Chickenize

Spaß mit node Manipulationen in LuaTEX

DANTE Herbsttagung 2011

Arno Trautmann

Inhalt

Vorteile von LuaTEX

- Native utf8-Unterstützung $\Longrightarrow$ nie mehr Probleme mit der „Schei?“ Kodierung

Vorteile von LuaTEX

- Native utf8-Unterstützung $\Longrightarrow$ nie mehr Probleme mit der „Schei?“ Kodierung
- (mit luaotfload) Laden beliebiger Systemschriften und Nutzen vielfältiger OpenType-Features
- Im Mathemodus: OpenType-Matheschriften
  (keine oder weniger separate Pakete für Zeichen)
- Verfügbarkeit einer „richtigen“ Programmiersprache

Vorteile von LuaTEX

- Native utf8-Unterstützung ⟹ nie mehr Probleme mit der „Schei?“ Kodierung
- (mit luaotfload) Laden beliebiger Systemschriften und Nutzen vielfältiger OpenType-Features
- Im Mathemodus: OpenType-Matheschriften
  (keine oder weniger separate Pakete für Zeichen)
- Verfügbarkeit einer „richtigen“ Programmiersprache
- Beeinflussung des Arbeitsablaufs von TEX möglich!

„Portieren“ von pdfLaTeX nach LuaLaTeX

Ohne Gewähr ...

- alle Dateien sollten utf8-kodiert sein
- `inputenc` und `fontenc` werden rausgenommen
- dafür wird `fontspec` (oder `luatextra`) geladen
- evtl. Hilfsprogramme auf utf8 anpassen

```
\documentclass{minimal}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\begin{document}
...
\end{document}
```

„Portieren“ von pdfLaTeX nach LuaLaTeX

Ohne Gewähr ...

- alle Dateien sollten utf8-kodiert sein
- `inputenc` und `fontenc` werden rausgenommen
- dafür wird `fontspec` (oder `luatextra`) geladen
- evtl. Hilfsprogramme auf utf8 anpassen

```
\documentclass{minimal}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\begin{document}
...
\end{document}
```

```
\documentclass{minimal}
\usepackage{fontspec}
\begin{document}
...
\end{document}
```

„Portieren“ von pdfLaTeX nach LuaLaTeX

Ohne Gewähr ...

- alle Dateien sollten utf8-kodiert sein
- `inputenc` und `fontenc` werden rausgenommen
- dafür wird `fontspec` (oder `luatextra`) geladen
- evtl. Hilfsprogramme auf utf8 anpassen

```
\documentclass{minimal}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\begin{document}
...
\end{document}
```

```
\documentclass{minimal}
\usepackage{fontspec}
\begin{document}
...
\end{document}
```

$\Longrightarrow$ „fertig“

unicode-math

```
\usepackage{unicode-math}
\setmathfont{XITS-math}
[...]
\[\oint ∫₀^∞ Σ ∑ = \oint \int_0^\infty \Sigma \sum\]
```

$$\oint \int_0^\infty \Sigma \sum = \oint \int_0^\infty \Sigma \sum$$

⟹ Direkte Eingabe z. B. mittels Neo-Tastaturlayout
gewohnte Befehle funktionieren aber weiter!
(bei `beamer`: Option `professionalfont` verwenden!)

breaking out: directlua

Um von TEX zu Lua zukommen: Primitive `\directlua`

```
1 \directlua{tex.print("hallo  welt")}
```

hallo welt

breaking out: directlua

Um von TEX zu Lua zukommen: Primitive `\directlua`

```
1 \directlua{tex.print("hallo welt")}
```

hallo welt

```
1 \directlua{
2 for i = 1,10 do
3   tex.print("hallo welt ".. i ..", ")
4 end
5 }
```

hallo welt 1, hallo welt 2, hallo welt 3, hallo welt 4, hallo welt 5, hallo welt 6, hallo welt 7, hallo welt 8, hallo welt 9, hallo welt 10,

breaking out: luacode

Paket luacode bietet Umgebung für längere Codestücke:

```
1 \def\hallo{Hallo Welt}
2 \begin{luacode}
3 for i = 1,10 do
4   tex.print("\hallo")
5 end
6 \end{luacode}
```

Hallo Welt Hallo Welt Hallo Welt Hallo Welt Hallo Welt Hallo Welt Hallo Welt Hallo Welt Hallo Welt Hallo Welt

Sehr lange Codestücke sollten in externer Datei `mycode.lua` gespeichert und dann aufgerufen werden:

```
\directlua(dofile("mycode.lua"))
```

Inhalt

Nodes in LuaTEX

- „Nodes“ sind der wichtigste Datentyp in LuaTEX:
- glyph, glue, hlist, vlist, rule, whatsit, ...

