Three things you can do with LuaTeX that would be extremely painful otherwise

Paul Isambert

Introduction

LuaTeX has made some typographic operations so easy one might wonder why it wasn't invented thirty years ago (probably because Lua didn't exist then).

Here I'm going to describe three simple features that would require advanced wizardry to do the same with any other engine. LuaTeX allows you to explore some of TeX's most intimate parts with a rather easy programming language, and the result is you can quite readily access things that were unreachable before. The three issues I'm going to address are:

- Turning lines into rules whose color depends on the line's original stretch or shrink.
- Underlining.
- Margin notes that align properly with the text.

I'll try to explain some of LuaTeX's basic functionality as we encounter these issues, but two of them are worth mentioning right now: callbacks and nodes.

First, we can control TeX's operations at various stages thanks to *callbacks*. These are points at which we can insert Lua code to modify or enhance TeX's processing. Callbacks range from processing TeX's input buffer (e.g. to accommodate a special encoding) to rewriting the paragraph builder and loading OpenType fonts.

Second, we can manipulate lists of *nodes*. To put it simply, nodes are the atoms that TeX uses to create pages: boxes, glyphs, glues, but also penalties, whatsits, etc. A list of nodes is a sequence of such atoms linked together. A simple paragraph, for instance, is a list made of horizontal boxes (the lines), penalties and glues. The boxes themselves are lists containing mostly glyph and glue nodes. Nodes are linked together like beads on a string, and the `prev` field of a node points to the preceding node in the list, whereas the `next` field returns the one that follows (there is an understandable exception for the first and last nodes of a list, whose `prev` and `last` fields respectively return `nil`). An important point to keep in mind is that when you query the content of, say, an `\hbox`, which in TeX's internal is a horizontal list, what you get is the first node of that list; you access the rest by sliding from `next` to `next`.

Author's note: I'm not a member of the LuaTeX team and this paper has no kind of official authority — it's just the result of experimentation by a LuaTeX user. Any error or misconception is mine.

Nodes also have several other *fields*, depending on their types. These types are recorded as a number in their `id` field, a numeric value. For instance, a glue node has `id` 10, whereas a glyph node has `id` 37. As long as LuaTeX hasn't reached version 1, though, such values might change. So, in order for our code to last, we must use the following workaround: the `node.id()` function, when fed a string denoting a node type, returns the associated `id` number. For instance, `node.id("glue")` returns `10`. Thus, when using symbolic names, we can get the right `id` value, regardless of changes in versions of LuaTeX. Another important field for nodes is `subtype`, which distinguishes between nodes with the same `id`. It's a numeric value, and for whatsits (which are numerous), one should use `node.subtype()` like `node.id()`.

Symbolic names won't change; they are listed in the LuaTeX reference manual, in the chapter called *Nodes*, available from the LuaTeX web site; they're also listed in the tables returned by `node.types()` and `node.whatsits()`. It's simpler to define variables beforehand rather than call `node.id` and `node.subtype` each time we need them. That's what we'll do here: the following declarations should start any file containing our code; it can also be made global by removing the `local` prefix and thus used anywhere once declared, but local variables are faster and safer. I use uppercase to mark their status.

```
local HLIST = node.id("hlist")
local RULE  = node.id("rule")
local GLUE  = node.id("glue")
local KERN  = node.id("kern")
local WHAT  = node.id("whatsit")
local COL   = node.subtype("pdf_colorstack")
```

The color of a page

Typographers speak of a page's color. While the color itself depends on several factors, its *evenness* depends on how lines are justified: loose lines make the page uneven in color, because large interword space creates holes in the overall greyness.

The code that follows takes the metaphor literally: it turns a page's color into a real color pattern. The idea is to replace each line with a rule of the same height and width, and whose color depends on the line's badness. If we take 0 as black and 1 as white, then a good line gets .5, tight lines approach 0 (which represents an overfull line) and loose lines tend to 1 (an underfull line). Now we have paragraphs and pages made of grey bars; the less contrast between them, the better the page.

To do this, we retrieve the horizontal boxes created by the paragraph builder, check the badness of each, then replace the box with the desired rule. This is easy to do in LuaTEX: we register a function in the `post_linebreak_filter` callback. This callback accesses the list of nodes output by the paragraph builder, i.e. the lines of text interspersed with interline penalties and glues, plus perhaps other things (whatsits, inserts, ...) that we'll ignore. Among these nodes we retrieve the ones we want, namely the lines of text, and replace them as described.

