SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Materials Science and Technology in Trnava

Reg. No.: MTF-5262-6440

Window Manager for the X Window System

Bachelor thesis

2013 **Tomáš Mészaroš**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Materials Science and Technology in Trnava

Reg. No.: MTF-5262-6440

Window Manager for the X Window System

Bachelor thesis

Study programme: Applied Informatics and Automation in Industry
Study field number: 2621
Study field: 5.2.14 Automation, 9.2.9 Applied Informatics
Training workplace: Institute of Applied Informatics, Automation and Mathematics
Thesis supervisor: Ing. Michal Sroka

Trnava 2013 **Tomáš Mészaroš**

Slovak University of Technology in Bratislava
Institute of Applied Informatics, Automation and Mathematics

Faculty of Materials Science and Technology in Trnava
Academic year: 2012/2013
Reg. No.: MTF-5262-6440

STU
MTF

BACHELOR THESIS TOPIC

Student:	**Tomáš Mészaroš**
Student's ID:	6440
Study programme:	Applied Informatics and Automation in Industry
Study branches combination:	5.2.14 Automation, 9.2.9 Applied informatics
Thesis supervisor:	Ing. Michal Sroka
Workplace:	UIAM MTF STU v Trnave

Topic: **Window Manager for the X Window System**

Specification of Assignment:

1. Analyze principles of X Window System application design.
2. Design and implement simple window manager for the X Window System.
3. Design and implement multi-screen emulation functionality for implemented window manager.
4. Provide documentation for designed system.

Assignment procedure from:	22. 02. 2013
Date of thesis submission:	31. 05. 2013

L. S.

Tomáš Mészaroš
Student

doc. Ing. Pavol Tanuška, PhD.
Head of department

doc. Ing. Peter Schreiber, CSc.
Study programme supervisor

Acknowledgement

I owe my deepest gratitude to my supervisor, Ing. Michal Sroka, whose guidance and support from the initial to the final stages enabled me to develop comprehension of the subject and write this thesis.

I am also deeply indebted to PhDr. Emília Mironovová for language consultancy.

Lastly, my best regards to all who supported me up to the completion of my project.

Author

Abstract

Mészaroš, Tomáš: *Window Manager for the X Window System.* [Bachelor Thesis] - Slovak University of Technology in Bratislava. Faculty of Materials Science and Technology in Trnava: Institute of Applied Informatics, Automation and Mathematics. - Thesis supervisor: Ing. Michal Sroka. - Trnava: MTF STU, 2013. 49 pp.

Key words: Window Manager, X Window System, Xlib, Automated testing

The present thesis deals with the window manager for the X Window System. Characteristics of application design and programming practices under the X Window System are demonstrated. One of the key objectives of the proposed thesis is to design and implement simple lightweight dynamic window manager. Both design and implementation of the submitted window manager are intended to meet the most common functional requirements of any power user. The specific functionality, multi-screen emulation using only one physical screen, is explained, designed and also implemented. Subsequently, methods and approaches used in the process of designing and executing automated tests are described.

Súhrn

Mészaroš, Tomáš: *Správca okien pre X Window System.* [Bakalárska práca] - Slovenská technická univerzita v Bratislave. Materiálovotechnologická fakulta so sídlom v Trnave: Ústav aplikovanej informatiky, automatizácie a matematiky. - Vedúci záverečnej práce: Ing. Michal Sroka. - Trnava: MTF STU, 2013. 49 s.

Kľúčové slová: správca okien, X Window System, Xlib, automatizované testovanie

Táto práca sa zaoberá správcom okien pre X Window System. Práca demonštruje špecifiká návrhu a vývoj aplikácie priamo pod X Window System. Jedným z hlavných cieľov tejto práce je navrhnúť a implementovať jednoduchého dynamického správcu okien. Návrh a implementácia spĺňajú základné požiadavky kladené na správcu okien pokročilým užívateľom. Špecifická funkcionalita (emulácia viac obrazoviek pomocou jednej fyzickej obrazovky) je vysvetlená, navrhnutá a následne implementovaná. Taktiež sú vysvetlené metódy a postupy použité v procese návrhu a vykonávania automatizovaných testov.

Contents

List of Tables

List of Figures

List of Code Listings

Introduction

"Window manager (WM) is a program to control the layout of windows on the screen, responding to user requests to move, resize, raise, lower, or iconify windows. The WM may also enforce a policy for window layout (such as mandating a standard window or icon size and placement) and provide a menu for commonly used shell commands." (Nye 1994)

In the proposed thesis, I explain the most important principles of the X Window System. Within this windowing system, the WM will be designed, implemented and tested.

Central concepts of the X Window System are described (along with the evolution of this particular windowing system[1]) in the section 1.1.

The most common used window management paradigms currently used in the GNU/Linux system are explained in the section 1.2.

Moreover, my motivation of using UNIX-like operating system (OS) along with the X Window System is described further in the sections 1.2 and 3.1.

Desired practical output of this thesis is a functional WM. Therefore, section 2.2 summarizes requirements specification for the thesis output, while section 2.1 is defining the key objectives.

I devote section 2.3 to the WM design. The whole section 2.3.2 deals with the key functionality of the output WM, which is the multi-screen emulation.

To ensure reliability and usability of the implemented WM, additional software responsible for the testing had to be designed and implemented. Section 2.4 describes my procedures and methods used in the automated testing.

Implementation process, methods and approaches used in the process of development are explained in the section 3.1. Subsequently development tools used during the stage

[1] The very first version of the X Window System was presented in the 1984 by Robert W. Scheifler. (Scheifler 1984)

of writing the source code are covered in the section 3.2.

The WM plays a very specific and essential role in the user experience[2] (UX) of every computer user. What is more, every group of computer users demands specific functionality and appearance, so that it is very difficult for developers to implement such a window manager, which will meet all users requirements.

Therefore, the area my of interest is to design and implement the WM targeted to the specific group of users, called power users[3]. The reasoning why this group of users is a target one is explained in the sections 1.2 and 2.3.

[2]User Experience (UX or UE) is defined in ISO 9241-210:2009 as *"a person's perceptions and responses that result from the use or anticipated use of a product, system or service". (ISO 2009)*

[3]According to the Dictionary.com: *"Power user is any person who knows enough about a computer or other device to take full advantage of its advanced features.". Dictionary.com (2013)*

Chapter 1

Window Manager

Window manager is major and very important component within a system's graphical user interface (GUI). Specific and most needed functionality for the user is mostly provided by the WM and other components, closely related to the WM or the particular windowing system. Hence, it is often the case, that window managers are *built just to meet specific purpose* and therefore there are many types of window managers.

1.1 X Window System

The WM is software within a particular windowing system. The windowing system I work with in the proposed thesis is The X Window System (commonly known as X). X is a computer software system and network protocol, which provides the base technology for developing graphical user interfaces[1].

"At a very basic level, X draws the elements of GUI on the user's screen and builds in methods for sending user interactions back to the application." (The Open Group 2013)

Later in the section 1.1.4 of this thesis, X.Org Server, an open source implementation of the X, which is currently the most widely used implementation of the X Window System, is introduced.

[1] According to the LinuxCounter.net (2013), there are approximately over 60 million worldwide Linux users. Currently, the wast majority of Linux users is using the X Window System in order to run any desktop environment.

1.1.1 Evolution

In the 1984 Robert W. Scheifler announced to the MIT Project Athena community, that he was *"writing a window system for the VS100"*. (Scheifler 1984)

Robert Scheifler led the development of X (which was developed jointly by MIT's Project Athena and Digital Equipment Corporation) also with contributors from many other companies. (Nye 1994)

The first snapshot of the Version 11 of the X Window System (commonly referred to as X11)[2] was released in September 1987 by the Massachusetts Institute of Technology. With X11 Release 2, control of X passed from MIT to the X Consortium. (Nye 1994)

X11 is a complete window programming package which offers flexibility in many areas, e.g. display features, window manager styles, multiple screens and what is more, it is also fully extensible[3]. (Nye 1994)

1.1.2 Concepts

Because of the complexity of the X Window System[4], this subsection is intended to describe some major concepts of the X.

Displays and Screens

X is a windowing system for bitmapped graphics displays. Color, monochrome and gray-scale displays are supported.

Display in the X is defined as a workstation consisting of a keyboard, a pointing devices (such as a mouse), and one or more screens. (Nye 1994)

Multiple screens can be accessed via mouse pointer simply by moving the pointer across physical screen boundaries.

Client-Server Model

Client-Server is a central model in the X. Because X is a network-oriented system, an application does not necessary need to be running on the same system that supports the display.

It is possible to execute applications on other machines, sending requests via the network to a targeted display and receiving input events from the system controlling

[2] X11 (still in active development) is one of the oldest and most important open-source projects.

[3] xfakeevent (program developed as a part of this thesis for purpose of automated testing) is using X extension Xtst, which provides a set of test functions for the X, see section 2.4.

[4] Release X11R7.7 of xorg xserver contains more than 350000 lines of source code.

the display. (Nye 1994)

The X server running on the local or remote systems acts as an intermediary between user programs (called clients or applications). The following tasks are performed by the server (without extensions):

- Allowing access to the display by multiple clients.
- Interpreting network messages from clients.
- Passing user input to the clients by sending network messages.
- Drawing graphics are performed by the display server rather than by the client.
- Maintaining complex data structures, including windows, cursors, fonts, and "graphics contexts," as resources that can be shared between clients and referred to simply by resources ID's.

