About the source code of TeXmacs

Table of contents

Chapter 1
General architecture of TeXmacs

1.1. Introduction

The TeXmacs program has been written in C++. You need `g++` and the `makefile` utility in order to compile TeXmacs. Currently, the source (in the `src` directory) of the TeXmacs implementation has been divided into the following parts:

- A set of basic and generic data structures in the `Basic` directory.
- Standard resources for TeXmacs, such as TeX fonts, languages, encodings and dictionaries, in the `Resource` directory.
- A documented graphical toolkit in the `Window` directory (although the documentation is a bit outdated).
- The extension language for TeXmacs in the `Prg` directory.
- The typesetting part of the editor in the directory `src/Typeset`.
- The editor in the directory `src/Edit`.
- The TeXmacs server in the directory `src/Server`.

All parts use the data structures from `Basic`. The graphical toolkit depends on `Resource` for the TeX fonts. The extension language is independent from `Resource` and `Window`. The typesetting part depends on all other parts except from `Prg`. The main editor and the TeXmacs server use all previous parts.

The TeXmacs data are contained in the directory `edit` which corresponds to the TeXmacs distribution without the source code. Roughly speaking, we have the following kind of data:

- Font data in `fonts` (encodings, `.pk` files, etc.).
- Language data in `languages` (hyphenation patterns, dictionaries, etc.).
- Document styles in `style`.
- Initialization and other Scheme programs in `progs`.

The directory `misc` contains some miscellaneous data like the edit icon (`misc/pixmaps/traditional/--x17/edit.xpm`).

1.2. Intern representation of texts

TeXmacs represents all texts by trees (for a fixed text, the corresponding tree is called the *edit tree*). The nodes of such a tree are labeled by standard *operators* which are listed in `Basic/Data/tree.hpp` and `Basic/Data/tree.cpp`. The labels of the leaves of the tree are strings, which are either invisible (such as lengths or macro definitions), or visible (the real text).

The meaning of the text and the way it is typeset essentially depend on the current environment. The environment mainly consists of a relative hash table of type `rel_hashmap<string,tree>`, i.e. a mapping from the environment variables to their tree values. The current language and the current font are examples of system environment variables; new variables can be defined by the user.

1.2.1. Text

All text strings in TEXMACS consist of sequences of either specific or universal symbols. A specific symbol is a character, different from `'\0'`, `'<'` and `'>'`. Its meaning may depend on the particular font which is being used. A universal symbol is a string starting with `'<'`, followed by an arbitrary sequence of characters different from `'\0'`, `'<'` and `'>'`, and ending with `'>'`. The meaning of universal characters does not depend on the particular font which is used, but different fonts may render them in a different way.

1.2.2. The language

The language of the text is capable performing a further semantic analysis of a text phrase. At least, it is capable of splitting a phrase up into *words* (which are smaller phrases) and inform the typesetter about the desired spaces between words and hyphenation information. In the future, additional semantics may be added into languages. For instance, spell checkers might be implemented for natural languages and parsers for mathematical formulas or programming languages.

1.3. Typesetting texts

Roughly speaking, the typesetter of TEXMACS takes a tree on input and produces a box, while accessing and modifying the typesetting environment. The `box` class is multifunctional. Its principal method is used for displaying the box on a post-script device (either the screen or a printer). But it also contains a lot of typesetting information, such as logical and ink bounding boxes, the positions of scripts, etc.

Another functionality of boxes is to convert between physical cursors (positions on the screen) and logical cursors (paths in the edit tree). Actually, boxes are also organized into a tree, which often simplifies the conversion. However, because of macro expansions and line and page breaking, the conversion routines may become quite intricate. Notice also that, besides a horizontal and vertical position, the physical cursor also contains an infinitesimal horizontal position. Roughly speaking, this infinitesimal coordinate is used to give certain boxes (such as color changes) an extra infinitesimal width.

1.4. Making modifications in texts

In `Edit/Modify` you find different routines for modifying the edit tree. Modifications go in several steps:

1. A certain input event triggers an action, such as `make_fraction`, which intends to modify the edit tree.

2. All modifications which `make_fraction` or its subroutines will make to the edit tree eventually break down to seven elementary modification routines, namely `assign`, `insert`, `remove`, `split`, `join`, `ins_unary` and `rem_unary`.

3. Before performing the required modification, the elementary modification routine first notifies all views of the same text of the modification.

4. On notification, each view updates several things, such as the cursor position. It also notifies the modification to the typesetter of the text, since the typesetter maintains a list of already typeset paragraphs.

5. When all views have been notified of the modification, we really perform it.

6. Each user action like a keystroke or a mouse click is responsible for inserting *undo points* between sequences of elementary modifications. When undoing a modification, the editor will move to the previous undo point.

For trees `t`, we notice that `t->label` yields the label of the tree and `t->a` the array of its children. The second argument of `<<` for trees is either a tree or an array of trees.

The implementation has been made such that the `<<` operation is fast, which is useful when considering arrays as buffers. Actually, the allocated space for arrays with more than five elements (words for strings) is always a power of two, so that new elements can be appended quickly. Notice that GNU malloc also always allocates blocks, whose sizes are powers of two. Therefore, we do not waste memory for small and large arrays.

2.3. Lists

Generic lists are implemented by the class `Tist< >`. The "nil" list is created using `list<T>()`, an atom using `list<T>(T x)` and a general list using `list<T>(T x, list<T> next)`. If `l` is a list, `l->item` and `l->next` correspond to its label and its successor respectively (`car` and `cdr` in lisp). The functions `nil` and `atom` tests whether a list is nil or an atom. The function `N` computes the length of a list.

The type `list<T>` is also denoted by `path`, because some additional functions are defined for it. Indeed, paths are used for accessing descendents in tree like structures. For instance, we implemented the function `tree subtree (tree t, path p)`.

2.4. Hash tables

The `hashmap<T,U>` class implements hash tables with entries in `T` and values in `U`. A function `hash` should be implemented for `T`. Given a hash table `H`. We set elements through

```
H(x)=y;
```

and access to elements through

```
H[x]
```

We also implemented a variant `rel_hashmap<T,U>` of hash tables, which also have a list-like structure, which makes them useful for implementing recursive environments.

2.5. Other data structures

- `command` implements abstract commands.
- `file` implements files.
- `iterator<T>` implements generic iterators.
- `rectangles` implements rectangles and lists of rectangles.
- `space` implements stretchable spaces.
- `timer` implements timers.

Chapter 3

Converters to other data formats

Currently, we have implemented imperfect converters between TeXmacs and LaTeX and from html to TeXmacs. We hope that someone else will be willing to write better converters from scratch. This chapter has been included in order to give some recommendations in that direction based on our experience from the implementation of the actual conversion programs. We also recommend to take a look at the current implementations in the directory `Convert`.

3.1. Parsing extern data formats

In order to write a converter from LaTeX html, xml, etc. to TeXmacs, a good first step is to write a parser for the extern data format. For html, xml, etc. this should be rather easy, but for LaTeX, you will probably need to be a real LaTeX guru (which I am not). We recommend the result of the parsing step to be a Scheme expression (something which is regrettable not the case for our actual converters), because this language is very well adapted for the implementation of the actual converter.