The code that follows, as all Lua code, should be fed to `\directlua` or stored in a `.lua` file.

```
local color_push = node.new(WHAT, COL)
local color_pop  = node.new(WHAT, COL)
color_push.stack = 0
color_pop.stack  = 0
color_push.cmd   = 1
color_pop.cmd    = 2
```

Here we have created two new whatsit nodes identified by their `subtype` as the Lua equivalents of `\pdfcolorstack`. They both modify stack 0 and `color_push` adds code to the stack while `color_pop` removes it. We'll use them to set the color of each line, with the exact content of the code added by `color_push` to be specified each time.

```
textcolor = function (head)
  for line in node.traverse_id(HLIST, head) do
    local glue_ratio = 0
    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 * line.glue_set
      end
    end
    color_push.data = .5 + glue_ratio .. " g"
```

Here's the beginning of our main function. It takes a node as its argument: it will be the first node of the list returned by the paragraph builder. That node, remember, denotes the entire list. We retrieve each line of text in this list, i.e. each node with `id HLIST`, and check its `glue_order` field; if it is 0, then the line has been justified with finite glue and we want to know how bad it is (if the line uses infinite glue then it is good by definition, as far as glue setting is concerned). We access `glue_sign` to know whether stretching or shrinking was used and `glue_set` to know the ratio (1 means the stretch/shrink was fully used; glues can also be overstretched, but we don't allow more than 1 in order to remain in the color range).

The last line sets the color of the line as the code to `color_push`, i.e. '*n* `g`', where *n* is a number between 0 and 1 and `g` a PDF operator setting the color in the grey model. In the rest of the loop we replace the line's content with a sequence of three nodes: `color_push`, a rule, and `color_pop`:

```
    local rule = node.new(RULE)
    rule.width = line.width
    local p = line.list
    line.list = node.copy(color_push)
    node.flush_list(p)
    node.insert_after(line.list,
      line.list, rule)
    node.insert_after(line.list,
      node.tail(line.list),
      node.copy(color_pop))
  end
```

What is done here is: first, we create a rule whose width is the same as the original line's (we could have created this rule beforehand with a width equal to `\hsize`, but this way we accommodate changing line widths). Then we set the line's list as a copy of `color_push` (we use a copy since we need that node for each line), and then we insert the rule node and a copy of `color_pop`. The first argument to `node.insert_after` is the list (denoted by its first node!) where we perform the insertion, the second one is the node in that list after which the insertion is performed, and the last one is the inserted node; `node.tail` returns the last node of its argument, so the third `node.insert_after` inserts at the end of the list.

The story with `p` is this: we retrieve the line's content before replacing it, so we can erase it from TEX's memory; it has no effect on the output.

Finally, and most importantly, we return the mutated list for TEX to continue its operations, and close the function.

```
  return head
end
```

Now, to use the function, we register it in the `post_linebreak_filter` callback:

```
\directlua{%
  callback.register("post_linebreak_filter",
    textcolor)}
```

Note that we could improve this code for the first and last lines of a paragraph, taking the indent and `\parfillskip` into account to create more faithful images of those lines. I leave it as an exercise to the reader, as is customary.

Underlining

The previous code was (hopefully) fun but not terribly useful (well, who knows?); let's do something (hopefully) more useful and no less fun.

Everybody knows that underlining is in bad typographic taste. That said, it may have its uses, and anyway allows us to investigate LuaTEX further. Underlining has been done in TEX (see Donald Arseneau's `ulem`, for instance); it requires great wizardry and has some limitations. With LuaTEX, it's (almost) child's play.

The problem with underlining in TEX is that you have to add the underline before the paragraph is built, and this hinders hyphenation. In LuaTEX we can do it after hyphenation is done: we retrieve the nodes to underline in the typeset lines. But how do we spot them? The answer lies with another basic LuaTEX functionality, namely attributes. These are very simple yet very powerful. An attribute is like a count register in that it holds a number. The difference with a count register is that nodes retain the values of all attributes in force *when they were created*. Thus, we can set an attribute to some value, input some text, and then reset the attribute; the text will have the value attached to it for the rest of TEX's processing.

This leads to the first definition:

```
\def\underline#1{%
  \quitvmode \attribute100 = 1 #1%
  \attribute100 = -"7FFFFFFF
  \directlua{callback.register(
    "post_linebreak_filter", get_lines)}%
}
```

It's important to use `\quitvmode` so that the indentation box is inserted before the attribute is set and not be underlined (in case the underlined text is the beginning of a paragraph).