Window Management

Window management is another important concept in the X. Applications do not control such a things as where a client window appears on the screen or what size it is.

Client itself usually gives hints about where he would like to be displayed or how long to stay on the screen.

WM is just another client program written in Xlib (or other library providing bindings to X, such as XCB) controlling other clients.

Events

Any mouse-driven windowing system must properly respond to variety of events, and therefore events are very important in the X.

User input such as mouse click, mouse move or keypress are represented as events.

Handling events in the X programs is a major difference between programming under a window system and traditional programming. Section 2.3 further inspects this topic.

Extensions

One of the features of the X is that it is extensible. Extensions are using standard X libraries such as Xlib.

Both, client and server side of the code are part of the extension. By using extension, user must query the server to see if the extension is supported.

1.1.3 Architecture

The most important concepts of the X were described in the section 1.1.2 and thus the X Window System architecture can be explained.

According to the figure 1.1 which shows X architecture, clients communicate via server by calls to low-level library (in our case Xlib).

Necessary functions needed for connecting to a particular display server, creating windows, responding to events, drawing graphics, and so on, are provided by the Xlib (or some other library supporting bindings to the X protocol, such as XCB[5]). According to Nye (1994), Xlib calls are translated to protocol requests and sent via TCP/IP either to the local server or to another server across the network.

It is possible to only use library such as Xlib in the process of development, but sets of higher-level subroutine libraries known as *toolkits* are also available.

Toolkits allow programming certain applications much easier, by implementing set of user interface features (such as buttons, menus, and so on) referred to as toolkits *widgets*. It is also possible for programmer to create new widgets.

1.1.4 X.Org Server

The proposed thesis deals with the X Window System and its open source implementation X.Org Server which is, according to the X.Org Foundation, currently the most widely used implementation of the X Window System.

The Xorg, maintained by the X.Org Foundation and hosted by the freedesktop.org community is the most popular implementation of the X Window System among Linux users. (X.Org Foundation 2013)

Therefore, the Xorg, mainly because of its open source development model, is the most popular choice for Linux users, resulting in the massive adoption from most Linux distributions.

1.2 Window Managers in GNU/Linux

There are many different types of window managers, working around different principles and paradigms. This thesis will focus only on window managers used in

[5]According to the freedesktop.org, XCB (The X protocol C-language Binding) *"is a replacement for Xlib featuring a small footprint, latency hiding, direct access to the protocol, improved threading support, and extensibility"*.

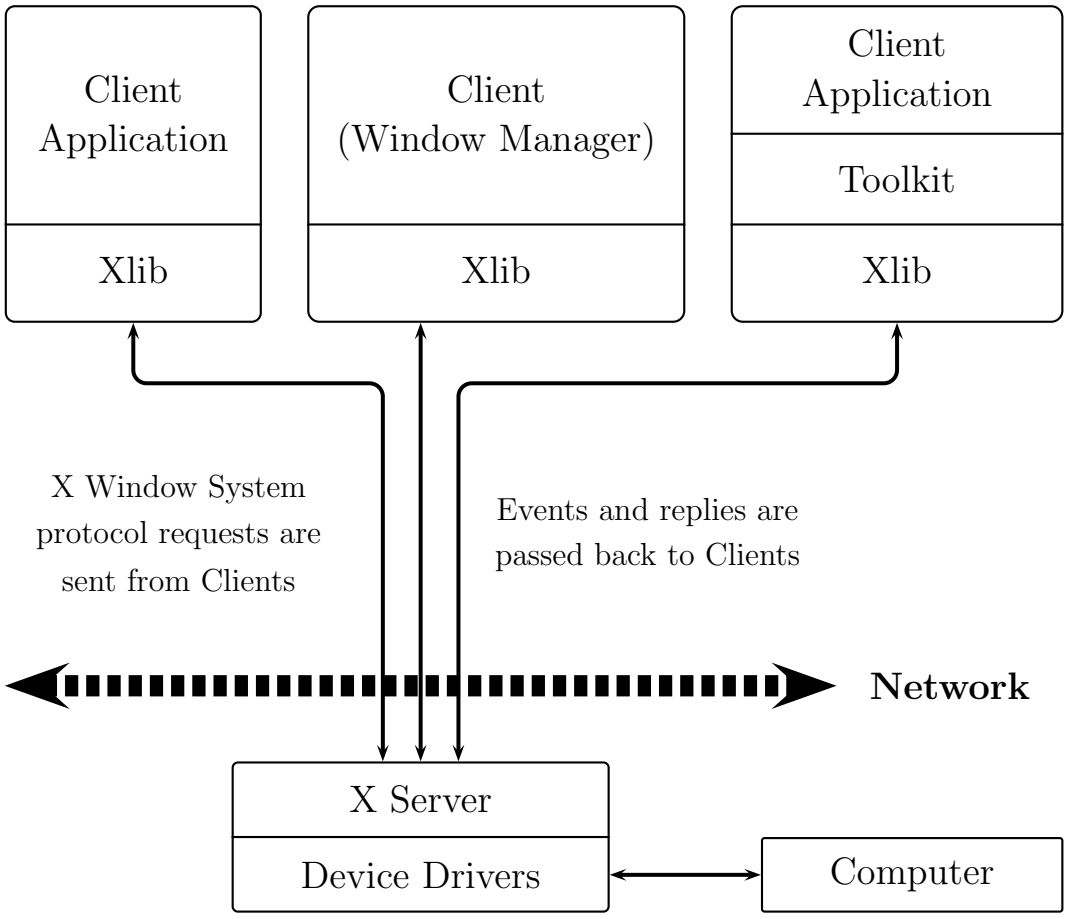

Figure 1.1: Clients communicate with the server via Xlib calls (Nye 1994)

the GNU/Linux environment, especially among power users community.

According to the Arch Linux Wiki (2013), we can divide window managers into the following categories:

1. **Stacking (also called floating) window managers**. Windows can be stacked on the top of each other (windows act like pieces of paper on a desk).
2. **Tiling window managers**. Windows are "tiled" by the window manager using specific algorithm, so they do not overlap each other and thus fill the entire screen.
3. **Dynamic window managers**. Ability to switch between floating and tiling window management layout.

It is very important to mention that power user, as a very specific computer user, has different requirements for any particular system, than ordinary computer user. It does not necessary mean, that power user is capable of programming or system administration. Nevertheless, every power user is likely to appreciate most of the, if not all, features and functionality as would programmer or system administrator.

Power users, specifically those among open-source community, are tightly bounded to

the Unix-like[6] operating system. KISS[7] is, according to Raymond (2003), the central philosophy principle.

In reference to the KISS principle as a central Unix philosophy, power users using Unix-like operating system are bounded to this central principle and what is more, they subordinate their demands on the system to this philosophy.

To be more specific, Gancarz (2003) summarized this Unix philosophy into the 9 simple points:

- Small is beautiful.
- Make each program do one thing well.
- Build a prototype as soon as possible.
- Choose portability over efficiency.
- Store data in flat text files.
- Use software leverage to your advantage.
- Use shell scripts to increase leverage and portability.
- Avoid captive user interfaces.
- Make every program a filter.

The output window manager is supposed to match the Unix philosophy by using the cleanest and simplest design possible. This approach should ensure that the final implementation would meet the software requirements specification specified later in the section 2.2.

1.2.1 Stacking Window Managers

A stacking (sometimes also called floating) window manager is a WM allowing windows (clients) overlap by drawing them in a specific order.

Stacking itself is done by the technique called painter's algorithm. (Foley et al. 1990)

Figure 1.2 illustrates situation, where client 2 is overlapping client 1. In agreement to the painter's algorithm, objects are sorted according to their depth and drawn in this order, farthest to closes.

In our situation, firstly client 2 is drawn following by the client 1. By using this algorithm, it is possible to solve the visibility problem.

[6]Sometimes referred as UN*X or *nix, is used to refer to any or all varieties of Unixoid operating systems (e.g. any distribution of Linux-based operating system). (Raymond 2013)

[7]According to Raymond (2013), KISS is an acronym for the design principle articulated by Kelly Johnson, "Keep It Simple, Stupid" or sometimes "Keep It Short and Simple".

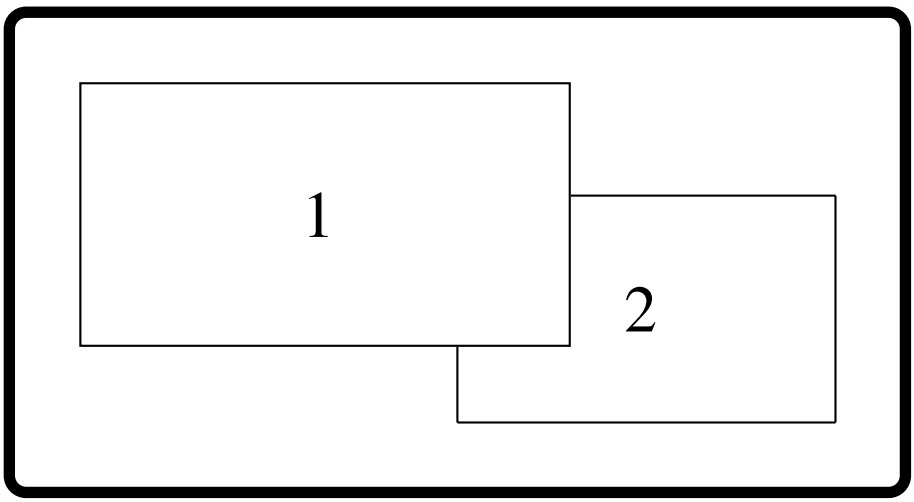

Figure 1.2: Two overlapping clients in the floating mode

1.2.2 Tiling Window Managers

A tiling window manager is a WM able to organize clients in such a way that windows do not overlap each other and therefore the entire screen can be filled.

