Litfass

v0.1.1

Contents

1 Tiling

The core of Litfass is a tiling layout mechanism. Every tiling basically is a tree structure of connected tiling actions. Roughly speaking a tiling action describes how to process a bounding box. The inner actions take a bounding box as an input, split or manipulate it and pass the new bounding box(es) further down to other actions until they reach leaf actions, which describe how to fill the bounding box they received from above. Leafs thus produce content and act as sinks for the flow of bounding boxes. Lets first get an intuition for how this tiling mechanism by looking at informal descriptions of the nodes and leaves that can be used to construct a tiling. Technically the functions shown below are of the form `f(params) = (ctx) => {/*some code working on a bounding box*/}` here `(ctx) => {/*...*/}` is the actual action, so `f()` actually generates an action based on `params`. Generators like this can be used to link actions. Let's take a generator of the form

```
f(params, left-action, right-action) = (ctx) => {
  let (ctx-left, ctx-right) = split(ctx, params)
  left-action(ctx-left)
  right-action(ctx-right)
}
```

with `left-action()` and `right-action` being of the type `(ctx) => {}` then `f()` is a branching point in the flow of bounding boxes in the tiling.

Available inner action generators are:

- `vs(cut: length, left-action, right-action)` – This generator produces a *vertically split* (vs) at the `cut` (default: `50%`) position and branches the flow to a `left-action` and `right-action`.
- `hs(cut: length, top-action, bottom-action)` – This generator produces a *horizontal split* (vs) at the `cut` (default: `50%`) position and branches the flow to a `top-action` and `bottom-action`.
- `pad(padding, action)` – This generator does not branch, wraps an existing `action` by capturing the bounding box and applying a `padding` to it before passing it on to the original `action`.

To stop the flow of bounding boxes, we need leaf actions which absorb the bounding box and produce `content`. Leaf actions are of the type `(ctx) => []`

Available leaf action generators are:

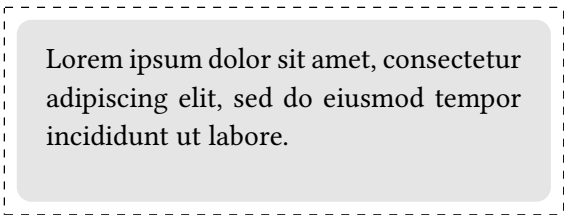
- `cbx(cnt: content, title: content)` – This leaf generator produces an action binding a content body and a title to a given bounding box.
- `blank()` – This leaf generator produces a blank action which just consumes the bounding box without displaying anyting

1.1 Examples

Let us allocate some screen realestate `bbox = (width: 19em, height: 7em)` to apply our tiling to:

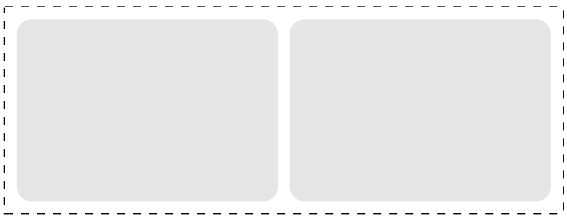
Now we start by defining a trivial tiling which just assinges one singular content box to the bounding box:

```
#let tiling = cbx([#lorem(10)])

#inline-box(tiling, ..bbox)
```



Now let's split in in half:

```
#let tiling = vs(
  cbx([]),
  cbx([])
)
#inline-box(tiling, ..bbox)
```