An attribute is 'set' if it has any value but `-"7FFFFFFF`. So setting it to 1 here would be the same thing as setting it to -45 (see the end of this section for an example of use for different values). Now all nodes produced by the argument to `underline` have the value 1 for attribute 100 — which was arbitrarily chosen. Attribute 458 would have been equally good. Actually, one should use attributes with greater care, i.e. they should be allocated with macros like `\newcount`, so that one never uses the same attribute for different tasks.

The last action performed by `\underline` is to register a function in the `post_linebreak_filter` callback. It does so because the Lua function used to underline clears the callback (as we'll see), so that it is called only on those paragraphs where it is required. It could be called on all paragraphs, but it'd waste TEX's time.

Let's now turn to the Lua functions:

```
get_lines = function (head)
  for line in node.traverse_id(HLIST, head) do
    underline(line.list, line.glue_order,
      line.glue_set, line.glue_sign)
  end
  callback.register
    ("post_linebreak_filter", nil)
  return head
end
```

This first function retrieves all lines in the paragraph and feeds their content to the `underline` function along with information about glue setting. It then clears the callback and returns the head. This part is nothing we haven't seen in the previous code.

Some nodes might have inherited the attribute's value, although we don't want to underline them: `\leftskip`, `\rightskip`, and `\parfillskip`. These are glue nodes and their `subtypes` are 8, 9 and 15, respectively. The following function is meant to filter them out. (Note: versions prior to v0.62 had a bug where `\leftskip` and `\rightskip` were not properly identified, so `item.subtype == 7` should be added to the `or` conditional below. Both TEX Live 2010 and MikTEX 2.9 uses v0.60, so they are affected.)

```
local good_item = function (item)
  if item.id == GLUE and
   (item.subtype == 8 or item.subtype == 9
   or item.subtype == 15) then
    return false
  else
    return true
  end
end
```

Now, here's how the `underline` Lua function starts:

```
underline =
  function (head, order, ratio, sign)
  local item = head
  while item do
    if node.has_attribute(item,100)
     and good_item(item) then
      local item_line = node.new(RULE)
      item_line.depth = tex.sp("1.4pt")
      item_line.height = tex.sp("-1pt")
```

The `while` loop is basically the same thing as traversing the list, but we'll sometimes want to skip nodes, so we'll set the `next` one by hand. We scan nodes, and once we've found one with the right value for the attribute (and which is not one of the glues above), we create our rule (with arbitrary dimensions). `tex.sp` turns a dimension (expressed as a string) into scaled points, the native measure for Lua code. How wide should the rule be? The length of the material starting at the current node up to the last node with the right attribute. To find this last node, we use the following loop, and then retrieve the length of that material via `node.dimensions`, which returns the material's length when it is typeset with the text line's glue setting. We use `end_node.next` because the function actually measures up to its last argument's `prev` node.

```
      local end_node = item
      while end_node.next and
       good_item(end_node.next) and
       node.has_attribute(end_node.next, 100) do
        end_node = end_node.next
      end
      item_line.width = node.dimensions
        (ratio, sign, order, item, end_node.next)
```

Finally we insert the line into the list. That's pretty simple: we insert a negative kern (with `subtype` 1, i.e. a handmade kern, not a font kern) as long as the line after the last underlined node, followed by the line itself. This is equivalent to using `\llap` in plain TEX. The end of the code sets the `next` node to be analyzed (including the false part of the overall conditional).

```
      local item_kern = node.new(KERN, 1)
      item_kern.kern = -item_line.width
      node.insert_after(head, end_node,
                        item_kern)
      node.insert_after(head, item_kern,
                        item_line)
      item = end_node.next
    else
      item = item.next
    end
  end
end
```

We could use different values of the attribute to distinguish different underlining styles. To do so, we would still use `node.has_attribute`, since it returns the value of the attribute, or `nil` if the attribute isn't set. That's another exercise left to the reader.

