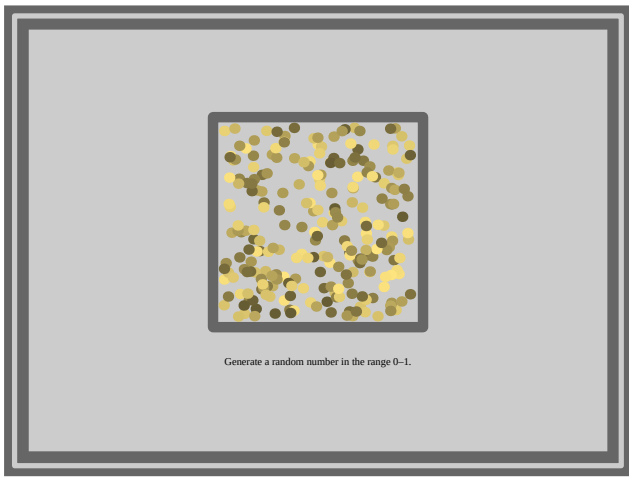


Figure 6 The random number screen.

Here the first reference is an internal one, often a chapter, a table or figure or a bibliography. The second one extends this class of references across documents. The third reference is a predefined internal one and the last reference gives access to viewer controls. As we can see: one scheme serves different purposes.

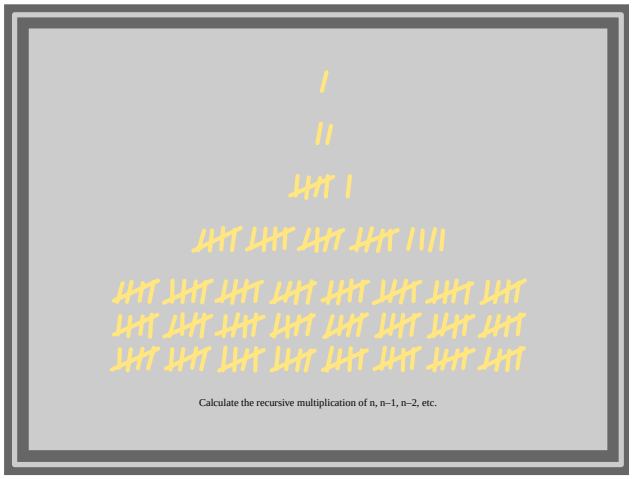


Figure 7 The period (.) screen.

Launching applications and following threads can quite easily be included in this scheme, but JAVASCRIPT support is different. In the calculator there are for instance 10 digit buttons that all do the same action and only differ in the digit involved. Here we want just one JAVASCRIPT to be reused 10 times. So instead of saying:

```
\button{0}[javascript 0]
\button{0}[javascript 1]
```

we want to express something like:

```
\def\SomeDigit#1%
  {\button{0}[javascript #1]}

\SomeDigit{4}
```

This means that in practice we need a referencing mechanism that:

- is able to recognize JAVASCRIPT
- is able to pass arguments to these scripts

So finally we end up with something:

```
\button{7}[JS(digit{7})]
```

This call tells the reference mechanism to access the JAVASCRIPT called `digit` and pass the value `7` to it. Actually defining the script comes down to just saying:

```
\startJScode{digit}
  Stack[Level] += String(JS_S_1);
  do_refresh(Level);
\stopJScode
```

One can pass as much arguments as needed. Here `JS_S_1` is the first string argument passed. Passing cross reference arguments is also possible. This enables us to let users jump to locations depending on their input. Such arguments are passed as `R{destination}` and can be accessed by `JS_R_1`.

Figure 8 The digit 7 screen.

In practice one will separate functions and calls by using preambles. Such preambles are document wide pieces of JAVASCRIPT, to be used whenever applicable.

```
\startJSpreamble{functions}
  // begin of common functions

  function do_digit(d)
    { Stack[Level] += String(d);
      do_refresh(Level) }

  // end of common functions
\stopJSpreamble
```

and:

```
\startJScode{digit}
  do_digit(JS_S_1);
\stopJScode
```

From these examples one can deduce that indeed the actual JAVASCRIPT code is included in the document source. It's up to TEX to pass this information to the PDF file, which in itself is not that trivial given the fact that one (1) has to strip comments, (2) has to convert some characters into legal PDF ones and (3) must pass arguments from TEX to JAVASCRIPT.