Managing clients by using this approach ensures that there is virtually no wasted screen space, which may potentially leads to the better productivity.

The idea of the tiling comes back to the 1980s. The 8010 Information System from Xerox (commonly known as "The Xerox Star") was capable to tile application windows. (Lineback 2013)

There are many possible ways how to tile clients. Probably the most common approach is to use vertical or horizontal tiling, but it is common nowadays to see mix of these concepts (as shown in the figure 1.3).

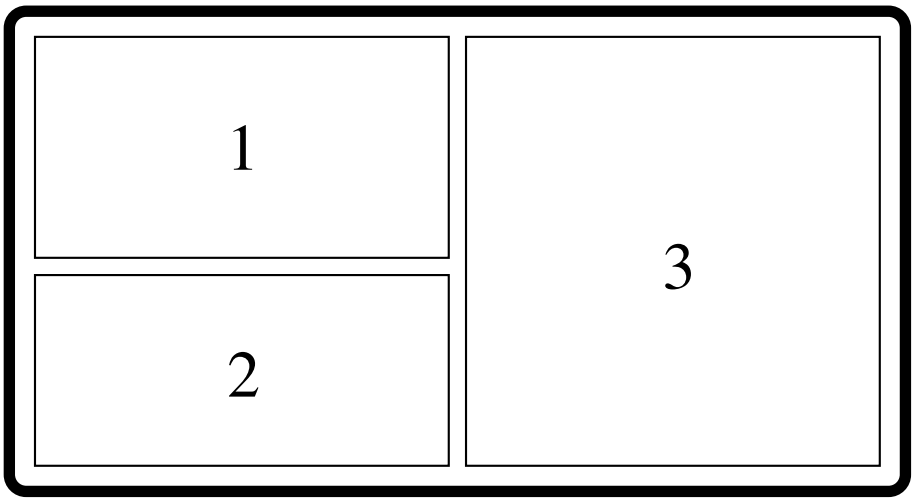

Figure 1.3: Three clients arranged on the screen according to some tiling algorithm

The figure 1.3 shows three clients tiled on the screen according to some tiling algorithm. There are plenty tiling algorithms used in window managers but because new window managers and new tiling algorithms are created relatively often, it is not possible to objectively categorize these algorithms.

1.2.3 Dynamic Window Managers

A dynamic window manager is a WM capable of managing clients by specific selected layout. Layout itself is only an abstraction for the particular algorithm responsible for managing clients position and behaviour.

Layouts are not limited just to tiling. In many cases (depends on the window manager implementation) it is possible to define layouts for tiling and floating mode.

As show in the figure 1.4, the WM is dynamically managing clients based on the current active layout.

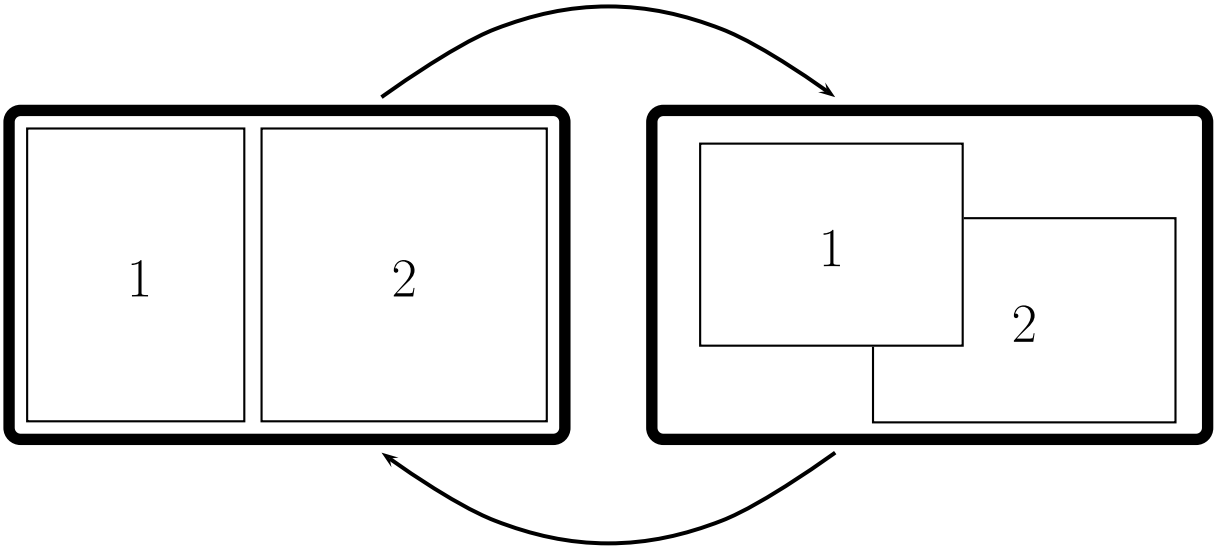

Figure 1.4: Dynamic switching from the tiling layout (mode) to floating and back

Dynamic switching is not limited only to two layouts. Depends on the WM implementation, it is usually possible to switch between multiple layouts and let the WM manage clients automatically.

Chapter 2

Objectives, Window Manager Design and the Methods Used

The following chapter deals with objectives related to the proposed thesis. These objectives come out from the thesis specification and are shortly explained in the first section (2.1) of this chapter. In the section 2.2, objectives are described more specifically.

The designed window manager is supposed to be fully functional and usable in the every day work of power users.

2.1 Key Objectives

Objectives related to the final window manager implementation can be summarized as follows:

- Analyse principles of X Window System application design.
- Design and implement simple window manager for the X Window System.
- Design and implement specific functionality for implemented window manager.
- Provide documentation for designed system.

Besides the objectives related to the window manager implementation, there are also objectives independent of the application itself

- Write user guide of the implemented window manager.
- Provide a source code documentation.

2.2 Software Requirements Specification

The window manager implementation must provide all necessary parts and functionality of the basic usable dynamic window manager and therefore meet all the required criteria.

According to analysis, the **functional requirements** can be found summarized in the Table 2.1.

Non-functional requirements:

1. Usability
    - Window manager has to be easy controllable for power users.
    - Included configuration file should be well commented and easy to change.
    - All available configuration should be done in the one configuration file.
    - Keyboard shortcuts should provide all necessary functionality to maximize work efficiency.
2. Performance
    - Implementation has to be able to perform very well and with minimal system resources.
    - Window manager should run very well on older hardware.
3. Reliability
    - Window manager should handle high system load without any significant problems.
    - Implementation should be well tested (using automatic tests).
4. Source code maintainability
    - Source code has to be licensed under the FOSS[1] license.
    - Source code and documentation should be easy to understand and accessible for future maintenance and potential collaborators.

[1]Free and open source software is software that is both free software and open source.

Table 2.1: Functional requirements specification

ID	Functional Requirement	Additional information	Priority
R01	Tiling mode	Ability to tile clients according to the specified algorithm	must
R02	Floating mode	Allow windows overlapping	must
R03	Mode/Layout switching	The WM implementation must allow to switch between tiling and floating mode	must
R04	Virtual desktops	The WM must implement virtual desktops	must
R05	Emulate multi-screen behavior on one physical screen	Provide independent virtual desktops within parent virtual desktop, so it will be possible to emulate classical multi-screen workspace behavior only with one physical screen	must
R06	Several tiling algorithms	Provide multiple tiling algorithms in the final implementation	must
R07	Configurable	Easy configurable through the configuration file	must
R08	Multi-monitor support	Possibility to support multi-monitor workspace	can
R09	Dynamic resolution adaptation	Ability to dynamically adapt on the screen resolution change	can
R10	System status bar	Implementation/support of the status bar showing system information	can
R11	Systray	The systray implementation (or support)	can
R12	External bar support	The support for the external status bar	can

2.3 Design of Window Manager

Window manager, as described in the chapter 1, needs more concrete description before it can be implemented.

There are two core parts of the proposed window manager implementation:

- Event Hadling and Processing
- Multi-Screen Emulation

The section 2.3.1 is devoted to the event handling and event processing, while the section 2.3.2 deals with the multi-screen emulation.

It is important to mention that the final WM implementation will be designed with respect to power users needs and requests.

There are several key properties which might be recognized by the power user community as an advantage or a disadvantage:

Disadvantages of current window managers

- As window manager implements more functionality, it might become more prone to errors and bugs.
- *"Bigger"*[2] window managers tend to perform less well in comparison to smaller, more *"lightweight"*[3] window managers.
- Inability to emulate multi-monitor behavior with one physical screen.

Advantages of current window managers

- Hundreds of different window managers with different functionality.
- Ability to tailor window manager's functionality to specific needs of the user.
- *"Lightweight"* window managers are usually very fast and stable.
- Usually easy to extend WM functionality (implement a module or write a patch).

2.3.1 Event Handling & Event Processing

Event handling and event processing is a key part of the WM and therefore it must be properly designed. The following paragraphs explain the basics of the events structure and behavior.

Approaches used in the process of designing structures representing event handling and event processing are also covered.

