Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Petr Koupý

Graphics Stack for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký

Study programme: Informatics

Specialization: Software Systems

Prague 2013

I would like to thank my supervisor, Martin Děcký, not only for giving me an idea on how to approach this thesis but also for his suggestions, numerous pieces of advice and significant help with code integration. Next, I would like to express my gratitude to all members of HelenOS developer community for their swift feedback and for making HelenOS such a good platform for works like this. Finally, I am very grateful to my parents and close family members for supporting me during my studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, March 27, 2013 Petr Koupý

Název práce: Graphics Stack for HelenOS
Autor: Petr Koupý
Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: Mgr. Martin Děcký

Abstrakt: HelenOS je experimentální operační systém založený na mikro-jádrové a multi-serverové architektuře. Před započetím této práce již HelenOS obsahoval početnou množinu moderně navržených subsystémů zajišťujících různé úkoly v rámci systému. Nicméně postrádal řádně navržený subsystém pro interakci s uživatelem. Zatímco vstupní část takového subsystému byla již v pokročilé fázi vývoje, výstupní část představovala pouze rychlé dočasné řešení, aby existoval alespoň nějaký prostředek jak vykreslit informace na obrazovku. Tato práce se zabývá vytvořením řádného grafického subsystému, který by nahradil ten dočasný. Výsledný nový grafický subsystém HelenOS je navržen podle moderních principů v dané problematice. I přes minimalistický přístup je počáteční implementace vysoce rozšiřitelná a již prakticky použitelná vývojářskou komunitou HelenOS. Práce pokrývá všechny důležité části grafického subsystému – rozhraní pro grafické ovladače, kreslící knihovnu, grafický server a knihovnu pro vytváření uživatelského rozhraní aplikací.

Klíčová slova: grafický subsystém, grafické uživatelské rozhraní, HelenOS

Title: Graphics Stack for HelenOS
Author: Petr Koupý
Department / Institute: Department of Distributed and Dependable Systems
Supervisor of the master thesis: Mgr. Martin Děcký

Abstract: HelenOS is an experimental operating system based on a micro-kernel multi-server architecture. Before the inception of this thesis, HelenOS already contained numerous modernly designed subsystems responsible for various system tasks. However, it lacked the proper subsystem for interaction with the user. While the input part of such subsystem was already in an advanced stage of development, the output part was just a quick and temporary solution in order to have at least some mean to populate the screen with information. This thesis deals with the creation of a proper graphics subsystem that would replace the temporary one. The resulting new HelenOS graphics subsystem is designed according to the state-of-the-art principles found within the problem domain. Although minimalistic, the initial implementation is highly extensible and already practically usable by the HelenOS developer community. The thesis covers all important parts of the graphics subsystem – graphic driver infrastructure, drawing library, graphics server and a toolkit for creation of application user interface.

Keywords: graphics subsystem, graphical user interface, HelenOS

Contents

1 Introduction **6**

1.1 Thesis Structure 6

1.2 Source Code 6

2 Context **8**

2.1 Key Concepts 8

2.2 Graphics Stack 10

2.3 Starting Point 11

2.4 Work Scope 14

3 Analysis **15**

3.1 Graphic Driver Infrastructure 15

3.2 Drawing Library 16

3.3 Graphics Server 18

3.4 Widget Toolkit 21

4 Design **24**

4.1 Overall Architecture 24

4.2 Graphic Driver Infrastructure 27

4.3 Drawing Library 30

4.4 Graphics Server 34

4.5 Widget Toolkit 40

5 Conclusion **48**

5.1 Accomplishments 48

5.2 Added Value 49

5.3 Future Work 49

1 Introduction

Interaction with the user is one of the many responsibilities any general-purpose operating system must cover in order to succeed. To accommodate ever widening range of devices, applications and usage scenarios, the subsystems for input handling and output generation have evolved into a fairly complex hierarchy of components residing on various levels within the operating system structure. In the case of output generation, such set of components is called *graphics stack*.

The common characteristic of graphics stacks found across mainstream operating systems is that alongside with the modernization they must also deal with the burden of backwards compatibility – either within the stack itself or for the sake of applications. As a result, these stacks are quite complicated. Then, there are experimental operating systems that usually either port existing solutions or do not have proper graphics stack at all as it is not a top priority.

The latter is true also for HelenOS, a research operating system originated at Faculty of Mathematics and Physics at Charles University in Prague, still actively developed by former and current students. HelenOS is characteristic by its portability and micro-kernel multi-server architecture which makes it very modular. By its experimental nature and modern design, HelenOS is an ideal platform to implement various subsystems from scratch without the need to consider an aspect of backwards compatibility.

The goal of this thesis is to design and implement a brand-new graphics stack for HelenOS. The proposed design reflects state-of-the-art approaches found within the problem domain and adapt them to HelenOS architecture. Because of limited manpower and wide extent of the topic, both design and implementation are approached with the expectation of future development. Overall, the thesis is focused on delivering minimalistic, yet practically usable and extensible product.

1.1 Thesis Structure

Chapter 1 introduces the thesis topic and provides the directions on how to retrieve the source code. **Chapter 2** serves as a basis for all subsequent chapters. It contains generic introduction to the graphics stack problem domain and explores HelenOS perspective regarding this matter. The chapter also sets the scope and goals of this thesis. **Chapter 3** analyses individual major components of a graphics stack with respect to the existing state-of-the-art or legacy solutions. Then, **Chapter 4** describes the overall architecture of the new HelenOS graphics stack and gives design and implementation details of the created components. **Chapter 5** summarizes the thesis achievements and suggests the future development of the presented graphics stack.

1.2 Source Code

HelenOS developer community utilizes distributed source control system called Bazaar (http://bazaar.canonical.com). The development effort is occurring simultaneously in numerous feature branches hosted on Launchpad (https://launchpad.net), each of which is sooner or later merged into the main project repository. In order to obtain, build and run any branch, please refer to the step-by-step description at [4].

The original development branch containing the graphics stack implementation can be obtained by Bazaar client from the `lp:~petr-koupy/helenos/gui`. Since the branch is also merged into the main repository, the most recent version of the code is available within `bzr://bzr.helenos.org/mainline`. For the purpose of just reviewing the code inside the web browser, one can do so at http://bazaar.launchpad.net/~petr-koupy/helenos/gui/files, or http://trac.helenos.org/browser respectively. Additionally, extracted source code repositories and compiled bootable images are located on the attached medium (**Appendix A**). However note that the medium content might be outdated and shall serve just for the purposes of this thesis, not any future development.

2 Context

This chapter describes the context in which the goal of this thesis is approached. It establishes the necessary terminology and provides the basic understanding of what is the graphics stack and how it is composed. The chapter also examines the implementation environment provided by HelenOS and specifies the goal of the thesis in more detail.

2.1 Key Concepts

The following list contains a brief explanation of the key concepts related to the graphics stack. These concepts are repeatedly referred to in the subsequent chapters of the thesis. Apart from giving an explanation, the list should also prevent ambiguity, as some of the concepts could be interpreted differently by various readers otherwise.

Graphic device

Dedicated or integrated graphic card with one or more video outputs. Can have some ability to accelerate 2D or 3D drawing operations.

Output device

Monitor or display connected to one of the video outputs on the graphic device.

Mode setting

Configuration of the video output supported by both graphic device and output device. This usually includes resolution, aspect ratio, color depth and refresh rate.

Frame buffer

Memory area that is displayed on the output device. Can be located in some region of system memory or graphic device memory. Sometimes, it has a form of a special-purpose dedicated memory on the graphic device. Its parameters correspond to the current mode setting.

Off-screen buffer

Region of system memory or graphic device memory used for preparation of content that will be at some point transferred to the frame buffer. Does not have to correspond to the current mode setting or to the frame buffer size, as it is usually transformed during the transfer and only used as one of the many assets to build a resulting scene.

Graphic driver

Driver controlling the graphic device. It is primarily responsible for providing access to the frame buffer, enumeration of output devices and enumeration of available mode settings. If the device has some acceleration capabilities, the driver can implement interface of some higher level entities in order to accelerate their drawing operations.

Drawing library

Provides a set of 2D or 3D drawing operations of various levels of complexity. To give an example, these operations include line drawing, curve drawing, polygon drawing, area filling, font glyph drawing, affine transformations, alpha compositing, etc. In case the library defines an interface that could be implemented by the graphic driver, some drawing operations can be accelerated by the underlying graphic device.

Graphics server

Allows multiple applications (i.e. clients) to share the frame buffer surface in a consistent manner, usually in the form of windows or tiles. Server also listens to input devices (e.g. mouse, keyboard) and feeds the input events to the clients according to their position on the screen. Some older graphics servers are completely responsible for drawing into the areas assigned to clients by executing the drawing operations received from clients. Modern graphics servers, or so called compositors, usually expect clients to draw for themselves and only integrate their off-screen buffers into the resulting scene. Since the server must draw at least itself (and sometimes even the clients), it must either implement its own set of drawing operations or use some drawing library. Again, some of these drawing operations can be accelerated if the server or the library defines the interface that could be implemented by the graphic driver.

Direct rendering

Application is drawing directly to the frame buffer or to the off-screen buffer, usually with the help of some drawing library.

Indirect rendering

Application is sending drawing commands to some third party, usually to the graphics server, that executes the received drawing operations instead of the application.

Software rendering

Drawing operations are executed on the central processing unit, instead of accelerated by the graphic device.

Widget toolkit

Library providing an infrastructure and a set of reusable building blocks (widgets) from which the graphical user interface of applications is composed. As an example, this includes buttons, text boxes, scroll bars, menus, etc. From the application viewpoint, widget toolkit is a high-level abstraction over drawing operations and input events provided by the lower level entities like drawing library or graphics server. Toolkit is also responsible for the distribution of input events among the widgets and for the mutual communication between the widgets.

Window manager

Usually a component of the graphics server that is responsible for placement and manipulation of client surfaces on the screen (e.g. resizing, moving, switching, closing, visibility). For this purpose, window manager must consume some input events that are consequently not delivered to the clients.

Window decorator

Either a component of the graphics server or a part of the widget toolkit that is responsible for drawing of a frame around the window. The frame is composed from widgets that communicate with window manager and allow the user to manipulate windows.

Desktop environment

One or more graphics server clients that together provide the means to launch other clients and to keep the user informed about the status of running clients. They are usually distributed over the entire screen, parts are in the background (behind all other clients), parts are in the foreground (in front of all other clients), they are not decorated and cannot be manipulated as ordinary clients. Because of the mentioned responsibilities, desktop environment must be launched automatically during the bootstrapping of the graphics stack. To give an example, desktop environment usually consists of an application launcher and a taskbar.

Terminal emulator

Special client of the graphics server, which purpose is to emulate low-level text modes that are usually provided by the computer firmware. If the operating system contains command-line interpreter and a set of console applications, these can communicate with the terminal emulator to be used even within non-text modes alongside with the graphical applications. More specifically, terminal emulator is responsible for rendering the text output of its clients and for sending them the keyboard strokes.

2.2 Graphics Stack

This section provides a basic insight to the internal structure of the generic graphics stack – how is it layered and what are the various ways how to distribute responsibilities among the layers. It should help later when giving real-world examples, making the major design decisions and choosing the appropriate subset of the graphics stack for the initial implementation.

When looking on the graphics stacks from the hindsight, one can usually distinguish three tiers, as demonstrated in **Figure 1**. First tier of the graphics stack is responsible for the management of physical resources of the underlying graphic device and for providing access to these resources to the second tier of the stack via some well-defined interface that is often enforced by the second tier entities. First tier is primarily inhabited by a vendor-specific graphic driver. In the simplest case, the driver expects to be used just by a single entity from the second tier and implements just a basic interface to allow access to the frame buffer and mode setting. In the most complex case, the driver must implement interfaces of numerous drawing libraries and accelerate multiple concurrent direct-rendering applications. Since such scenario requires the driver to act as a graphic kernel that performs advanced memory management, context switching and scheduling of the clients to share the graphic device between them, some of this advanced functionality might be provided by the operating system in the form of a graphic driver infrastructure. Moreover, the drawing libraries can be implemented against a common interface or there can be some unification layer for them, so the graphic driver is required to implement just one interface instead of many. Depending on the overall architecture of the operating system, the communication between first and second tier can be done via system calls or via inter-process communication.

Second tier contains drawing libraries and a graphics server. There can be multiple drawing libraries, some of them specializing on 2D operations, others on 3D operations. While it could be enough for 2D libraries to use software rendering, 3D libraries have to be accelerated through the graphic driver. As was already said, there can be a unification layer between the graphic driver and drawing libraries. Apart from unifying the interfaces, such layer usually also contains functionality reusable by various drawing libraries (e.g. state machine management for 3D libraries). Graphics server can accommodate direct rendering clients, indirect rendering clients, or both. Similarly as the drawing library, graphics server can also use software rendering or hardware acceleration. Often, the graphics server itself is also linked to some accelerated drawing library and therefore shares the graphic device with other direct-rendering applications. As a minimum, graphics server must distribute input events to the clients and share the frame buffer surface among them. With the increasing complexity, server can also provide compositing features, window management and possibly even window decorations. The communication between graphics server and its clients is based either on synchronous or asynchronous inter-process communication, sometimes with the possibility to tunnel this communication over the network to/from the remotely running client. The communication protocol and interface is usually defined by the server.

While for some multimedia and 3D applications, it is enough to be linked just to the drawing library and therefore be dependent only on the first and the second tier of the graphics stack; most of the applications additionally need the widget toolkit. Because the widget toolkit is a high-level abstraction over the interface provided by a drawing library and over the input events coming from the graphics server, it can be considered as a third and final tier of the graphics stack. Again, there can be multiple widget toolkits in the third tier, each dependent on different drawing library and each providing different level of sophistication that often even exceeds the context of graphics stack (e.g. databases, networking, web, audio, scripting).

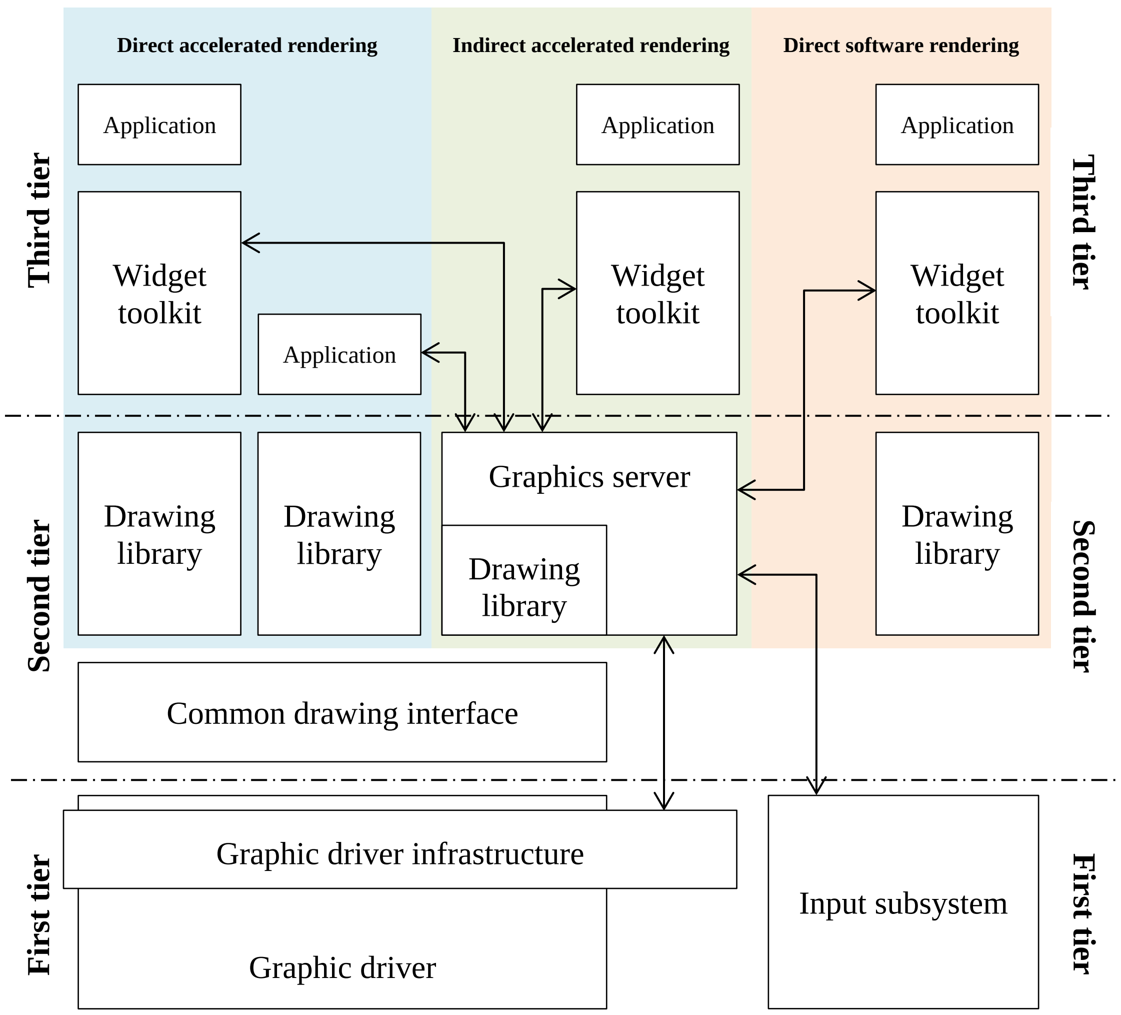


Figure 1 Generalized layout of a graphics stack showing the most important concepts.