This first step should be able to process any correct file having the extern data format; possible incompatibilities should only come into play during the actual conversion. In the case of LaTeX, one should not expand the macros and keep all macro definitions, because TeXmacs will be able to take advantage out of this.

3.2. The actual converter

We recommend the actual converter to proceed in several steps. Often it is convenient to start with a rough, structural, conversion step, which is "polished" by a certain number of additional steps. These additional steps may take care of some very particular layout issues which can not be treated conveniently at the main step.

Actually, the main difficulties usually come from exceptional text, like verbatim, and layout issues which are handled differently in the extern data format and TeXmacs. A good example of such a difference between LaTeX and TeXmacs is the way equations or lists are handled. Consider for instance the following paragraph:

Text before.

$$a^2 + b^2 = c^2.$$

Text after.

In LaTeX, the equation is really seen as a part of the paragraph. Indeed, there will not be any blank line between "Text before" and the equation. However, for efficiency reasons, it is better to see the paragraph as three paragraphs in TeXmacs, because the lines can be typesetted independently. Nevertheless, the equation environment will disable the indentation of "Text after".

As a result of this anomaly, converted texts have to be postprocessed, so as to insert paragraph breaks at strategic places. It should be noticed that this step may be independent from the format which is actually being converted and that a similar reverse step may be implemented for backward conversions. We also notice that one needs an exhaustive list of all similar exceptional environments for this postprocessing step. Actually, a future version of TEXMACS might come with an additional feature, which permits the automatic detection of such environments. This is also important from a semantical point of view, because one should be able to detect that the above example logically forms only one and not three paragraphs.

3.3. Backward conversions

Conversions from TEXMACS to an extern data format are usually easier to implement, because the TEXMACS data format is semantically rich. However, conversions to an extern data format without a TEX-like macro facility give rise to the problem of macro expansion of non supported TEXMACS functions or environments. We plan to write a facility for this, which you will be able to use when writing a converter from TEXMACS to something else.

Chapter 4

The graphical user interface

4.1. Introduction

4.1.1. Main architecture

The graphical toolkit used by Edit has two main components: an abstract window interface, which is very similar to X Window, and the actual toolkit. At this moment an abstract window interface for X Window has been implemented, but others might be added in the feature.

The abstract window interface consists of two main classes: `display` and `window`. The `display` class is responsible for

- The connection with the (X-)server.
- Managing server resources such as colors and fonts.
- Inter-window communication (i.e. selections and so).
- Redirection of input/output (pointer and keyboard grabs).

The `window` class is responsible for

- The part of the layout of a window, which is negotiated with the window manager.
- Providing a basic set of postscript-compatible graphical routines.
- Implementation of some clipping and region translation routines.
- Delegation of events to the widget associated to the window.

In particular, the `window`-class inherits from the `ps_device`-class, which is responsible for providing the basic set of postscript-compatible graphical routines. Hence, user applications will draw their graphics on a `ps_device`, since this will both allow them to visualize them in a window or on a printer.

The graphical toolkit built on top of the abstract window interface is widget oriented. A large number of widget classes are implemented, which all inherit from the abstract `widget` class. Widgets may have a finite number of children and they are responsible for

- Their, and their children, sizes and positioning in the window.
- Reaction on events, which are either redraw requests, keyboard or pointer events or other miscellaneous events.
- Other functionalities, which depend on the particular widget class.

Later a special widget-style manager will be implemented, which will be able to present widgets according to different “styles”.

4.1.1.1. A simple example

In order to create a window "Test" with the text "Hello world" in it, one first opens the display, then create the widget with the text and finally the window, with the widget attached to it

```
display dis= open_display ();
widget  wid= text_widget ("Hello world");
window  win= plain_window (wid, dis, "Test");
```

Technically speaking, the window creation amounts to several actions:

- The widget is questioned for an appropriate size for it (actually, only minimal, default and maximal size hints are computed).
- The window is created.
- The widget is "attached" to the window.
- The subwidgets are "positioned" at their appropriate places and they are given appropriate sizes within their parents.

The next step is to make the window visible by

```
win->map ();
```

At this points repaint request events are generated, which are handled when starting the event loop by

```
dis->event_loop ();
```

Eventually, the window `win` will be destroyed using

```
delete win;
```

At this point, control is handled back by the event loop and we close and destroy the display by

```
close_display (dis)
delete dis;
```

The user only has to bother about destroying window and displays; widgets are automatically destroyed if they are no longer being referenced to.

4.1.2. Widgets and event processing

From the implementation point of view, widgets are pointers to instances of the abstract widget representation class `widget_rep`. Moreover, the widget class supports reference counting. The `widget_rep` class contains information about the window it is attached to and its location within this window, its size, the position of the origin of its local coordinates (`north_west`, `north`, etc.) and its children. The `widget_rep` class also provides a virtual event handler `widget_rep::handle`. This handler returns true if the event could be handled.

The implemented widget representation classes are organized in a hierarchy, which contains both concrete and abstract classes. The abstract classes reimplement the virtual event handler `widget_rep::handle` in such a way that if the event is recognized, then the event is dispatched to a more particular virtual event handler, possibly after some processing.

For example, instances of the `basic_widget_rep` class can handle the most common events, such as keyboard, mouse, repaint events and so on. If a key is pressed, then the virtual function `basic_widget_rep::handle_keypress` is called with an argument of type `keypress_event`. The default implementation of this virtual function does nothing.

The user can also create his own events, which he can pass to any widget. For instance in order to invalidate a region for redrawing, one creates an invalidate event using `emit_invalidate` and sends it to the appropriate widget using the operator `<<`. Notice that the user is responsible for sending events to widgets which can handle them. Bad matches are only discovered at run time, in which case an error is generated by `<<`.

4.1.2.1. A simple example

Suppose that we want to make a widget, which tracks keyboard events and which displays them. Such a widget must have a construction routine and keyboard and repaint handlers:

```
class track_widget_rep: basic_widget_rep {
  string last_key;
  track_widget_rep ();
  void handle_keypress (keypress_event ev);
  void handle_repaint (repaint_event ev);
};
```

The constructor is taken to be empty and places the origin at the center of the widget

```
track_widget_rep::track_widget_rep (): basic_widget (center) {}
```

In particular `last_key` is initialized by the empty string. We also define the function

```
void
track_widget () {
  return new track_widget_rep ();
}
```

in order to create an instance of a `track_widget`.

The event handler for keyboard events should just reset the string `last_key` and invalidate the entire widget region.

```
void
track_widget_rep::handle_keypress (keypress_event ev) {
  last_key= ev->key;
  this << emit_invalidate_all ();
}
```

The event handler for repainting first determines the string to be repainted as a function of `last_key`, computes its extents and repaints it at the center.

```
void
track_widget_rep::handle_repaint (repaint_event ev) {
  string s= (last_key == ""? "No key pressed": "Pressed " * last_key);

  SI x1, y1, x2, y2;
  win->get_extents (s, x1, y1, x2, y2); // CHECK THIS

  win->set_color (black);
  win->fill (ev->x1, ev->y1, ev->x2, ev->y2);
  win->set_color (white);
  win->draw_string (s, -(x1+x2)>>1, -(y1+y2)>>1);
}
```

4.2. The abstract window interface

4.2.1. Displays

4.3. Widget principles

4.3.1. The widget class

Widgets are pointers to instances of the abstract widget representation class `widget_rep`. Widgets support reference counting, so that a widget is automatically destroyed if it is not used any more (except in the case of circular referencing; see below). As a general rule, the user does not have to worry about the creation and destruction of widgets.

