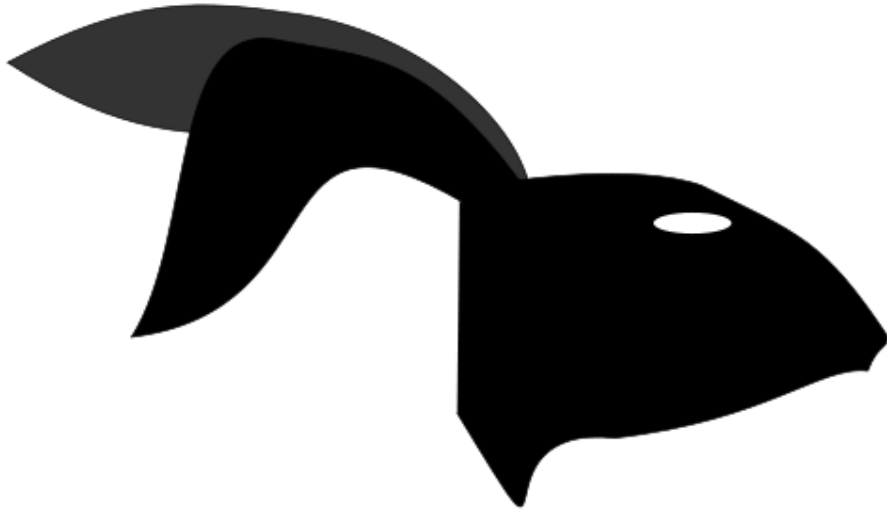


ViTE

June 16, 2009



Technical Manual

Contents

1	Convention for making code	3
1.1	Convention to code	3
2	Parser module	5
2.1	Production of tokens	5
2.1.1	Introduction	5
2.1.2	Definition	5
2.1.3	Words of the language	5
2.1.4	Automaton of the language's words	5
2.1.5	Implementation	6
2.1.6	Tests	7
2.2	Representation of the file trace line by line	7
2.2.1	Introduction	7
2.2.2	Definition	7
2.2.3	Implementation	7
2.3	Representation of the definitions described in the trace file	8
2.3.1	Introduction	8
2.3.2	Implementation	8
2.4	Acknowledge of definitions' lines	8
2.4.1	Introduction	8
2.4.2	Definition	8
2.4.3	Automaton of definition	8
2.4.4	Specification	8
2.4.5	Tests	9
2.5	Event parser	9
2.5.1	Introduction	9
2.5.2	Implementation	9
2.6	Handling of a trace	9
2.6.1	Introduction	9
2.6.2	Implementation	9
2.6.3	Specification	9
2.6.4	Tests	9
3	The data structure	11
3.1	The Main classes	11
3.1.1	The class Trace	12
3.1.2	The class container	12
3.1.3	The class State	12
3.1.4	The class event	12
3.1.5	The class Link	13
3.2	The lists	13
3.3	The trees	13

3.4	Algorithms on the trees :	14
3.4.1	Browse tree	14
3.4.2	Getting informations on a clicked element	15
4	Human-Computer Interface	16
4.1	The Interface class	16
4.2	The <i>ViTE</i> notify message system : the <i>Message</i> class	16
4.3	Console User Interface : class <i>Interface_console</i>	17
4.3.1	Description	17
4.3.2	How it works	17
4.3.3	Managing of the arguments	18
4.3.4	Graphical User Interface : class <i>Interface_graphic</i>	18
5	The render system	19
5.1	The render opengl	19
5.1.1	The drawing methods	19
5.1.2	The scrolling and scaling methods	20
5.2	Render out Svg file	22
6	Some possible ameliorations	25
6.1	Using filters	25
6.2	Export in another image file format	25
6.3	A better entry format?	25
6.4	Allocation by lot	26
6.5	Reading at the flight	26
6.6	Saves of configurations	26
6.7	Interface enhancement	26

Chapter 1

Convention for making code

1.1 Convention to code

Generalities

- The software is totally written using C++ language.
- Source files must be named **.cpp* and headers **.hpp*
- Code must have commentaries in order to create automatically documentation using the *Doxygen* tool.
- The indentation is 4 spaces.
- Directories are organized using modules. Each contains a *Makefile*.

Variables

- Variables' names only contain letters and numbers. Letters belong to [a-z] and [0-9], excluding [A-Z]. *eg : temp, size2*
- If a name is composed of some words, then they must be separated with the **underscore symbol**, and all the letters are in [a-z][0-9] as said above. *eg : number_of_item, size_of_window, ...*
- Boolean variables must have a name that remains their functionality, so names must start with the prefix *is_* or *has_*. *eg : is_enabled*
- Attributes of classes or structures must begin with the symbol **underscore**. *eg : _name, _is_enabled, _width*

Constants

- Constants are written using only [A-Z] and [0-9].
- If a constant is composed by more than one word, then they must be, as for variables, separated using the **underscore symbol**.

Functions and methods

- Functions' names only contain letters and numbers. Letters belong to [a-z] and [0-9], excluding [A-Z]. *eg : set_width(), get_size()*
- The names of the functions must always contain an action verb.