2.3 Starting Point

This section explores the state of HelenOS graphics stack before the inception of this thesis. It also enumerates and briefly describes HelenOS features and properties that are leveraged or exploited within the implementation of the brand-new stack. This section also mentions what is yet missing in the HelenOS and therefore what complicated or otherwise limited the initial implementation of the new stack.

HelenOS is a micro-kernel multi-server operating system; therefore almost all the functionality resides in user-space servers that communicate with each other via the asynchronous kernel-based inter-process communication. To ease the development of various servers, HelenOS C library contains so called *async framework* [2] that is providing a relatively high-level abstraction over the low-level communication primitives. Most importantly, async framework allows to group individual messages into more complex logical compounds [3] and enables various independent entities within the server to be isolated from each other when communicating with other servers.

HelenOS also provides advanced means [1] to parallelize and synchronize the execution of the entities within the server. Threads that are preemptively scheduled by the kernel can be further divided into cooperatively scheduled fibrils that provide purely user-space parallelism. This is accompanied by a wide range of synchronization primitives (atomics, mutexes, futexes, read-write locks, conditional variables). User-space parallelism and synchronization is heavily leveraged by the async framework and by the communication parts of the servers to avoid the cost of otherwise quite expensive context switches.

In order to establish communication between servers, there are two special servers in the system designated to this purpose – *ns*[1] *server* and *loc*[2] *server*. While the ns server keeps track over the system-wide singleton servers, loc server is concerned by dynamically created entities within the servers. In more detail, loc server accepts registration of those entities from other servers, puts them into the requested category and organizes them into the tree-like structure within the virtual file system. To simplify interaction with ns server and loc server, HelenOS C library contains convenient interfaces to look up and connect to a particular entity in a remote server.

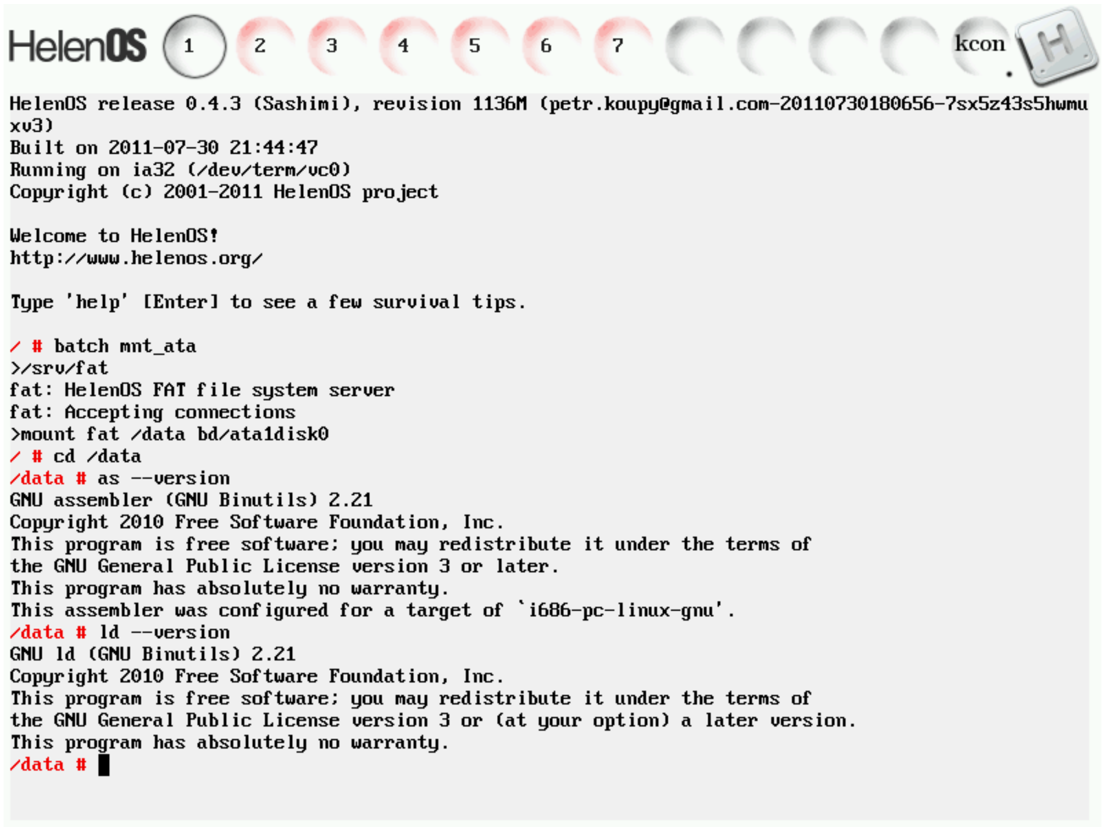
Despite the already mentioned assets, HelenOS lacks some background functionality that is relatively important for more advanced features of the graphics stack but which extent is too large to be included in this thesis. First of all, HelenOS C library, despite being relatively extensive, is implemented in a lazy manner – i.e. new functionality is added only once it is required for the first time. As of writing this, C library is completely missing math functions that might be useful for floating point arithmetic used in the drawing library. Also there is no library for extending the C language with object-oriented features, nor is it currently possible to use C++ to implement some parts of the system. While it is possible, to some extent, to simulate object-oriented principles in C, this could be a limiting factor when implementing the widget toolkit. Lastly, it prevails throughout the system the lack of support to terminate things properly or to terminate them at all – be it termination of applications, unregistering of services or shut down of the entire system.

After the description of what implementation environment is available in HelenOS, it is now possible to focus on the state of the actual graphics subsystem that was part of HelenOS before the beginning of this thesis. On the first sight, HelenOS was offering just a text-based user interface as can be seen in **Figure 2**. In more detail, there was a *console server* managing multiple character-based tabs (i.e. two dimensional rectangular character arrays) for the clients. Although the tabs were top-down scrollable (in a sense of a circular buffer), no history was stored for the content that disappeared from the visible part of the screen. Each tab was shared between clients in a cooperative manner – i.e. claimed by *bdsh*[3] *command interpreter* that would temporarily yield the tab for the sake of newly executed application. Some of the applications were using just standard input and output acquired from the virtual file system

[1] Name service
[2] Location service
[3] Brain-dead shell

and backed by bdsh, others were using the console server interface directly in order to have more accurate control over the output. The console server was getting input from the *input server* as a queue of keyboard events that were further distributed to the application in the active tab. Active tab of the console server was shared with the *fb*[4] *server* that was mainly responsible for rendering tab‘s textual content by some of the embedded graphic drivers, either text-based or pixel-based. It should be noted that those embedded drivers were not full-featured graphic drivers and that they did not have the same status as other device drivers in the system – first of all, they were all dependent on the firmware-provided graphic mode initialized by the kernel during bootstrapping (e.g. VESA[5] mode, EGA[6] mode), secondly, they were not based on the *device driver framework* and *devman*[7] *server* as other device drivers in HelenOS. When using pixel-based driver, fb server was also able to render mouse pointer, images and simple animations (image sequences). For this purpose, console server was also handling mouse events, delivered from the input server, and relayed them to the fb server in order to depict mouse movement and image clicking.

Using the already established terminology, it is possible to say that the console server was actually a terminal emulator residing in the place of a graphics server. Continuing the analogy, fb server was a strange combination of a primitive graphics server and graphic driver infrastructure, residing in the place of a graphic driver. Overall, the original design of HelenOS graphic subsystem was clearly meant just as a quick temporary solution to help develop other parts of the system.



```
HelenOS release 0.4.3 (Sashimi), revision 1136M (petr.koupy@gmail.com-20110730180656-7sx5z43s5hwmu
xv3)
Built on 2011-07-30 21:44:47
Running on ia32 (/dev/term/vc0)
Copyright (c) 2001-2011 HelenOS project

Welcome to HelenOS!
http://www.helenos.org/

Type 'help' [Enter] to see a few survival tips.

/ # batch mnt_ata
>/srv/fat
fat: HelenOS FAT file system server
fat: Accepting connections
>mount fat /data bd/ata1disk0
/ # cd /data
/data # as --version
GNU assembler (GNU Binutils) 2.21
Copyright 2010 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `i686-pc-linux-gnu'.
/data # ld --version
GNU ld (GNU Binutils) 2.21
Copyright 2010 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) a later version.
This program has absolutely no warranty.
/data # █
```

Figure 2 Original user interface of HelenOS before starting this thesis.

[4] Frame buffer
[5] Video Electronics Standards Association
[6] Enhanced Graphics Adapter
[7] Device manager

2.4 Work Scope

This section declares the scope of the thesis and the extent of the accompanying implementation of the graphics stack. It should be obvious from the description in the previous sections that design and implementation of the graphics stack is a very wide topic and therefore both scope and extent must be very carefully considered, especially in the context of a relatively limited time and manpower. Consider that each part of the graphics stack, assuming the full-featured implementation, could be a stand-alone thesis topic by itself.

From the HelenOS viewpoint, it is crucial to ensure design completeness and balanced minimalistic implementation of the final product. More specifically, resulting design shall be complete enough so the graphics stack could be gradually enhanced by the HelenOS developer community in the aftermath of the thesis without the need for some disruptive and extensive redesigning. Initial implementation, while minimalistic, shall be complete enough to cover the practical usage scenarios occurring within HelenOS developer community on a daily basis, mainly the ability to have multiple terminal emulator windows opened on the desktop simultaneously and to interact with them in a comfortable manner. At the same time, the initial implementation shall be able to demonstrate the key aspects of the modern graphics stack, at least in some basic sense. This way, there is a chance that the resulting graphics stack could be merged into HelenOS mainline instead of becoming just an experimental side project.

Considering the goals stated above, the initial implementation shall span all three tiers of the stack with the strict focus on the functionality that is required to reach the state of basic, yet practical usability. Therefore, majority of effort shall be invested into upper tiers of the stack, mainly to the graphics server in the second tier, as it is the central component of the entire stack and also the prerequisite for further development of user applications and other stack components. Both drawing library and widget toolkit, while designed properly, shall be implemented in a lazy manner according to the requirements of the graphics server and the initial set of applications.

Design phase of the project shall be accompanied by the exploration of existing solutions across all the tiers of the stack. This exploration shall yield historical context and modern trends that could help in choosing the appropriate properties when designing the HelenOS graphics stack. While the existing solutions shall be mainly a source of inspiration, some of them might be even considered to be ported into HelenOS – especially some full-featured drawing library and widget toolkit could be useful to bring more applications to HelenOS from other operating systems. However, as the porting effort could be quite time demanding from the thesis perspective and since HelenOS might not yet be ready to accommodate some large external project, it is enough to just assess such possibility for the future.

The initial implementation shall be fast enough to be practically usable on both physical and virtual machines. However, the optimization shall not be a primary focus of the project as it usually decreases the design flexibility and code readability, both of which are important for the future development of the stack. Regarding the optimization, it should be also noted that HelenOS is a multiplatform operating system. Therefore, the implementation shall not be over optimized just for a single platform.

3 Analysis

This chapter explores various possibilities on how to design individual components of the graphics stack. In addition, it gives real-world examples on existing solutions and considers what features and properties are the most important with respect to this thesis.

3.1 Graphic Driver Infrastructure

During the last decade, there was a lot of development happening in the bottom layer of the graphics stacks across the mainstream operating systems. Originally, it was common to expect graphic driver to be responsible for all the required functionality. There was virtually no assistance from the operating system, perhaps apart from frame buffer management and mode setting. Such an approach was acceptable because there were usually only two drawing libraries in the system (2D library embedded in the graphics server and a stand-alone 3D library) and direct rendering was expected from at most one client at a time (e.g. full-screen 3D game).

Over the time, due to the gradually increasing sophistication of both hardware and software, the set of responsibilities that had to be covered by the graphic driver significantly increased in complexity. Suddenly, there was a demand to have multiple applications directly rendering into the off-screen buffers through various accelerated drawing libraries, all running at the same time. For a while, graphic drivers kept up with this progression and gradually became more and more complex as they had to implement multiple interfaces and provide scheduling and memory management for their direct-rendering clients.

Since a lot of this new functionality was implemented over and over again in various graphic drivers, the reaction of mainstream operating systems to this trend was to extract such functionality, unify it and provide it as a reusable graphic driver infrastructure. To give real world examples, such graphic driver infrastructure is called DRI[8] in Linux and WDDM[9] in Windows. As can be seen in [6] and [8], in both cases, the infrastructure manages command queues of individual clients, schedules those command queues on the device and incorporates dedicated device memory into the system paging mechanism. In order to simplify the driver even more, there has been an effort to unify various drawing libraries over a single interface. This resulted in Gallium3D in Linux and DXGI[10] in Windows. Apart from the interface unification, [7] and [9] states that those new layers also provide certain low-level building blocks to simplify the implementation of drawing libraries (e.g. state tracker).

From the perspective of this thesis as was outlined in the scope definition, both graphic driver infrastructure and common drawing interface are not nearly as important as the components in the second and the third tier of the stack. For the graphic driver infrastructure, it means that it must provide frame buffer management and mode setting enumeration but the support for accelerated direct rendering can be left only in an early design phase because HelenOS currently does not contain a full-featured graphic driver. Since implementation of the full-featured graphic driver is obviously out of the scope of this thesis, already existing HelenOS drivers have to be reused and adapted against the new graphic driver infrastructure. The support for accelerated direct rendering could be implemented later together with the full-featured graphic driver as a part of a more narrowly focused project. As for the common drawing interface, it will be significant only once some of the existing drawing libraries are

[8] Direct Rendering Infrastructure
[9] Windows Display Driver Model
[10] DirectX Graphics Infrastructure

ported to HelenOS, and even then it will be reasonable to port them alongside with their common interface from the mainstream operating system.

3.2 Drawing Library

As there is quite a lot of drawing libraries available, it is first necessary to narrow down the kind of drawing libraries we are interested in. It is possible to distinguish three main categories – 3D libraries, 2D gaming libraries and 2D general-purpose libraries. While 3D libraries (e.g. OpenGL, Direct3D) are primarily intended for drawing 3D graphics, it is starting to be common in mainstream operating systems that they are also used by graphics servers or even by widget toolkits. Those libraries are however not suitable for this project because of their complexity and required acceleration. Then there are 2D libraries that are intended for game development (e.g. Allegro, SDL[11]). For this purpose, they provide also sound and input handling in addition to drawing 2D graphics. The common property of those libraries is that the drawing is usually accelerated and optimized for advanced animations. In contrast to that, general-purpose 2D libraries are rather focused on *device independence* and *resolution independence* despite being slower. Such libraries are suitable for drawing widget-based user interfaces, text and mostly static graphics. The important aspect is that the device and resolution independence helps in achieving WYSIWYG[12] effect and allows application not to be concerned whether its output is created by processor, graphic card, printer or plotter. Obviously, general-purpose 2D libraries are the ones that are of interest to this thesis.