Marginal notes

When a document has comfortable margins and notes are infrequent and short, marginal notes are an elegant and convenient alternative to footnotes. They are best typeset with their first line level with the line in the text to which they refer. However, such a rule cannot be absolute. Suppose for instance that a note is called on the last line of a page, and itself is made of more than one line. If we follow the rule then the note will invade the bottom margin and ruin the design of the page. So it should be shifted up so that its last line is level with the last line of the page. Doing this is also an improvement when the text doesn't fill the page, e.g. at the end of a chapter, even though there might remain space on the page to accommodate the note. The page looks better that way: a note is a note and would be too conspicuous if it were allowed to run without the main text by its side. Ideally, a note should also be shifted up if it runs along a section break, but I'll ignore that case, to keep things simpler. (For an alternative approach in LATEX, see Stephen Hicks' article in *TUGboat* 30:2.)

Generally marginal notes are typeset in a smaller font size and on a smaller leading than the main text. Since the leading is smaller, some lines of the notes won't be level with the textblock's lines; however, there should be some 'cyclical synchronicity' between the two blocks, so that for instance three lines of the main text have the same height as four lines of the note (in TEX terms it would mean, for instance, `\baselineskip` at 12pt and 9pt respectively), and the following lines are level again.

Here, however, I will typeset notes with the same leading as the main text to avoid complications. Extra calculations are required to achieve what's been previously described — nothing very complicated, though. I'll simply use italics to distinguish the notes from the main text.

Margin notes so numerous that they sometimes overlap each other and must be shifted upward should probably be converted to footnotes, all the more as they'll require a number or symbol so the reader can spot where in the main text they refer to — whereas sparse notes don't need such a mark, since they're supposed to start on the same line as the text they comment, with the known exception we're investigating here. However, we can use the code below to shift notes whatever the reason, so we'll leave æsthetics aside and shift all notes (the shift might go wrong if there are stretchable vertical glues on the page, e.g. `\parskip`; that can be amended, and it's left as yet another exercise). We won't allow more than one note per line, though, because that definitely doesn't make sense.

Here's the TEX part of the code:

```
\newcount\notecount
\suppressoutererror=1
\def\note#1{%
  \advance\notecount 1
  \expandafter\newbox
    \csname marginnote_\the\notecount\endcsname
  \expandafter\setbox
    \csname marginnote_\the\notecount\endcsname=
    \vtop{\hsize=4cm
      \rightskip=0pt plus 1fil
      \noindent\it #1}%
  \bgroup
  \attribute100=\expandafter\the
    \csname marginnote_\the\notecount\endcsname
  \vadjust pre {\pdfliteral{}}%
  \egroup
}
```

This might be somewhat unfamiliar, even to advanced TEXies, because what we're doing is preparing the ground for Lua code. First, we choose not to insert the note directly in the paragraph (to be shifted later if necessary). Instead, we store the note in a box. For each note, we create a new box; that might seem somewhat resource-consuming, but there are 65,536 available boxes in LuaTEX, so a shortage seems only a distant possibility. Alternatively, we could store only the source code for the note (in a macro), and typeset it in a box only when we place notes on the page in the output routine, but the asynchronicity between the processing of the main text and the note might lead to trouble.

So we create boxes instead, with proper settings (mostly, a reduced `\hsize`). To allow `\newbox` to appear inside a macro definition in plain TEX, we suppress the `outer` error beforehand; then we set the note in its box with a uniquely defined name (thanks to `\newcount`), and most importantly we set an attribute to the value of the box register and `\vadjust` a literal with that attribute. This literal's only role is to mark the line it comes from, so we'll be able to spot lines with margin notes when needed, along with the box's number (the value of the attribute).

The following Lua function, to be inserted in the `post_linebreak_filter` callback, does exactly that: our special `\pdfliteral`s give their attributes to the lines they come from, and are removed. Now, the reader might have wondered why we used the `pre` version of `\vadjust` instead of the default: it's because of a bug in the actual version of LuaTEX (to be fixed in v0.64, I am told): some `prev` fields are sometimes wrong, as would be the case here, and we couldn't link each literal to its line if the latter was before the former. So we use `next` instead. Note that we can't just take for granted that the first `next` node is the line, first because '`pre-\vadjust`ed' material is inserted before the baselineskip glue, and because there might be more adjusted material between the literal and the line. So we recurse over `next` fields until we find a line (i.e. a node `id HLIST`).

```
mark_lines = function (head)
  for mark in node.traverse_id(WHAT, head) do
    local attr = node.has_attribute(mark, 100)
    if attr then
      local item = mark.next
      while item do
        if item.id == HLIST then
          node.set_attribute(item, 100, attr)
          item = nil
        else
          item = item.next
        end
      end
      head = node.remove(head, mark)
    end
  end
  return head
end
```