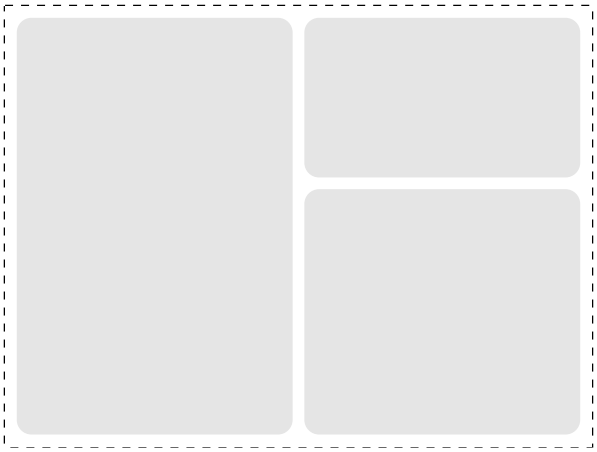


```
#let tiling = hs(
  cbx([]),
  cbx([])
)
#inline-box(tiling, ..bbox)
```

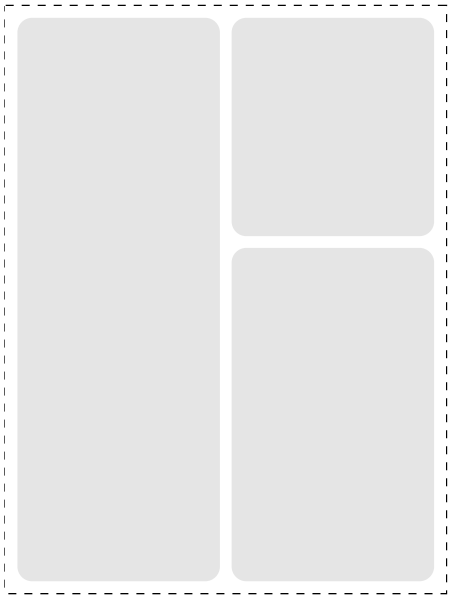
Note how we plugged two `cbx([])` into the split generators and remember that `cbx()` is actiualy also just an action generator. Instead of a leaf generator like `cbx()` we can also pass an inner generator to the splitting functions, effectively nesting the tiling.

```
#let tiling = vs(
  cbx([]),
  hs(cut: 40%, cbx([]), cbx([]))
)
```

```
#inline-box(tiling, width: 20em,
            height: 15em)
```



```
#inline-box(tiling, width: 20em,
            height: 15em)
```



2 Render

In the section above we showed how to construct a tiling, which is a function `(ctx) => [...]` mapping a context object (not to be confused with `context` in typst) to content. In our case `ctx` is a struct

holding dimensions (a rect of paper real estate), a theme defining to display content and other technical internals which are not important right now. The context gets manipulated and passed on to lower levels while it flows through the tiling actions.

To render a tiling, the `apply(tiling, ctx)` function needs to be called. A context can be created with the utility function `create-context(dims, theme: default-theme)`, where `dims` is:

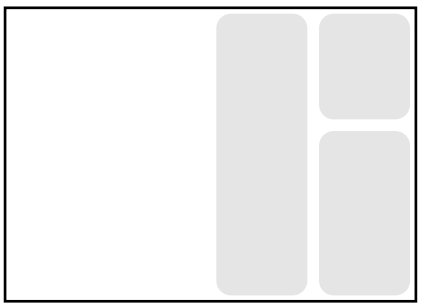
```
dims = (
  pos: (x: length, y: length),
  wh: (width: length, height: length)
)
```

This is the universal dimensions pattern throughout Litfass. So, on to a concrete example:

```
#let tiling = vs(
  cbx([]),
  hs(cut: 40%, cbx([]), cbx([]))
)

#let ctx = create-context(
  (
    pos: (x: 0pt, y: 0pt),
    wh: (width: 100%, height: 100%)
  )
)
#box(
  width: 14em,
  height: 10em,
  {
    apply(tiling, ctx)
  }
)
```


Here we created a typst `box()` and completely filled it with the tiling defined above. To achieve this, we created a context, to place the tiling at the top left corner of the box (`pos: (x: 0pt, y: 0pt)`) and span the whole area of the box (`wh: (width: 100%, height: 100%)`). We could for instance also only use the right half of the box by creating a context like:

```
#let ctx = create-context(
  (
    pos: (x: 50%, y: 0pt),
    wh: (width: 50%, height: 100%)
  )
)
#box(
  width: 14em,
  height: 10em,
  stroke: black,
  {
    apply(tiling, ctx)
  }
)
```



2.1 Convenience Functions

Usually though, you probably just want to fill a box of a specific width and hight or even a complete document. The first one you have already seen in the examples of the Tiling section.