In order to achieve device and resolution independence, drawing libraries provide abstract drawing interface (pens, brushes, fonts, color palettes, graphic primitives, etc.) and so called *device contexts* (also referred to as *rendering backends*) in which the actual drawing operations are carried out at a certain resolution. Apart from translating the abstract interface into the device driver operations, device contexts can also convert coordinates and colors used by the application into the ones that could be handled by the device (consider for example conversion of floating point coordinates or floating point pixels into integers, or adjustment of coordinates with respect to the different point of origin). There are usually three types of device contexts commonly found within the drawing libraries – i.e. contexts of printing devices, graphic devices and software renderers. Regarding the performance, optimized software renderers are sufficient for most of the scenarios of widget-based applications, assuming modern contemporary hardware. However, at least partial acceleration of the operations dealing with pixel buffers (copying, masking, blending, overlaying, etc.) can be still significant when the drawing library is used for window manipulation within the graphics server.

Depending on when the drawing operations are executed, it is possible to distinguish two modes of operation within various drawing libraries. In the simpler mode of operation, or so called *immediate mode*, drawing operations are instantly sent to the device context and from there to the device driver – i.e. an operation is applied onto the resulting visual medium and then forgotten. In contrast to that, more sophisticated drawing libraries support also a *retained mode* in which the drawing operations are remembered as part of a model allowing to modify or reconstruct the scene again later. In other words, retained mode allows decoupling the scene building from the actual rendering to the visual medium.

To give an example of a retained-mode model used in practice, consider storing a scene as a tree-like data structure, where each leaf corresponds to some graphic primitive and each node (including leaves) is represented by parameters describing how to render that particular part of the overall scene (transformation matrix, Cartesian coordinates relative to the parent, etc.).

[11] Simple DirectMedia Layer
[12] What You See Is What You Get

Alternatively, scene creation can be modeled as a directed acyclic graph, where nodes represent the operations and edges represent intermediate images, or simply as a recorded list of applied drawing functions and their arguments.

When looking at the historical development of mainstream operating systems, there is a clear trend in improving the flexibility and the quality of visual output created by their native 2D libraries. Originally, the libraries were based just on the device contexts and supported only immediate mode. Over the last decade, however, most of these established libraries were superseded by new libraries that, in addition, support also retained mode and vastly improve the visual quality. It is common for those new libraries that they represent coordinates and pixels by floating point data types, render the scene into off-screen buffers, apply antialiasing to smooth down edges and utilize double buffering to avoid image tearing. Real world example of the described progress is represented by the transition from QuickDraw to Quartz 2D in MacOS, as illustrated by [10] and [11]. On Linux and Windows, traditional drawing libraries like Xlib [12] and GDI[13] [13] are still available but becoming less relevant for application development as they serve only as one of the backends for modern drawing libraries, like for example Cairo.

Speaking of Cairo, it could be considered as a state-of-the-art drawing library in many aspects. According to Cairo documentation [14], it supports multiple platforms, uses vector-based scene description and provides multiple rendering backends each suitable for different situation. As was already mentioned, even the legacy drawing libraries can be used as such rendering backends to provide backwards compatibility. Then, there are rendering backends for SVG[14] format and PostScript language, for both of which it is convenient to have vector-based scene description. For the rendering of pixel-based application windows, Cairo can use its accelerated OpenGL backend or an embedded software renderer called Pixman.

Last thing that deserve attention when exploring the drawing libraries is font rendering. What was originally embedded within a drawing library is nowadays isolated into the dedicated font library and only referenced from the actual drawing library. Such step was logical because of the increasing complexity of font rendering – consider strong demand for high quality vector fonts, various font file formats, Unicode encoding, non-Latin characters etc. To give examples of such font libraries, there are multi-platform open-source libraries like FreeType or Pango, or platform-specific libraries like DirectWrite on Windows or Core Text on MacOS. By linking against a dedicated font library, drawing library is obviously freed from a significant burden. However, for example Cairo provides, in addition to external bindings, also its own less sophisticated font support to increase the portability of the library.

Now, it is finally possible to outline how to bring the functionality covered by a drawing library into HelenOS. Although porting of some existing library to HelenOS is out of scope of this thesis due to the intricacies related to any porting effort, let us consider such possibility for the sake of some future stand-alone project. Considering which existing library would be suitable for HelenOS, Cairo seems to be a particularly sound candidate as it is written in C language, it is open source and it is sufficiently portable according to the preliminary investigation. From the HelenOS viewpoint, it should be enough to improve the math support in C library and perhaps create a fork of Cairo to adjust its build system and remove its non-optional dependency on GLib (which is seemingly unnecessary if bindings to other languages are not important). Overall, such minimalistic port of Cairo would provide only a single software rendering backend based on Pixman and a basic embedded font engine. As an alternative to Cairo, there is an interesting drawing library from Google, called Skia [15]. Although Skia

[13] Graphics Device Interface
[14] Scalable Vector Graphics

is also highly portable and provides similar features as Cairo, it is written in C++ which is currently not supported by HelenOS.

Until some existing library is ported to HelenOS, the new graphics stack obviously needs its own native drawing library. However, to fit into the time frame available for the thesis, only the most important functionality of the drawing library can be implemented. Assuming the drawing library shall be implemented in a lazy manner according to the needs of graphics server and widget toolkit, the most important part seems to be a routine for pixel transfers. If such routine is sophisticated enough (i.e. supporting affine transformations, rectangular clipping, bitmap masking, basic filtering and basic compositing), it will enable the advanced manipulation of pixel surfaces and the drawing of basic graphic primitives (rectangles, horizontal lines, vertical lines and bitmap font glyphs), which should suffice for both the window handling within a graphics server and the rendering of basic widgets within a widget toolkit. Therefore, the initial implementation does not necessarily have to contain any advanced drawing primitives or retained scene description.

3.3 Graphics Server

As a central component of a graphics stack, graphics server is responsible for integrating the output from its clients into a single logical desktop. While the properties of such logical desktop can exactly match the frame buffer, it is usually the case that frame buffer serves only as a viewport into a much larger logical desktop. There are two commonly used approaches how to define the logical desktop – *switching desktop* and *oversized desktop*. In the switching desktop approach, there are multiple isolated desktops each of which usually matches the frame buffer dimensions. On the contrary, oversized desktop approach considers desktop as a single continuous surface which is either significantly larger than the frame buffer or even logically infinite. While the oversized desktop is suitable for multi-monitor setups as it allows seamless transition of client windows between the monitors, switching desktop is convenient when user needs multiple isolated work spaces. Note that both of these approaches can be combined together by providing switching between multiple oversized desktops.

Regarding the output from clients, there has been much development in the last decade on the form in which such output is communicated between the client and the server. Originally, it was common that server represented the scene as a very fine grained hierarchy of sub-client objects. In other words, server was aware of the internal structure of the client's user interface. To give an example, X server [16] on Linux represents the user interface of all the clients as a tree of logical regions within the desktop. The visual content of those regions is drawn by the X server itself according to the commands sent from the clients – i.e. client is the one who controls visual appearance of widgets but the server is the one who actually draws it. Even more centralized approach could be found within older versions of Windows operating system – there, USER subsystem contained both drawing library and widget toolkit, so the client just requested to draw a certain widget at certain coordinates and referred to it by an opaque handle afterwards.

The common characteristic of the older graphics servers, or so called *stacking graphics servers*, is that desktop manipulation must be orchestrated with the help of its clients – i.e. whenever user moves the desktop window of some client, server must notify any client whose been (partially) exposed by such action to find out how to fill in the uncovered areas. While such cooperation between the server and its clients is efficient with respect to memory and computing resources (no pixel buffers need to be allocated or copied), it is not very robust because just a single non-cooperative client has the potential to destroy the entire desktop.

Since a stacking graphics server must be instructed by its client on how to assemble or draw the exposed areas, there must be a rich communication protocol established between both parties. Interesting examples of such protocols can be found in some no longer existing projects – while Fresco windowing system was based on CORBA middleware, NeWS[15] from Sun Microsystems leveraged Postscript language so it could be actually programmed by clients as described in [24]. However, history shows that such protocols were too complicated to be widely adopted. Instead, more straight-forward protocols were preferred, most notably X protocol on top of which Xlib [12] drawing library is built (Xlib serves as a proxy to the drawing operations embedded inside X server).

While stacking graphics servers served its purpose in the past, increasing sophistication of hardware and software both motivated and allowed to replace them by so called *compositing graphics servers*, or compositors in short. Characteristic property of compositors is that the structure of client's user interface is completely opaque to them. More specifically, each client has its own off-screen buffer that is shared with the server to provide read-only access – i.e. client is fully responsible for drawing into the buffer and only informs the server about what parts of it need to be refreshed in the frame buffer to keep the desktop updated. This approach has three important consequences. First of all, the communication between server and client is very simple – i.e. input events are propagated from the server to the client and damage notifications from the client to the server. Secondly, server has a complete freedom in how to incorporate client off-screen buffers into the desktop – e.g. it can apply various transformations to them, make them transparent, reuse them on multiple places etc. And finally, the client is free to use any drawing library or widget toolkit it wants.

Clearly, compositing brings superior flexibility over stacking; however there might be concerns about its overall performance as there is a lot of pixel copying involved. It is important to realize, that clients do not have to redraw themselves once they are re-exposed – they draw into their off-screen buffers only when there is some internal reason to change their visual appearance, without any communication overhead. Assuming that server does not use any transformations, the exposure of client surface could be resolved by a plain pixel buffer copy directly within the server, again without any communication overhead. Note that pixel copying by itself is not significantly more expensive as it involves executing fewer instructions overall and at most twice as many memory references than any drawing operation on the same area, assuming that the drawing operation involves filling the background of the area with some color. In practice, however, compositors apply quite complex transformations on the client buffers or even consider the desktop as a 3D scene. Therefore, drawing libraries used by compositors are usually either fully accelerated or at least partially accelerated to provide hardware assisted pixel transfers (sometimes referred to as *blitting*). The important assumption in the analysis above is that the client's off-screen buffer is shared between the client and the server via a virtual memory mapping mechanism that does not involve any copying of the actual physical memory. This however does not hold in the networked scenario where client and server are located on different machines. In this case, the rich communication protocols of stacking graphics servers might outperform compositing, because the drawing commands are much smaller than the actual rendered pixel buffers. Although the pixel buffers might be transferred over the network in a compressed form, it is still the most notable drawback of compositing.

As of writing this, all major mainstream operating systems incorporated compositing into their graphics stack. On MacOS, this transition is represented by introduction of Quartz Com-

[15] Network extensible Window System

positor [18]. Similarly, Windows was equipped with DWM[16] compositor [17]. On Linux, proper compositor is yet to be adopted. In the meantime, X server was improved by various extensions and third-party window managers to support compositing as well. The development actually got to the point where X server no longer fulfills the task it was originally designed for and acts only as an unnecessary communication hub between input subsystem, graphic driver, clients, window manager and all the extensions. As a reaction to this situation, Wayland project [19] recently introduced new communication protocol, compositor framework and demonstration compositor (based on OpenGL) with the intention to replace X server ecosystem altogether. Currently, there are signs that it might be indeed adopted by major Linux distributions.

Despite the wide adoption of compositing in mainstream operating systems, its potential is often not fully leveraged to increase ergonomics or productivity but rather to provide more eye candy. As written in [21], authors of Metisse compositor took an innovative approach to what features should the compositor provide to the user. Instead of providing purposeless 3D effects, Metisse is more focused on how to efficiently distribute the screen space among the client applications in a flexible and intuitive manner. As can be seen in the screencast [22], Mettise can apply affine transformations like scaling or rotation on both viewports and client windows which is especially useful for various mobile devices or horizontal screen surfaces.

Another interesting compositor worth mentioning is the Nitpicker from Genode framework, as described in [23]. Apart from giving a security credit to compositing in general, the article explains that Nitpicker considers the client pixel buffers and the client desktop windows (called *views* in the article) as two independent entities, which is not the case for most of the other existing compositors. This generalization provides another level of flexibility, for example when dealing with a virtual machine client that is hosting another operating system. The pixel buffer representing a desktop of the virtualized operating system could be cut into multiple views, each view perhaps mapped to an area with the output from some virtualized application. This way, the Nitpicker is able to present those virtualized applications in a way indistinguishable from the native clients.

Before focusing on HelenOS, let us revise the extent of the compositor responsibilities. As was already explained, both drawing operations and widgets moved from the graphics servers to the clients. Compositing graphics server must definitely redistribute input events, manage client windows and incorporate their pixel buffers into the frame buffer, as all of those things are tightly connected together due to the various transformations that might be applied to the windows. In other words, compositor must be able to decide what window is the receiver of the input event (e.g. mouse click) and how to react to it – whether to apply some transformation on the window or redirect the event to the client. Similarly, it must be able to decide what part of the frame buffer is affected by the damage received from the client that is perhaps having a heavily transformed window.

One responsibility, which is window decorations, is however causing some controversy among the compositor creators as it is not clear whether it should be assigned to the compositor or to the client-side widget toolkit. In the stacking graphics servers, windows decorations are taken care of within the server by reparenting the client's top-level region in the desktop scene tree. Such an approach is natural for a stacking server because similarly as the server understands the internal structure of the client's user interface, it can do the same for the decorations. In contrast to that, compositor is not equipped to do such thing naturally as it works just with the pixel buffers and does not contain any scene tree. Surely, compositor could be linked to a widget toolkit in order to take care of the decorations, but it would damage purity

[16] Desktop Window Manager

of its design. On the other hand, taking care of decorations seems completely natural for the widget toolkit within the client, assuming it is able to tell the compositor when the decoration element was clicked on. From the user viewpoint, server-side decorations can provide consistent look-and-feel among all clients and can prevent decorations from being affected by certain transformations. However, it is not clear whether such properties are actually advantages or drawbacks. While new compositors like Wayland or Nitpicker prefer client-side decorations, DWM compositor or any X-based compositor are using primarily server-side decorations. Nevertheless, server-side and client-side decorations are not mutually exclusive and can be both available within the same graphics stack. However, client-side decorations are easier to implement and could be considered more natural choice for a compositing graphics server.

To conclude the section, let us consider what to choose from all the mentioned properties for HelenOS new graphics server. Because of the reasons described above, it shall be definitely a compositing graphics server. While it would be very beneficial to support as flexible viewports and client windows as could be seen in Metisse and Nitpicker, such goal might be too ambitious for the initial implementation, assuming limited manpower and lack of full-featured accelerated drawing library. It is therefore reasonable to choose a subset of these features that would be most practical for HelenOS usage scenarios in a near future. For now, it should be sufficient to support multiple viewports (to cover multi-monitor scenario) that could be moved (but not scaled) within the oversized desktop. The reason for not providing viewport scaling right away is that it would be unbearably slow without aggressive optimization or hardware acceleration (consider significantly zoomed out viewport with the client resized to fill it entirely), not to mention that HelenOS is mostly being executed within a virtual machine. The reason for choosing an oversized desktop rather than a switching desktop is that it is synergetic with the multi-viewport design and could be extended later to provide switching desktop as multiple oversized desktops. Regarding the client pixel buffers, they shall have an identity relationship with the desktop windows as there is currently no practical motivation for Nitpicker's views in HelenOS and if such motivation emerges in the future it should be easy to implement without any significant redesigning. When incorporating pixel buffers into the desktop, HelenOS compositor shall allow them to be moved, resized, scaled, rotated and made partially transparent – all of which shall be facilitated by the initial implementation of the drawing library. As for window decorations, compositor shall assume client-side decorations and shall be able to cooperate with the client's widget toolkit when such decorations are grabbed or clicked by the mouse. Server-side decorations shall not be a part of the initial implementation, because they are more complicated and do not bring any significant advantage, however they might be added to the compositor later if necessary.

3.4 Widget Toolkit

Widget toolkit is a very good example of a library that can significantly leverage object oriented principles (inheritance, polymorphism, etc.) and other advanced features of the modern programming languages (exceptions, meta-programming, reflection, etc.). Because of that, most of the existing widget toolkits are implemented in languages like C++ or Java, and often they even extend those languages with new keywords and other constructs to decrease the verbosity of the resulting source code written by an application programmer. Therefore, only a very few widget toolkits are implemented in C language since they would have to also implement their own object oriented extension over the C. In practice, such object extension is either embedded in the widget toolkit or more often it is available as an independent object library – for example Xt[17] from Linux environments, BOOPSI[18] from AmigaOS or multi-platform GLib.