Nodes in LuaTEX

- „Nodes“ sind der wichtigste Datentyp in LuaTEX:
- glyph, glue, hlist, vlist, rule, whatsit, ...
- Aus einzelnen Nodes werden hierarchisch weitere nodes (oder Listen von nodes) gebildet:
  glyph, glue, kern, ... $\Longrightarrow$ hlist (Absatzumbruch)
  hlist, glue, ... $\Longrightarrow$ vlist (Seitenaufbau)

Nodes in LuaTEX

- „Nodes“ sind der wichtigste Datentyp in LuaTEX:
- glyph, glue, hlist, vlist, rule, whatsit, ...
- Aus einzelnen Nodes werden hierarchisch weitere nodes (oder Listen von nodes) gebildet:
  glyph, glue, kern, ... ⟹ hlist (Absatzumbruch)
  hlist, glue, ... ⟹ vlist (Seitenaufbau)
- Die Eigenschaften jedes nodes können gelesen und (teilweise, je nach Zusammenhang) auch geschrieben werden.
- Neue nodes können vom Nutzer erzeugt und geändert werden – ohne einen einzigen TEX-Befehl!
  ⟹ siehe `publisher` von Patrick Gundlach („TEX without TEX“)

Callbacks

- Callbacks bieten die Möglichkeit, an verschiedenen Stellen in TEX einzugreifen
- z. B. beim Font-Laden ($\Longrightarrow$ OpenType)

Callbacks

- Callbacks bieten die Möglichkeit, an verschiedenen Stellen in TEX einzugreifen
- z. B. beim Font-Laden (⟹ OpenType)
- aber auch vor, während und nach dem Zeilenumbruch: `pre_linebreak_filter`, `post_linebreak_filter`

Callbacks

- Callbacks bieten die Möglichkeit, an verschiedenen Stellen in TEX einzugreifen
- z. B. beim Font-Laden (⟹ OpenType)
- aber auch vor, während und nach dem Zeilenumbruch: `pre_linebreak_filter`, `post_linebreak_filter`
- Beide Callbacks sind normalerweise leer.
- Ermöglichen Manipulationen, Auswertungen und Ergänzungen für *jeden* Absatz (auch Überschriften etc.!)

Callbacks registrieren

- Funktionen können in Callbacks registriert werden

Callbacks registrieren

- Funktionen können in Callbacks registriert werden
- Dazu: entweder low-level Interface:
  `callback.register("pre_linebreak_filter",myfunction)`

Callbacks registrieren

- Funktionen können in Callbacks registriert werden
- Dazu: entweder low-level Interface:
  `callback.register("pre_linebreak_filter",myfunction)`
- oder high-levle Interface mit `luatexbase` (LaTeX und plainTeX):
  ```
  luatexbase.add_to_callback(
        "pre_linebreak_filter",myfunction,"meine Funktion",1)
  ```
- `add_to_callback` kann mehrere Funktionen in einen Callback registrieren; Löschen von Funktionen mit `remove_from_callback`
- üblicher Ablauf: Callback erhält eine Nodelist, verarbeitet sie, und gibt eine (geänderte oder neue) Nodelist zurück
- Nötige Werte und Rückgabe der Funktion $\Longrightarrow$ siehe LuaTeX Dokumentation

pre_linebreak_filter

Aus der Dokumentation:

This callback is called just before LuaTEX starts converting a listof nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

pre_linebreak_filter

Aus der Dokumentation:

This callback is called just before LuaTEX starts converting a listof nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

Aus der Dokumentation:

```
function(<node> head, <string> groupcode)
 return true | false | <node> newhead
end
```

pre_linebreak_filter

Aus der Dokumentation:

This callback is called just before LuaTEX starts converting a listof nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

Aus der Dokumentation:

```
function(<node> head, <string> groupcode)
 return true | false | <node> newhead
end
```

Eine Definition, die nichts macht:

```
donothing = function(head)
  return head
end
luatexbase.add_to_callback(
  "pre_linebreak_filter",donothing,"nichts"
)
```

pre_linebreak_filter

Eine Funktion, die alle Zeichen durchgeht:

```
allglyphs = function(head)
  for i in node.traverse_id(37,head) do
    -- beliebige Manipulation der Nodes
  end
  return head
end
luatexbase.add_to_callback(
  "pre_linebreak_filter",allglyphs,"alle Zeichen"
)
```

post_linebreak_filter

Aus der Dokumentation:

This callback is called just after LuaTEX has converted a list of nodes into a stack of \hboxes.