- Accessors must start with the *get_* verb followed by the nature of the object returned. *eg : get_lenght(), get_value()*
- Mutators must start with the *set_* verb followed by the field modified. *eg : set_size(), set_value()*
- Interrogatives functions must start with *is_*.
- Each opening of bloc must be on the same line as the prototype of the function, the condition, whereas the end must be on a new line.

Eg :

```
if(bool) {
}
else {
}
```

Classes

- The names of the classes must begin with an upper-case followed with only lower-case.

.cpp files

- Namespaces are declared in the *.cpp* and not in the *.hpp* file.
- For classes, each *.cpp* file correspond to a *.hpp* file, with exactly the same name and that is the only file include within the *.cpp*
- The file contain the implementation of all the functions declared in the *.hpp* file (unless virtual functions).

.hpp files

- Classes are defined within a *.hpp* files of same name, no function is implemented within (unless templates).
- The file begins with a *#ifndef* instruction followed by *CLASS_NAME_HPP*
- Then a *#define* instruction followed by *NAME_OF_CLASS_HPP* as above.
- Then a cluster of includes.
- In the class, there are at the beginning the declarations of the variables, then the constructors and the destructor, and finally the declarations of the methods.
- The *private* keyword must clearly be written such as *public*.
- The file must end by the *#endif* instruction.

Chapter 2

Parser module

2.1 Production of tokens

2.1.1 Introduction

The input of `TokenSource` is an ASCII character file. `TokenSource` splits text into lexical units : tokens which belong to the Pajé file format. Here are some words from the Pajé file format : `'EventDef'`, `'%'`, `'10'`, `'''10 10 10'''`.

2.1.2 Definition

ASCII character set is reduced to the following sets :

- LATIN set, the set of character belonging to the regular expression `[a-zA-Z0-9]`.
- ESCAPE set, including `' '`, `'\t'`, `'EOF, end of file mark'`, `'end of file mark'`.
- the escape character `'''`.
- the mark character `'%'`.
- the mark character `'\n'`.

2.1.3 Words of the language

A set of character `m` is a Pajé file format lexical units if and only if :

- `m` is `"%"` token, or
- `m` is `''\n''` token, or
- `m` is braced with the escape character `" "` and contains characters from LATIN or ESCAPE, or
- `m` is braced with two spaces `' '` and contains characters from LATIN

2.1.4 Automaton of the langage's words

An automaton founded on these descriptions which recognize the Pajé file format lexical units is described on the figure above. It explains the algorithm used in `TokenSource` object.

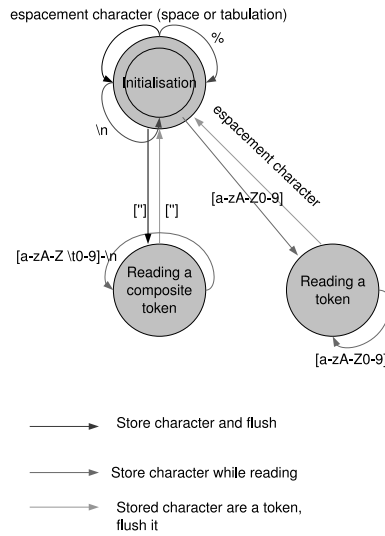


Figure 2.1: The lexer automaton

2.1.5 Implementation

TokenSource object supplies the lexical abstraction which encapsulates a file. `next_token()` call returns a string which is the next token.

Algorithm `next_token`

Implementation faithfully uses the automaton described. For example, when in the automaton, a Latin character is read, it reaches the state building-simple-token which corresponds to the `build_simple_token()` method. It accepts when eof file mark is read.

Enhancement of reading of memory

A ring buffer is used to store temporarily characters read from the file. An empty buffer is loaded from the file. Automaton algorithm applied in buffer stores characters read in a string buffer, returned by `next_token()`.

Control of the characters read

Each step in ring buffer is followed by `ensure_capacity()` test method which can load characters from a file.

Specification

A TokenSource can be instantiated with a filename in its constructor. A TokenSource instantiated by an empty constructor requires the `open()` method to be called so as to link a file to this object. Each `read_token()` call returns a '\0' ended character string, NULL if the file ended is read, or NULL if no file is opened.

2.1.6 Tests

A test produces tokens using `TokenSource`. Trace file Character are splitted and tokens belong to the Pajé file trace format.

2.2 Representation of the file trace line by line

2.2.1 Introduction

A line is a syntactical unit in a file following the Pajé trace format. `TokenSource` supplies tokens, `Line` object handles tokens and supplies them line by line. In Pajé file trace format, a line is the beginning of a definition, the end of a definition, a definition or an event.

For example :

- `'%EventDef PajeDefineContainerType 1'`
- `'% Name string'`
- `'111 1.03305 MT 315146706:1131391392 315146706-135096520 1131391392'`

2.2.2 Definition

A Pajé's line is a `'\n'` ended by a set of tokens.

2.2.3 Implementation

`Line` object stands for a line. It encapsulates a `TokenSource`. The `newline()` method must be used to read the file line by line. `newline()` loads the next file line and enables the `item()` method to access line's tokens : the `item()` method is an indexed access to line's token.

Construction of a line

A line is built by `newline()` that consists in calling `read_token()` as long as the current token read is different from `'\n'`.