Widget toolkits can be classified according to whether they are native or extraneous toolkits on a certain operating system. In the case of cross-platform widget toolkits, they can be actually both native toolkit on one system and extraneous on the different one. While native toolkits must use a drawing library in order to draw the widgets, extraneous toolkits have more freedom in this regard – they can either be implemented also on top of a drawing library (and possibly emulate the look and feel of the host platform) or they can use the native toolkit and map their own widgets onto the native ones. Contemporary widget toolkits usually support multiple such backends – i.e. they can work on top of various drawing libraries and widget toolkits, both native and extraneous. Regarding contemporary widget toolkits, it is also important to note that some of the projects that are commonly referred to as widget toolkits actually provide much more functionality than just widgets – there is often support for databases, multimedia, networking, web, scripting etc. If the toolkit providing such a broad spectrum of functionality also supports multiple platforms, it can be considered as an *operating system abstraction layer* from the application point of view.

When exploring existing widget toolkits, there are actually not so many radically different properties or mutually exclusive approaches that could be identified within other graphics stack components from the previous chapters. Rather, all widget toolkits are based on the same basic architecture and only extend it to various levels of abstraction and sophistication. In the core of any widget toolkit, there is object oriented hierarchy of widgets, widget scene tree and event loop. Events that are queued within an event loop can originate from both the graphics server and any instantiated widget. Events are dispatched one-by-one from the queue and delivered to a particular widget either directly or by traversing the tree in a top-bottom manner according to some event-specific property (e.g. mouse events are delivered by comparing the pointer position with widget's position and size). A reaction of the widget to the event can be then propagated both upwards and downwards within the scene tree, so other widgets affected by the change can adapt. While it is possible to create a complex user interface with just single scene tree and single event loop, modern widget toolkits support multiple pairs of scene tree and event loop, each pair assigned to a separate thread.

An important property of any widget in the scene tree is its size and position with respect to the root of the tree. The placement of widgets can be specified by an application programmer either directly by hardwiring the coordinates into the application code or logically by leveraging the so called *layout widgets*. Layout widget is intended to be located in a non-leaf position of the scene tree and to dynamically distribute the space assigned to it among its descendants according to the chosen logical layout (e.g. grid, horizontal stack, vertical stack, etc.).

To make widget communication and application code more flexible, contemporary widget toolkits provide the so called *signal-slot mechanism*. Without signals and slots, message sender must specify both the message itself and the receiver of the message, which makes the receiver hard-wired into the sender. Signal-slot mechanism alleviates this by letting both sender and receiver to be concerned just with the message, and let the addressing be done by a third party – e.g. consider a button widget having a signal that reports the button has been clicked on, a label widget having a slot that changes its caption text and finally an application code (i.e. the third party) that connects the signal and the slot of the two particular instantiated widgets. Signal-slot mechanism can support synchronous communication, asynchronous

[17] X Toolkit Intrinsics
[18] Basic Object Oriented Programming System for Intuition

communication or both. Synchronous signal is usually implemented as ordinary function call and asynchronous signal as posting the event into the event loop.

Recently, widget toolkits started providing even more sophisticated abstractions. As criticized in [25], widget toolkits employ too simple model for mapping events to actions when not considering the internal state of the application (i.e. whether a particular action is valid with respect to the internal state). In other words, the responsibility to handle the state and consistency of the user interface is left entirely to the programmer without forcing any formal model. In practice, such approach leads to inconsistencies and bugs. Therefore, some widget toolkits optionally allow describing the user interface more formally as a finite state automaton. Another novel feature of widget toolkits is the possibility to program the user interface dynamically at runtime without recompiling the application – i.e. widget toolkit contains an interpreter of some scripting language that describes the user interface. When used correctly, such added flexibility can be leveraged to decouple application business logic from the visual appearance.

When looking around for a widget toolkit that could be ported to HelenOS, there are not as many possibilities as one would expect. Established cross-platform widget toolkits like Qt [26], GTK+[19] [27] or wxWidgets [28] are too complex (to various extent they can be all considered as operating system abstraction layers), too large and have too many dependencies to be considered reasonable candidates for HelenOS at this point. Then, there are more light-weight toolkits like IUP [29] that seem promising at first look, but closer examination reveals they are built on top of the native widget toolkits. Basically the only toolkit that is sufficiently light-weight and requires just some drawing library underneath is FLTK[20] [30]. Its only downside is that it is implemented in C++ language which is currently not supported by HelenOS. However, FLTK avoids using advanced runtime features of C++ like exceptions or RTTI[21] in order to increase its portability, which is indeed convenient in case it would be ported to HelenOS in the future.

As with the drawing library, in order to keep the thesis scope bearable, HelenOS graphics stack shall be equipped with its own minimalistic, yet extensible and practically usable widget toolkit. While the initial implementation shall be driven primarily by the needs of the initial set of applications, which should include application launcher and terminal emulator, there shall be enough functionality and examples so the developers of the future applications could focus on implementing their own widgets (and perhaps drawing operations) not having to consider the toolkit (or drawing library) as a whole. Therefore, the widget toolkit implementation shall contain the necessary core infrastructure, but does not necessarily have to offer a wide range of widgets – assuming the mentioned applications, it is enough to implement just label widget, button widget and some dynamic layout widget (probably a grid). Note that since the implementation have to be based on C language, it is necessary to partially simulate some object oriented features, at least inheritance and polymorphism. Regarding the core infrastructure, it shall support multi-window applications by allowing multiple pairs of scene tree and event loop, each pair assigned to a separate fibril. Because the goal is to create a client-side widget toolkit, it will have to communicate with the compositor to receive input events, to report damaged pixel buffer areas and to inform which window decorations are grabbed by mouse. Furthermore, the initial implementation shall provide a signal-slot mechanism, both synchronous and asynchronous.

[19] GNU Image Manipulation Program Toolkit
[20] Fast Light Toolkit
[21] Run-time type information

4 Design

This chapter describes the architecture and design of the new HelenOS graphics stack. First, the stack is described as a whole. Then, detailed design of individual components is provided.

4.1 Overall Architecture

Since the basic architectural traits of modern graphic stack were hinted in the previous chapters, it is now possible to focus specifically on of what had to be done in order to transform the original HelenOS graphics subsystem into the proper graphics stack while keeping the already existing components and applications available.

As was already mentioned in the introductory chapter, HelenOS graphic drivers were originally embedded into the fb server. In this regard, fb server acted as a graphic driver infrastructure, because it implemented the communication routines and expected drivers to implement just a small number of device-specific functions (e.g. how to draw a character into the frame buffer). Most of the drivers within fb server provided just text modes intended for character-based output devices. The only driver that actually manipulated individual pixels was the basic VESA driver. On top of the fb server, there was the console server that distributed keyboard events (received from the input server) to the client applications and populated screen-sized scrollable character buffer (shared with the fb server) according to the output from the client applications. Other than that, the console server regarded each client to be running in its own dedicated tab. The tabs could be discovered by clients via the loc server. To support tab switching, each tab had to be backed by its own off-screen character buffer. Notably, console server did not support dynamic runtime resizing of the character buffers nor did it store any history that had been scrolled away. The overall situation is illustrated in **Figure 3**.

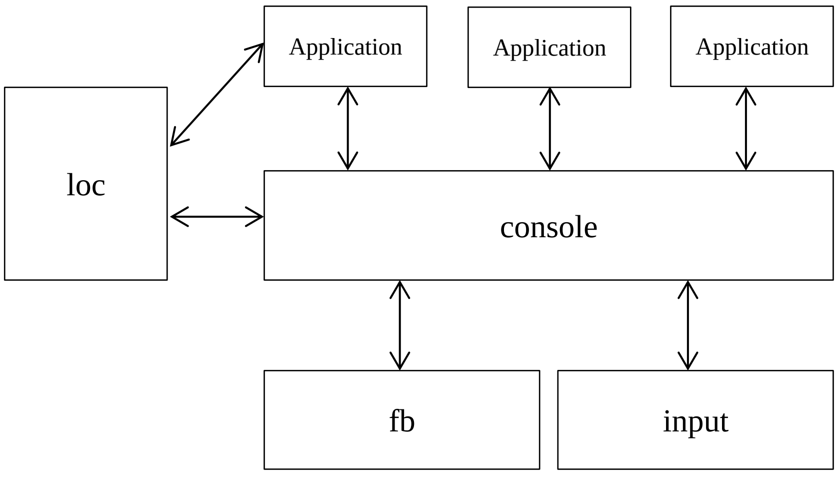


Figure 3 Original text-based graphics subsystem of HelenOS.

From the beginning, it was clear that fb server is insufficient to be a foundation of the new stack and must be replaced by a graphic driver based on the device driver framework and graphic driver infrastructure. Instead of rendering characters from the single character buffer shared with the console server, new drivers had to be capable of sharing multiple pixel buffers with the compositor and also of refreshing the damaged areas in the corresponding frame buffers. However, keeping the old text-based console server functional was also desirable in order to support character-only output devices. At the same time, it was necessary to ensure that the existing console applications would be compatible with the new terminal emulator running as a compositor client. Now, there are two approaches how to do that. Both approaches were actually implemented, the first was tried within the feature branch, second was applied when merging the branch into the mainline.

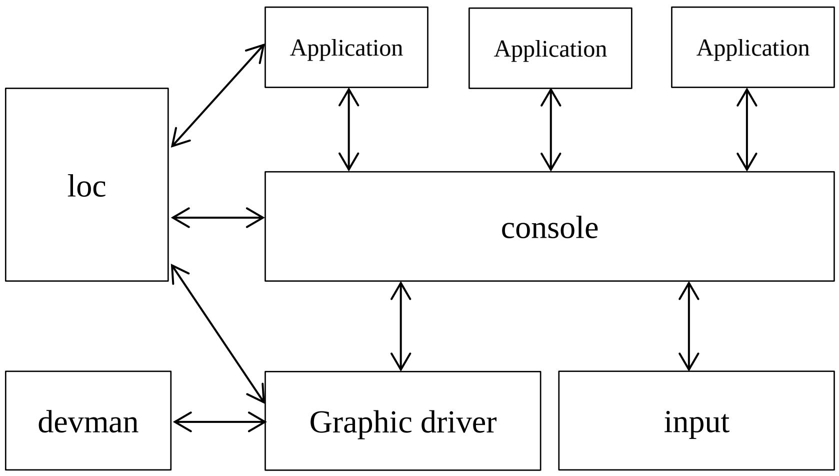


Figure 4 Console mode of the new graphics stack (feature branch version).

In the first approach, the architecture is driven by code reuse and strict verticality of the stack. It is assumed that already powerful console server will be further improved to support history and dynamic resizing, in addition to multiple tabs and scrolling. Because it does not seem reasonable to reimplement all the console functionality in the terminal emulator, the console server was ported from the fb server to the new graphic drivers and the terminal emulator actually emulates the driver to accommodate the console server instead of emulating the console server to accommodate console clients. For this to work, graphic driver infrastructure abstracts from pixels and characters and considers the shared buffers as opaque cell arrays, where the cell type is part of the mode setting provided by the actual graphic driver (i.e. driver can support multiple pixel modes and text modes).

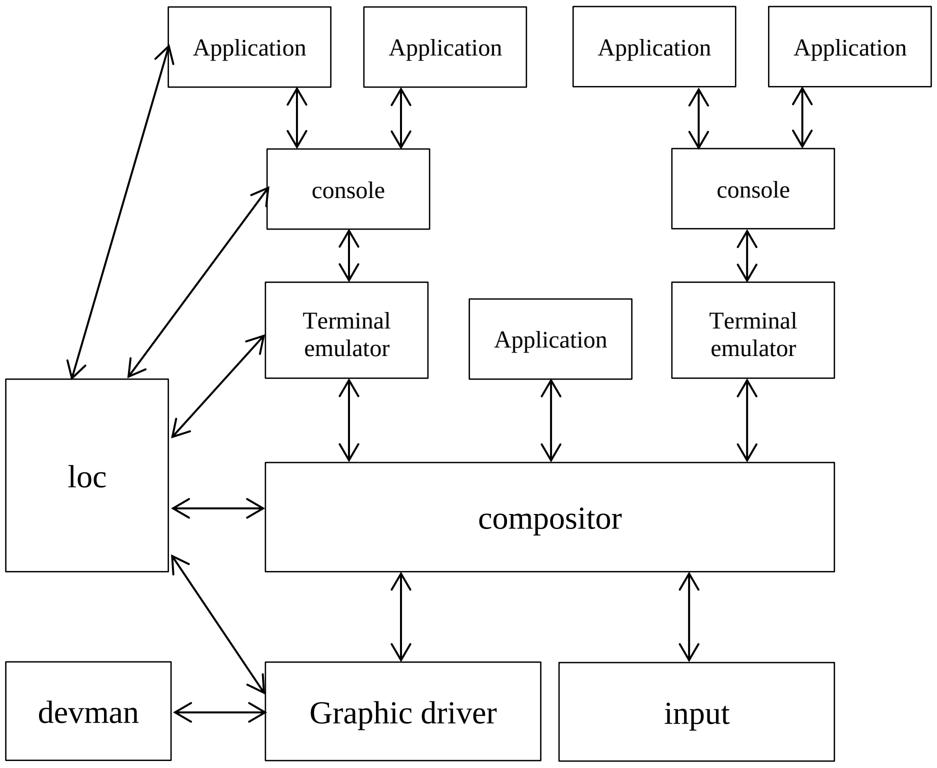


Figure 5 Compositor mode of the new graphics stack (feature branch version).

As can be seen in **Figure 4** and **Figure 5**, first approach considers the compositor and console server as two mutually exclusive components in the second tier of the stack. The disputable property of the first approach is that each terminal window on the desktop requires its own instance of the console server running – that is one process for the terminal emulator, second process for the console server and then another process for the actual application – altogether three layers of inter-process communication until the information gets to the compositor. Because of the above reasons and perhaps other more subjective reasons, the first approach was not widely accepted within HelenOS developer community despite its cleaner design.

Instead, more pragmatic and performance oriented architecture was devised during merging of the feature branch into the mainline, as illustrated in **Figure 6**. There, only pixel mode drivers were ported to graphics driver infrastructure (and device driver framework) while text mode drivers were kept in the old fb server which was renamed to *output server*. This way, console server and compositor server can theoretically coexist in the second tier of the stack, if the input server would support such scenario. Regarding the terminal emulator, now it must emulate the entire console server, which leads to some code repetition, but the advantage is there are only two processes and communication hops required per application to reach the compositor. Because of this architectural change, some features from the feature branch are not ported to the mainline yet – i.e. terminal emulator does not support multiple tabs and dynamic resizing. If not said otherwise, the following chapters describe the mainline version of the graphics stack, so it can serve as a documentation for the future developers.

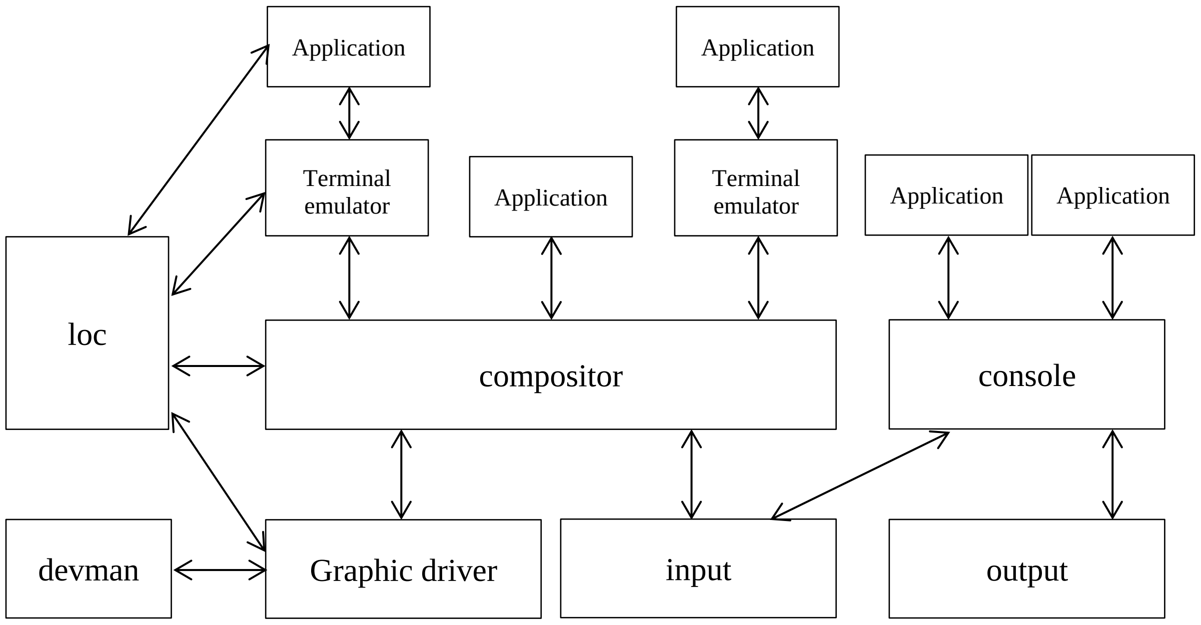


Figure 6 Final mainline version of the new graphics stack.