Aus der Dokumentation:

```
function(<node> head, <string> groupcode)
  return true | false | <node> newhead
end
```

post_linebreak_filter

Eine Funktion, die alle Zeichen *im Absatz* durchgeht:

```
allglyphs_post = function(head)
  -- zuerst durch alle hbox' gehen
  for line in node.traveres_id(0,head)
    -- dann durch alle Zeichen in der Liste,
      -- deren Anfang line.head ist
    for i in node.traverse_id(37,line.head) do
      -- beliebige Manipulation der Nodes
    end
  end
  return head
end
luatexbase.add_to_callback(
  "post_linebreak_filter",allglyphs_post,"alle Zeichen"
)
```

Inhalt

Aufgabe

„Ersetze alle *e* und *E* durch *3*“
[und weitere 1337-Umschrift]

Aufgabe

„Ersetze alle *e* und *E* durch *3*“
[und weitere 1337-Umschrift]

- Ansatz: Für jeden Node `x` vom Typ `glyph` mit `x.char = e` $\Longrightarrow$ ändere `x.char` zu `3`.
- Callback: `post_linebreak_filter`, da sonst die Trennungen nicht richtig funktionieren.

1337

Aufgabe

„Ersetze alle *e* und *E* durch *3*“
[und weitere 1337-Umschrift]

- Ansatz: Für jeden Node `x` vom Typ `glyph` mit `x.char = e` $\Longrightarrow$ ändere `x.char` zu 3.
- Callback: `post_linebreak_filter`, da sonst die Trennungen nicht richtig funktionieren.
- Problem: Da die Zeichen unterschiedliche Breiten haben, stimmt die Zeilenbreite nicht!
- Mögliche lösung: Lies Breite der Zeichen aus und justiere nachträglich ...
- Code: Siehe Paket `chickenize`

Farben einfügen

Aufgabe

„Mache alle Großbuchstaben farbig!“

- Ansatz: Für jeden glyph-node prüfe, ob Großbuchstabe (mittels `.char`)
- dann: füge einen „Farbe-einfügen-node“ vor dem glyph ein
- füge einen „Farbe-aufheben-node“ nach dem glyph ein
- Callback: `post_linebreak_filter`, da sonst die Trennungen nicht richtig funktionieren.
- Nachteil: keiner ☺

Farben einfügen

- „Farb-Knoten“ sind nodes vom Typ `whatsit`
- `whatsit`-nodes haben einen `subtyp`, hier: `pdf_colorstack`

Farben einfügen

- „Farb-Knoten“ sind nodes vom Typ `whatsit`
- `whatsit`-nodes haben einen `subtyp`, hier: `pdf_colorstack`

Vorgehen:

- erzeuge einen `push`-node und einen `pop`-node
- weise alle relevanten Eigenschaften zu (Farbe)
- füge *Kopien* in die node-Liste ein (mittels `.prev` und `.next`)
- Rückgabe der fertigen Liste

Farben einfügen

- „Farb-Knoten“ sind nodes vom Typ `whatsit`
- `whatsit`-nodes haben einen `subtyp`, hier: `pdf_colorstack`

Vorgehen:

- erzeuge einen `push`-node und einen `pop`-node
- weise alle relevanten Eigenschaften zu (Farbe)
- füge *Kopien* in die node-Liste ein (mittels `.prev` und `.next`)
- Rückgabe der fertigen Liste
- füge weitere Spielereien ein: Zufallsfarben, Berücksichtigen von Kapitälchen, Regenbogenfarben, ...
- Code: Siehe Paket `chickenize`

Grauwert anzeigen

Aufgabe

„Stelle die Dehnung/Stauchung jeder Zeile, also den Grauwert, mit einem grauen Balken dar!“

Grauwert anzeigen – Glue

- Ansatz: Lies für jede Zeile die `glue_ratio` aus

Grauwert anzeigen – Glue

- Ansatz: Lies für jede Zeile die `glue_ratio` aus
- Erstelle eine breite Linie (rule)
- färbe die Linie mit einer Farbe, die dem `glue_ratio` entspricht
- füge das ganze als Knoten vor der betreffenden Zeile ein

Grauwert anzeigen – Glue

Codebeispiel

```
if line.glue_order == 0 then
  if line.glue_sign == 1 then
    glue_ratio = .5 * math.min(line.glue_set,1)
  else
    glue_ratio = -.5 * math.min(line.glue_set,1)
  end
end
color_push.data = .5 + glue_ratio .. " g"
```

Grauwert anzeigen – Expansion

- Ansatz: Suche den ersten Buchstaben `g` der Zeile
- Lies dessen Breite aus: `g.width`

Grauwert anzeigen – Expansion

- Ansatz: Suche den ersten Buchstaben `g` der Zeile
- Lies dessen Breite aus: `g.width`
- vergleiche dies mit der Breite des Buchstabens aus dem aktuellen Font
- Das Verhältnis ist die Font Expansion!