Storage

At present, a private `std::vector<std::string>` stores line's tokens. Actually there is no use of an indexed access. The `item()` method could simply supply the next token of a line.

Specification

A `Line` can be instantiated with a filename as parameter for its constructor. A line instantiated by an empty constructor requires the `open()` method to be called, to link a file to the object. When `item()` returns, its output is the token required into the `std::string&` parameter supplied.

2.3 Representation of the definitions described in the trace file

2.3.1 Introduction

A Definition is the definition of events found while parsing the file. It is always composed by a field "name", with other characteristics of the event such as type or values. The set of definition is stored in a table. The definition and its identifiers are paired.

Eg : A file trace : %EventDef PajeDestroyContainer 8 % Time date % Name string % Type string %EndEventDef %EventDef PajeDefineStateType 3 % Alias string % ContainerType string % Name string %EndEventDef

The Table : (8, d1),(3,d2)

d1 définition : "PajeDestroyContainer",f1,f2,f3

f1 field : "Alias", "string"

2.3.2 Implementation

Extensible Definition Fields are stored in a std::vector. Couples of definition and id are available in a std::map<int,Definition> hash table.

2.4 Acknowledge of definitions' lines

2.4.1 Introduction

ParserDefinitionDecoder performs a received Line which starts with percent.

2.4.2 Definition

A field Line is a Line with the two first tokens that are the name of a variable and a name. A EventDef Line is a Line with the first token that is EventDef. A EndEventDef Line is a Line with the first token that is EndEventDef.

A definition is a set of lines that begins by an EndEventDef Line, have an indefinite number of fields Line and ends with an EndEventDef Line.

2.4.3 Automaton of definition

Implementation faithfully uses the automaton described.

Automaton reaches "in a definition" when an EventDef Line is read and instantiated a new definition which enables :

- reading a field Line : which is translated and read fields are added to the current definition.
- reading an EndEventDef Line : which consists in storing the current definition in the table and fall in "out of a definition".

2.4.4 Specification

A ParserDefinitionDecoder can be instantiated by an empty constructor.

The store_definition() method requires a filled Line Object which starts with percent tokens and returns when this line is read.

get_definition() enables access to the table of definition.

2.4.5 Tests

A test reads a trace file and prints the table of definition built. The input file and the stored definition match.

2.5 Event parser

2.5.1 Introduction

The parser provides a Line which represents an event and the definition of this event. The purpose of the event parser is to give the values of this event to the data structure in order to store it.

2.5.2 Implementation

The event parser contains only one method which fills the data structure step by step with the lines (Line) provided. To do that, there are as many local variables as known field names in Pajé's events. The variables which fit the event defined by the definition (Definition) are filled ordered by the fields of the definition by converting the strings read in the line to the defined type. The values which do not fit some known field names are stored in a vector. Then, the event parser uses the data structure methods to fill it. The variables given as arguments for these methods depend on the Pajé event name.

2.6 Handling of a trace

2.6.1 Introduction

ParserPaje reads a file trace and sends messages to Trace accordingly.

2.6.2 Implementation

ParserPaje opens a file (instanciate a Line), instanciates a ParserEvenDecoder and a ParserDefinitionDecoder. It reads lines until the end of the file. Each line is parsed by ParserEventDecoder or ParserDefinitionDecoder whether or not the line starts with % token.

2.6.3 Specification

A ParserPaje can be instanciated by an empty constructor. The parse() method requires a filename to be opened and an instance of a trace object to proceed.

2.6.4 Tests

A test reads a trace file, uses the ParserDefinitionDecoder and ParserEventDecoder and prints the calls to the trace object. Each event produces a call as expected.

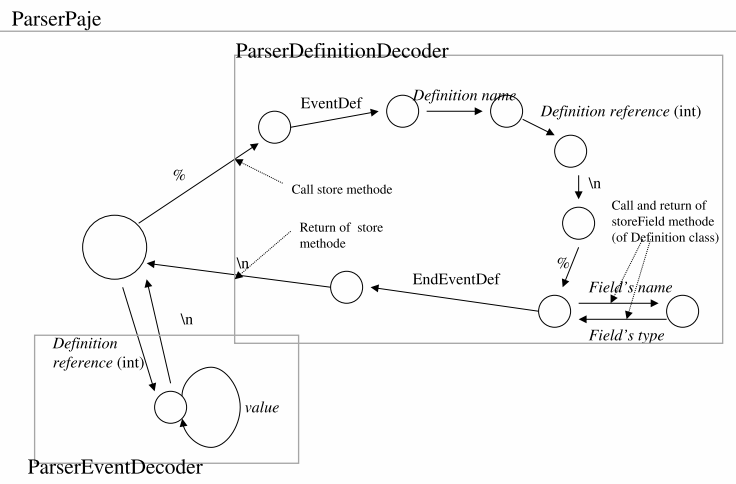


Figure 2.2: The Parser automaton

Chapter 3

The data structure

The data structure is composed by 2 main parts :

- The first one where while parsing double chained list are built,
- The second are trees that are built above the lists.

3.1 The Main classes

To represent the diversity of the Pajé trace format, we have created a wide variety of class that represent the various abstractions on objects :

- Trace
- Container
- State
- Event
- Variable
- Entity

Moreover, we had to create some other classes that help creating these objects such as their type.