Citing Nye (1994): *"An event, to quote the Oxford English Dictionary, is an "incident*

[2]bigger = implementation consists of many lines of code or many included software libraries
[3]lightweight = implementation is rather simple, short and easy to understand

of importance" or a "consequence, result, or outcome." This definition holds for X. An event reports some device activity or is generated as a side effect of an Xlib routine."

According to Nye (1994), an *event reports* can be described from the programmer's point of view as follows:

- Something that your program needs to know about, such as user input or information available from other clients.
- Something your program is doing that other clients need to know about, such as making text available for pasting to another client.
- Something the window manager needs to know, such as a request by your program for a change to the layout of the screen by mapping a window.

"There are three important steps in a program's handling of events. First, the program selects the events it wants for each window. Then it maps the windows. Finally, it provides an event loop which reads events from the event queue as they occur." (Nye 1994)

The Events Structure

The code listing 2.1 represents the simplest event structure[4]. Using this structure, it is possible to implement event as a packet of information.

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
} XAnyEvent;
```

Code Listing 2.1: XAnyEvent: The simplest event structure

There are over 30 different types of event structures and all of them share the following attributes:

- `type`: indicates the type of an event.
- `serial`: identifies the last protocol request processed by the server.
- `send_event`: indicates whether the event was sent from the server (False) or from the another client (True).
- `display`: identifies the connection to the server that the event came from.

[4]Actual implementation can be found in X11/Xlib.h.

- `window:` indicates the window that selected and received the event.

(Nye 1994)

All the event structures are contained within the `XEvent` union (Appendix A, code listing A.1). The `XEvent` union and each event within the `XEvent` union begins with the event `type`. By looking at the `type` member of the `XEvent`, a client determines the character of the event, which ensures the proper branching to specific code for appropriate event type. (Nye 1994)

The Event Loop

In order to get program act accordingly, it has to be able to read events from the event queue.

As shown in the figure 2.1, there are two queues: a server queue and a Xlib queue.

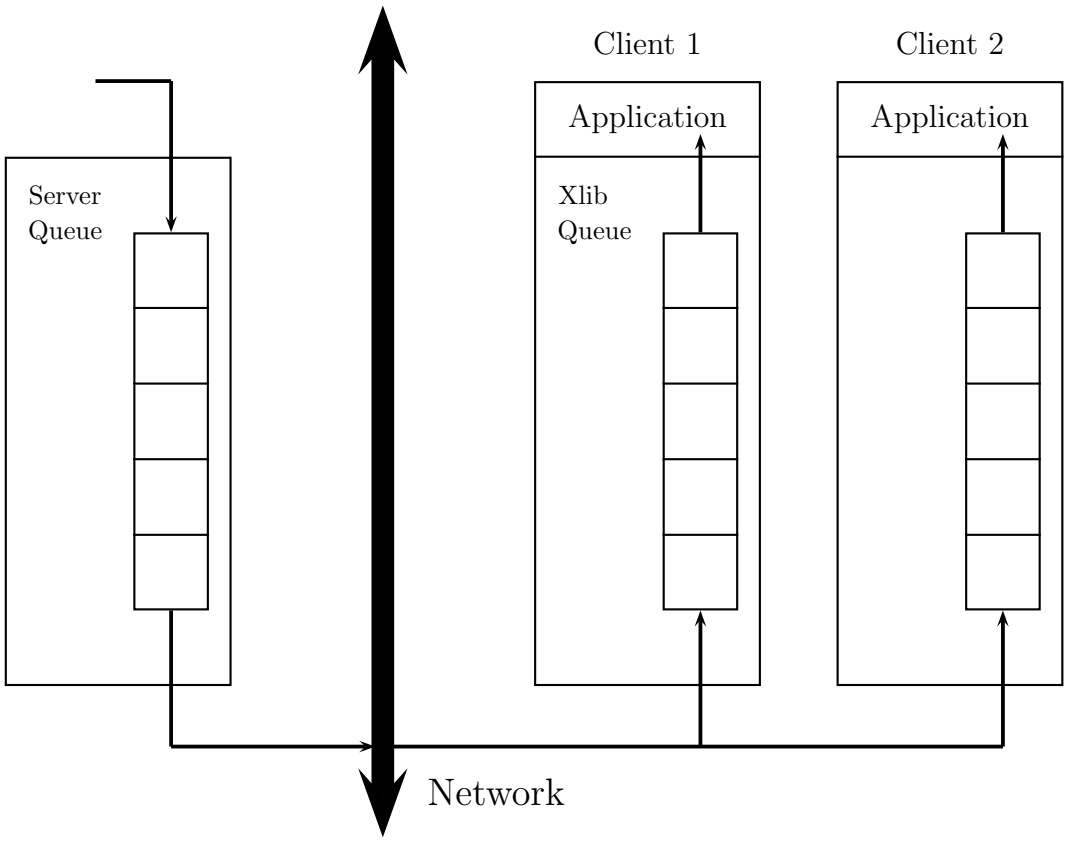

Figure 2.1: The server's event queue and client's event queue (Nye 1994)

The event structures are firstly placed on the server queue maintained by the server. Afterwards, events in the server's queue are transferred (periodically) over the network and placed on the Xlib queues.

If multiple clients select to receive same events, it is possible for them to get copies of the same events. Events that arrive on the client's event queue can be handled and

further processed by the event-receiving loop. (Nye 1994)

```
while (running && !XNextEvent(dpy, &e))
        if (events[e.type])
                events[e.type](&e);
```

Code Listing 2.2: Event Loop

The most important structure involved in the process of handling events is an *event loop*. Such a loop is shown in the code listing 2.2. The design of the event loop is up to the programmer, but there are certainly some constructs that are necessary.

One of the core constructs is the `XNextEvent` library routine. According to Nye (1994), *"The XNextEvent routine gets the next event on the queue for our client or waits until one appears before returning."*. The `XNextEvent` is wrapped within loop, so it is possible to continuously handle and process incoming events.

Events are handled one by one, while each event's type is checked for further possible processing. This is done by the simple check. If there is a stored function pointer corresponding to the type of currently processed event, program calls the desired function with the particular event structure as its argument.

As shown in the code listing 2.3, the function pointers are stored in the array of function pointers named `events`. By this approach, it is easy to extend functionality, so the WM can process more different events.

```
void (*events[LASTEvent]) (XEvent *e) = {
        [ButtonPress] = buttonpress,
        [ConfigureNotify] = configurenotify,
        [ConfigureRequest] = configurerequest,
        [DestroyNotify] = destroynotify,
        [EnterNotify] = enternotify,
        [Expose] = expose,
        [KeyPress] = keypress,
        [MapRequest] = maprequest
};
```

Code Listing 2.3: Array of function pointers

The Event Processing

When the appropriate event processing function has been called, the flow of the program is passed from the event loop and `events` array to this particular

event processing function.

Suppose that the `KeyPress` event has been triggered. The appropriate function (in our case `keypress`) is called within the event loop. As every event processing function, `keypress` further process triggered event using some code (code listing 2.4).

```
for (i = 0; i < LENGTH(keys); i++) {
        if (keysym == keys[i].keysym
         && keys[i].mod == ke->state
         && keys[i].func)
                keys[i].func(&(keys[i].arg));
}
```

Code Listing 2.4: keypress

Firstly, `keypress` takes this triggered event structure as its argument. After that, event structure is examined and its core attributes (key type, key modifier and related function) are checked against the `keys` array structure stored in the `config.h`. Because `config.h` is under the jurisdiction of the user, so it is possible for the user to easily change or add content to the `keys`.

The snipped of the `keys` array containing an example member is showed in the code listing 2.5.