  $\Longrightarrow$ Als Farbe kodieren und als zweiten Balken ausgeben lassen

Grauwert anzeigen – Expansion

Codebeispiel

```
  local f = font.getfont(font.current()).characters
[...]
    if colorexpansion then
      local g = line.head
        while not(g.id == 37) do
         g = g.next
        end
      exp_factor = g.width / f[g.char].width
      exp_color = .5 + (1-exp_factor)*10 .. " g"
```

Aufgabe

„Ersetze jedes Wort durch ‚chicken‘“

Aufgabe

„Ersetze jedes Wort durch ‚chicken‘“

- Ansatz: Finde Wörter
  (fängt an mit einem Buchstaben, hört auf mit „Nicht-Buchstaben“)
- Vorsicht: Auch Trennstellen und Kerning sind Nicht-Buchstaben!

Aufgabe

„Ersetze jedes Wort durch ‚chicken‘“

- Ansatz: Finde Wörter
  (fängt an mit einem Buchstaben, hört auf mit „Nicht-Buchstaben“)
- Vorsicht: Auch Trennstellen und Kerning sind Nicht-Buchstaben!
- pre_linebreak_filter, um korrekten Satz zu ermöglichen
- Merke Knoten vor und hinter dem Wort
- Füge das Wort `chicken` inklusive Trennstellen als Node-List ein
- Baue alles zusammen – „fertig“!

Wörter bauen in Lua

- Siehe am besten: „TEX without TEX“ von Patrick Gundlach

$\Longrightarrow$ Man kann allein auf der Lua-Seite arbeiten

Wörter bauen in Lua

- Siehe am besten: „TEX without TEX“ von Patrick Gundlach

⟹ Man kann allein auf der Lua-Seite arbeiten

- `table` Konstruktion: `chicken =`
- erzeuge `glyph`-Nodes und weise Schrift zu
- weise Buchstaben in einer Schleife zu
- Einfügen von Trennstellen mittels `lang.hyphenate`
- [Kerning und Ligaturen fehlen noch]

Inhalt

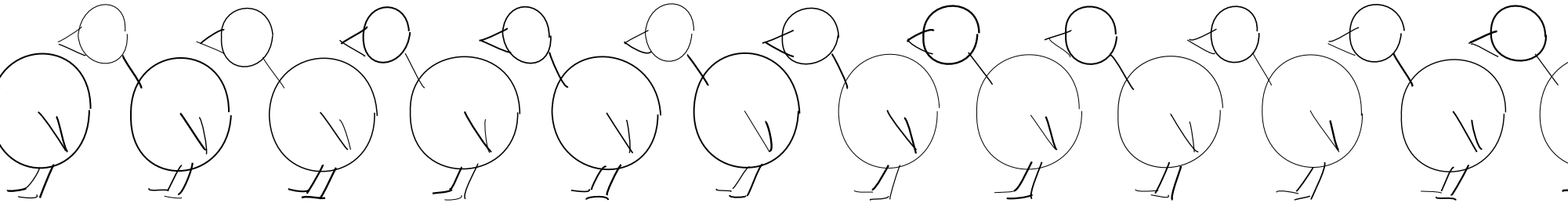
Überblick – was kann chickenize momentan?

- colorstretch
- chickenize, leetspeak
- random – uclc, fonts, chars, color
- rainbow, uppercase – color
- viele Parameter für die Anpassung (nur zufällige Wörter ersetzen/färben)
- Einfügen von Pfannkuchenrezepten und gezeichneten Hühnchen (geplant/experimentell)
- [viele kreative Vorschläge der Anwesenden ...]

Überblick – was kann chickenize momentan?

- colorstretch
- chickenize, leetspeak
- random – uclc, fonts, chars, color
- rainbow, uppercase – color
- viele Parameter für die Anpassung (nur zufällige Wörter ersetzen/färben)
- Einfügen von Pfannkuchenrezepten und gezeichneten Hühnchen (geplant/experimentell)
- [viele kreative Vorschläge der Anwesenden ...]

- bald (evtl. nach Überarbeitung des Codes) auf CTAN



HAPPY TEXING!