Simple cases like the digit code fragment, can also be passed as reference: `JS(digit{1})`. By default CONTEXT converts all functions present in the preambles into such references. One can organize JAVASCRIPTS into collections as well as postpone inclusion of preambles until they are actually used.

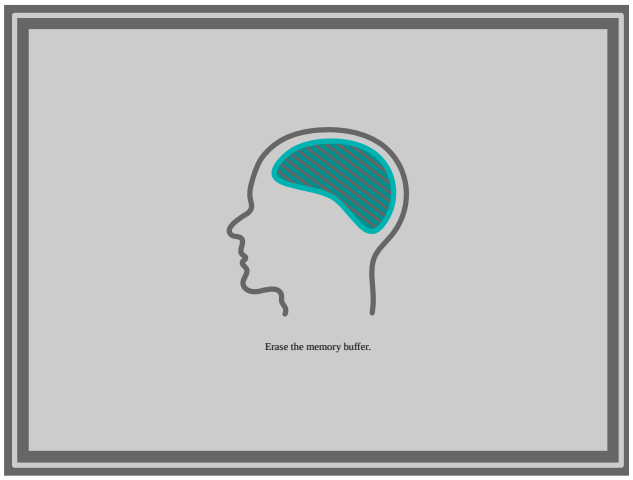


Figure 9 The memory erase screen.

Currently the only problem with including preambles lays in the mere fact that ACROBAT pdfmarks[4] not yet offer a mechanism to enter the JAVASCRIPT entries in the appropriate place in the document catalog, without spoiling the collected list of named destinations. Because CONTEXT can be instructed to use page destinations when possible, I could work around this (temporary) ACROBAT pdfmark and PDFTEX limitation. At the time this article is published, PDFTEX probably handles this conceptual weak part of PDF in an adequate way.

METAPOST graphics

All graphics are generated at run time using METAPOST. Like the previous mentioned script, METAPOST code is included in the source of the document. For instance, the graphic representing π is defined as:

```
\startuseMPgraphic{pi}
  pickup pencircle scaled 10;
  draw fullcircle
    scaled 150
    withcolor .4white;
  linecap := butt;
  ahlength := 25;
  drawarrow halfcircle
    scaled 150
    withcolor \MPcolor{action};
\stopuseMPgraphic
```

and called

```
\useMPgraphic{pi}
```

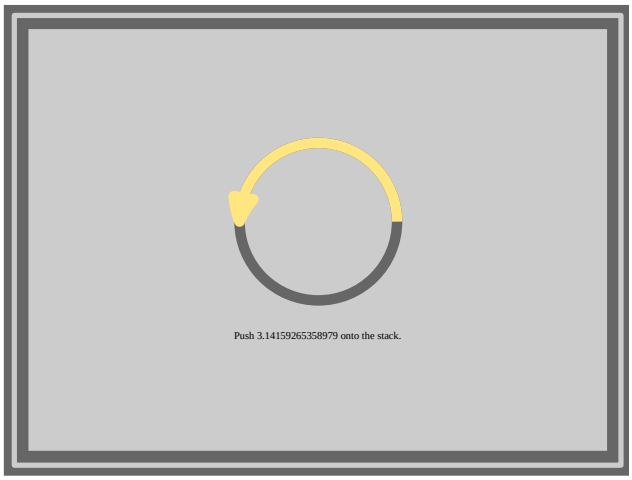


Figure 10 The π screen.

Just like the JAVASCRIPT preamble we can separate common METAPOST functions by defining inclusions. The next one automatically loads a module with some

[4] These are extensions to the POSTSCRIPT language.

auxiliary macros.

```
\startMPinclusions
  input mp-tool;
\stopMPinclusions
```

The mechanism for including METAPOST graphics is also able to deal with reusing graphics and running METAPOST itself from within TEX. In CONTEXT all processed METAPOST graphics are automatically translated into PDF by TEX itself, colors are converted to the current color space, and text is dealt with accordingly. Of course one needs to take care of proper tagging, but the next macro does this well:

```
\def\SomeShape#1#2%
  {\startreuseMPgraphic{shape:#1#2}
     draw fullcircle
       xscaled #1
       yscaled #2
   \stopreuseMPgraphic
   \reuseMPgraphic{shape:#1#2}}
```

Now we can say:

```
\SomeShape{100pt}{200pt}
\SomeShape{150pt}{180pt}
\SomeShape{120pt}{110pt}
```

Which just inserts three graphics with different sizes but similar line widths.