2.1.1 inline-box

With `inline-box(tiling, width, height, theme: default-theme)` you, as the name suggests, draw an inline box right at the place of the function cal by applying a tiling to it.

```
#let tiling = vs(
  cbx([]),
  hs(cut: 40%, cbx([]), cbx([]))
)

#inline-box(tiling, width: 14em, height: 10em)
```


2.1.2 poster

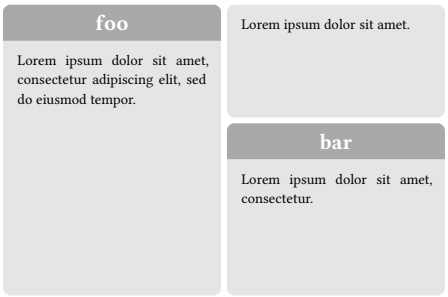
With `poster(tiling, theme: default-theme)` the tiling gets applied to the whole page dimensions. An example usage could look like:

```
#set page(paper: "a2")
#set text(18pt)

#let tiling = /* define poster content and layout */

poster(tiling)
```

3 Themes

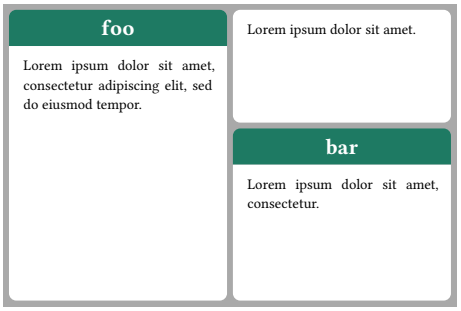
Litfass supports the use of themes to style the content laid out by a tiling. Predefined themes can be imported from `import litfass.themes: *`. As for now, one basic default theme is provided under `litfass.themes.basic`. For this manual we used a modified version of the `basic` theme. By default the `basic` theme calculates the background color of the box panel based on the title background color. Thus to define a colored box, you have to set the `box.title.background` entry of the theme.

```
#let theme = {
  let theme = litfass.themes.basic
  theme.padding = 0.2em
  theme.box.padding = 0.2em
  theme.box.title.background = gray

  theme
}
#let inline-box = inline-box.with(
  theme: theme
)

#let tiling = vs(
  cbx(lorem(12), title: [Foo]),
  hs(
    cut: 40%,
    cbx(lorem(5)), cbx(lorem(6), title: [Bar])
  )
)

#inline-box(tiling, width: 6cm, height: 4cm)
```



For the tiling section this theme was OK, to visualize the box ares. For text though, the gray boxes seem a little dark. So, let's tweak it a little, by making the overall background gray and the background of the boxes white. Further, we decide to go for a green title background.

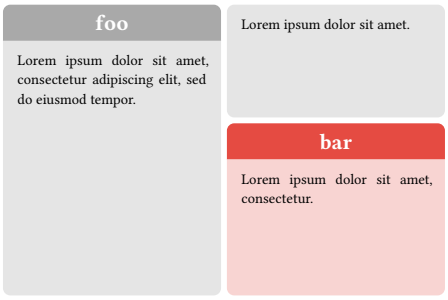
```
#let new-theme = {
  let thm = theme
  thm.box.title.background = rgb("1e7a63")
  thm.background = gray
  thm.box.background = white
  thm
}

#inline-box(tiling, width: 6cm, height: 4cm,
theme: new-theme)
```