Before focusing on the stack components in detail, let us briefly describe how the stack works as a whole. During system boot, devman server spawns the graphic driver which in turn finds all output devices available through the graphic device and advertises them via the loc server. When compositor is launched, it connects to the loc server, discovers the advertised input and output devices, establishes communication with both the graphic driver and the input server, initializes the output devices according to a default mode setting and finally advertises a window bootstrapping mechanism via the loc server. Any executed application then connects to the loc server to discover the window bootstrapping mechanism, establishes communication with the compositor, asks the compositor to create new window and input queue, initializes the newly created communication channels, constructs the user interface and finally starts executing an application code. Input events are then propagated from the input server, through the compositor server, up to the application. Damage messages are propagated from the client-side widget toolkit, through the compositor, to the graphic driver. There is an off-screen buffer shared between each application and the compositor. Furthermore, there is one off screen buffer per output device shared between the compositor and the graphic driver. Only the graphic driver has an access to the actual frame buffers and therefore has the responsibility to convert the compositor pixels (ARGB32) to the format required by the selected mode setting. Therefore, any pixel written by the client-side drawing library must be copied two more times until it appears on the screen – during the first copy, the pixel is exposed to the compositor transformations, during the second copy, it is converted for the output device.

4.2 Graphic Driver Infrastructure

The graphic driver infrastructure is implemented within the *libgraph* library which is intended to be linked to the graphic driver. In fact, the library defines the interfaces that have to be implemented by the driver. For the following description, let us assume a hypothetical scenario in which the graphic driver is responsible for handling two graphics cards – one having two video outputs, the other having only one video output. Overall, there would be two graphic devices with acceleration capabilities and up to three output devices. To represent the devices for the second tier of the graphics stack, libgraph defines two data types – *visualizers* (for output devices) and *renderers* (for graphic accelerators). The given example is illustrated in the **Figure 7** – i.e. three visualizers and two renderers. Regarding renderers, it is up to the graphic driver whether to represent the two graphic cards as one renderer and handle load balancing internally, or as two renderers and leave the responsibility for load balancing to the upper tiers of the stack, or whether to make acceleration capabilities accessible at all (i.e. no renderer).

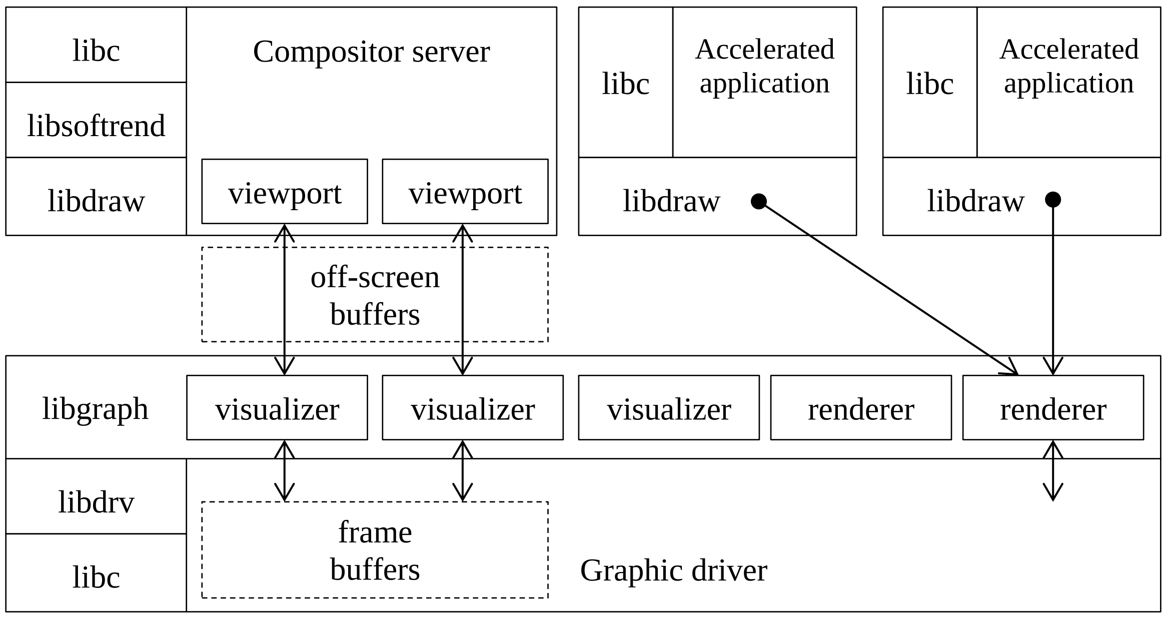


Figure 7 Detail of the interface between the first and the second tier of the graphics stack.

Since the whole concept of accelerated direct-rendering applications in HelenOS graphics stack is currently left just in a very early design stage and not actually implemented, let us briefly elaborate first on how it could be done. The renderer is supposed to manage client command queues, memory management and scheduling. For each client, there would be connection fibril, command queue, condition variable and device context. Connection fibril would put the command into the command queue and block on the condition variable. Command queues would be consumed by a scheduling thread that decides what client to serve and then switches the corresponding device context into the accelerator. After that, the command would be executed on the accelerator. Once the command execution is finished, the condition variable would be signaled so the connection fibril could answer the client. Operations that are not implemented in the hardware of the accelerator could be carried out by worker threads managed by the scheduling thread. If the physical memory of the accelerator is mapped to the address space of the driver, it could be extended by allowing scheduling thread to page out the device memory into the system memory and to handle subsequent page faults. It is not entirely clear which parts should be implemented in the libgraph and which in the device driver. As a rough sketch, the connection fibril routine, command queue, scheduling strategies and memory paging should be handled by the libgraph. The device context, context switching routine and the actual graphic operations should be provided by the driver.

As for visualizers, these are completely implemented and ready to be used by the compositor. The visualizer is basically a container to store mode setting, device context and pixel buffer shared between the driver and the compositor, as can be seen in the following code snippet:

```
typedef struct {
   // ...
   list_t modes;
   vslmode_t current_mode;
   sysarg_t default_mode_idx;
   visualizer_ops_t ops;
   pixelmap_t cells;
   void *dev_ctx;
} visualizer_t;
```

Most of the visualizer management and its communication with the compositor is implemented by the libgraph routines. The driver is only expected to partially initialize the visualizer, which mainly includes filling the mode setting list and creating the device context, and to implement the following device-specific functions:

```
typedef struct {
   int (* claim)(struct visualizer *vs);
   int (* yield)(struct visualizer *vs);
   int (* change_mode)(struct visualizer *vs, vslmode_t new_mode);
   int (* handle_damage)(struct visualizer *vs,
       sysarg_t x, sysarg_t y, sysarg_t width, sysarg_t height);
   int (* suspend)(struct visualizer *vs);
   int (* wakeup)(struct visualizer *vs);
} visualizer_ops_t;
```

That is, besides initialization functions and power management functions, the driver is mainly responsible for refreshing the frame buffer upon damage and for adjusting the frame buffer whenever the mode setting change is requested. As for the mode setting, it is defined by the following data type:

```
typedef struct {
   sysarg_t index;
   sysarg_t refresh_rate;
   aspect_ratio_t screen_aspect;
   sysarg_t screen_width;
   sysarg_t screen_height;
   aspect_ratio_t cell_aspect;
   cell_visual_t cell_visual;
} vslmode_t;
```

Notice the specification of the pixel visual type. While most of the graphics stack components use ARGB32 pixels as a preferred and default choice (and currently also the only one), graphic driver is free to use any pixel type it wants as long as it is able to convert the types between each other.

The above description shows how the graphic driver infrastructure looks from the driver perspective. From the other side, the compositor can invoke the following functions to manipulate visualizers managed by the graphic driver infrastructure.

```
int visualizer_claim(async_sess_t *);
int visualizer_yield(async_sess_t *);
int visualizer_enumerate_modes(async_sess_t *, vslmode_t *, sysarg_t nth);
int visualizer_get_default_mode(async_sess_t *, vslmode_t *);
int visualizer_get_current_mode(async_sess_t *, vslmode_t *);
int visualizer_get_mode(async_sess_t *, vslmode_t *, sysarg_t index);
int visualizer_set_mode(async_sess_t *, sysarg_t index, void *cells);
int visualizer_update_damaged_region(async_sess_t *,
    sysarg_t x, sysarg_t y, sysarg_t width, sysarg_t height);
int visualizer_suspend(async_sess_t *);
int visualizer_wakeup(async_sess_t *);
```

As was mentioned in the previous chapter, visualizers are registered to the loc server during graphic driver initialization. From there, compositor can find them and connect to them via inter-process communication. Once the connection is established, compositor claims the visualizer, discovers the default mode setting and sets the default mode. Claiming the visualizer triggers its initialization, both within the graphic driver infrastructure and within the graphic driver. The only thing that is not prepared after claiming the visualizer, are the mode-specific parts of the visualizer. Setting the mode involves preparation of the frame buffer within the driver and allocation of a matching pixel buffer to be shared between the visualizer and the compositor. After the mode is set, visualizer is prepared to receive damage events from the compositor. Apart from retrieving the default mode, notice that compositor can also enumerate through the other modes available for a particular visualizer. While such feature is currently not utilized, it is intended for the future use once there will be a desktop environment in the graphics stack through which the user will be able to switch the mode setting by himself.

The last important thing that could be noticed in the **Figure 7** is that the driver is linked to *libdrv* library, which makes it part of the *device driver framework*. Detailed information about the driver framework and how to use it could be found in [5]. In short, the driver is invoked by the devman server and initialized by generic functions within libdrv. Apart from the generic interface provided by libdrv (driver addition, driver removal, device addition, etc.), driver is also expected to define its own interface for device-specific functions. In this regard, libgraph serves as an additional abstraction layer between libdrv and the actual graphic driver because, as was said above, the graphic driver must implement interfaces dictated by libgraph (i.e. visualizer and renderer interface). Although the driver is free to implement even its own interfaces, the minimal requirement for the graphic driver to work is to implement the visualizer interface and, in order to pass the control from libdrv functions to libgraph functions, it is necessary to include `ops/graph_dev.h` from libdrv and to provide the following code snippet:

```
static graph_dev_ops_t graph_vsl_dev_ops = {
   .connect = (connect_func) &graph_visualizer_connection
};

ddf_dev_ops_t graph_vsl_device_ops = {
   .interfaces[GRAPH_DEV_IFACE] = &graph_vsl_dev_ops
};
```

There, the `graph_visualizer_connection` is the entry function of the libgraph. While it is preferable for the driver to be based on the device driver framework, libgraph supports also the situation where the libdrv is not present. This can happen for example in a driver emulator that is not part of the device driver framework. As a substitute for similar functions normally provided by the libdrv, libgraph implements the following functions:

```
visualizer_t *graph_alloc_visualizer(void);
int graph_register_visualizer(visualizer_t *);
visualizer_t *graph_get_visualizer(sysarg_t);
int graph_unregister_visualizer(visualizer_t *);
void graph_destroy_visualizer(visualizer_t *);
```

In such situation, the driver emulator would have to register the main communication function to the loc server on its own – only this time, it would be `graph_client_connection` function instead of `graph_visualizer_connection`.

4.3 Drawing Library

There are currently two libraries that together create a drawing library – *libdraw* and *libsoftrend*. While libdraw is supposed to contain the actual drawing library, libsoftrend is intended to provide various basic routines and building blocks reusable by the software renderers. In case the libdraw becomes accelerated in the future, the functionality from libsoftrend will be provided by a graphic driver. As of writing this, libsoftrend offers the following:

- Support for 2D affine transformations – i.e. translation, scaling, rotation, multiplication and inversion of 3x3 matrices. Rotation is limited to 90-degree steps due to the missing trigonometric functions within HelenOS libc.
- Porter-Duff [20] compositing operators. Currently, the only non-trivial operator implemented is the OVER operator (sometimes referred to as *alpha blending*) which can be used to create a transparency effect.
- Texture filtering methods. Only the nearest-neighbor method is a part of the initial implementation, mainly for its simplicity and performance. The inherent blockiness of the nearest-neighbor method can be resolved by implementing more sophisticated method like bilinear filtering.
- Pixel format conversions. Wide range of 8-bit, 16-bit and 32-bit conversions is already available because the routines were mostly adapted from the original HelenOS graphic subsystem.

The drawing library is currently completely pixel based. With some effort, it should be possible to supplement it with a vector support. Alternatively, libdraw as it is could be used just as one of the backends for the entirely new vector library built on top of it. Such approach would be also more in line with the potential porting of some existing drawing library to HelenOS.

The drawing model of libdraw is based on three principal objects – *surfaces*, *sources* and *drawing contexts*. The surface is a thin wrapper over a raw pixel buffer. It tracks the damaged region of the buffer and is able to establish the sharing of the buffer between multiple servers. Damaged region is currently tracked as a bounding rectangle of all the damaged pixels. In the future, more fine-grained tracking could be implemented. From the perspective of the drawing model, surface serves either as a drawing canvas onto which the library applies individual drawing operations or as a template that is transferred onto another surface.

```
typedef struct {
  surface_flags_t flags;
  surface_coord_t dirty_x_lo;
  surface_coord_t dirty_x_hi;
  surface_coord_t dirty_y_lo;
  surface_coord_t dirty_y_hi;
  pixelmap_t pixmap;
} surface_t;
```

Responsibility of the source is to determine a color for a particular coordinates on the target surface. There are numerous combinations how the color could be specified within the source. The simplest case is when the color is specified just by a single ARGB32 pixel. Optionally, color transparency can be specified separately from the color pixel as a single alpha value. Color can be also specified by an entire ARGB32 texture, in which case it is necessary to declare the relationship between coordinate systems of the source and the surface by setting the transformation matrix and filtering method. Again, the transparency of the texture can be specified separately by an alpha mask. It is actually even possible to combine color pixel with alpha mask and color texture with alpha value. Furthermore, alpha values within the color pixel or color texture are not in conflict with the separately specified ones – both alpha values are combined together when determining the resulting source pixel. To provide even more flexibility, both the color texture and the alpha mask are actually surfaces (i.e. they could be prepared at some point in the past the same way as the current surface is being drawn onto). Moreover, their sizes do not have to match and either of them can be optionally tiled.

```
typedef struct {
  transform_t transform;
  filter_t filter;

  pixel_t color;
  surface_t *texture;
  bool texture_tile;

  pixel_t alpha;
  surface_t *mask;
  bool mask_tile;
} source_t;
```

Last piece of the puzzle is the drawing context. Its main purpose is to decide if and how to blend the target pixel from the surface with the pixel retrieved from the source. To limit the surface areas exposed to the source, drawing context can be equipped with a rectangular clipping region or a bit mask. Note that the size of the bit mask must correspond to the size of the surface. If a particular coordinates are not clipped or masked, surface and source pixels are combined together by a compositing operator.

```
typedef struct {
  //...
  surface_t *surface;
  compose_t compose;
  surface_t *mask;
  source_t *source;
  font_t *font;

  sysarg_t clip_x;
  sysarg_t clip_y;
  sysarg_t clip_width;
  sysarg_t clip_height;
} drawctx_t;
```

The whole process of transferring the single pixel is illustrated in **Figure 8**. The most basic operation offered by libdraw is to transfer a rectangular area of pixels via `drawctx_transfer` function. Such area is always specified in the surface coordinate system. That way, surface pixels can be iterated and for each pixel, it is determined what pixel from the source corresponds to it, as was described above. It is obvious that the flexibility involved in pixel transfer is quite costly considering the amount of pixels that have to be transferred in the usual scenarios. Therefore, `drawctx_transfer` is able to distinguish and optimize the special case when the whole transfer is actually a plain copying of two pixel buffers (i.e. no masking, clipping, transformation and transparency). Such optimization is primarily important for the compositor, because the most frequent operation is expected to be a translation of otherwise non-transformed opaque windows.