The basic types used in the data structure are defined in the *values* directory. These types are :

- Color
- Date
- Double
- Hex
- Integer
- Name
- String
- Value

The definitions for the tree and the interval are in the *tree* directory.

3.1.1 The class Trace

The class *Trace* represents the trace object generated by the user. It contains the elements :

- List of type of container
- List of container
- List of states
- List of events
- List of links
- List of variables

Moreover an attribute that contains the date when the latest action happens in the trace has been created. The methods only enable the construction of the trace, the possibility to get the values of the attributes, and to research some elements in the trace.

3.1.2 The class container

It represents an object of type container. In this class, there are :

- A name
- The date of the beginning and end of the container
- Type of container
- A father
- A list of children
- Set of states, events, links, and their cardinal

The methods are accessors.

3.1.3 The class State

It represents a state of a processor, a container. This class contains the only object of Pajé that is not directly presented. Because the use of trees of punctual events, we do not consider the states but the change of states as a relevant facts. This class has :

- Dates of beginning and end
- Type of state
- A descriptor of the value of the state, contains the optionnal fields.

The methods are accessors and a constructor attribute by attribute.

3.1.4 The class event

It represents an event that occurs in the trace. An event is made by :

- A date when it occurs
- A type of event
- A descriptor of the value of the event, contains the optionnal fields.

Here, the only methods are accessors.

3.1.5 The class Link

It represents a communication between 2 containers in the trace. The attributes are :

- Dates of beginning and end
- A type of link
- The containers source and destination of the communication
- A descriptor with the value of the link(contains the optionnal fields). Methods are accessors.

3.2 The lists

The lists are based on the `std::list` template. They are built in the logical order, each time an element is sent by the parser, it is added in last position in the right list (after making sure the data are rights).

The states are not seen as having a duration but punctual, but as brief moments that represent either the beginning or the end of the action. So, to remain logical, a binary tree is not made with states, but with elements from the *StateChange* class, because a state is not punctual whereas a change is, and represents the same information.

The lists may be :

- List of links, that are contained in the container that is related to the container where the communication starts and where it ends.
- List of containers
- List of States/Events, but they are transformed in binary trees after their construction

3.3 The trees

The trees do not concern the list of communications. The tree is binary and balanced.

Assuming there is a list, a class is used to build the tree reading only one time each element of the list.

The algorithm of construction is based on binary incrementation. To sum up the main idea, 1 element in the list on 2 is a leaf, as the lowest bit in binary, 1 element on 4 is a father of leaves, as the second lowest bit in binary, etc ...

Assuming we know the lenght of the list, we can calculate the height of the tree (because it is balanced) and then the number of leaves in the lowest level. While we have not reached this number of leaves, we build a binary tree linking the nodes depending on their position in the list (in binary). After this limit, a shift is necessary but the algorithm remain the same.

And so we have a binary tree, linked with an infix order kept : each node happen after all the nodes down in its left side, before all the nodes down in its right side, and these latter happen before the father date.

Then, there is a class that links the data structure and the interface, to be able to get all the events within an interval without being forced to visit each previous node.

3.4 Algorithms on the trees :

3.4.1 Browse tree

The main idea is a recursive browse of the tree. Going down we select the nodes that are in the interval, and going up we make sure the node is displayable, and prepare some data for the father to know if it can be displayed.

Algorithm :

```
function browse_recurisif ( root, interval, zoom)

if the node is in the interval
  If the interval is wide enough to display the node

    browse_recurisif ( left child )
    browse_recurisif ( right child )

    If there is a problem on the left side of the left child
      Copy the problematic data in the node
    end if

    If there is a problem on the right side of the left side and
    the left side of the right child
      Making sure we can display the node, or be added to an
      interval of conflict that is then displayed
    end if

    If there is a problem on the right of the right child
      Copy the problematic data in the node
    end if

    If the node has not been displayed yet
      Display the node
    end if

  else // Not wide enough to display
    Current interval set as problematic in both problematic area
  end if

Else if node is after the interval
  browse_recurisif ( left child )
  Copy the data of conflict on the left in the left child

Else // Node before the interval
  browse_recurisif ( right child )
  Copy the data of conflict on the right in the right child

end if

end function
```

3.4.2 Getting informations on a clicked element

Principe : First we browse for the links (because if we start by something else : state, counter or event, we would not be able to find the links because they are over the others elements and thin enough not to hide them). Because the use of a distance, sometimes to get the right information, it is important to zoom enough.

Algorithm :

```
function get_information( positionX, positionY)

calculate the lowest container in the tree of container where
the clic corresponds

for each container father of the container clicked
  look for a communication close enough
  if one is found
    return this communication
  end if
end for each

Search in the list of events of the lowest container
  if one corresponds
    return it
  end if
end search

Search in the list of states of the lowest container
  if one corresponds
    return it
  end if
end search

return no information found

end function
```


Chapter 4

Human-Computer Interface

The interface is divided into two classes : the *console.interface* and the *graphical.interface* inherited from the *Interface* class.