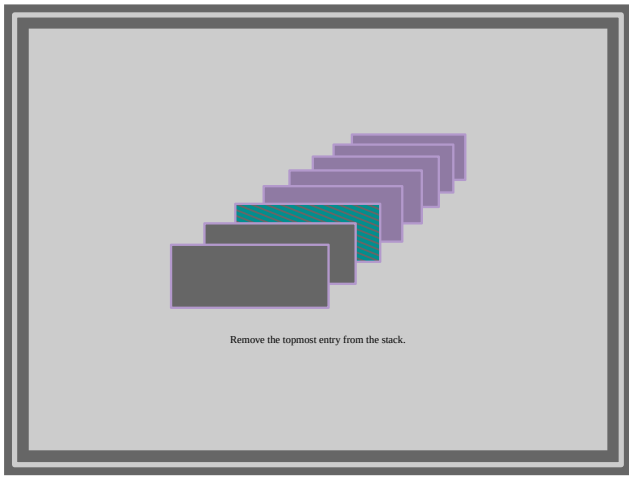


Figure 11 The pop stack screen.

Backgrounds

Now how do we attach such shapes to the buttons? Here we introduce a feature common to all framed things in CONTEXT, called overlays. Such an overlay is defined as:

```
\defineoverlay
  [shape]
  [\MPshape
     {\overlaywidth}
     {\overlayheight}
     {\overlaycolor}]
```

The shape called \MPshape is defined as:

```
\def\MPshape#1#2#3%
  {\startreusableMPgraphic{fs:#1#2#3}
     path p ;
     p := unitsquare
       xscaled #1
       yscaled #2;
     color c ;
     c := #3 ;
     fill p
       withcolor c ;
     draw p
       withpen pencircle scaled 1.5
       withcolor .8c ;
   \stopreusableMPgraphic
   \reuseMPgraphic{fs:#1#2#3}}
```

Such an overlay is bound to a particular framed thing by saying:

```
\setupbuttons[background=shape]
```

Here the right dimensions are automatically passed on to the overlay mechanism which in turn invokes METAPOST.

The calculator demo proved me that it is rather useful to have stacked backgrounds. Therefore the buttons, which have both a background (the METAPOST drawn shape) and behind that a sort of help button that is activated by clicking on the surroundings of the button, have their backgrounds defined as:

```
\setupbuttons
  [background={infobutton,shape}]
```

Actually we're stacking from back to top: an info button, the key bound button, the background graphic and the text. One rather tricky side effect is that stacked buttons interfere with the way active areas are

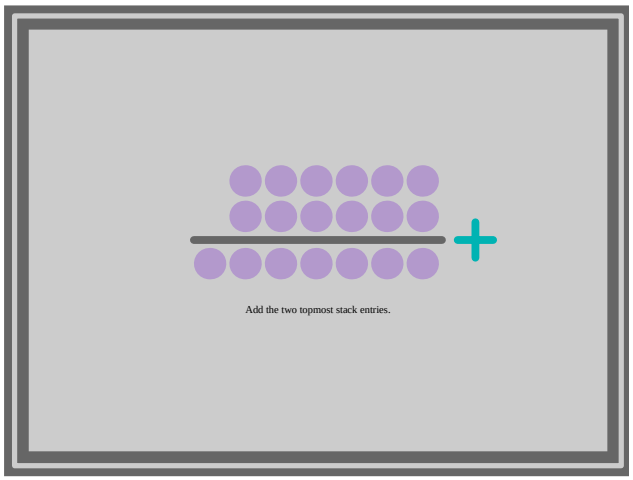


Figure 12 The addition (+) screen.

output. In this particular case we have to revert the order of the active areas by saying `\reversegototrue`.