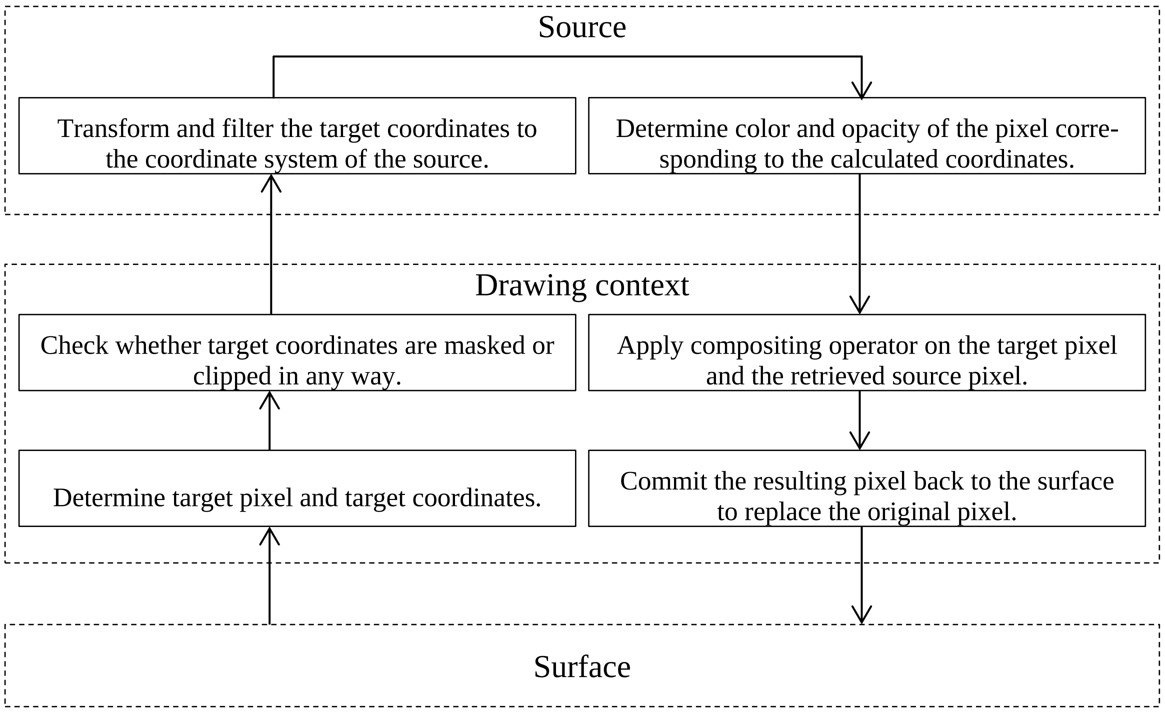

Source
Transform and filter the target coordinates to the coordinate system of the source.
Determine color and opacity of the pixel corresponding to the calculated coordinates.
Drawing context
Check whether target coordinates are masked or clipped in any way.
Apply compositing operator on the target pixel and the retrieved source pixel.
Determine target pixel and target coordinates.
Commit the resulting pixel back to the surface to replace the original pixel.
Surface

Figure 8 Subtasks involved in the single pixel transfer.

Other, more sophisticated drawing operations like line drawing or curve drawing can be based on pixel transferring by supplying the appropriate bit mask. The bit mask can be prepared either manually or by providing its abstract retained description via a *path* mechanism. The path is basically a list of steps, where each step is characterized by starting coordinates, ending coordinates, step type and any additional type-specific parameters.

```
typedef enum {
    PATH_STEP_MOVETO,
    PATH_STEP_LINETO
    // additional step types
} path_step_type_t;

typedef struct {
    link_t link;
    path_step_type_t type;
    double to_x;
    double to_y;
    // additional step descriptors
} path_step_t;

typedef struct {
    list_t list;
    double cur_x;
    double cur_y;
} path_t;
```

For example, line drawing can be described just by MOVETO and LINETO steps, both of which do not have any additional parameters. If there was a CURVETO step, it would require additional parameters to describe the curvature of the curve. Currently, path mechanism is prepared just for drawing lines and rectangles. Also, it serves only as a placeholder for the future extension because libdraw is currently not able to render the mask described by the path. Idea is that once the path is described, either `drawctx_stroke` or `drawctx_fill` is called to render the mask as an outline of the path, or respectively as an area bounded by the path. However, both functions are not part of the initial implementation.

Apart from pixel transferring, drawing context is also able to render a text according to the supplied *font object*. In libdraw, the font object is considered a set of glyph masks rendered at a particular point size. In order to abstract over the different font sources (i.e. various bitmap or vector font formats), the font object must be equipped with a *font decoder*.

```
typedef struct {
   uint16_t points;
   uint16_t glyph_count;
   surface_t **glyphs;
   font_decoder_t *decoder;
   void *decoder_ctx;
} font_t;

typedef struct {
   void (*init)(char *filename, uint16_t *glyph_count, void **decoder_ctx);
   uint16_t (*resolve)(const wchar_t character, void *decoder_ctx);
   surface_t *(*render)(uint16_t glyph_idx, uint16_t points);
   void (*release)(void *decoder_ctx);
} font_decoder_t;
```

The task of the font decoder is to translate characters to glyph indices and to render the glyph masks according to the point size supplied from the font object. In other words, font decoder is a rendering engine of the font object. Since the glyph masks are rendered in a lazy manner, the font object can be actually considered as a font cache. The initialized font object can be used to determine a bounding box size for a particular text string, and subsequently to draw such string via the pixel transfer mechanism.

```
void font_get_box(font_t *, char *text, sysarg_t *width, sysarg_t *height);
void font_draw_text(font_t *, drawctx_t *, source_t *,
    const char *text, sysarg_t x, sysarg_t y);
```

While the above font functions can be used directly, drawing context also provides convenience wrapper function `drawctx_print`. The initial implementation contains only a single font decoder for the non-proportional bitmap font embedded into the libdraw library. Note that to support proportional fonts, libdraw would have to be considerably extended to account for all the glyph properties like bearing, baseline, ascent, descent, etc. Another important extension could be an implementation of the system-wide font cache to decrease the memory consumption and to avoid redundant rendering of glyphs within different applications.

In a similar way as with font object, libdraw provides a *cursor object*. While it can be certainly used in various other scenarios, its main purpose is to handle visual depiction of the mouse pointer for the compositor. Assuming the cursor can be animated, cursor object is in fact a wrapper over multiple surfaces that together create such an animation. To abstract over the file formats that describe cursors or animations, cursor object is dependent on the *cursor decoder* which is responsible for rendering of those surfaces. As with fonts, libdraw currently offers only single decoder for the embedded non-animated bitmap cursor depicting the arrow.

```
typedef struct {
   uint8_t state_count;
   surface_t **states;
   cursor_decoder_t *decoder;
   void *decoder_data;
} cursor_t;
```

```
typedef struct {
    void (*init)(char *filename, uint8_t *state_count, void **decoder_ctx);
    surface_t *(*render)(uint8_t state_idx);
    void (*release)(void *decoder_ctx);
} cursor_decoder_t;
```

Last functionality covered by libdraw is the encoding and decoding of the image file formats. While the initial implementation actually provides only a TGA[22] format decoder, it is expected that for each format there would be both the encoder and the decoder (e.g. `decode_tga` and `encode_tga`) to convert between the image format and plain libdraw surfaces.

To conclude this chapter, let us see the following usage example of libdraw. It shows how to center a semi-transparent 50x50 image onto a white 100x100 surface.

```
surface_t *surface = surface_create(100, 100);
surface_t *image = decode_tga( /* ... */ ); // 50x50

source_t background;
source_init(&background);
source_set_color(&background, PIXEL(255, 255, 255, 255));

transform_t transform;
transform_identity(&transform);
transform_translate(&transform, -25, -25);

source_t foreground;
source_init(&foreground);
source_set_transform(&foreground, transform);
source_set_filter(&foreground, filter_nearest);
source_set_texture(&foreground, image, false);
source_set_alpha(&foreground, PIXEL(128, 0, 0, 0));

drawctx_t context;
drawctx_init(&context, surface);

drawctx_set_compose(&context, compose_src);
drawctx_set_source(&context, &background);
drawctx_transfer(&context, 0, 0, 100, 100);

drawctx_set_compose(&context, compose_over);
drawctx_set_source(&context, &foreground);
drawctx_transfer(&context, 25, 25, 50, 50);
```

4.4 Graphics Server

As was concluded in the analysis chapter, HelenOS compositor is a manager of the oversized desktop into which it places viewports from graphic drivers and windows from application clients. The oversized desktop is considered logically infinite, however in practice the coordinates are saved as 32-bit unsigned integers. To give an illusion of infinity when placing viewports or windows onto the desktop and to ensure the convenience of having only positive coordinates (without overflows or underflows), the origin of the coordinate system is put into the middle of the value space provided by the aforementioned data type. Individual pixels are mapped onto the coordinates in such a way that pixels reside in-between the coordinates and the identity of the pixel is determined by the coordinate of its top-left corner. Such approach is the most convenient when implementing the graphics stack in C-like language. The drawback of the approach is that pixel might change its identity after the application of certain affine

[22] Truevision Graphics Adapter

transformation – consider for example clock-wise rotation of 90 degrees – the rotated coordinate will no longer represent the original pixel, but its top neighbor (which becomes the right neighbor after the rotation). This inconvenience could be resolved either by mapping the coordinate onto the center of the pixel or simply by accounting for it in the implementation to avoid off-by-one errors. The latter approach was chosen because the top-left mapping is otherwise more convenient than the center mapping.

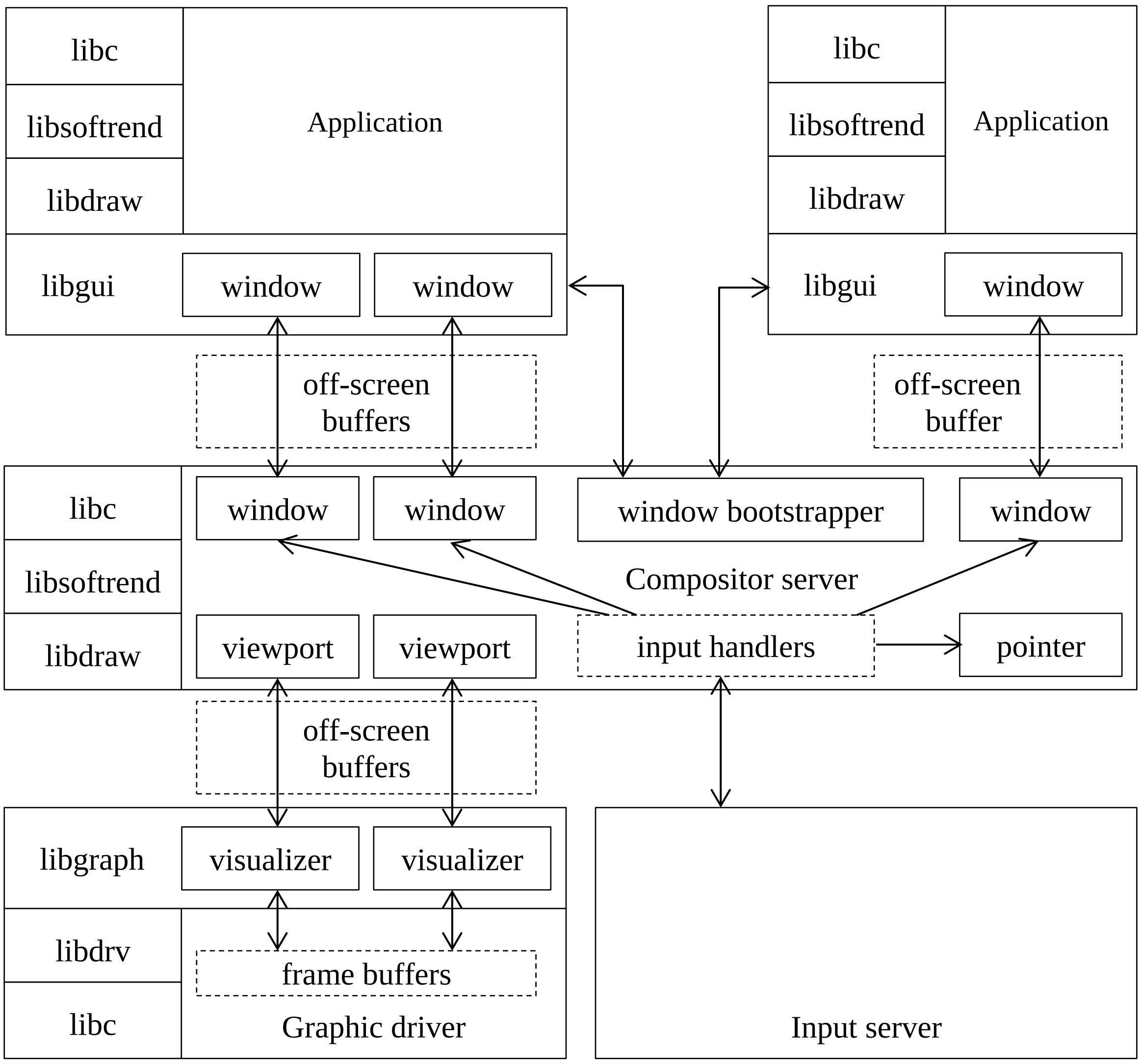


Figure 9 Interfaces between compositor, widget toolkit, graphic driver and input subsystem.

As can be seen in the **Figure 9**, there are three principal object types managed by the compositor – *viewports*, *windows* and *pointers*. Each viewport represents one of the output devices provided by the graphic driver. As shown in the following code snippet, viewport consists just from the shared pixel buffer and its position within the oversized desktop.

```
typedef struct {
   // ...
   vslmode_t mode;
   desktop_point_t position;
   surface_t *surface;
} viewport_t;
```

Copy of the selected mode setting of the output device is currently not used, but can be utilized in the future to let user change the mode setting via some sort of a manager. In the initial implementation, viewports are automatically created for all the output devices available at the compositor startup – each output device is set to its default mode setting and the corresponding viewport is placed to the origin of the compositor coordinate system.

Pixel buffers shared with the compositor clients are wrapped by a window object. Each client can have one or more windows opened. Apart from pixel buffer, window object consists also from communication channels, event queue and information about applied transformations.

```
typedef struct {
   // ...
   service_id_t input_channel;
   service_id_t output_channel;
   prodcons_t event_queue;
   transform_t transform;
   double dx; // horizontal translation
   double dy; // vertical translation
   double fx; // horizontal scaling
   double fy; // vertical scaling
   double angle; // rotation angle
   uint8_t opacity;
   surface_t *surface;
} window_t;
```

The event queue serves as a temporary buffer for the events that are addressed to the client but were not dispatched yet. While events from the event queue are sooner or later relayed to the client, transformations are kept exclusively in the compositor (i.e. client does not know how its windows are transformed or where they are located in the desktop). Notice that transformations are expressed both in an aggregated form (transformation matrix) and parametric form, which is necessary due to the fact that each variant is useful in different scenario and the conversion between the variants is costly and complicated (especially the extraction of the transformation parameters from the transformation matrix). Windows are created on the client's request via a window bootsrapper mechanism advertised via the loc server – i.e. client connects to the bootstrapper and requests the creation of the window upon which the compositor creates a new window object and establishes window-specific communication channels with the client.

Apart from the client windows, compositor is also responsible for drawing the pointers representing input positioning devices. Although the input handlers within the compositor can currently handle just a single pointer (i.e. mouse pointer), rest of the compositor is prepared for the eventuality of having multiple pointers (consider for example multi-touch input device).

```
typedef struct {
   // ...
   sysarg_t pointer_id;
   desktop_point_t current_position;
   sysarg_t pressed_button;
   desktop_point_t click_position;
   desktop_vector_t click_distance;
   sysarg_t grab_flags;
   uint8_t visual_state;
   cursor_t textures;
   window_t ghost;
   // ...
} pointer_t;
```

Appearance of the pointer is defined by the pointer state and the corresponding texture stored within the cursor object provided by the drawing library. Among other items of the pointer object is the current position of the pointer and the context of the current position. The position context includes information like whether any button is pressed, where the last click occurred, how it relates to the current position, whether the click triggered any window transformation and how to depict such transformation. More detailed description of how the window transformations are handled and depicted follows later in the text.