4.3.1.1. The widget representation class

The definition of the `widget_rep` class goes as follows:

```
    struct widget_rep: rep_struct {
      window        win;           // underlying window
      SI            ox, oy;        // origin of widget in window
      SI            w, h;          // width and height of widget
      gravity       grav;          // position of the origin in the widget
      array<widget> a;             // children of widget
      array<string> name;          // names for the children

      widget_rep (array<widget> a, array<string> name, gravity grav);
      virtual ~widget_rep ();

      virtual operator tree () = 0;
      virtual bool handle (event ev) = 0;

      SI       x1 (); SI y1 (); // lower left window coordinates of widget
      SI       x2 (); SI y2 (); // upper right window coordinates of widget
      bool     attached ();     // tests whether (win != NULL)
      volatile void fatal_error (string mess, string rout="", string
fname="");

      friend class widget;
    };
```

The `win` field specifies the window to which the widget is attached (`win=NULL`, by default). The origin `(ox,oy)` of the widget is specified with respect to the windows origin. Next come the width `w` and the height `h` of the widget. The gravity `grav` determines where the origin of the widget is located (`north_west`, `north`, etc.). The array `a` specifies the children of the widget. The array `name` gives names to the children of the widget. This is useful for addressing children by comprehensible names; the names are also useful for designing menu widgets.

The virtual type casting operator for trees is used for debugging purposes; mainly in order to print widgets. The virtual member function `handle` processes an event which is send to the widget and returns `TRUE` if the event could be handled and `FALSE` if not.

4.3.1.2. The widget class

The definition of the `widget` class goes as follows:

```
struct widget {
#import null_indirect_h (widget, widget_rep)
  inline widget (widget_rep* rep2): rep (rep2) {
    if (rep!=NULL) rep->ref_count++; }
  inline widget operator [] (int i) { return rep->a[i]; }
         widget operator [] (string s);
  inline operator tree () { return (tree) (*rep); }
  inline bool operator == (widget w) { return rep == w.rep; }
  inline bool operator != (widget w) { return rep != w.rep; }
};
```

Widgets may be constructed in two ways. First, we may construct a symbolic "nil" widget by `widget ()`. The function `bool nil (widget)` is provided in order to test whether a widget is "nil". Secondly, we may construct a widget from a pointer of type `widget_rep*`.

The reference counting mechanism ensures widgets to be destroyed when they are no longer pointed to. An important exception is when two widgets point one to each other, which fools the reference counter (for instance a scrollbar and the widget which is scrolled need to point one to each other). In order to deal with such "circular dependencies", one works directly with `widget_rep*` pointers if one does not want to the pointer to be taken into account in the reference counter.

Child widgets can again be accessed to in two ways. First, we have the direct way, using its index in the array `a`. Secondly, we can access to a child via its name. Actually, when using this method, a `get_widget` event is generated. In the basic widget class, the default action for this event is to search in the name array for the child. However, the user may override this default action and provide another child searching method.

4.3.2. The event class

Events are pointers to instances of the abstract `event_rep` class, which supports reference counting. Actually, concrete event representation classes just contain some information. Hence, events actually provide a safe and generic way to store and communicate information.

4.3.2.1. The event representation class

The definition of the `event_rep` structure is as follows:

```
struct event_rep: public rep_struct {
  int    type;  // the event type
  inline event_rep (int type2): rep_struct (0), type (type2) {}
  inline virtual ~event_rep () {}
  virtual operator tree () = 0;   // for displaying events (debugging)
};
```

The `type` field gives the type of the event. A complete list of the event types if given in the file

```
Window/Event/event_codes.hpp
```

For each project which uses new event types, an analogue file should be made and the numbers of the event types should all be different. Unfortunately, there is no safe way in order to let this job be done by the compiler.

4.3.2.2. The event class

The `event` structure is defined by

```
struct event {
#import indirect_h (event, event_rep)
  inline event (event_rep* rep2): rep (rep2) {
    if (rep!=NULL) rep->ref_count++; }
  inline operator tree () { return (tree) (*rep); }
};
#import indirect_cc (event, event_rep)
```

4.3.2.3. Concrete event classes

Concrete event classes again come into two parts: the class itself and its representation class. For instance, the representation class for `get_widget` events is defined by

```
struct get_widget_event_rep: public event_rep {
  string which; widget& w;
  get_widget_event_rep (string which, widget& w);
  operator tree ();
};
```

The corresponding `get_widget_event` class is defined by

```
#import event (get_widget_event, get_widget_event_rep)
```

The module `event` with two parameters is defined by

```
#module event (T, R)
struct T {
  R* rep;
  T (T& ev);
  ~T ();
  T (event& ev);
  operator event ();
  R* operator -> ();
  T& operator = (T ev);
};
#endmodule // event (T, R)
```

The important thing to notice is that we have converters from and to the generic `event` class. Moreover, the generic `event` class and the specific `get_widget_event` class are compatible from the reference counting point of view.

The implementation of the `get_widget_event_rep` class is as follows:

```
get_widget_event_rep::get_widget_event_rep (string ww, widget& w2):
  event_rep (GET_WIDGET_EVENT), which (ww), w (w2) {}
get_widget_event_rep::operator tree () {
  return tree ("get_widget_event", which); }
#import code_event (get_widget_event, get_widget_event_rep)
```

The actual events are created by

```
event get_widget (string which, widget& w) {
  return new get_widget_event_rep (which, w); }
```

4.3.2.4. Event handlers

Implementations of the generic event handler `bool widget_rep::handle(event)` usually do the following

- Determine the event type.
- Perform some action, depending on the event type and the widget.
- Dispatch the event to a concrete or abstract specific event handler or to the generic event handler of some other widget representation class.

For instance, the event handler for composite widgets is as follows:

```
bool
composite_widget_rep::handle (event ev) {
  switch (ev->type) {
  case CLEAN_EVENT:
    handle_clean (ev);
    return TRUE;
  case INSERT_EVENT:
    handle_insert (ev);
    return TRUE;
  case REMOVE_EVENT:
    handle_remove (ev);
    return TRUE;
  }
  return basic_widget_rep::handle (ev);
}
```

The member function `handle_insert` is implemented as follows:

```
void
composite_widget_rep::handle_insert (insert_event ev) {
  a << ev->w;
  name << ev->s;
}
```

In particular, we can retrieve the fields `w` and `s` from `insert_event_rep` from the `insert_event` in the member function.