```
static Key keys[] = {
        ...
        /* modifier       key      function    argument */
        {  MOD1|SHIFT,    XK_q,    quit,       { 0 }},
        ...
}
```

Code Listing 2.5: keys

Exactly the same principle applies to the `buttonpress`. This routine process the `ButtonPress` event using the `buttons` array structure stored in the `config.h` configuration file.

Snippet of the `buttonpress` and `buttons` can be found in the appendix B.

2.3.2 Multi-Screen Emulation

As stated in the key objectives (section 2.1), specific functionality has to be designed and later also implemented. Multi-screen emulation is the central concept across the

proposed WM. All structures used in the design have to be compatible with this specific functionality.

The idea of multi-screen emulation comes from the fact, that the high resolution wide screen monitors are easily accessible for most of the computer users. The fact that the typical power user usually uses almost exclusively command line tools to get the job done makes this wide screens sometimes inefficient for the job. Therefore, more and more people are using two screens to get more space but keep the work efficiency.

As shown in the figure 2.2, multi-screen emulation treats one physical monitor as a dual screen setup. This approach ensures possibility of using one screen the same way as a setup made out of two screens.

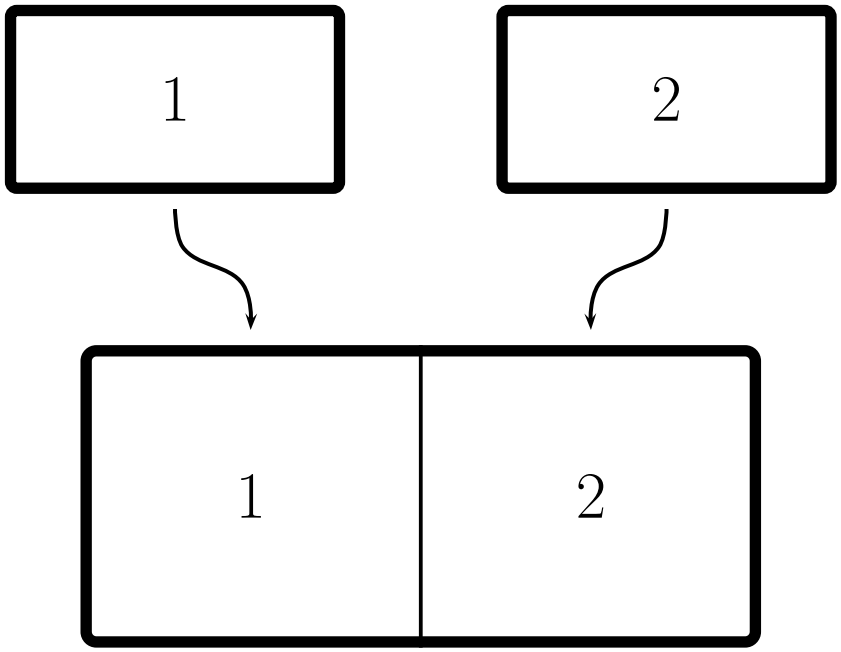

Figure 2.2: Multi-screen emulation using single monitor

Switchable Virtual Desktops

In order to provide requested functionality for the power users, it is necessary to design *switchable virtual desktops*.

Virtual desktop is a paradigm of managing computer's screen space. Switchable virtual desktops usually implement such a functionality, that it is possible to make virtual copies of the screen space and therefore expand computer screen space in general.

Design of the virtual desktops used in the output WM is shown in the figure D.1. Each virtual desktop is referenced by a *tag*. The tag can be a number or a string (defined in the `config.h`) and what is more, each tag is accessible thought the keybind

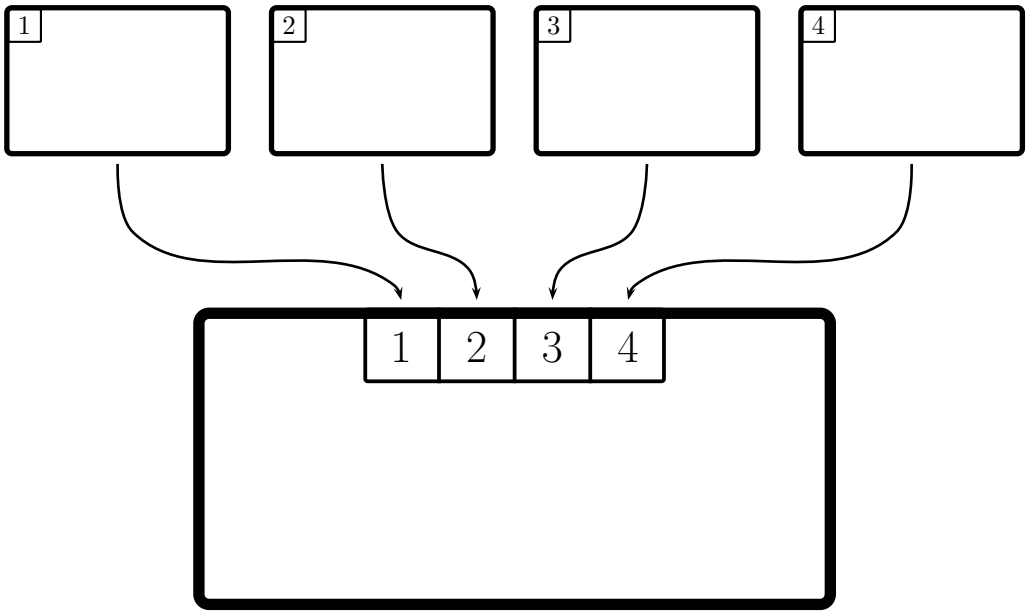

Figure 2.3: Switchable virtual desktops

shortcut (also defined in the `config.h`). This design is used in many lightweight window managers because it is effective and simple way how to design virtual desktops.

2.4 Automated Testing

It is necessary to ensure that the final implementation will be well tested, so the software requirements specification can be met (section 2.2).

Because the WM design is based on the event handling and processing, window managers in general are heavily dependent on the users input (usually mouse and keyboard). There are infinite amount of possible inputs combinations, therefore it is needed to design and implement automated tests. Only automated testing can cover enough cases and thus provide good feedback.

In order to test stability of the implementation, only the black box testing[5] seems like a reasonable solution. Generating huge amount of events in the short period of time and let the WM handle them without crashing should provide sufficient stability.

Design of the software responsible for the testing is covered in the following paragraphs. Results can be found in the section 4.1.

[5]More about software testing can be found in the free online Udacity course, cs258 Software Testing. (Udacity, Inc. 2013)

2.4.1 xfakeevent

Appropriate software responsible for the testing has to be designed and implemented. As mentioned earlier, the WM is heavily dependent on the input events, which means that some software able to artificially generate events on demand is needed in order to successfully and responsibly test the implementation.

Software responsible for generating artificial events has to meet the following criteria:

- Ability to generate button, mouse and key events on demand.
- Capable of covering the same range of possible inputs as the user.
- Command line user interface.

There are many applications implementing one or some of the requirements but unfortunately, there is no such a software which meets all the criteria mentioned above.

Therefore, an application, `xfakeevent` had to be written, so all the requirements can be met and automated testing for the WM can be effectively performed.

"xfakeevent is an easy way to send fake button, motion and key events to the X."

`xfakeevent` is written in C using the `xlib` along with the X11 extension `XTest.h`. `XTest.h` provides set of testing functions for the X. (cgit.freedesktop.org 2013)

By using this extension, it is possible to send fake events to the X. These fake events are processed as regular events, which is required for the testing.

Source code of the `xfakeevent` can be found in the `GitHub` repository: `https://github.com/examon/xfakeevent`.

2.4.2 test_splitwm

However, `xfakeevent` can not perform full required testing because all it can do is to send fake events and so `text_splitwm` had to be written.

`test_splitwm` is written in Python using only Python Standard Library. The reasons for choosing Python programming language are following:

- Python provides much higher level of abstraction than C.
- Interpreted nature, ideal for scripting.
- Extensive standard library.

(Python Software Foundation 2013)

Using set of structures representing alphanumeric symbols, buttons identifiers and mods identifiers (an example can be found in the appendix C.1), `test_splitwm`

randomly generates combinations of button, mouse and keyboard events.

`test_splitwm` uses external `xfakeevent` calls to perform set of random tests based on fake events sent by the `xfakeevent`. Randomly generated sequences of fake events consists of:

- Button events (mouse click).
- Mouse events (mouse movement).
- Key events (key press).
- Launching and closing random applications.

`test_splitwm` is available in the `splitwm` repository[6] under the `test/test_splitwm` folder.

[6] `splitwm` is currently hosted at the `https://github.com/examon/splitwm`.

Chapter 3

Implementation

The present chapter is devoted to the implementation of the window manager and thus describes tools used in the process of development of this particular application.

This chapter does not replace source code or user documentation, which can be found in the appendix E.

3.1 Implementation of Window Manager

The design of the WM is described in the section 2.3 along with the event handling structures and multi-screen emulation functionality.

Implementation of the WM was done under the project name `splitwm` and is available under the open source license at the `GitHub`: `https://github.com/examon/splitwm`.

The whole implementation has the following two central structures:

- Event handling and processing (this topic was covered in the section 2.3.1). Structures related to events can be found in the source code files `src/event.c` and `src/event.h`.
- Client, Desktop and View.

3.1.1 Client, Desktop & View

Each one of the following structures handles one important aspect of the window manager.

Client

`Client` structure is representing one window on the screen. Code listing 3.1 shows the actual code for one client[1]. Structure also contains the window's title and title length

[1] Client code structure can be found in the `src/client.h` file.

(this is needed in order to be able to correctly draw window's name on the bar) and what is more, it also stores pointers to the next and previous windows (needed for the window switching).

```
typedef struct Client {
        struct Client *next;
        struct Client *prev;
        char *title;
        int title_len;
        Window win;
} Client;
```

Code Listing 3.1: `Client` structure

All windows from one desktop are linked into the doubly linked list, so is easily possible to cycle through the linked windows forwards and also backwards using appropriate keyboard shortcuts.

Desktop

Each switchable virtual desktop (section 2.3.2) is represented by the `Desktop` structure shown in the code listing 3.2.

```
typedef struct {
        Client *head;
        Client *curr;
        int master_size;
        int layout;
        int tile_or_float;
} Desktop;
```

Code Listing 3.2: `Desktop` structure

`Desktop` contains set of clients linked into the doubly linked list (as was mentioned in the previous section) and information about the current windows layout.

In order to send client from one virtual desktop to another, it has to be possible to detach client from one desktop and attach it to targeted desktop. This, and similar functions (such as changing virtual desktops) are implemented in the `src/client.c` and `src/desktop.c` source code files.

View

`View` structure handles the multi-screen emulation functionality. As shown in the code listing 3.3, each `View` contains two `Desktop` structures, which are representing two monitors, one `Desktop` for each monitor.