Object reuse

The button and background graphics are generated once and used more than once. We already mentioned that METAPOST graphics can be reused. In practice this comes down to producing the graphic once and including it many times. In PDF however, one can also include the graphic once and refer to it many times. In PDF such reused objects are called forms, a rather unfortunate naming. So, in the calculator demo, all buttons with common shapes as well as the backgrounds are included only once. One can imagine that extending TEX with such features leads to interesting discussions on the PDFTEX discussion list.

Forms

Although still under construction, CONTEXT supports PDF fill-in-forms. The calculator demo demonstrates that such forms can be used as a (two way) communication channel to the user. Stack values, statistics and memory content are stored and presented in form fields, defined by saying something like:

```
\definefield[Stack.1][line][Results]
```

followed by

```
\field[Stack.1]
```

The characteristics of this line field are set by:

```
\setupfield
  [Results]
  [horizontal,frame]
  [width=fit,
   height=.5\ButtonWidth,
   background=shape,
   backgroundcolor=\MPcolor{stack},
   frame=off]
  [width=3.5\ButtonWidth,
   frame=off]
  [width=3.5\ButtonWidth,
   frame=off]
```

The reader needs some fantasy to grab the meaning of this rather overloaded setup. The first argument tags the characteristics, and can be considered something like a class in object oriented languages. The second argument tells CONTEXT how to typeset the field when labels are used, while the last three arguments specify the way fields, their labels and the envelop that holds them both together are typeset. In the calculator, the labels are suppressed.

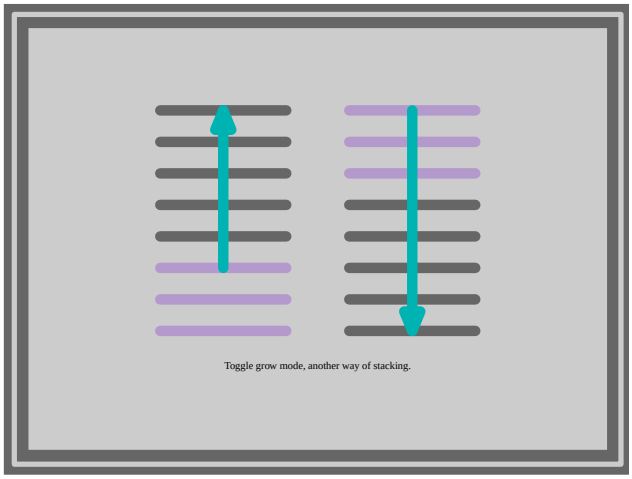


Figure 13 The grow mode screen.

One reason for decoupling definition and setup, that is, not attaching characteristics to individual fields, lays in the fact that I have applications in mind with thousands of fields and saving characteristics at the field level that would definitely overload TEX.

Where do we go

The previous examples show us quite clearly that, although being of old age in terms of computer programs, TEX is among the few applications that are able to adapt themselves rather fast to current develop-

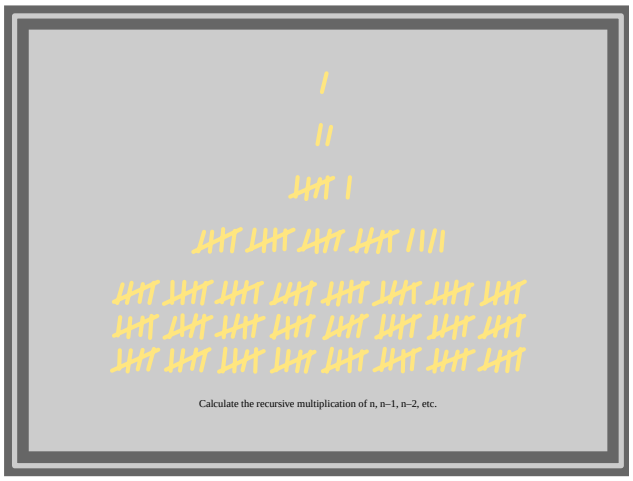


Figure 14 The *n*! screen.

ments while at the same time preserving the high quality and stability its users are accustomed to. As TEX gave mathematicians the means of circumventing the often lousy text editing and desk top publishing output in the early days of computing, TEX can give its users the high quality and stable authoring platform they need in this multi–media age. As demonstrated here, TEX can do a wonderful job not only in producing interactive documents, but in producing intelligent documents too.