3.1 Local Overriding

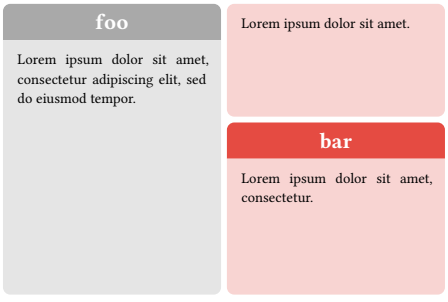
Sometimes, we only want to change the appearance of a specific element. Take for example a key box you want to draw special attention to. In Litfass you can locally override themes. To do so, you have to define the changes. Then, when passed to a local tiling action generator through the `theme` argument, the theme of the current context will be updated with those changes.

```
#let highlight-box-style = (
  title: (
    background:
      red.desaturate(10%).darken(10%)
  )
)
#let tiling = vs(
  cbx(lorem(12), title: [foo]),
  hs(
    cut: 40%,
    cbx(
      lorem(5)),
      cbx(
        lorem(6),
        title: [bar],
        theme: (box: highlight-box-style)
      ),
  )
)
#inline-box(tiling, width: 6cm, height: 4cm)
```



By the very nature of how the context gets passed further down, local theme overrides will cascade further down the tiling hierarchy.

```
#let tiling = vs(
  cbx(lorem(12), title: [foo]),
  hs(
    cut: 40%,
    cbx(
      lorem(5)),
      cbx(lorem(6), title: [bar]
    ),
    theme: (box: highlight-box-style)
  )
)
```



```
#inline-box(tiling, width: 6cm, height: 4cm)
```

3.2 Internals

In many cases one wants to define their one theme. For this, let's have a look into how a theme is defined. The `basic` theme is defined as:

```
#let theme = (
  padding: 1em,
  box: (
    padding: 1em,
    background: 90%,
    inset: 1em,
    radius: 0.5em,
    stroke: 0em,
    footnote: (
      text: (
        size: 0.8em,
      ),
    ),
    title: (
      text: (
        size: 1.5em,
        color: white,
      ),
      background: rgb("#1e417a"),
      inset: 0.75em,
    )
  ),
  background: white,
  p-block: _p-block
)
```

Here `p-block` is a function taking a `ctx` object, a body and a title content and returning a `context [...]`, `p-block` gets called internally by the content box action generated by `cbx()`.

Contract to fulfill when creating custom themes: A p-block method must be provided to draw context boxes, it has to follow the following template.

```
#let _p-block(ctx, cnt, title: none) = context {
  /*
  * Render the content provided by cnt (body) and title
  * constraint to the width and height provided by ctx.dims.wh
  */
}
```

4 Util

4.1 Footnotes

Litfass provides a custom footnotes function, which is scoped at the context box level. By calling `box-footnote(cnt)` the location of its call can be annotated with a footnote at the bottom of the box, just like regular `footnote(cnt)` but at a box level.

```
#let tiling = hs(
  cbx([
    Lorem ipsum dolor sit amet,
    consectetur#box-footnote[You can]
    adipiscing elit, sed do eiusmod
    tempor incididunt#box-footnote[
      place random footnotes
    ] ut labore et dolore magnam
    aliquam#box-footnote[
      throughout a box!
    ] quaerat voluptatem.
  ]),
  cbx([
    Lorem ipsum dolor sit amet,
    consectetur#box-footnote[
      Footnotes are _box-local_.
    ] adipiscing elit, sed do eiusmod
    tempor incididunt#box-footnote[
      Footnote counters get reset
      at each box.
    ] ut labore et dolore magnam
    aliquam quaerat voluptatem.
  ]),
)
```

Lorem ipsum dolor sit amet, consectetur[1] adipiscing elit, sed do eiusmod tempor incididunt[2] ut labore et dolore magnam aliquam[3] quaerat voluptatem.

[1] You can
[2] place random footnotes
[3] throughout a box!

Lorem ipsum dolor sit amet, consectetur[1] adipiscing elit, sed do eiusmod tempor incididunt[2] ut labore et dolore magnam aliquam quaerat voluptatem.

[1] Footnotes are *box-local*.
[2] Footnote counters get reset at each box.