4.3.2.5. Adding your own event classes

Summarizing, in order to add your own new event classes, you have to take care of the following steps:

- Add a new event type to some `event_codes` file.
- Declare and implement the event type and its representation type.
- Declare and implement the event creation functions.
- Reimplement the generic event handler `bool widget_rep::handle(event)` in the abstract or concrete widget representation class, where you want to use your new event class.

4.3.3. The main event loop

The main event loop does the following

- As long as the application did not destroy all its windows, wait for a new event to occur.
- If an event occurs, handle all events on the queue, by creating the appropriate events and sending them to the appropriate widgets (job of the window interface implementation).
- If there are no events left, send an `inquire_event` to each window. This is useful for complex applications, where regions of the window may be invalidated during this phase. Indeed, the event handling phase may consist of many complex actions, so that the regions to invalidate may be determined easier *a posteriori*.
- Requests are emitted in order to repaint the regions which have been invalidated during the event processing stages.

4.3.4. Coordinates

4.3.4.1. Coordinates, pixels and rounding

All coordinates and sizes are represented by instances of type `SI`, which is nothing but another name for `int`. The `SI` constant `PIXEL`, which is a power of two `1 << PIXEL_SHIFT` denotes the size of a pixel on the screen. Since `PIXEL>1`, coordinates and sizes are not necessarily integer multiples of the pixel size. However, the coordinates of the origin and the size of a widget should always be such multiples.

In order to achieve this, some rounding functions are provided. The function `round (SI&)` rounds the argument to an integer multiple of `PIXEL`. Furthermore, the `window` member functions

```
void inner_round (SI& x1, SI& y1, SI& x2, SI& y2);
void outer_round (SI& x1, SI& y1, SI& x2, SI& y2);
```

transform a rectangle into a new one with integer multiple of `PIXEL` coordinates, which is enclosed resp. encloses the original rectangle.

4.3.4.2. Local and global coordinates

Each widget has an origin `(ox,oy)` with respect to the window to which it has been attached. This is the origin of the "local coordinates". The origin of the "global coordinates" is the origin of the window. The location of the local origin in the widget is determined by the widget's gravity, which is either one of `north_west`, `north`, `north_east`, `west`, `center`, `east`, `south_west`, `south` or `south_east`.

As a general rule, events are transmitted in global coordinates. Nevertheless, in widgets which are derived from the abstract basic widget class, by default, all computations are done with respect to local coordinates. This is due to two reasons

- When an event has to be processed, the abstract widget event handler translates global into local coordinates and calls the appropriate virtual event handler using local coordinates.
- When an event has to be emitted, the abstract widget provides event construction routines w.r.t. local coordinates, which override the global event construction routines w.r.t global coordinates.

4.3.4.3. Screen coordinates

For some very particular purposes, such as popping up windows, one has to perform computations with respect to the screen coordinates. Given a point `(x,y)` in the coordinates of some window `win`, the screen coordinates of `(x,y)` are obtained by adding the windows origin, which is obtained by calling `win->get_position (ox,oy)`.

4.3.5. Attaching and positioning widgets

4.3.5.1. Attaching widgets

When a widget is created, the `win` field of its representation is set to `NULL`, since it is not yet attached to a window. In order to attach a widget `w` to a window `win`, one emits an `attach_window_event`:

```
w << emit_attach_window (win);
```

Notice that taking `win==NULL` results in detaching the widget. Notice also that a widget may be attached to at most one window: attempts to reattach a widget, which is already attached, to another window, result in a fatal error.

Some events can be handled by widgets which are not yet attached to a window, such as:

- "get size" events, which determine the default, minimal and maximal size of a widget. Such an event is generated before the creation of the window to which the widget will be attached in order to determine the size of the window.
- "attach window" events, in order to attach (or detach) a window.
- Events for setting (and getting) attributes: after the creation of a widget some attributes of the widget may be given some default value. In order to change them, one might wish to set them to other values before attaching the widget to a window.

- Events for modifying the composite structure of a widget: these events are used for instance in order to construct menus.

For some of these events, such as attribute changes, it may be necessary to emit invalidate events in case when the widget had been attached to some window. In order to test this one uses the member function

```
bool widget_rep::attached ();
```

4.3.5.2. Positioning widgets

When an appropriate size `(w,h)` has been determined for a widget (using "get size" events) and when a widget has been attached to some window, the widget is positioned in the main window. By default, all children are recursively positioned at the top left of the window at sizes `(w,h)`. But for complex widgets with children, a specific positioning routine usually has to be implemented.

Such a routine involves positioning of the children within the parent. This is done by emitting position events to the children. For instance,

```
a[i] << emit_position (x[i], y[i], w[i], h[i], center);
```

positions the `i`-th child, such that the origin of `a[i]` is at position `(x[i], y[i])` w.r.t. the local coordinates of `this` and such that the origin is situated in the center of `a[i]`. The width and height of `a[i]` are set to `w[i]` resp. `h[i]`.

4.3.5.3. Repositioning widgets

During execution, it may happen that a particular widget has changed, so that it obtains a different size and/or position. In this case, one emits an `update_event` to the closest ancestor, whose position and size did not change.

For instance, consider the case of a footer `footer`, which consists of a left footer `footer["left"]`, followed by some glue `footer["middle"]` and a right footer `footer["right"]`. When the left footer changes:

```
footer << set_widget ("left", text_widget ("new text"));
```

the size of `footer["left"]` changes, and the size and position of the glue should also be changed. Nevertheless, the size and position of `footer` remain unaltered, whence we update `footer`:

```
footer << emit_update ();
```

Updating an attached widget results in three actions to take place:

- The widget is reattached to its own window. Indeed, some children of the widget might need be attached.
- The widget is repositioned at its current position and size. Again this will actually affect the children.
- The widget is invalidated, so that it will be repainted.

4.3.6. The keyboard

4.3.6.1. Keyboard focus

Each window `win` on the screen determines a main widget `win->w` which is attached to it and a descendant `win->kbd_focus` of this widget, which handles the keyboard input directed to the window. This latter widget `win->kbd_focus`, which is set to `win->w` by default, is said to have keyboard focus, if the window `win` has keyboard focus (i.e. if all keyboard events are sent to this window). Consequently, the widget which has keyboard focus receives all keyboard events.

When the keyboard focus of a window `win` changes, a `keyboard_focus_event` is sent to `win->kbd_focus`. The field `ev->flag` of this event `ev` is `TRUE` if the window got the focus, and `FALSE` if the window lost focus.

The keyboard focus widget `win->kbd_focus` associated to a window can be changed by calling the `window_rep` member function

```
void window_rep::set_keyboard_focus (widget);
```

Setting the input focus to another widget than `win->w` is useful, for instance, if a particular text input field of some form needs keyboard focus after a mouse click on it.

4.3.6.2. Keyboard events

When a widget has the keyboard focus, and a key is pressed, it receives a `keypress_event`. The `keypress_event_rep` class contains a field `key`, which contains a comprehensible string corresponding to the key which was pressed.