The following function scans the content of a vertical list, probably box 255, finds the lines that have attribute 100 set to some value, and adds the margin notes to those lines. Remember that our goal is to avoid margin notes running into the space below the textblock (either the bottom margin or the vacant space at the end of a chapter). So we must compute how much space remains to accommodate the note. To do so, we scan the box (the page), starting at the bottom, and accumulate the height and depth of lines and the width of kerns and glues — except kerns and glues that might appear before the last line, i.e. space filling the page. To do so, we have a `first` boolean that is `true` as long as a line hasn't been found and prevents adding the width of glues and kerns. With `node.slide` we grasp the last node of the list, since we're reading it backward.

```
process_marginalia = function (head)
  local remainingheight, first, item =
    0, true, node.slide(head)
  while item do
    if node.has_field(item, "kern") then
      if not first then
        remainingheight = remainingheight
          + item.kern
      end
    elseif node.has_field(item, "spec") then
      if not first then
        remainingheight = remainingheight
          + item.spec.width
      end
```

Now, if we find a line, we add its depth if and only if it's not the first one we encounter (i.e. the last one on the page), because in that case its depth belongs to the bottom margin. Its height is added later, if and only if the line doesn't take a note.

```
    elseif node.has_field(item, "height") then
      if first then
        first = false
      else
        remainingheight = remainingheight
          + item.depth
      end
```

If attribute 100 is set to some value, then the line takes a note. In that case, we retrieve the box, measure its depth, and compare it to the remaining height. Note that the depth of the box is all its material barring the height of its first line (since we used a `\vtop`), which is exactly what we want: its first line can't go wrong, since it's level with the main text's line from whence it came. We also remove the depth of the last line, since its going into the bottom margin is perfectly ok.

```
     local attr = node.has_attribute(item, 100)
     if attr then
        local note = node.copy(tex.box[attr])
        local upward = note.depth
          - node.tail(note.list).depth
        if upward > remainingheight then
          upward = remainingheight - upward
        else
          upward = 0
        end
```

Now we insert the note box after the line: first, we add a negative vertical kern to account for the upward shift (possibly 0), plus the line's depth and the note's height (i.e. the height of its first line), so it is level with the line. We then set the note's height and depth to 0, so it doesn't take up space on the page. (Since the kern becomes the head of the list, we have to explicitly set `note.list` to it, otherwise TEX still thinks the previous head is the good one.)

```
        local kern = node.new(KERN, 1)
        kern.kern = upward - note.height
          - item.depth
        node.insert_before(note.list,
                           note.list, kern)
        note.list = kern
        note.height, note.depth = 0, 0
```

Finally, we insert the note and set its horizontal `shift` (here it goes into the right margin, but this should depend on whether the page is even or odd), and reset `first` and `remainingheight`, the latter to `upward` so the vertical shift of the current note (if any) is taken into account for the following one. The rest of the code is the end of the `attr` conditional (false, so we add the line's height to the `remainingheight`) and the end of the main loop.

```
        node.insert_after(head, item, note)
        note.shift = tex.hsize + tex.sp("1em")
        first = true
        remainingheight = upward
      else
        remainingheight = remainingheight
          + item.height
      end
    end
    item = item.prev
  end
end
```

When a page is found good, before we ship it out (and before we add inserts too), we feed it to the function, so notes are added. For instance, a very simple output routine would be:

```
\output{%
  \directlua{%
    process_marginalia(tex.box[255].list)
    }%
  \shipout\box255}
```

The important part is, of course, the Lua code.

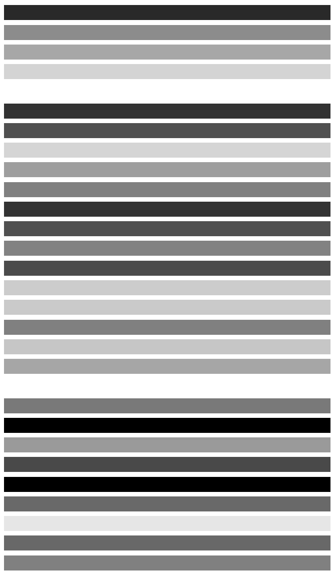
Conclusion

LuaTEX has much to offer: UTF-8 encoding, non-TFM fonts, a comfortable programming language, ... Access to TEX's internals is, to me, one of its most valuable features: it enables the user to do things that were previously unthinkable, and gives such control over typography that the software's limitations almost vanish, as if we were working on a hand press — except we don't manipulate metal, but nodes.