```
typedef struct {
        Desktop ld[10];
        Desktop rd[10];
        int curr_left_id;
        int prev_left_id;
        int curr_right_id;
        int prev_right_id;
        int curr_desk;
        float split_width_x;
        float split_height_y;
        Bool both_views_activated;
        Bool left_view_activated;
        Bool right_view_activated;
} View;
```

Code Listing 3.3: `View` structure

Other necessary information about the screen composition are also included in the `View`. Structure itself, along with the functions responsible for the multi-screen emulation, can be found in the `src/view.c` and `src/view.h`.

3.1.2 Tiling & Floating Mode

The implementation is capable of managing windows according to tiling or floating mode. Switching between these modes is accessible through the appropriate keyboard shortcut (this shortcut is saved in the configuration file `config.h`).

There are currently two tiling algorithms implemented. The first one tiles windows according to the master and slave area, while the second one uses the grid tiling layout. It is possible to implement more tiling algorithms by writing new functions and adding them into the `src/tile.c` source code file.

3.1.3 Configuration

All configuration available to the user is done through the configuration file `config.h`. This configuration file is stored within the `src` folder along with the other source code

files. In order to correctly configure the WM, it is needed to modify the `config.h` and recompile the whole project using `Makefile` by executing the following commands within the `splitwm` folder:

```
cd src
make
sudo make install
```

This may seem inconvenient for the ordinary user, but for the simplicity of the implementation, power users prefer this design (many window managers are implemented this way, included `dwm`, `monsterwm` and others).

3.1.4 Status Bar

Built-in status bar is part of the implementation. It serves as an indicator for the user, displaying information about desktops, views, currently used layouts and window titles.

Adding external bar is also supported. The only prerequisite is to change the settings `EXTERNAL_BAR_POSITION` and `EXTERNAL_BAR_HEIGHT` in the `config.h`, so the WM automatically make space for the external bar.

By using external bar, it is possible to get system information bar or systray, which are not part of implementation yet.

3.2 Development Tools

The following section deals with the development tools used in the process of implementation.

C programming language is used for the implementation, because this language provides low-level access to the memory and require minimal run-time support, therefore it is very well suitable for writing such a low-level system application as the window manager.

3.2.1 C Programming Language

Regarding Kernighan & Ritchie (1988), *"C is a general-purpose programming language. It has been closely associated with the UNIX system where it was developed, since both system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a "system programming language" because it is very useful for writing compilers*

and operating systems, it has been used equally well to write major programs in many different domains."

The key reasons for choosing C as an implementation language are:

- low-level access to the computer memory;
- low-level data types;
- language simplicity;
- static type system;
- run-time efficiency;
- libraries support.

3.2.2 Xlib

Regarding Gancarz (2003), *"Xlib is a C subroutine library that application programs (clients) use to interface with the window system by means of stream connection."*

The Xlib library is currently used in many GUIs for variety of Unix-like operating systems.

According to the X.Org Foundation (2013), *"For low-level X development, XCB, the X C Bindings provide a clean low-level protocol binding. Its older cousin Xlib (or libX11), is not recommended for new development."*

However, Xlib is used in window manager implementation, because it is still widely used library and also with bigger community and support than XCB.

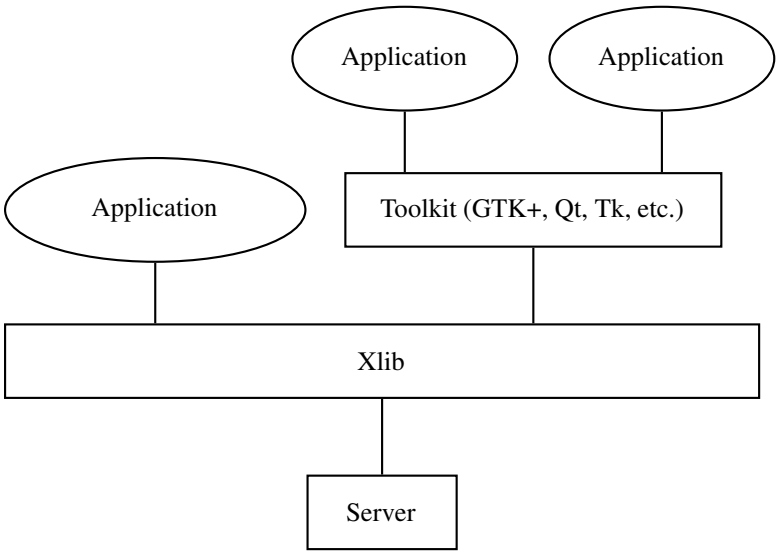

Figure 3.1: Application using Xlib

Xlib provides relative simple interface to the X. It is possible to use directly Xlib interface in the phase of development or for higher abstraction, there is a possbility to use various toolkits, as shown in the figure 3.1.

XTest.h

As was mentioned earlier in the section 2.4.1, an extension named `XTest.h` was used in the `xfakeevent` application. According to the cgit.freedesktop.org (2013), `XTest.h` provides a set of functions able to send fake events to the X. By this approach, it is possible to artificially trigger desired event without any other requirements and therefore this extension is suitable for automated tests.

3.2.3 Source Code Management

`Git` is used as a version control system. As explained in project's homepage , *"Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency."* (Git 2013)

Because of its availability and functionality, `Git` was used in conjunction with the `GitHub`[2].

Source code and documentation is hosted and available under the open source license at the implementation homepage: `https://github.com/examon/splitwm`.

[2]GitHub is a web-based hosting service for software development projects that use the Git revision control system.

Chapter 4

Results

The earlier parts of this thesis covered the X Window System concepts and window management in the GNU/Linux ecosystem. Design and implementation of the WM was also explained.

The subsequent chapter will evaluate the project. More specifically, results of the automated testing and possible future application improvements.

4.1 Results of Automated Testing

Tools and methods used in the process of automated testing are covered in the section 2.4. The following section deals with the results of the testing.

As shown in the table 4.1, six types of tests were executed by the `test_splitwm`[1]. Using `test_splitwm` it is possible to test around 10000 events in about 193 seconds. This approach extensively screens the WM and its ability to handle high workload. Also, the amount of tested events performed in relatively short period of time makes this testing method really effective. Testing single, double and triple key combinations replaces manual testing of every possible WM key shortcuts.

Results of testing are kept in log files for the further analysis. This is done by the following procedures:

1. `splitwm` is started as follows:

   ```
   exec splitwm >> splitwm.log
   ```

   The above command starts `splitwm` and redirects the output generated by the window manager's `dbg()` function to the file `splitwm.log`.

2. `test/start_splitwm_test.sh` is executed within running `splitwm` as follows:

[1] `test_splitwm` was covered earlier in the subsection 2.4.2.