More precisely, `key` is either a one character string, or a symbolic name like `"<return>"`, `"<right>"`, `"<del>"`, etc. or a composed name like `"<shift-F1>"`, `"<ctrl-esc>"` or `"<meta-x>"`. The complete list of keys is as follows:

```
"<F1>", "<F2>", "<F3>", "<F4>", "<F5>", "<F6>",
"<F7>", "<F8>", "<F9>", "<F10>", "<F11>", "<F12>",
"<esc>", "<tab>", "<less>", "<gtr>", "<del>", "<return>",
"<ins>", "<home>", "<end>", "<page-down>", "<page-up>",
"<left>", "<up>", "<down>", "<right>"
```

The keys `"<less>"` and `"<gtr>"` correspond resp. to `"<"` and `">"`. The allowed modifiers are `"shift"`, `"ctrl"` and `"meta"` or combinations of these.

4.3.7. The mouse

4.3.7.1. Mouse events

A mouse event `ev` occurs on a button change or a mouse movement. The `ev->type` field contains the type of the event and `ev->x` and `ev->y` the corresponding coordinates of the mouse. Finally, the states of the mouse buttons can be questioned using the routine `ev->pressed (string)`.

The possible values of `ev->type` on button change events are the following:

```
"press-left", "press-middle", "press-right",
"release-left", "release-middle", "release-right"
```

The possible values for mouse movement events are

```
"move", "enter", "leave"
```

The `"enter"` and `"leave"` events occur when the mouse enters resp. leaves the widget. Finally, the states of the left, middle and right mouse buttons can respectively be obtained using the calls

```
ev->pressed ("left")
ev->pressed ("middle")
ev->pressed ("right")
```

4.3.7.2. Grabbing the mouse

For some applications such as popup menus or scrollbars, it is useful to direct all mouse events to a particular widget `w`. This is done by grabbing the mouse by emitting the event

```
w << emit_grab_mouse (TRUE)
```

After such a grab, all mouse events are directed to `w`, even those events which occurred before the grab (contrary to X Window). The mouse grab is released by

```
w << emit_grab_mouse (FALSE)
```

Actually, the display keeps track of a list of widgets for which a mouse grab occurred: if the mouse is grabbed by widgets `w1` next `w2`, and again ungrabbed by `w2`, then all mouse events are again sent to `w1`. This feature is useful for successive grabs by recursive popup menus.

When a widget `w1` grabs the mouse, and a previous mouse grab on a widget `w2` is still active, then a `"leave"` event is sent to `w2` and an `"enter"` event to `w1`. Similarly, if `w1` releases the grab, then a `"leave"` event is sent to `w1` and an `"enter"` event to `w2`.

4.3.8. The screen

Each window keeps track of a list of rectangles to be repainted (moreover, redundant rectangles are eliminated automatically and adjacent rectangles are transformed in larger rectangles). During the repaint stage in the event loop, the widget is requested to repaint these rectangles.

4.3.8.1. Repainting rectangles

The repaint handler takes on input a `repaint_event ev`, which determines the rectangle to be repainted. Moreover, `repaint_event_rep` contains a boolean field `stop`, which can be set in order to indicate that the repaint process was stopped somewhere in the middle.

Indeed, for widgets which take a long time to be repainted, it may be useful to abort repainting if a key is pressed. The arrival of an event which aborts repainting can be checked directly on the postscript device `dev`:

```
if (dev->check_event (EVENT_STATUS)) { ... } // CHECK THIS
```

In the case of a window, such an event may signify that a key has been pressed; in the case of a printer, it might suggest the printer being turned off.

If the application decides to abort repainting, it sets `ev->stop` to `TRUE`. The rectangle which was being repainted is put back on the invalid rectangles list in the event loop; it will be processed again during the next pass through the repaint phase.

4.3.8.2. Invalidation of rectangles

When window is mapped on the screen or when a region is exposed, the window interface automatically invalidates the corresponding rectangle. The user may also invalidate a rectangle by using either one of the routines

```
event emit_invalidate_all ();
event emit_invalidate (SI x1, SI y1, SI x2, SI y2);
```

The first routine creates an event to invalidate the entire widget area; the other routine invalidates a specified region.

4.3.9. The toolkit

4.3.9.1. Other standard widget classes

Many widgets from the toolkit are derived from some other standard abstract widget classes, which can handle some other special events.

4.3.9.2. Composite widgets

These widgets allow to add or remove children to or from a widget. This makes them particularly useful for menu widgets. They respond to `clean`, `insert` and `remove` events.

4.3.9.3. Attribute widgets

These widgets allow to set window attributes of some common types such as integers, strings, commands, points, etc. They can be used for instance to retrieve an input string or in order to set the scroll position in a canvas widget.

4.3.9.4. Glue widgets

Glue widgets are created by

```
widget glue_widget (bool hext=TRUE, bool vext=TRUE, SI w=0, SI h=0);
```

The first two arguments determine whether the widget is extensible horizontally resp. vertically. The last two elements determine the default and minimal size of the widget.

4.3.9.5. Text widgets

Text widgets are created using

```
widget text_widget (string s);
```

They just display the text `s`.

4.3.9.6. Buttons

Two types of buttons have been implemented. First, command buttons are created using

```
widget command_button (string s, command cmd);
```

They display the text `s` and execute the command `cmd` when pressed. Secondly, we implemented popup buttons, which popup some window when pressed. Popup buttons are created by one of

```
widget pulldown_button (string s, widget m);
widget pullright_button (string s, widget m);
```

depending on where the popup window should popup. The main widget attached to the popup window should be created using

```
widget popup_widget (widget w, gravity quit);
```

The `quit` argument specifies that the popup window should disappear as soon as the pointer leaves the widget in the `quit` direction.

4.3.9.7. Menus

Horizontal and vertical menus are created using

```
widget horizontal_menu ();
widget vertical_menu ();
```

By default, they are empty. Subsequently, they can be modified as composite widgets.

4.3.9.8. Canvas widgets

Canvas widgets are created using

```
widget canvas_widget (widget w);
```

Canvas widget consist of a portion of the widget `w` and scrollbars, which enable to scroll `w`. The events

```
event set_scrollable (widget w);
event set_extents    (SI ew, SI ey);
event set_scroll_pos (SI x, SI y);
event get_extents    (SI& ew, SI& eh);
event get_visible    (SI& x1, SI& y1, SI& x2, SI& y2);
```

enable to change `w`, to set the extents of `w`, to set the scroll position, to get the extents of `w` and to get the rectangle of `w`, which is currently visible.

4.3.9.9. Input widgets

Input widgets enable to type a string and to retrieve it when finished. They are created using

```
widget input_text_widget (command call_back);
```

Some initial text can be put in it using

```
event set_input_string (string s);
```

The command `call_back` is executed when typing has been finished or aborted (by typing return, escape or ctrl-c). The typed string can then be retrieved using

```
event get_input_string (string& s);
```

Usually, the returned `s` is a string enclosed between quotes. If typing was aborted, `s` contains the string `"cancel"`.

Chapter 5

TEXMACS fonts

5.1. Classical conceptions of fonts

The way TEXMACS handles fonts is quite different from classical text editors and even from TEX. Let us first analyze some classical ways of conceiving fonts.