4.1 The Interface class

The class Interface declares three pure virtual functions that are used by inherited class to display messages for users :

- `virtual void error(const string) const = 0;`
- `virtual void warning(const string) const = 0;`
- `virtual void information(const string) const = 0;`

These functions are defined inside the Console User Interface (*Interface_console* class) and the Graphical User Interface (*Interface_graphic* class) which both inherit of the *Interface* interface.

In the first case, message is displayed in the Operating System terminal. In the second class, Qt text feature is used to display message inside a window: *Info.window*.

Nevertheless, *ViTE* modules (like the Parser, the Data Structure, the Render area, etc.) need to:

- Have an easy way to display message.
- Does not care if message should be displayed in the System terminal or in a graphical window.

Thus, to respect these two requirements, the *Message* class was created.

4.2 The *ViTE* notify message system : the *Message* class

The *Message* class was developed with the following leitmotiv: provides an easy-to-use way to display message for user anywhere inside the *ViTE* source code.

For all the *ViTE* parts, just a one *Message* class instance is needed. Thus, this class is a singleton. To pass many different argument types (such as *int*, *string*, *bool*), the *stringstream* class is used.

To display message just recover an instance (the only instance) and used it like

`cout` or `cerr` of the *STD* library. Nevertheless, you need to finish your message to pass a special object instance: it will be `Message::endi` (for informative message), `Message::endw` (for warning message) or `Message::ende` (for error message).

Message processing is different depending of the special object. For example, it influences the message color (orange for warning, red for error) in the graphical interface or add `Warning` or `ERROR` at the beginning of the message for the console interface.

Moreover, it exists an other special object: `Message::endsi` (for selection informative message). It is used to display information about the mouse selected entity in the graphical interface only.

NOTE: in graphical interface, messages are displayed in the *Info window*.

Example 1: *You need to notify the user that an error occurred.*

```
*Message::get_instance() << "An error occurred." << Message::ende;
```

Figure 4.1: Message render: **Left:** in the System terminal - **Right:** in the *Info window*.

Example 2: *You need to notify the user that the trace is misformed. Use a warning message.*

```
*Message::get_instance() << "Incorrect value 1." << line << " of \
file: " << file << "." << Message::endw;
```

Take advice that you can define macro to use as debug :

```
#define message *Message::get_instance() << "(" << __FILE__ << " 1." \
<< __LINE__ << "): "
```

Thus, just use `message` instead of `*Message::get_instance()` in the source code.

4.3 Console User Interface : class `Interface_console`

4.3.1 Description

The *Interface_console* class is the most critical class of the software. It is the *ViTE* core since it manages and connect all the application modules. For example, it analyses the command line, launches window interface, creates a parser and triggers the data structure filling. Moreover, it is used to broadcast message between the render area (in OpenGL) and the graphical interface (in Qt). Moreover, it triggers the SVG export of an opened file.

ViTE module connections can be seen as a star where the *Interface_console* class is set on the middle. This organisation was designed to provide an easy and safe way to manage threads.

4.3.2 How it works

First, the *main* function of *ViTE* creates a console interface and gives it to the command line parameters.

Then the console interface analyses the parameters from its *constructor*. It gives them the `get_state()` function, which returns a code number of the right action

to be executed. This latter is given the `launch_action()` function which executes the action corresponding to the code number.

For example: if user calls *ViTE* with the command line:

```
vite -h
```

`get_state()` returns `STATE_DISPLAY_HELP` code number. Then from this latter, `launch_action()` will be called and will apply `display_help()` function that displays in the terminal the *ViTE* synopsis.

4.3.3 Managing of the arguments

We saw, in the last section, an example of using *ViTE* by command line to launch the help.

The arguments are managed by a loop in the `get_state()` function and the action to do is set in the integer `state`. The loop can be considered like an automaton which recognize the argument passed in parameter.

We have two kind of arguments : the ones beginning with a "-" and the others. The ones beginning with a "-" are the options of *ViTE* and are predefined in the `get_state()` function. The others are the arguments of these options or, by default, the file to open.

When the option is known, `state` value will be changed depending on the option and the option consumes the next argument it will may need. If the option is unknown, the help is printed.

For example, to export a file in svg you will tape : `./bin/vite -f in -e out` . The loop starts by reading the `-f` option. This option consumes the next token which corresponds to the file to open (`in` here). Then, because of the consumption of the argument `in`, the loop will read the `-e` option and set the next argument as the name of the destination file.

The loop makes a comparison between the argument and all the pre-defined options.

So, it is easy to add options : First of all, depending on the option aim, you will need to add a new state in the header (for example `_STATE_OPEN_FILE` is used to set the state for opening a file). Then, in the `get_state()` function, add the case you want and what you want to do with it. If you want to change a parameter of another class, the easier is to do a static method in this class and call it in the case (it is what has be done to add the epsilon option for the export in svg and this method do not need a new state). Do not forget to consume the next parameters if necessary (check before if they exist). Maybe, you will need to need to do another thing and to set a new state, and moreover you will need to return it without preoccupying of the other ones. The action to set then has to be put in the `launch_action()` function. Do not forget to add the option in the help and it is ok.

Some states are compatible, like for example the opening or exporting a file in an interval which are two different arguments. The choice made is to use binary operands for them and to do binary OR when we have the "-t" argument. To help to the comprehension, two states were added to specify that we are in an interval.