```
time -p -o time.log ./start_splitwm_test.sh
```

`time` utility measures time needed to finish the task executed by the `start_splitwm_test.sh` and stores this information into the `time.log`. `start_splitwm_test.sh` is a `Bash` script used to start `test/test_splitwm.py` and redirect its output to the `test_splitwm.log`.

After testing, the following logs are available:

- `splitwm.log`, produced by the WM itself, logs mostly function calls/returns.
- `time.log`, produced by the `time` UNIX utility, logs elapsed time in seconds.
- `test_splitwm.log`, produced by the `test/test_splitwm.py`, logs tested events.

Table 4.1: Results of Automated Testing

Test Type	**Number of Tests**
Mouse button press	2049
Mouse movement	1991
Single key shortcut	2010
Double key shortcut	1957
Triple key shortcut	1222
Launch/Close application	1111
Total tests	10340
Testing time (in seconds)	193
Tests per second	54.6

In case of the WM crash during the automated testing, three logs mentioned above are available for the inspection, so the cause of the crash can be found and fixed.

By executing automated tests and analysing saved logs, verification of meeting functional requirements R01-R06 and performance demands has been made.

Example video showing automated testing can be found in the appendix E or at the web page: `https://vimeo.com/58268388`.

4.2 Possible Application Improvements

Although the final WM implementation is complete and fully functional, there are some possible application improvements.

Independent Multi-Monitor Support

Multi-monitor support could not only dynamically adapt on the resolution change (as was requested earlier in the software requirements specification, section 2.2), but also treat multiple physical monitors independently (such as `awesome` window manager or `dwm`).

Systray & System Monitor Bar Integration

Currently, the only way how to get systray and system monitor bar within the WM implementation is though an external bar. Future improvement could include native implementation of such a bar, so any third party software would not be needed in order to provide this functionality.

Xlib vs XCB

Rewriting the WM implementation using the XCB instead of Xlib should provide potential future advantages (such as better threading support, latency hiding and direct access to the protocol (freedesktop.org 2013)). XCB is becoming more popular and some window managers are already completely rewritten using XCB (for example the `awesome` window manager (awesome project 2013)).

Conclusion

Introduction and common terminology to the topic of window managers and windowing systems in the GNU/Linux system, along with the current state and general philosophy behind the topic was discussed in the chapter 1.

Chapter 2 was dedicated to the key objectives and software requirements specification of the proposed thesis. Specially, section 2.1 summarized only key objectives related to the thesis, where section 2.2 extended key objectives by defining software requirements specification.

- Multi-screen emulation functionality is designed and implemented. (Sections 2.3.2 and 3.1.1)
- Concept of switchable virtual desktops is covered and its implementation is explained. (Sections 2.3.2 and 3.1.1)
- Configuration of the window manager implementation is described. (Section 3.1.3)
- Dynamic resolution adaptation is part of the implementation. (Section 4.2)
- Multi-monitor support is implemented, although this could be improved in the future. (Section 4.2)
- External bar is supported by changing settings in the configuration file. (Section 3.1.4)
- System status bar and systray functionality are possible to achieve by using external bar. (Section 3.1.4)
- Two tiling algorithms were implemented as well as floating mode. (Section 3.1.2)

Design of the key parts was covered in the subsequent sections. Section 2.3 dealt with the event handling and processing as well as multi-screen emulation design.

Results of automated testing were explained in the section 4.1 and possible future implementation improvements were proposed in the subsequent paragraphs

(section 4.2).

Finally, the chapter 3 explained all tools, methods and approaches used in the process of implementation. Various tools used in the phase of development are analysed and selection of these tools is clarified. Screenshot of the implementation in practice can be found in the appendix D.

The implementation was positively reviewed by the community, agreeing that the multi-screen functionality is an interesting concept.

Resumé

Úvod tejto práce sa zaoberá teoretickým opisom oknového systému X Window System (ďalej len X). X je počitačový systém a sieťový protokol, ktorý poskytuje technológiu pre vývoj grafického užívateľského rozhrania. Fakt, že tento systém je prakticky jediným stabilným oknovým systémom nasadzovaným v Linuxových a všeobecne UNIXových operačných systémoch z neho robí veľmi dôležitú súčasť moderného desktopu. História vzniku tohto systému siaha až do roku 1984, kedy Robert Scheifler oznámil prvú verziu.

X sa skladá z mnohých komponentov a je celkovo veľmi rozsiahly. Displej je v X definovaný ako pracovná stanica pozostávajúca z klávesnice, myše a jedného alebo viac monitorov (zostava pozostávajúca z viac monitorov je v X podporovaná). Architektúra klient-server je centrálnym modelom, v rámci ktorého X pracuje. Vďaka tomu, že X je sieťovo orientovaný, je možné spustené aplikácie na serveri zdielať cez sieť na iný počítač vybavený klientom a displejom.

Oknový systém musí správne reagovať na veľké množstvo rôznych udalostí, ktoré sa rozdielne v čase vyvolávajú ako dôsledok konania užívateľa (užívateľ napríklad stlačí klávesu a tak vyvolá určitú udalosť). Vyhodnocovanie a spracúvanie udalostí je hlavným rozdielom oproti bežnému aplikačnému programovaniu a programovaniu priamo pre X.

Spravovanie okien je ďalším z dôležitých konceptov v rámci X. Aplikácie samotné zväčša neovplyvňujú to kde alebo na ako dlho sa objavia na obrazovke. Túto funkcionalitu zabezpečuje program nazývaný správca okien. Správca okien je v X len ďalším programom, resp. klientom, ktorý má špeciálne privilégiá a právomoci (napríklad manipulácia s inými aplikáciami v rámci X).

Existuje veľké množstvo správcov okien napísaných pre X, pričom veľa z nich pracuje na úplne odlišných princípoch. Táto práca sa zaoberá správcami okien výhradne pre operačný systém GNU/Linux, pričom viac špecificky, správcami okien pre skupinu užívateľov nazývanú power users (pokročilí užívatelia).

Správcov okien je možné rozdeliť na nasledujúce tri základné kategórie:

- Plávajúci správca okien: Okná je možné na seba ukladať podobne ako papiere na stole (najpoužívanejší koncept medzi bežnými užívateľmi).

- Dlaždicový správca okien: Okná sú automaticky ukladané podľa preddefinovaných vzorov pripomínajúcich dlaždice. Okná sa neprekrývajú a tak je celá plocha obrazovky efektívne využitá.
- Dynamický správca okien: Možnosť prepínania medzi rôznymi štýlmi spravovania okien (napríklad medzi plávajúcim a dlaždicovým).

Kedže základná terminológia a teória súvisiaca s oknovým systémom X Window System a spravovaním okien v tomto systéme bola opísaná, je možné bližšie špecifikovať ciele tejto práce:

- Analyzovať princípy uplatňujúce sa pri návrhu aplikácie pre X Window System.
- Navrhnúť a implementovať jednoduchého správcu okien pre X Window System, ktorý musí spĺňať nasledujúce funkčné požiadavky:
  - Podpora dlaždicového módu s minimálne dvoma implementovanými dlaždicovými algoritmami.
  - Podpora plávajúceho módu.
  - Umožňovanie dynamického prepínania medzi módmi.
  - Implementácia prepínateľných virtuálnych plôch.
  - Nastavitelnosť správcu okien pomocou užívateľsky prístupného konfiguračného súboru.

  Nasledujúce funkčné požiadavky na systém sú volitelné:

  - Podpora viacerých monitorov.
  - Dynamické prispôsobenie sa na zmenu rozlíšenia monitora.
  - Implementácia/podpora oznamovacej oblasti a systémového panelu.
  - Podpora externého panelu.
- Navrhnúť a implementovať špecifickú funkcionalitu pre navrhnutého správcu okien. Táto funckionalita musí využívať emuláciu dvoch obrazoviek v rámci jednej fyzickej obrazovky.
- Poskytnúť dokumentáciu k navrhnutému systému.

Všetky hore uvedené povinné ako aj voliteľné funkčné požiadavky kladené na správcu okien sú implementované a splnené.

Špecifickou vlastnosťou navrhovaného správcu okien je emulácia dvoch monitorov za použitia jedného fyzického monitora. Žiadaná functionalita je motivovaná tým, že na širokouhlých monitoroch vzniká zbytočne veľa nezaplnených plôch, ktoré by mohli byť efektívne využité tým, že sa bude jeden širokouhlý monitor správať ako zostava dvoch monitorov a tak poskytovať všetky výhody dvojmonitorovej zostavy.

Požadovaná funkcionalita bude, podobne ako iný správcovia okien pre X,

implementovať prepínateľné virtuálne plochy. Virtuálna plocha je koncept, ktorý umožňuje rozšíriť pracovný priestor tým, že pracovná plocha je nakopírovaná a uložená v pamäti, pričom je možné kedykoľvek vyvolať akúkoľvek požadovanú plochu a tak rozšíriť niekoľkonásobne pracovný priestor.

Kedže je potrebné zaistiť stabilitu implementovaného systému, je nutné navrhnúť testovacie scenáre. Pre úspešné testovanie je teda potrebné vytvoriť ďalšie programy, ktoré budú zodpovedné za testovanie.

Ako bolo uvedené vyššie, správca okien je veľmi závyslý na vyhodnocovaní a následnom spracúvaní udalostí vyvolaných užívateľom. Pre potreby testovania je však nutné generovať veľký objem udalostí v krátkom čase a tak otestovať čo najväčší počet možných udalostí a ich následkov. Kvôli týmto požiadavkám musí byť vytvorený špeciálny program, ktorý bude zodpovedný za umelé generovanie udalostí. Ďalší napísaný program musí náhodne vyberať aké udalosti sa budú generovať a všetko zapisovať do textového súboru.

Programy `xfakeevent` a `test_splitwm` boli teda vytvorené pre tento účel. `xfakeevent` je napísaný v jazyku C a využíva rozšírenie pre X nazvané `XTest.h`. Toto rozšírenie poskytuje testovacie funkcie, ktoré dokážu odosielať umelé udalosti na X server. `xfakeevent` je teda zodpovedný za umelé generovanie udalostí. `test_splitwm` je napísaný vo vysokoúrovňovom skriptovacom jazyku Python a je zodpovedný za náhodný výber a zapisovanie záznamu.

Samotný priebeh testovania je plne automatizovaný. So spustením skriptu `start_test_splitwm.sh` (tento skript sa spoločne s programom `test_splitwm` nachádza v adresári `test`) sa splustí program `test_splitwm`, ktorý za pomoci programu `xfakeevent` náhodne generuje umelé udalosti. Celý priebeh testovania sa zaznamenáva do logovacích súborov pre prípad zlyhania a následného pádu správcu okien. Zaznamenaná činnosť testovacích programov, ako aj samotného správcu okien, neskôr poslúži k reprodukovaniu chyby, ktorá spôsobila pád správcu okien a k následnému odstráneniu vzniknutej chyby.

Ako je možné vidieť v tabuľke 4.1, programy vykonávajúce automatické testovanie dokážu otestovať približne 10000 testov za 193 sekúnd. Ak by neboli použité automatické testy, bolo by prakticky nemožné otestovať až 55 testov za sekundu, čo robí z tohto prístupu k testovaniu veľmi výhodnú a rýchu metódu ako otestovať program akým je správca okien.

Všetky programy podieľajúce sa na testovaní, spolu s ich zdrojovými kódmi, je možné nájsť v prílohách tejto práce.

Konfigurácia správcu okien je prístupná užívateľovi cez konfiguračný súbor `config.h`. V tomto súbore sa (okrem iného) nachádza napríklad zoznam klávesových skratiek

pomocou ktorých je možné ovládať správcu okien.

Implementačný jazyk použitý pri vývoji správcu okien je C, pretože poskytuje dostatočnú vrstvu abstrakcie a výhodné dátové typy a štruktúry pre tento typ projektu.

Samotný projekt implementácie správcu okien je vedený pod názvom `splitwm`. Projekt je uložený na serveri `GitHub`, ktorý poskytuje úložný priestor pre projekty spolu s verzovacím systémom `Git`. Implementácia správcu okien je vydaná pod open-source licenciou MIT/X.

Snímok zachytávajúci správcu okien `splitwm` pri emulácii dvoch obrazoviek je priložený v prílohe D.

Zdrojový kód implementácie, spolu s dokumentáciou k projektu, sú priložené v prílohách práce.

References

Arch Linux Wiki 2013. *Window manager*. [online], Available on Internet: `https://wiki.archlinux.org/index.php/Window_manager`, [cit.2013-04-05].

awesome project 2013. *Awesome documentation*. [online], Available on Internet: `http://awesome.naquadah.org/wiki/Main_Page`, [cit.2013-04-05].

cgit.freedesktop.org 2013. *libxtst*. [online], Available on Internet: `http://cgit.freedesktop.org/xorg/lib/libXtst`, [cit.2013-04-05].

Dictionary.com 2013. *Power user definition*. Available on Internet: `http://dictionary.reference.com/browse/power%20user`, [cit.2013-01-04].

Foley, J. D., van Dam, A., Feiner, S. K., et al. 1990. *Computer Graphics: Principles and Practice*. Addison-Wesley, ISBN 0-201-12110-7.

freedesktop.org 2013. *Xcb*. [online], Available on Internet: `http://xcb.freedesktop.org`, [cit.2013-04-05].

Gancarz, M. 2003. *Linux and the Unix Philosophy*. Digital Press, ISBN 1555582737.

Git 2013. *Git project homepage*. Available on Internet: `http://www.git-scm.com`, [cit.2013-01-04].

ISO 2009. *Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems*. ISO 9241-210:2009, International Organization for Standardization, Geneva, Switzerland.

Kernighan, B. W., Ritchie, D. M. 1988. *C Programming Langage (2nd Edition)*. Prentice Hall, ISBN 0131103628.

Lineback, N. 2013. *Xerox star*. [online], Available on Internet: `http://toastytech.com/guis/star.html`, [cit.2013-04-05].

LinuxCounter.net 2013. *Linuxcounter.net*. [online], Available on Internet: `http://linuxcounter.net`, [cit.2013-04-05].

Nye, A. 1994. *Xlib Programming Manual, Rel. 5, 3rd Edition*. O'Reilly Media, ISBN 1565920023.

Python Software Foundation 2013. *Python programming language*. [online], Available on Internet: `http://www.python.org`, [cit.2013-04-05].

Raymond, E. S. 2003. *The Art of Unix Programming*. Addison-Wesley, ISBN 0-13-142901-9.

Raymond, E. S. 2013. *The Jargon File, version 4.4.8*. Available on Internet: `http://catb.org/jargon/`, [cit.2013-01-04].

Scheifler, R. W. 1984. *window system x*. [online], Available on Internet: `http://www.talisman.org/x-debut.shtml`, [cit.2013-04-05].

The Open Group 2013. *About the X Window System*. Available on Internet: `http://www.opengroup.org/desktop/x`, [cit.2013-01-04].

Udacity, Inc. 2013. *Software testing*. [online], Available on Internet: `https://www.udacity.com/course/cs258`, [cit.2013-04-05].

X.Org Foundation 2013. *The X.Org project*. Available on Internet: `http://www.x.org/wiki`, [cit.2013-01-04].

Appendix A

The XEvent union