- Physical fonts are just given by the name of a file, which contains a character set, i.e. a list of bitmaps. Usually the size of a character set is limited by 256 (or 65536).
- True type fonts essentially work in the same way, except that the bitmaps can now be computed for any desired size.
- In the X-window system, the name of the font is replaced by a more systematic name, which explicitly contains a certain number of font parameters, such as its size, series and shape. This makes it easier for applications to select an appropriate font. However, character sets are still limited in size.
- In TEX, symbols are seen as commands, which select an appropriate physical font (which corresponds to a `.tfm` and a `.pk` file), based on symbol font declarations and environment variables (such as size, series and shape).

Clearly, among all these methods, TEX provides the largest flexibility. However, philosophically speaking, we think that it also has some drawbacks:

- There is no distinction between usual commands and commands to make symbols: the current time might be considered as a symbol.
- The encoding of the font is fixed by the names of the commands. For instance, for mathematical symbols, no clean general encoding scheme is provided, except the default naming of symbols by commands.
- For beginners, it remains extremely hard to use non standard fonts.

Actually, in TEX, the notion of "the current font" is ill-defined: it is merely the superposition of all character generating commands.

5.2. The conception of a font in TeXmacs

Philosophically speaking, we think that a font should be characterized by the following two essential properties:

1. A font associates graphical meanings to *words*. The words can always be represented by strings.

2. The way this association takes place is coherent as a function of the word.

By a word, we either mean a word in a natural language, or a sequence of mathematical, technical or artistic symbols.

This way of viewing fonts has several advantages:

1. A font may take care of kerning and ligatures.
2. A font may consist of several "physical fonts", which are somehow merged together.
3. A font might in principle automatically build very complicated glyphs like hieroglyphs or large delimiters from words in a well chosen encoding.
4. A font is an irreductable and persistent entity, not a bunch of commands whose actions may depend on some environement.

Notice finally that the "graphical meaning" of a word might be more than just a bitmap: it might also contain some information about a logical bounding box, appropriate places for scripts, etc. Similarly, the "coherence of the association" should be interpreted in its broadest sense: the font might contain additional information for the global typesetting of the words on a page, like the recommended distance between lines, the height of a fraction bar, etc.

5.3. String encodings

All text strings in TEXMACS consist of sequences of either specific or universal symbols. A specific symbol is a character, different from `'\0'`, `'<'` and `'>'`. Its meaning may depend on the particular font which is being used. A universal symbol is a string starting with `'<'`, followed by an arbitrary sequence of characters different from `'\0'`, `'<'` and `'>'`, and ending with `'>'`. The meaning of universal characters does not depend on the particular font which is used, but different fonts may render them in a different way.

Universal symbols can also be used to represent mathematical symbols of variable sizes like large brackets. The point here is that the shapes of such symbols depend on certain size parameters, which can not conveniently be thought of as font parameters. This problem is solved by letting the extra parameters be part of the symbol. For instance, `"<left-(-1>"` would be usual bracket and `"<left-(-2>"` a slightly larger one.

5.4. The abstract font class

The main abstract `font` class is defined in `font.hpp`:

```
struct font_rep: rep<font> {
  display  dis;              // underlying display
  encoding enc;              // underlying encoding of the font
  SI       design_size;      // design size in points/256
  SI       display_size;     // display size in points/PIXEL
  double   slope;            // italic slope
  space    spc;              // usual space between words
  space    extra;            // extra space at end of words
```

```
    SI       y1;              // bottom y position
    SI       y2;              // top y position
    SI       yfrac;           // vertical position fraction bar
    SI       ysub;            // base line for subscripts
    SI       ysup;            // base line for superscripts

    SI       wpt;             // width of one point in font
    SI       wquad;           // wpt * design size in points
    SI       wunit;           // unit width for extendable fonts
    SI       wfrac;           // width fraction bar
    SI       wsqrt;           // width horzontal line in square root
    SI       wneg;            // width of negation line

    font_rep (display dis, string name);
    font_rep (display dis, string name, font fn);
    void copy_math_pars (font fn);

    virtual void   get_extents (string s, text_extents& ex) = 0;
    virtual void   draw (ps_device dev, string s, SI x, SI y) = 0;

    virtual SI     get_sub_base (string s);
    virtual SI     get_sup_base (string s);
    virtual double get_left_slope  (string s);
    virtual double get_right_slope (string s);
    virtual SI     get_left_correction  (string s);
    virtual SI     get_right_correction (string s);
    virtual SI     get_lsub_correction (string s, double level);
    virtual SI     get_lsup_correction (string s, double level);
    virtual SI     get_rsub_correction (string s, double level);
    virtual SI     get_rsup_correction (string s, double level);

  void var_get_extents (string s, text_extents& ex);
  void var_draw (ps_device dev, string s, SI x, SI y);
  virtual bitmap_char get_bitmap (string s);
};
```

The main abstract routines are `get_extents` and `draw`. The first routine determines the logical and physical bounding boxes of a graphical representation of a word, the second one draws the string on the the screen.

The additional data are used for global typesetting using the font. The other virtual routines are used for determening additional properties of typesetted strings.

5.5. Implementation of concrete fonts

Several types of concrete fonts have been implemented in TEXMACS:

TeX text fonts. See `src/Resource/Fonts/tex_font.cc`.

TeX rubber fonts. See `src/Resource/Fonts/tex_rubber_font.cpp`.

X fonts. See `src/Resource/Fonts/ps_font.cpp`.

Mathematical fonts. See `src/Resource/Fonts/math_font.cpp`.

Virtual fonts. See `src/Resource/Fonts/virtual_font.cpp`.

In most cases, the lowest layer of the implementation consists of a collection of bitmaps, together with some font metric information. The font is responsable for putting these bitmaps together on the screen using some appropriate spacing. The `ps_device` class comes with a method to display bitmaps in a nice, anti-aliased way, or to print them out.

5.6. Font selection

After having implemented fonts themselves, an important remaining issue is the selection of the appropriate font as a function of a certain number of parameters, such as its name, series, shape and size. For optimal flexibility, TEXMACS comes with a powerful macro-based font-selection scheme (using the SCHEME syntax), which allows the user to decide which parameters should be considered meaningful.

At the lowest level, we provide a fixed number of macros which directly correspond to the above types of concrete fonts. For instance, the macro

```
(tex $name $size $dpi)
```

corresponds to the constructor

```
font tex_font (display dis, string fam, int size, int dpi, int
dsize=10);
```

of a TEX text font.

At the middle level, it is possible to specify some rewriting rules like

```
((roman rm medium right $s $d) (ec ecrm $s $d))
((avant-garde rm medium right $s $d) (tex rpagk $s $d 0))
((x-times rm medium right $s $d) (ps adobe-times-medium-r-normal $s
$d))
```

When a left hand pattern is matched, it is recursively substituted by the right hand side. The files in the directory `progs/fonts` contain a large number of rewriting rules.

At the top level, TEXMACS calls a macro of the form

```
($name $family $series $shape $size $dpi)
```

as a function of the current environment in the text. In the future, the top level macro call might change in order to enable the user to let the font depend on other environment variables.