4.3.4 Graphical User Interface : class `Interface_graphic`

Graphical window interface is created by *Qt designer* in a `.ui` file (`main_window.ui`). The interface graphic loads it and displays the window generated from the file.

In a same way, the informative window is loaded from `info_window.ui` file.

In the `_render_area_layout`, a `Render_area` object is created. It displays an OpenGL scene.

Chapter 5

The render system

The render system is the most important *ViTE* module since it provides graphical output of the trace files. Currently, there are two ways to render a trace file: the *direct render* or the *in-file save render*.

The *Render* class defines several *passives* displaying methods. What is a *passive* displaying method? It is a method which just performs a displaying action (calling *OpenGL* functions or in-file writing function for *SVG* output) according to its parameters.

Thus, these methods do not call any other functions to browse, for example, the entity tree.

5.1 The render opengl

5.1.1 The drawing methods

Render_opengl uses the `QGLWidget` to display graphics using the OpenGL API. With three Qt functions, the scene is initialized (`initializeGL()`), resized when the main window is resized (`resizeGL()`) and displays the trace (`paintGL()`). Two functions are used to display a trace :

- `build()` - > creates the display of a trace loaded in the data structure.
- `unbuild()` - > releases the scene.

The `build()` function `build()` is called when a new trace needs to be drawn. It releases the waiting screen (currently a rabbit turning) and the timer associated with the waiting screen (for animation).

The `unbuild()` function This function is the complementary of the `build()` function : it releases the trace design and displays the waiting screen (and also restarts the timer for the rabbit animation).

The drawing function To enhance the *ViTE* performance the drawing functions - called by the Data Structure while it browses the trace tree - are inlined functions and are defined in *render_opengl.hpp*. There must be an order to respect the OpenGL Display List mechanism. So, each display list should be opened, filled then closed.

Currently, there are three OpenGL Display Lists used for the render area: the container, state and counter Display Lists.

Nevertheless, arrows and events do not belong to a Display List since they must be drawn according to the current scale. (Else, the triangle of the arrow will be scaled and mask the states under it!)

Following, the list of calls is given and must be respected for each kind of objects else OpenGL errors should be lifted. Thus, for *container* (and also *container text*), *state*, *arrow* and *counter*, the drawing function (i.e. *draw_**()) must be called between *start_draw_**() and *end_draw_**():

1. *start_draw*()
2.
 - (a) *start_draw_containers*()
 - (b) *draw_container*()
 - (c) *draw_container_text*()
 - (d) *end_draw_containers*()
3.
 - (a) *start_draw_states*()
 - (b) *draw_state*()
 - (c) *end_draw_states*()
4.
 - (a) *start_draw_arrows*()
 - (b) *draw_arrow*()
 - (c) *end_draw_arrows*()
5. *draw_event*()
6.
 - (a) *start_draw_counter*()
 - (b) *draw_counter*()
 - (c) *end_draw_counter*()
7. *end_draw*()

5.1.2 The scrolling and scaling methods

One of the most important feature of *ViTE* is to allow user to freely move in the trace render. They are two primitives available to move in the trace: the scrolling and the scaling.

NOTE: the origin of render area is at the top-left of the *QGLWidget*.

NOTE: please refer to the *ViTE* API documentation to know each following attribute meanings.

- *Scrolling* allows the user to move the image to the left, the right, the top or the bottom in the render area.
- *Scaling* allows user to resize the image, for example to perform a zoom in particular point.

Thus the user can visualize any part of an image trace just with his keyboard and his mouse. Nevertheless, *ViTE* needs to know what kind of action (push on a touch, move the mouse, etc.) the user does. The information is given by *Qt*. Since the *render_opengl* is a *Qt* object (a *QGLWidget*), it receives some of these events (like mouse movement over the render area or mouse click). The others (some keyboard event) are caught by the graphical interface since they are bound as a shortcut then send to the render area with the *Qt* signal and slot mechanism.

Now, we just present the different attributes and methods used to perform scrolling and scaling.

All of them have the same type: *Element_pos*.

- – `_screen_width`: the *Qt QGLWidget* width (in pixel).
- `_screen_height`: the *Qt QGLWidget* height (in pixel).
- – `_render_width`: the render area width (in OpenGL unit).
- `_render_height`: the render area height (in OpenGL unit).

Both of the `_screen_*` and `_render_*` represent for the user the same area in his monitor but with a different point of view.

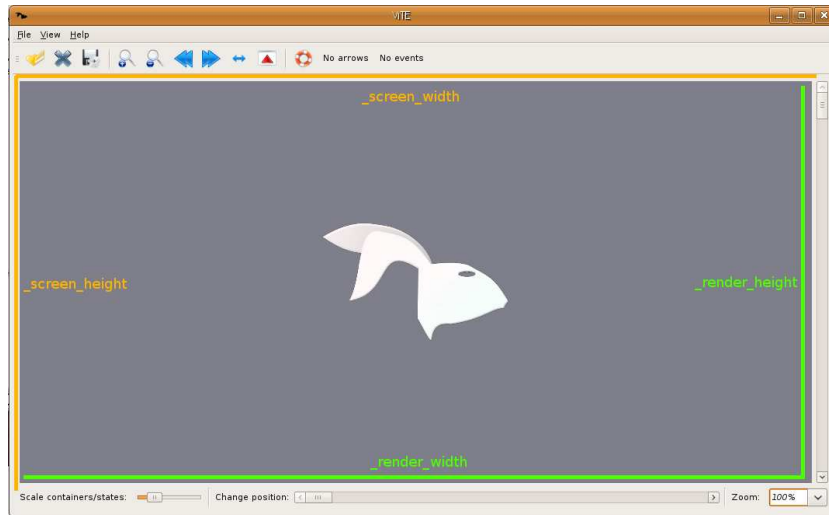


Figure 5.1: The render area dimensions. The both measures (green and orange) are the same but in different units.