```
typedef union _XEvent {
    int type;
    XAnyEvent xany;
    XKeyEvent xkey;
    XButtonEvent xbutton;
    XMotionEvent xmotion;
    XCrossingEvent xcrossing;
    XFocusChangeEvent xfocus;
    XKeymapEvent xkeymap;
    XExposeEvent xexpose;
    XNoExposeEvent xnoexpose;
    XGraphicsExposeEvent xgraphicsexpose;
    XVisibilityEvent xvisibility;
    XCreateWindowEvent xcreatewindow;
    XDestroyWindowEvent xdestroywindow;
    XUnmapEvent xunmap;
    XMapEvent xmap;
    XMappingEvent xmapping;
    XMapRequestEvent xmaprequest;
    XReparentEvent xreparent;
    XConfigureEvent xconfigure;
    XGravityEvent xgravity;
    XResizeRequestEvent xresizerequest;
    XConfigureRequestEvent xconfigurerequest;
    XCirculateEvent xcirculate;
    XCirculateRequestEvent xcirculaterequest;
    XPropertyEvent xproperty;
```

```
    XSelectionClearEvent xselectionclear;
    XSelectionRequestEvent xselectionrequest;
    XSelectionEvent xselection;
    XColormapEvent xcolormap;
    XClientMessageEvent xclient;
} XEvent;
```

Code Listing A.1: The ***XEvent*** *union*

Appendix B

buttonpress, buttons

```
for (i = 0; i < LENGTH(buttons); i++) {
        if ((buttons[i].mask == e->xbutton.state)
         && (buttons[i].button == e->xbutton.button)
         && buttons[i].func)
                buttons[i].func(&(buttons[i].arg));
}
```

Code Listing B.1: `buttonpress`

```
static Button buttons[] = {
        /* event mask button   function   argument */
        {  MOD1,       Button1, mousemove, { .i = MOVE }},
        {  MOD1,       Button3, mousemove, { .i = RESIZE }},
        {  MOD4,       Button1, mousemove, { .i = MOVE }},
        {  MOD4,       Button3, mousemove, { .i = RESIZE }}
};
```

Code Listing B.2: `buttons`

Appendix C

min_set

```
alphabet = [ "0", "1", "2", "3", "4",
             "5", "6", "7", "8", "9",
             "a", "b", "c", "d", "e",
             "f", "g", "h", "i", "j",
             "k", "l", "m", "n", "o",
             "p", "q", "r", "s", "t",
             "u", "v", "w", "x", "y",
             "z", "A", "B", "C", "D",
             "E", "F", "G", "H", "I",
             "J", "K", "L", "M", "N",
             "O", "P", "Q", "R", "S",
             "T", "U", "V", "W", "X",
             "Y", "Z" ]

butns = [ "Space",
          "Tab",
          "Return",
          "F1",
          "F2",
          "F3",
          "F4",
          "F5",
          "F6",
          "F7",
          "F8",
          "F9",
          "F10",
```

```
           "F11",
           "F12" ]

mods = [ "Shift_L",
         "Shift_R",
         "Control_L",
         "Control_R",
         "Meta_L",
         "Meta_R",
         "Alt_L",
         "Alt_R",
         "Super_L",
         "Super_R" ]
```

Code Listing C.1: ***min_set***

Appendix D

`splitwm` screenshot

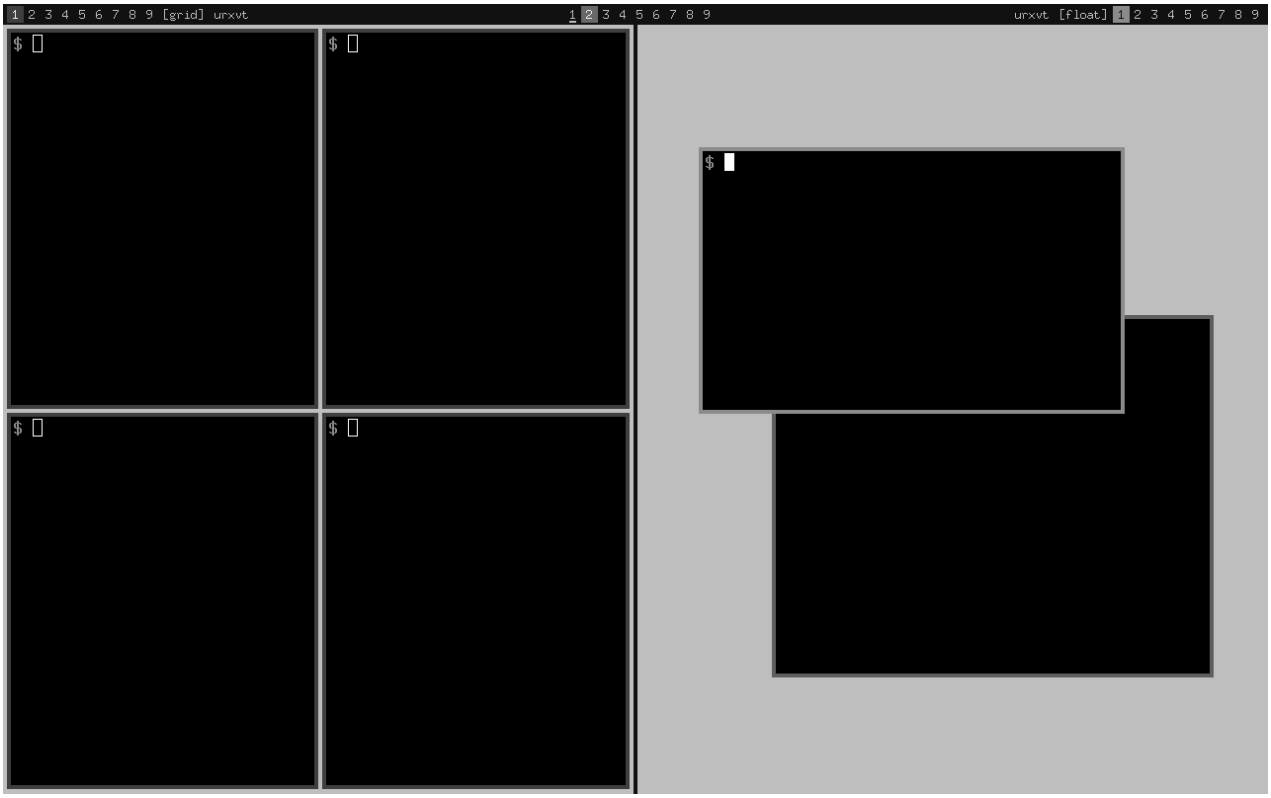

Figure D.1: `splitwm` screenshot

Appendix E

Source CD

Contents

Statutory Declaration

This is to declare that I have elaborated this thesis autonomously, have not used other than the listed sources, and that I listed all the sources quoted either literally or by content from the sources used.

Trnava, 29th May 2013

_____

Signature