The communication interface between the client and the compositor is shown in the following code snippet. The client-to-compositor direction of the communication is covered by calling the listed functions apart from `win_get_event` function. The other direction, i.e. compositor-to-client, is based on queuing events within the corresponding window object in the compositor from where the client fetches them by calling `win_get_event`. Each event is defined by its type and additional type-specific data (e.g. keycode for `ET_KEYBOARD_EVENT` or mouse pointer position for `ET_POSITION_EVENT`). One by one, the event types are further explained in the following text. Note that event types `ET_SIGNAL_EVENT`, `ET_WINDOW_REFRESH` and `ET_WINDOW_DAMAGE` are used only internally within the client-side widget toolkit, not for the client-compositor communication.

```
typedef struct {
   link_t link;
   window_event_type_t type;
   window_event_data_t data;
} window_event_t;

typedef enum {
   ET_KEYBOARD_EVENT,
   ET_POSITION_EVENT,
   ET_SIGNAL_EVENT,
   ET_WINDOW_FOCUS,
   ET_WINDOW_UNFOCUS,
   ET_WINDOW_RESIZE,
   ET_WINDOW_REFRESH,
   ET_WINDOW_DAMAGE,
   ET_WINDOW_CLOSE
} window_event_type_t;

int win_register(async_sess_t *, service_id_t *in, service_id_t *out,
    sysarg_t x_offset, sysarg_t y_offset);
int win_get_event(async_sess_t *, window_event_t *);
int win_damage(async_sess_t *,
    sysarg_t x, sysarg_t y, sysarg_t width, sysarg_t height);
int win_grab(async_sess_t *, sysarg_t pointer_id, sysarg_t grab_flags);
int win_resize(async_sess_t *,
    sysarg_t width, sysarg_t height, void *cells);
int win_close(async_sess_t *);
int win_close_request(async_sess_t *);
```

First of all, let us explore the lifecycle of the window. As was already mentioned, window is first bootstrapped by calling `win_register` function. Apart from establishing the communication channels, `win_register` also specifies the initial position of the window relatively to the origin of the compositor coordinate system. Next, `win_resize` is called to specify the window size and to establish the sharing of the pixel buffer. Note that any time the window gets resized, it involves deallocating the old buffer and calling the `win_resize` with the new one.

Destruction of the window can be initiated either within the client or within the compositor. If it is initiated within the client, client simply calls `win_close_request` by which it notifies the compositor about the intention to destroy the window. After that, the process is the same as if it was initiated within the compositor. First, the compositor posts `ET_WINDOW_CLOSE` event into the client's event queue. Once the close event is fetched by the client's widget toolkit, client closes the input communication channel, deallocates any window-related resources and calls `win_close` function, upon which the compositor also deallocates its side of window resources and terminates the output communication channel.

During window lifetime, the basic case of interaction between the client window and the compositor is shown in **Figure 10**. While the compositor feeds the window with keyboard (`ET_KEYBOARD_EVENT`) and mouse (`ET_POSITION_EVENT`) events, window reports via `win_damage` function the area of the shared pixel buffer that was damaged by the reaction of widget toolkit to any such event. Note that both mouse events and damage rectangles must be converted between coordinate systems of the client window and the compositor with respect to any transformations applied on the window. That way, client can live in its own flat world not concerned about how its pixel buffer is integrated into the composited desktop.

As noted in **Figure 10**, the main damage routine of the compositor currently uses *painter's algorithm* to compose the desktop. Each desktop element is transferred onto the viewport off-screen buffers in the order described by the following pseudo code:

```
HANDLEDAMAGE($rectangle, background, viewports, windows, pointers$)
1 for each $v \in viewports$ do
2     let $v_r \leftarrow rectangle \cap v.rectangle$
3     transfer $background$ to $v.surface$ within $v_r$
4     for each $w \in windows$ do {in bottom to top order}
5         let $w_r \leftarrow v_r \cap w.rectangle$
6         transfer $w.surface$ to $v.surface$ within $w_r$
7     for each $p \in pointers$ do
8         let $p_r \leftarrow v_r \cap p.rectangle$
9         transfer $p.surface$ to $v.surface$ within $p_r$
```

In contrast to non-compositing graphics servers, the painter's algorithm is necessary to account for the situation when the desktop contains transparent and transformed windows despite it is not efficient when all the windows are opaque and non-transformed. While the algorithm could be optimized by determining the visibility of pixels, it would get significantly complex due to all the possible window transformations. That is the reason why the accelerated compositors in mainstream operating systems model the desktop as a 3D scene even if there would be only 2D transformation. Then, the visibility and depth of pixels can be determined via techniques like Z-buffering. While the plain painter's algorithm is sufficient for the purpose of the initial implementation, it should be definitely replaced by something better once there is an accelerated 3D drawing library available in HelenOS.

Another thing apparent from the **Figure 10** is the consumption of certain input events by the compositor itself, which is necessary for window management. User can control windows both via keyboard and mouse. Let us first explore the keyboard controls:

Key	Action
Alt + Tab	Switch active window
Alt + A, S, W, D	Move window (left, down, up, right)
Alt + T, G, B, N	Resize window (lower, higher, narrower, wider)
Alt + X	Close window
Alt + F, R	Scale window (bigger, smaller)
Alt + Q, E	Rotate window (clockwise, counter-clockwise)
Alt + C, V	Window opacity (more transparent, more opaque)
Alt + J, K, I, L	Move viewport (left, down, up, right)
Alt + O, P	Select active viewport

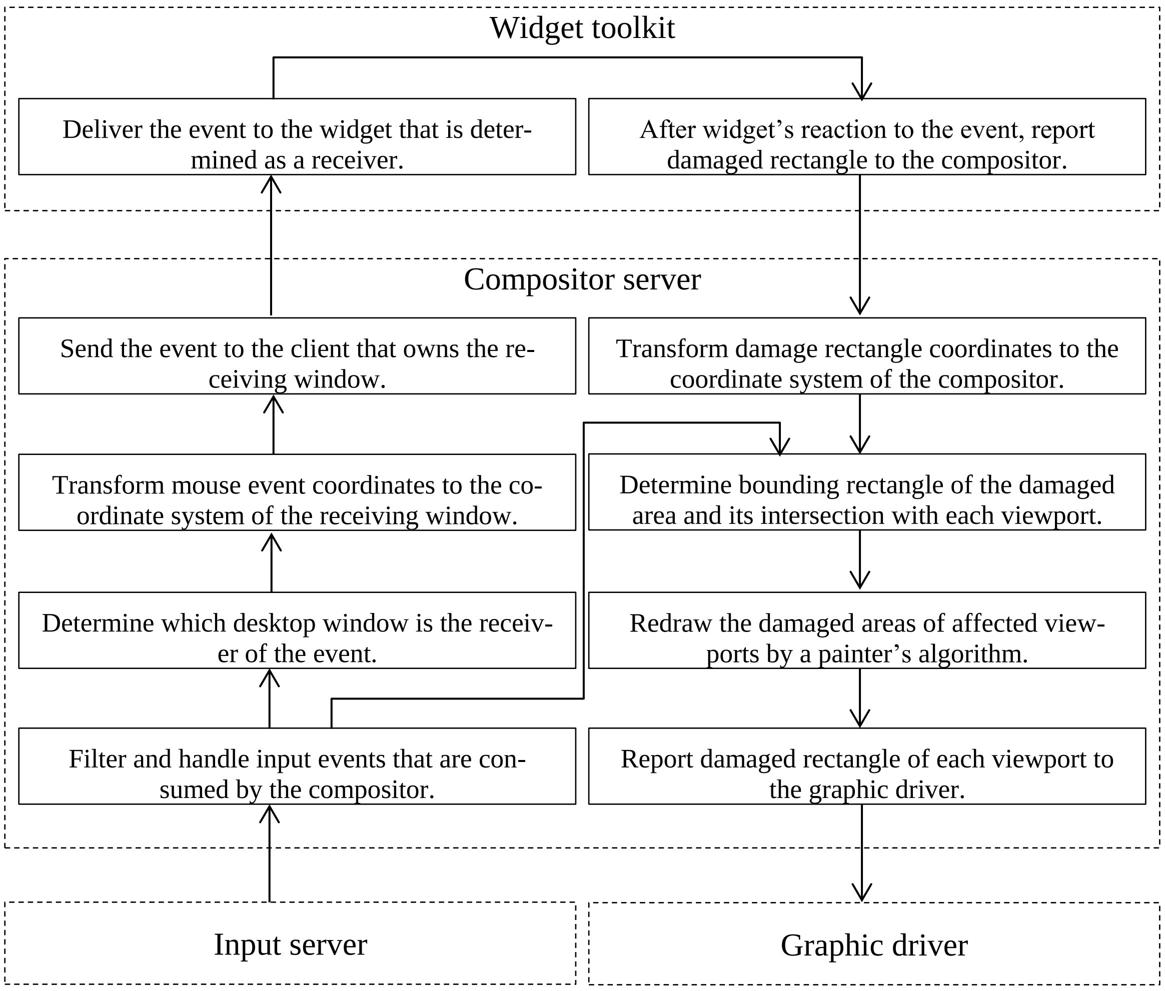

Widget toolkit
Deliver the event to the widget that is determined as a receiver.
After widget's reaction to the event, report damaged rectangle to the compositor.
Compositor server
Send the event to the client that owns the receiving window.
Transform damage rectangle coordinates to the coordinate system of the compositor.
Transform mouse event coordinates to the coordinate system of the receiving window.
Determine bounding rectangle of the damaged area and its intersection with each viewport.
Determine which desktop window is the receiver of the event.
Redraw the damaged areas of affected viewports by a painter's algorithm.
Filter and handle input events that are consumed by the compositor.
Report damaged rectangle of each viewport to the graphic driver.
Input server
Graphic driver

Figure 10 Reaction of the compositor and related components to the input event.

The important aspect of controlling windows via keyboard is that it does not require almost any cooperation with the client-side widget toolkit as opposed to the mouse controlling. Apart from window resizing, window switching and window closing, all the actions listed in the table above are handled directly within the compositor. Window closing was already described (i.e. the case when the window destruction is initiated by the compositor). Window resizing was also described further above, only this time the client is requested to resize via `ET_WINDOW_RESIZE` event (new size is delivered as the event parameter). Window switching involves putting the current top-level window to the back of the window list (which makes the next window in the list a new top-level window) and informing both affected windows about the change via `ET_WINDOW_UNFOCUS` and `ET_WINDOW_FOCUS` events, respectively, so the widget toolkit can perhaps change the colors of window decorations.

Not all the actions provided by the keyboard are currently available for mouse controlling – window transparency, window rotation and viewport position can be adjusted only by the keyboard. Window switching is triggered whenever the mouse click event occurs on the window that is currently not top-level, but then the situation is handled in the same way as with the keyboard. On the contrary to the keyboard case, window closing is initiated within the client once the widget toolkit detects that closing button within the decoration was clicked.

Moving, resizing and scaling windows via mouse are the most complicated actions that need further cooperation between the client and the compositor. The reason is that the meaning of the action is not determined only by the pressed button, but also by the pointer position within the client window. Since window decorations are currently handled by the client-side widget toolkit, action meaning cannot be determined within the compositor. Instead, client informs the compositor about the occurring action via `win_grab` function and the following flags:

```
typedef enum {
  GF_EMPTY = 0,
  GF_MOVE_X = 1,
  GF_MOVE_Y = 2,
  GF_RESIZE_X = 4,
  GF_RESIZE_Y = 8,
  GF_SCALE_X = 16,
  GF_SCALE_Y = 32
} window_grab_flags_t;
```

In other words, generic mouse event is forwarded to the client; client decides whether such event grabs any window decoration and if so, the compositor is informed that any subsequent mouse movement should be considered as a particular window transformation. In the same way, client informs the compositor when the decoration is not grabbed anymore.

Notice that once any grabbing flag is set, mouse movement becomes much more costly operation. Suddenly, the damage to the viewport is caused not only by the small pointer but by the entire window. Furthermore, window resizing involves also communication with the client that subsequently reallocates and redraws the entire pixel buffer of the affected window. Clearly, such actions are too expensive in the absence of any hardware acceleration to be executed several times per second during the mouse movement. Despite the lacking acceleration, the initial implementation is able to animate window transformations during the mouse movement. Since window resizing could be slow even with the acceleration, it is substituted by window scaling during the mouse movement and the window is only resized once the grabbing flags are cleared.

Anyway, animations are currently turned off by default so there are no slowdowns. As you can see in the feature branch, the missing indication of the progressing window transformation decreases overall intuitiveness of the user interface. Therefore, mainline version of the compositor is additionally equipped with the wireframe windows, or so called *ghosts*. The ghost is basically a window object that is associated with the pointer and that does not have any pixel buffer allocated. When any grabbing flag is set, the ghost mimics the transformations applied on the active window. In the main damage handling routine, ghost drawing takes place after all the windows but before the pointers. Since the ghost is depicted only by four thin lines representing the window boundary, the viewport damage caused by mouse movement could be represented only by 5 rather small rectangles (one rectangle for the pointer texture, rest for the ghost lines). That way, it is possible to increase the intuitiveness of window transformations while not affecting the performance significantly.

4.5 Widget Toolkit

HelenOS widget toolkit consists of two principal types of objects – *windows* and *widgets*. In server-side widget toolkits, window is usually also considered a widget. However, a client-side toolkit requires window to take more responsibility than an ordinary widget, therefore it is more convenient to declare it as a separate type. In more detail, each window maintains event queue, widget tree and communication channels with the compositor. Each window is also associated with two fibrils as can be seen in **Figure 11**. One of the fibrils listens exclusively to the compositor and puts the received events into the event queue. Second fibril acts as an event loop – i.e. it dispatches events from the queue, delivers them to a particular widget and executes a corresponding event handler. Note that the event loop fibril is not strictly an event consumer because the widgets can post events to the event queue as well.

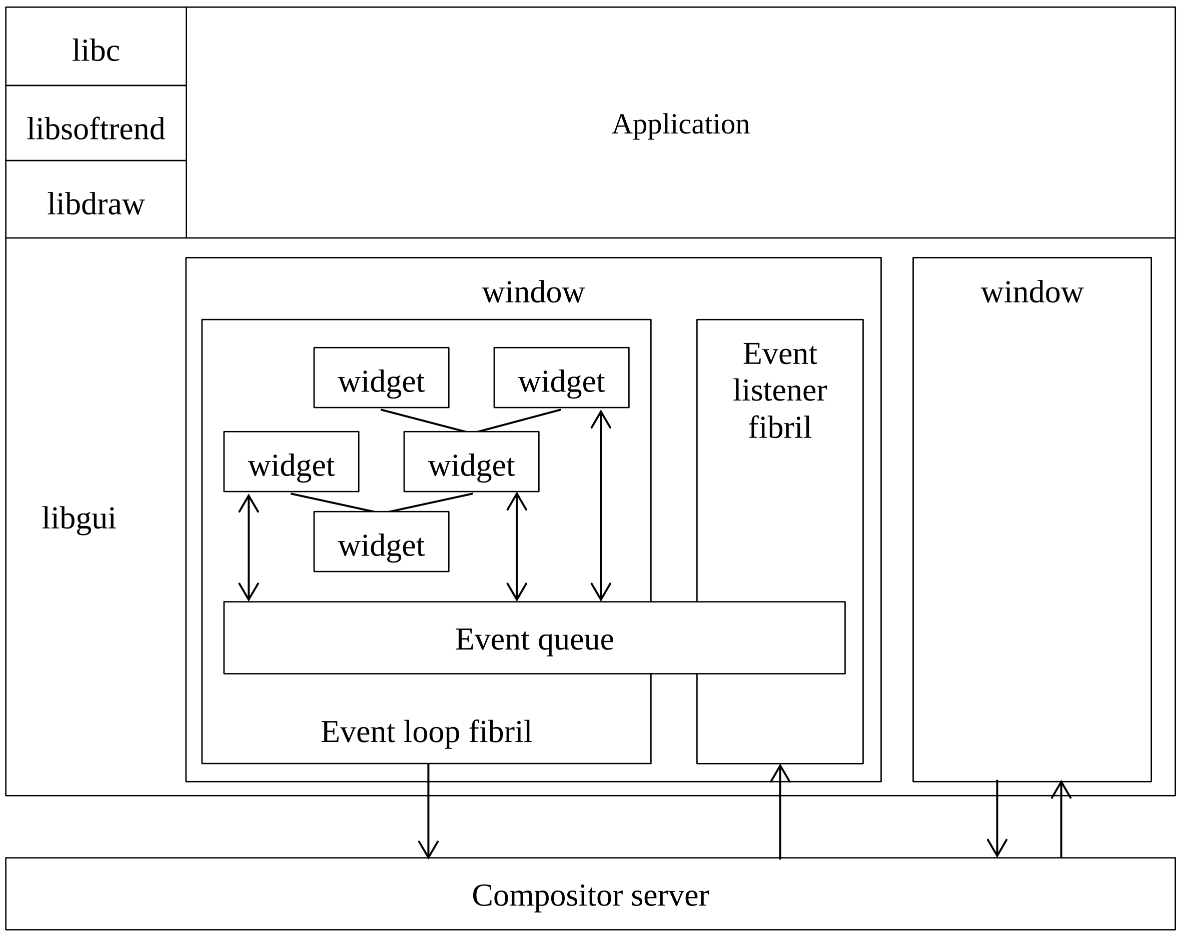


Figure 11 Internal structure of the widget toolkit.