To convert screen units to render units, you have to use `screen_to_render_x()` and `screen_to_render_y()` (please refer to the *ViTE* API documentation for a whole synopsis). To pass from render to screen, user `render_to_screen_x()` and `render_to_screen_y()`.

Moreover, we need to get the trace coordinates to know, for example, from which entity the user has just clicked. So, there are two methods to convert from render units to trace units: `render_to_trace_x()` and `render_to_trace_y()` and also `trace_to_render_x()` and `trace_to_render_y()`.

Another important coordinates are the trace coordinates. There are several attributes to defines a render :

NOTE: all of the following coordinates are in the trace unit. (and not in OpenGL unit)

- – **Container coordinates**
 - `_container_x_min`
 - `_container_x_max`
 - `_container_y_min`
 - `_container_y_max`
- – **State coordinates**
 - `_state_x_min`
 - `_state_x_max`

- `_state_y_min`
- `_state_y_max`

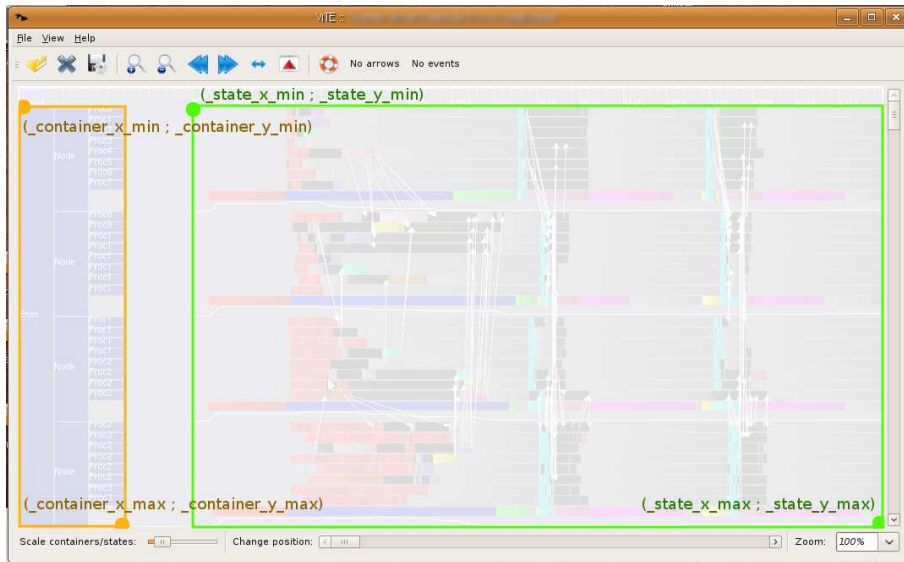


Figure 5.2: Container and state dimensions in a trace. (in **trace** unit)

Finally, the trace entity positions : **NOTE: all of the following coordinates are in the OpenGL unit.**

- `_default_entity_x_translate`
- `_ruler_y`
- `_ruler_height`
- `_x_scale_container_state`

5.2 Render out Svg file

Introduction

As `render_opengl`, `render_svg` must implements the drawing functions. It consists in outputting data in a `svg` file. Data to be printed in the file, are buffered in differents `std::ostringstream` buffer. Main elements are stored in `_buffer`, flushed regularly by `print()`. In a `svg` viewer, the last element in the file are displayed in the front : the thinnest element must be printed as latest as possible : these are stored in `_thin_element_buffer`. Counter are stored in `_chronogramme`, and the scale is stored in `_time_scale`.

Required settings

Render in `svg` may produce a high weight file due to numbered elements and default scale setting may hide some relevant events. That is why `render_svg` allows user to set the width, heigth and accuracy of the render. These are set by default with the following static functions :