Chapter 6

Mathematical typesetting

6.1. Introduction

In this chapter we describe the algorithms used by TEXMACS in order to typeset mathematical formulas. This is a difficult subject, because esthetics and effectiveness do not always go hand in hand. Until now, TEX is widely accepted for having achieved an optimal compromise in this respect. Nevertheless, we thought that several improvements could still be made, which have now been implemented in TEXMACS. We will shortly describe the motivations behind them.

In order to obtain esthetic formulas, what criteria should we use? It is often stressed that good typesetting allows the reader to concentrate on what he reads, without being distracted by ugly typesetting details. Such distracting details arise when distinct, though similar parts of text are typesetted in a non uniform way:

Different base lines. The eye expects text of a similar nature to be typesetted with respect to a same base line. For instance, in $x + y + z$, the bottoms of the x and z should be at the same height as the bottom of the u-part in the y. This should again be the case in $2^x + 2^y + 2^z$.

Unequal spacing. Different components of text with approximately the same function should be separated by equal amounts of space. For instance, in $a^2 + f^2$, the typesetter should notice the hangover of the f. This should again be the case in $e^a + e^f + e^x$. Similarly, the distance between the baselines of the a and the i in a_i should not be disproportially large with respect to the height of an x.

Additional difficulties may arise when considering automatically generated formulas, in which case line breaking has to be dealt with in a satisfactory way.

Unfortunately, the different esthetic criteria may enter into conflict with each other. For instance, consider the formula $x_p + x_p^2$. On the one hand, the baselines of the scripts should be the same, but the other hand, the first subscript should not be "disproportionally low" with respect to the x. Unfortunately, this dilemma can not been solved in a completely satisfactory way without the help of a human for the simple reason that the computer has no way to know whether the x_p and x_p^i are "related". Indeed, if the x_p and x_p^i are close (like in $x_p + x_p^i$), then it is natural to opt for a common base line. However, if they are further away from each other (like in $x_p + \sum_{i=0}^{\infty} c_i x_p^i$), then we might want to opt for different base lines and locally optimize the rendering of the first x_p.

Consequently, TEXMACS should offer a reasonable compromise for the most frequent cases, while offering methods for the user to make finer adjustments in the remaining ones. We provide the constructs Format→Adjust→Move and Format→Adjust→Resize to move and resize boxes in order to perform such adjustments. For instance, if the brackets around the two sums

$$\phi\left(\sum_i a_i x^i\right) = \psi\left(\sum_j b_j y^j\right)$$

have different sizes, then one may resize the bottom of the subscript j of the second sum to `0fn`. Alternatively, one may resize the bottoms of both the i and j subscripts to (say) `-0.3fn`. For easier adjustments you may use Format→Adjust→Smash and Format→Adjust→Swell to automatically adjust the size of the contents to the height of the character "x" and the largest one in the font respectively.

Notice that one should adjust by preference in a structural and not visual way. For instance, one should prefer `-0.3fn` to `-2mm` in the above example, because the second option disallows you to switch to another font size for your document. Similarly, you should try not change the semantics of the formula. For instance, in the above example, you might have added a "dummy subscript" to the i subscript of the sum. However, this would alter the meaning of the formula (whence make it non suitable as input to a computer algebra system) In the future, we plan to provide additional constructs in order to facilitate structural adjusting. For instance, in the case of a formula like

$$1 + x_1 + x_1^2 + \cdots + x_2 + x_1x_2 + x_1^2x_2 + \cdots x_2^2 + x_1x_2^2 + x_1^2x_2^2 + \cdots,$$

one might think of a construct to enclose the entire formula into an area, where all scripts are forced to be double (using dummy superscripts wherever necessary).

6.2. The font parameters

Several font parameters are crucial for the correct positioning of the different components. The following are often needed:

quad. The main font reference space `1fn`, which can be taken as the distance between successive lines of text.

y1 and y2. The bottom and top level for the font (we have `y2-y1=quad`).

sep. The reference minimal space between distinct components, like the minimal distance between a subscript and a superscript. In fact, `sep=quad/10`.

wline. The width of several types of lines, like the fraction and square root bars, wide accents, etc.

yfrac. The height of the fraction bar, which is needed for the positioning of fractions and big delimiters. Usually, `yfrac` is almost equal to `yx/2` below.

The following parameters are mainly needed in order to deal with scripts:

yx. The height of the x character, which is needed for the positioning of scripts. All the remaining parameters are actually computed as a function of `yx`.

ysub lo base. Logical base line for subscripts.

ysub hi lim. Subscripts may never physically exceed this top height.

ysup lo base. Logical base line for superscripts.

ysup lo lim. Superscripts may never physically exceed this bottom height.

ysup hi lim. Suggestion for a physical top line for superscripts.

yshift. Possible shift of the base lines when we are inside fractions or scripts.

The individual strings in a font also have several important positioning properties. First of all, they always admit left and right slopes. Furthermore, they admit left and right italic corrections, which are needed for the positioning of scripts or when passing from text in upright to text in italics (or vice versa).

6.3. Some major mathematical constructs

6.3.1. Fractions

The following heuristics are used:

- The horizontal middles of the numerator and the denominator are taken to be the same.
- The vertical spaces between the numerator resp. denominator and the fraction bar is at least `sep`.
- The depth (resp. height) of the numerator (resp. denominator) is descended (resp. increased) to `y1` (resp. `y2`) if necessary. This forces the base lines of not too large numerators resp. denominators to be the same in presence of multiple fractions.
- The fraction bar has a overhang of `sep/2` to both sides and the logical limits of the fraction are another `sep/2` further. The logical left limit is zero.

The italic corrections are not taken into account during the positioning algorithms, because this may create the impression that the numerator and denominator are not correctly centered with respect to each other. Nevertheless, the italic corrections are taken into account in order to compute the logical bounding box of the fraction (whose has italic slopes vanish at both sides).

6.3.2. Roots

The following heuristics are used:

- The vertical space between the main argument and the upper bar is at least `sep`.
- The root itself is typesetted like a large delimiter. The positioning of a potential script works only is very dependent on the usage of TEX fonts.
- The upper bar has a overhang of `sep/2` at the right and the logical right limit of the root is situated another `sep/2` further to the right.

We take the logical right border plus the italic correction of the main argument in order to determine the right hand limit of the upper bar. The left italic correction is not needed.

6.3.3. Negations

The following heuristics are used:

- The negation bar passes through the logical center of the argument.

- The italic corrections of the argument are only taken into account during the computation of the logical limits of the negation box (which has zero left and right slopes).

6.3.4. Wide boxes

The following heuristics are used:

- We use TEX fonts for small accents and an *ad hoc* algorithm for the wider ones.
- The distance between the main argument and the accent is at least `sep` (or a distance which depends on the TEX font for small accents).
- The accent is positioned horizintally according to the right slope of the main argument.
- The slopes for the accented box are inherited from those of the main argument and the italic corrections are adjusted accordingly.
- All script height parameters of the accented box are inherited from the main argument. The only exception is `ysup_hi_lim`, which may be increased by the height of the accent, or determined in the generic way, whichever leads to the least value. It is indeed better to keep superscripts positioned reasonably low, whenever possible.