A final note: in this paper, functions have been added to callbacks with LuaTEX's bare mechanism. If two functions are added to the same callback this way, the second erases the first. To do this properly, the `luatexbase` package can be used for plain TEX and LATEX, and it is taken care of in ConTEXt.

The next page shows examples of our three programs. First comes the page color, displaying a typeset text and its translation to shades of grey; the second text uses font expansion to show the resulting improvement in justification. Then are examples of underlining and marginal notes. The text used is the first page of Robert Coover's novel *The Adventures of Lucky Pierre.*

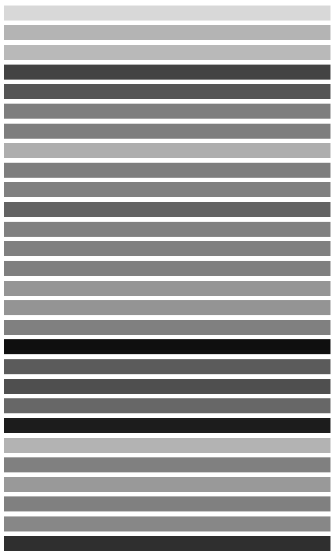
◇ Paul Isambert
Université de la Sorbonne Nouvelle
France
`zappathustra (at) free dot fr`



In the darkness, softly. A whisper becoming a tone, the echo of a tone. Doleful, incipient lament blowing in the night like a wind, like the echo of a wind, a plainsong wafting silently through the windy chambers of the night, wafting unisonously through the spaced chambers of the bitter night, alas, the solitary city, she that was full of people, thus a distant and hollow epiodion laced with sibilants bewailing the solitary city.

And now, the flickering of a light, a pallor emerging from the darkness as though lit by a candle, a candle guttering in the cold wind, a forgotten candle, hid and found again, casting its doubtful luster on this faint white plane, now visible, now lost again in the tenebrous absences behind the eye.

And still the hushing plaint, undeterred by light, plying its fricatives like a persistent woeful wind, the echo of woe, affanato, piangevole, a piangevole wind rising in the fluttering night through its perfect primes, lamenting the beautiful princess become an unclean widow, an emergence from C, a titular C, tentative and parenthetical, the widow then, weeping sore in the night, the candle searching the pale expanse for form, for the suggestion of form, a balm for the anxious eye, weeping she weepeth.



In the darkness, softly. A whisper becoming a tone, the echo of a tone. Doleful, incipient lament blowing in the night like a wind, like the echo of a wind, a plainsong wafting silently through the windy chambers of the night, wafting unisonously through the spaced chambers of the bitter night, alas, the solitary city, she that was full of people, thus a distant and hollow epiodion laced with sibilants bewailing the solitary city.

And now, the flickering of a light, a pallor emerging from the darkness as though lit by a candle, a candle guttering in the cold wind, a forgotten candle, hid and found again, casting its doubtful luster on this faint white plane, now visible, now lost again in the tenebrous absences behind the eye.

And still the hushing plaint, undeterred by light, plying its fricatives like a persistent woeful wind, the echo of woe, affanato, piangevole, a piangevole wind rising in the fluttering night through its perfect primes, lamenting the beautiful princess become an unclean widow, an emergence from C, a titular C, tentative and parenthetical, the widow then, weeping sore in the night, the candle searching the pale expanse for form, for the suggestion of form, a balm for the anxious eye, weeping she weepeth.

And now, the flickering of a light, a pallor emerging from the darkness as though lit by a candle, a candle guttering in the cold wind, a forgotten candle, hid and found again, <u>casting its doubtful luster on this faint white plane</u>, now visible, now lost again in the tenebrous absences behind the eye.

And still the hushing plaint, undeterred by light, plying its fricatives like a persistent woeful wind, the echo of woe, **affanato**, **piangevole**, a piangevole wind rising in the fluttering night through its perfect primes, lamenting the beautiful princess become an unclean widow, an emergence from C, a titular C, tentative and parenthetical, the widow then, weeping sore in the night, the candle searching the pale expanse for form, for the suggestion of form, a balm for the anxious eye, weeping she **weepeth**.

‘Affanato’ means ‘anguished’

‘Piangevole’ means ‘plaintive’

‘Weepeth’ is an archaic form of ‘weeps’