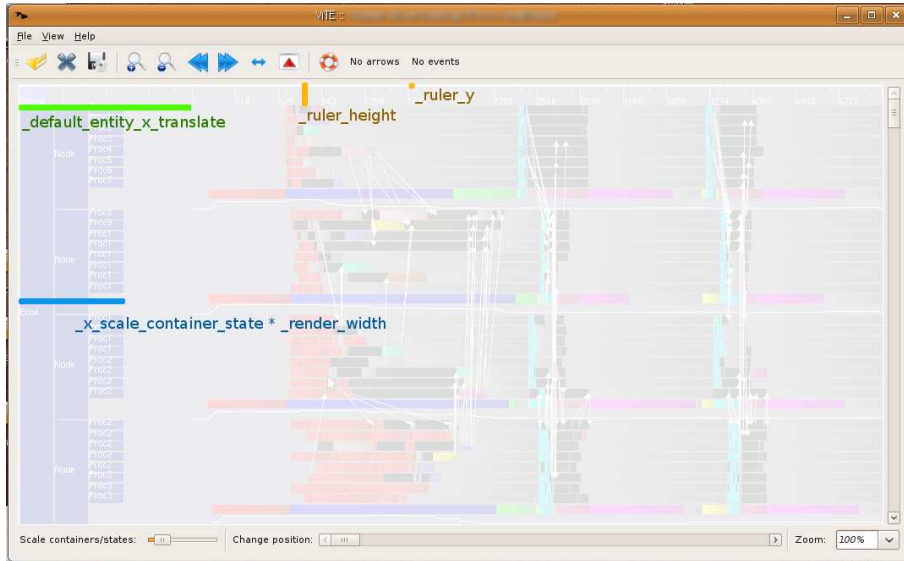


Figure 5.3: Container and state positions in a trace. (in **OpenGL** unit)

- `Svg::set_height_factor(double)`
- `Svg::set_wide_factor(double)`
- `Svg::set_interval(double, double)`
- `Svg::set_accuracy(double)`
- `Svg::set_scale_frequency(double)`.

Initialisation

`render_svg` opens the file specified by `Svg::init(const char *path)` and output the header. `Css` embedded stylesheet reduces style attributes in each elements.

drawing

The procedure to implement functions such as `draw_arrow`, `draw_container` or `draw_state` is the same.

`render_svg` controls that the elements to be displayed are in the appropriate range, set by `Svg::set_interval(double, double)`. In this release, the render reshapes itself too tall elements, but in the next version of Vite, the reshaping can be replaced by warnings (a correct `draw_trace` function doesn't need this control).

Then, `render_svg` computes positions and colors according to parameters. Any elements are widen by `_wide_factor` to convert time in ms to pixel, heights are grown by `_height_factor` for the same reason.

Positions are translated to `_container_width_max` : the left region occupied by containers and `MARGINTOP` the upper region reserved for scale. Color are converted from floats to RGB.

Whenever a time selection is set, are translated to (-) `_start_interval.time` (in ms, ie `-_start_interval.time*_wide_factor` pixel).

The computed position allows the call to basic shape painting functions such as

```
Svg::rectangle(const char* name, Element_pos w, Element_pos h,
              Element_pos x1, Element_pos y1, unsigned int r,
```



```
unsigned int g, unsigned int b)
```

which corresponds to write in the svg output :

```
<rect class='name' title='name' width='w' height='h'  
x='x1' y='y1' fill='rgb(r,g,b)'/>
```

drawing counter

Drawing counter is an automaton. See the schema for details. Svg paths are used to display counter.

drawing scale

Each element position corresponds to a time. The latest element indicates the total time. The total time multiplies by width factor equals the total width. The scale consists in drawing a time value regularly.

ending function

`Svg::end()` flushes left buffered informations, writes the scale, the cursor and ends the svg file. The cursor is only available on javascript enabled svg viewers (e.g. firefox).

Chapter 6

Some possible ameliorations

This section is for the maintenance of *ViTE* and describes some features that could be added to it.

Of course, there could be some features not written here that could be implemented. Moreover, some of these features are already in *Pajé*.

6.1 Using filters

Unlike *Pajé*, there is no filters in *ViTE*.

We could add some like :

- filtering on the printed data : in order to improve the legibility of the trace, we could hide some events, nodes, links ... printed in the trace.
- filtering on the trace during parsing to do not keep some events too little or some entities (links for example) to improve the time spent to parse and show only what is useful.
- filtering by container to only show the ones we want...

6.2 Export in another image file format

ViTE can export a trace in a svg format.

The advantage of this kind of format (vectorial) is that the quality of the trace is perfect because we can zoom in or out with the precision we want.

The problem is that the output file can be more than three times bigger than the original file (for an export of the whole trace).

So, it can be difficult to open it after. The export in a non-vectorial format can reduce the output size and make easier to open with only a slighty difference of quality.

6.3 A better entry format?

Traces generated are text files and they follow the *Pajé* trace format. This format has a lot of qualities like genericity. But, these files can be very big and that can enhance the time to parse them.

A format less constraining which takes less places could be created to replace the *Pajé*'s one, for example, a binary format in order to improve the interaction between the user and the software.

6.4 Allocation by lot

For the moment all the different events are stored in memory one by one.

It could be very slow for very big traces. The idea, in order to do a new object for each event, is to do a tabular which could store a group of event.

The aim is to reduce the number of allocations which cost a lot.

6.5 Reading at the flight

For a better interactivity with *ViTE*, you would like to be able to see the progress of a loading file or watch the about menu. For now, this is not possible because *ViTE* is not multi-threaded and it parses the file and then shows it.

6.6 Saves of configurations

Maybe, for a trace that you will use a lot of time, you would like to always have the same filters or anything else preloaded without setting them when you load the trace. In order to do it, a configuration file by trace could be the solution and be loaded while the trace is loaded automatically.

6.7 Interface enhancement

The interface will have to have a menu built such as a tree, listing all the containers, and enabling by selecting one of them, to display only its children.