6.4. SUBSCRIPTS AND SUPERSCRIPTS

The positioning of subscripts and superscripts is a complicated affair, due to the conflict between locally and globally optimal esthetics mentioned above. The base line for a subscript is determined as follows:

1. Always pretend that the subscript has height at least `y2-yshift` in the script font (actually we should use the height of an M instead).
2. Try to position the script at the base line given by the main argument.
3. If the top limit (given by the main argument) is physically exceeded by the subscript, then the base line is moved further down accordingly.

The base line for a superscript is determined as follows:

1. Try to physically position the superscript beneath the suggested top line. Usually, this will place the superscript to far down.
2. Move the superscript up to the logical base line if necessary. This will usually occur: most of the time, the logical base line is the just the height of an x-script below the suggested top line.
3. If the superscript physically descends below the physical under limit given by the main box, then we move the superscript further upwards.

If both a subscript and a superscript were present, then we still have to adjust the base lines: if the top of the subscript and the bottom of the superscript are not physically separated by `sep`, then we both move the subscript and the superscript by the same amount away from each other. Because of step 1 in the positioning of the subscript, the base lines of double scripts will usually be the same in formulas with several of them.

The right slope and italic correction of a script box may be non trivial. In order to compute them, we first determine the script (or main argument), whose right limit (taking into account its italic correction) is furthest to the right (this may be the main box, in the case of a big integral with a tiny subscript). Then the right slope of the main box is inherited by the right slope of this script (or main argument). As to the italic correction, it is precisely the difference between the right offset of the script plus its italic correction minus the logical right coordinate of the entire box. The italic correction should be at least zero though. The left slope and italic correction are computed in a similar way.

6.5. Big delimiters

The automatic positioning and computation of sizes of big delimiters is again complicated because of potential conflicts between locally and globally optimal esthetics.

First of all, TEX fonts come only with a discrete set of possible sizes for large delimiters. This is an advantage from the point of view that it favorites delimiters around slightly different expressions to have the same baselines. However, it has the disadvantage that delimiters are easily made "one size to large". For this reason, we actually diminish the height and the depth of the delimited expression by the small amount `sep`, before computing the sizes of the delimiters.

Secondly, it is best when the vertical middles of big delimiters occur at the height of fraction bars. However, in a formula like

$$f\left(\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{x}}}}\right),$$

it may be worth it to descend the delimiters a bit. On the other hand, slight vertical shifts in the middles of the delimiters potentially have a bad effect on base lines, like in

$$f\left(\sum_{i=1}^{b} X_i\right)+g\left(\sum_{j=1}^{a} Y_j\right).$$

In TEXMACS, we use the following compromise: we start with the middle of the delimited expression as a first approximation to the middle of the delimiters. The real middle is obtained by shifting this middle towards the height of fraction bars by an amount which cannot exceed `sep`.

From a horizontal point of view, we finally have to notice that we adapted the metrics of the big delimiters in a way that potential scripts are positioned in a better way. For instance, according to the TEX `tfm` file, in a formula like

$$\left(A+\left(\sum_{i=1}^{10} B_i\right)^2\right),$$

the square rather seems to be a left superscript of the second closing bracket than a right superscript of the first one. This is particularly annoying in the case of automatically generated formulas, where this situation occurs quite often.

3. Some boxes, such header and footers, or certain results of macro expansions, may not be "accessible". Although one should be able to find a reasonable cursor position when clicking on them, the contents of this box can not be edited directly.

4. The correspondence has to be reasonably complete (see the next section).

The first difficulty forces us to store a path in the source tree along with any box. In order to save storage, this path is stored in a reversed manner, so that common heads can be shared. This common head sharing is also necessary to quickly change the source locations when modifying the source tree, for instance by inserting a new paragraph.

In order to cope with the third difficulty, the inverse path may start with a negative number, which indicates that the box can not directly be edited (we also say that the box is a decoration). In this case, the tail of the inverse path corresponds to a location in the source tree, where the cursor should be positioned when clicking on the box. The negative number influences the way in which this is done.

7.2.2. The three kinds of paths

More precisely, we have to deal with three kinds of paths:

Tree paths. These paths correspond to paths in the source tree. Actually, the path minus its last item points to a subtree of the source tree. The last item gives a position in this subtree: if the subtree is a leaf, i.e. a string, it is a position in this string. Otherwise a zero indicates a position before the subtree and a one a position after the subtree.

Inverse paths. These are just reverted tree paths (with shared tails), with an optional negative head. A negative head indicates that the tree path is not accessible, i.e. the corresponding subtree does not correspond to editable content. If the negative value is -2, -3 or -4, then a zero or one has to be put behind the tree path, depending on the value and the cursor position.

Box paths. These paths correspond to logical paths in the box tree. Again, the path minus its last item points to a subbox of the main box, and the last item gives a position in this subtree: if the subbox corresponds to a text box it is a position in this text. Otherwise a zero indicates a position before the subbox and a one a position after it. In the case of side boxes, a two and a three may also indicate the position after the left script resp. before the right script.

7.2.3. The conversion routines

In order to implement the conversion between the three kinds of paths, every box comes with a reference inverse path `ip` in the source tree. Composite boxes also come with a left and a right inverse path `lip` resp. `rip`, which correspond to the left-most and right-most accessible paths in its subboxes (if there are such subboxes).

The routine:

```
virtual path box_rep::find_tree_path (path bp)
```

transforms a box path into a tree path. This routine (which only uses `ip`) is fast and has a linear time complexity as a function of the lengths of the paths. The routine:

```
virtual path box_rep::find_box_path (path p)
```

does the inverse conversion. Unfortunately, in the worst case, it may be necessary to search for the matching tree path in all subboxes. Nevertheless, in the best case, a dichotomic algorithm (which uses `lip` and `rip`), finds the right branch how to descend in a logarithmic time. This algorithm also has a quadratic time complexity as a function of the lengths of the paths, because we frequently need to revert paths.

7.3. The cursor and selections

In order to fulfill the requirement of being a "structured editor", TEXMACS needs to provide a (reasonably) complete correspondence between logical tree paths and physical cursor positions. This yields an additional difficulty in the case of "environment changes", such as a change in font or color. Indeed, when you are on the border of such a change, it is not clear *a priori* which environment you are in.

In TEXMACS, the cursor position therefore contains an x and a y coordinate, as well as an additional infinitesimal x-coordinate, called δ. A change in environment is then represented by a box with an infinitesimal width. Although the δ-position of the cursor is always zero when you select using the mouse, it may be non zero when moving around using the cursor keys. The linear time routine:

```
virtual path box_rep::find_box_path (SI x, SI y, SI delta)
```

as a function of the length of the path searches the box path which corresponds to a cursor position. Inversely, the routine:

```
virtual cursor box_rep::find_cursor (box bp)
```

yields a graphical representation for the cursor at a certain box path. The cursor is given by its x, y and δ coordinates and a line segment relative to this origin, given by its extremities (x_1, y_1) and (x_2, y_2).

In a similar way, the routine:

```
virtual selection box_rep::find_selection (box lbp, box rbp)
```

computes the selection between two given box paths. This selection comprises two delimiting tree paths and a graphical representation in the form of a list of rectangles.