Significant part of the window behavior was already described in the compositor chapter, especially its lifecycle. This chapter explores the window from widget toolkit perspective. Apart from communication channels, event queue and widget tree, the window also maintains pixel buffer and decorations:

```
typedef struct {
  bool is_main;
  bool is_decorated;
  bool is_focused;
  char *header_caption;
  async_sess_t *input_session;
  async_sess_t *output_session;
  prodcons_t event_queue;
  widget_t root;
  widget_t *mouse_grab;
  widget_t *keyboard_focus;
  fibril_mutex_t surface_guard;
  surface_t *surface;
} window_t;
```

Client can open multiple windows in the compositor. Windows can be opened either from an application-specific code or from within some widgets (consider for example context menus or list boxes). One of the client's windows should be always set as a main window, because closing of such window terminates the entire application. In order to deliver the events to the widgets, window maintains references to the widget tree root and to the widget that has keyboard focus or mouse grab.

The possibility to bind a widget with keyboard and mouse serves as a shortcut when delivering events – i.e. instead of propagating the event throughout the whole widget tree, the event is delivered directly to a particular widget. It is the responsibility of each widget to (un)grab a mouse or to get a keyboard focus. While it is entirely in the hands of the widget implementer, the good practice is to get a keyboard focus upon receiving any mouse click event (if it makes sense for the widget functionality). On the other hand, mouse grabbing is intended only for special scenarios required by certain advanced widgets (consider for example scrollable drawing canvas) and should be used with caution. Note that once the mouse is grabbed by a widget, all mouse events are delivered to such widget no matter what coordinates the event occurred at, as long as it was somewhere within the window. Normally, mouse events are propagated throughout the widget tree in a top-bottom manner according to the widget positions and sizes. If there is no widget having a keyboard focus, keyboard events are delivered to the root widget where they are currently ignored.

Root of the widget tree is actually a special widget embedded directly into the window. While the root can have multiple children, it is expected that it will have only single child which would be a layout widget. The reason is that the root widget, in contrast to the layout widgets, does not have any internal logic to determine which of its children should receive the event. Therefore, if there are multiple children, the events are broadcasted to all of them. Perhaps the biggest responsibility of the root widget is to draw window decorations and to inform the compositor when the user interacts with the decorations. Note that window decorations can be disabled for a window. As of writing this, the window decoration consists of window boundary, window header and close button. The window boundary is used either for window resizing or window scaling, depending on whether the left or the right mouse button is pressed while dragging the mouse, respectively. While the window header is primarily used for window moving, it also contains a caption string with the application name. The background color of the header corresponds to whether the window is a top-level desktop window or not (widget toolkit is informed about that via `ET_WINDOW_FOCUS` and `ET_WINDOW_UNFOCUS` events). For information on window transformations, refer to the compositor chapter.

The following listing enumerates the functions that can be applied on a window either from the widgets or from the application code:

```
window_t *window_open(char *, char *window_bootstrapper,
    bool is_main, bool is_decorated, const char *caption,
    sysarg_t x_offset, sysarg_t y_offset);
void window_resize(window_t *, sysarg_t width, sysarg_t height);
void window_refresh(window_t *);
void window_damage(window_t *);
widget_t *window_root(window_t *);
void window_exec(window_t *);
surface_t *window_claim(window_t *);
void window_yield(window_t *);
void window_close(window_t *);
```

Window opening and closing was already explained in the compositor chapter. To quickly remind the window opening, `window_open` establishes the communication channels, creates the window object within the compositor and places it at particular coordinates (relative to the origin of the compositor coordinate system). Immediately after that, `window_resize` must be called to allocate and share out the appropriately sized pixel buffer with the compositor. Note that window resizing can be also initiated anytime during window lifetime both from the compositor and from the client – the only difference is whether `ET_WINDOW_RESIZE` event is posted into the event queue by the fibril listening to the compositor or by any other fibril of the client (usually the event loop fibril of that particular window). Once the window and the widget hierarchy are both fully initialized, application code is expected to call `window_exec` function that creates the window fibrils and schedules them for execution within the fibril manager. When application `main` function has nothing left to initialize, it must call `async_manager` function which will ultimately lead to execution of the window event loop.

During the window lifetime, widgets react to various events by damaging parts of the window surface. Whenever the surface content is altered, responsible widget is obliged to post `ET_WINDOW_DAMAGE` event into the event queue by calling `window_damage` function. Note that the actual damage region is tracked by the window surface, so it is not a problem if there are multiple consecutive damage events posted into the queue by various widgets – all the events are aggregated into the single call to the compositor.

Since there can be widgets that draw onto the surface from their own private fibril, access to the window surface must be properly synchronized by all widgets by surrounding the drawing operations with `window_claim` and `window_yield` calls. Speaking of multi-fibril widgets, some of them might prefer to draw into their own private pixel buffer to avoid slowing down the event loop fibril or to avoid storing the information required to redraw their visual manifestation. Example of such widget might be drawing canvas, video surface, terminal emulator etc. The actual drawing onto the window surface in the event loop fibril then consists only from transferring the content of the widget's private buffer onto the surface. To trigger the transfer, widget's private fibril must post `ET_WINDOW_REFRESH` event into the event loop by calling `window_refresh` function. As a reaction to the event, all widgets in the widget tree are requested to redraw themselves in a top-bottom order. While it might seem inefficient to redraw all the widgets, the above mentioned examples show that multi-fibril buffered widgets usually encompass majority of the window surface. If more fine grained approach is still desired, it is possible for multi-fibril widget to notify itself via a signal-slot mechanism (which is described later) without calling `window_refresh` at all.

The window closing cascade was mostly described in the compositor chapter but it lacked the detailed description on what is happening in the widget toolkit. Function `window_close` is actually just a wrapper over `win_close_request` function and is expected to be called from the window decoration. After receiving `ET_WINDOW_CLOSE` event back from the compositor, the listening fibril posts the event to the event queue and terminates. Upon dispatching the event, the event loop fibril deallocates all window-related resources (including the widget tree) and terminates as well. Moreover, if the window is in fact the main window of the application, the whole client process is terminated.

Knowing how the window works, next important thing to explain is the widget tree. Each widget (apart from the root widget) is expected to have single parent and arbitrary number of children. Widget's position is expressed in the window coordinate system (top-left corner of the window is the origin). It is assumed that the bounding rectangle of the widget is encompassed by the bounding rectangle of its parent, so the entire widget tree can be drawn in a top-bottom manner via the painter's algorithm.

Note that the size of the widget is not determined exclusively by the widget itself but instead by the cooperation with its parent and other ancestors that are closer to the root. The most common situation is when the internal nodes of the widget tree consist of layout widgets and the leaf nodes are the actual widgets that are visible to the user. In such scenario, layout widgets dynamically distribute the available window space among their children according to the layout (be it grid layout, stacked layout, or something more elaborate). Because window size might be arbitrary, each widget must be able to draw its visual manifestation at any size that is assigned to it. While some layout widgets might be authoritative when deciding the size of its children (e.g. grid layout), there are also layouts that might consider the size preferred by the child widget (e.g. vertically stacked layout might consider the preferred heights of its children). For such purpose, each widget might optionally declare the minimum, maximum and ideal size as a hint to the parent widget. Once again, while the preferred range of sizes might be optimal for the widget (i.e. widget can draw all of its elements to ensure ergonomic interaction), widget must be able to draw itself at arbitrary size (including zero).

```
typedef struct {
   link_t link;
   widget_t *parent;
   list_t children;
   window_t *window;

   sysarg_t hpos, vpos;
   sysarg_t width, height;
   sysarg_t width_min, width_ideal, width_max;
   sysarg_t height_min, height_ideal, height_max;

   void (*destroy)(widget_t *);
   void (*reconfigure)(widget_t *);
   void (*rearrange)(widget_t *,
         sysarg_t hpos, sysarg_t vpos, sysarg_t width, sysarg_t height);
   void (*repaint)(widget_t *);
   void (*handle_keyboard_event)(widget_t *, kbd_event_t);
   void (*handle_position_event)(widget_t *, pos_event_t);
} widget_t;
```

The above code snippet shows the top-level base class of any widget object. In order to simulate inheritance and basic polymorphism, it is necessary for any derived widget class to store its base class as a first data member (so the casting is possible). Derived widget might extend the base class by its own data members and functions. However, the derived widget must always implement the functions from the above code snippet (in object-oriented language, these functions would be declared as abstract). Additionally, each widget must provide initializer, deinitializer and constructor function outside of the object. For example the button widget would define:

```
bool init_button(button_t *, widget_t *parent, /* ... */);
button_t *create_button(widget_t *parent, /* ... */);
void deinit_button(button_t *);
```

Note that the destructor (`destroy` function from `widget_t`) must be part of the object in order to be virtual (i.e. it could be overloaded by the destructor of the derived class) and to make possible the recursive bottom-top destruction of the entire widget tree. Also note that the initializer is given a pointer to the parent widget in order to add itself to the parent's children list. Similarly, the deinitializer is responsible for removing the widget from parent's widget list.

The reason why the initializer (deinitializer) is not an integral part of the constructor (destructor) is to separate object allocation from its initialization. In fact, the constructor (destructor) is expected to be just a simple wrapper that (de)allocates the widget object and calls the (de)initializer. Splitting object construction this way is necessary for the situation when a derived class needs to initialize a base class as a part of its own construction – since the base class is a data member of the derived class, it is already allocated but not initialized.

```
typedef struct {                      typedef struct {
   widget_t widget;                      button_t button;
   // ...                                // ...
} button_t;                           } my_button_t;
```

Abstract functions from `widget_t` are necessary for event delivery and subsequent reaction propagation within the widget tree. Input events from the compositor are delivered to the widget via `handle_keyboard_event` and `handle_position_event` handlers. As mentioned in **Figure 12**, the handlers are invoked either directly on the widget that has keyboard focus or mouse grab (usually a leaf node in the widget tree), or on the root widget in which case the event is then propagated through the hierarchy of layout widgets down to some leaf node. Currently, only mouse events are propagated this way because the natural way of how to determine the next receiver of the event is to compare the position of mouse pointer with the position and size of the child widgets. That is, keyboard events are currently discarded within the root widget if there is no widget having a keyboard focus. Each widget has the opportunity to obtain the keyboard focus (or to grab the mouse) in the `handle_position_event` handler through the `window` pointer from the `widget_t`.

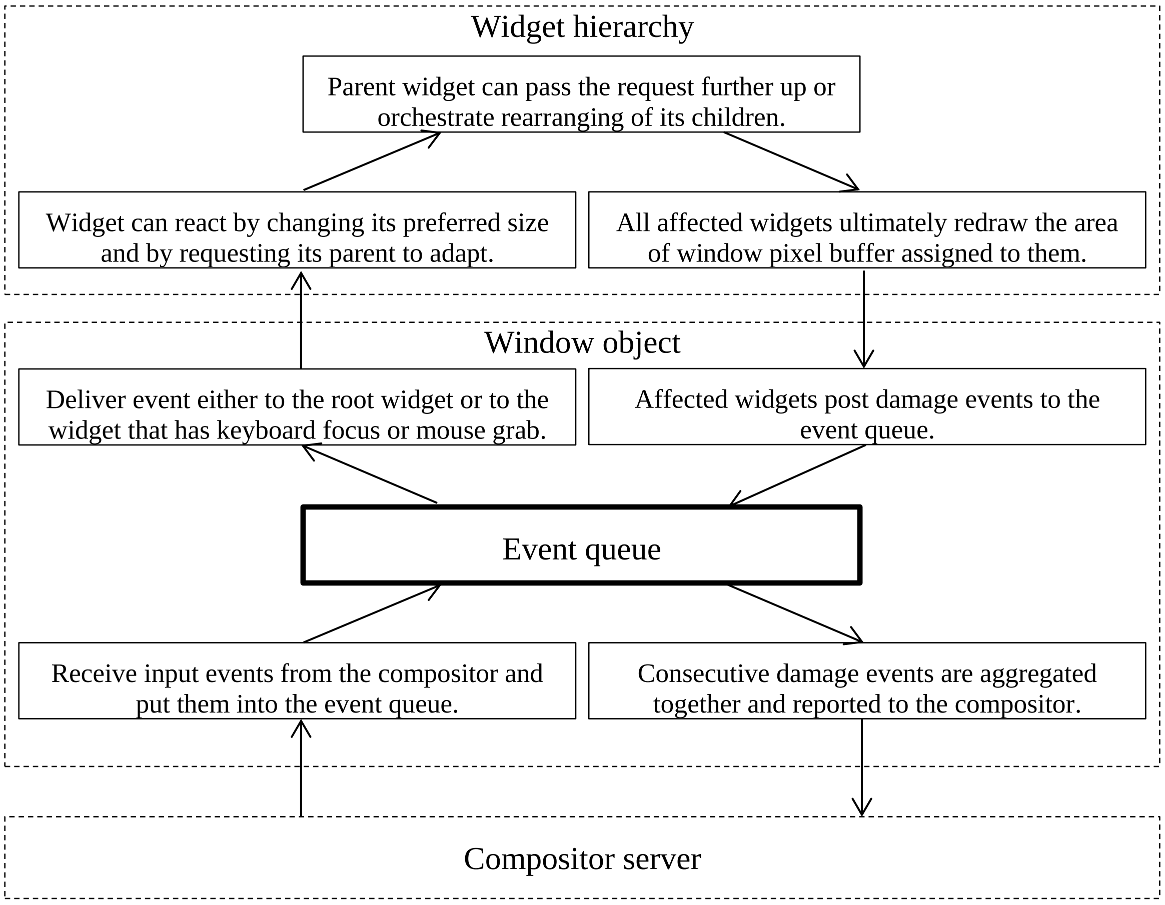

Widget hierarchy
Parent widget can pass the request further up or orchestrate rearranging of its children.
Widget can react by changing its preferred size and by requesting its parent to adapt.
All affected widgets ultimately redraw the area of window pixel buffer assigned to them.
Window object
Deliver event either to the root widget or to the widget that has keyboard focus or mouse grab.
Affected widgets post damage events to the event queue.
Event queue
Receive input events from the compositor and put them into the event queue.
Consecutive damage events are aggregated together and reported to the compositor.
Compositor server

Figure 12 Reaction of the widget toolkit to the input event.

To further explain the situation shown on the **Figure 12**, consider that widget changes its preferred size upon receiving the input event (for example edit box widget can increase its preferred width as a part of the reaction in the `handle_keyboard_event`). In order to report such change to upper layers of the widget tree, the widget should call `reconfigure` function of its parent. Assuming the parent would be a layout widget, it can assign new positions and sizes to its children by calling `rearrange` function on each of them. Alternatively, layout widget can change its preferred size as well and pass the `reconfigure` call further up. In other words, `reconfigure` calls are propagated up until some widget is able to handle the situation without changing its own size, then `rearrange` calls are propagated from that widget down to all of its descendants. As a reaction to `rearrange` call, each widget should redraw itself according to the new position and size. Note that `rearrange` calls are also propagated from the root widget whenever the window size is changed. Also note that while `rearrange` function must be implemented by all widgets, `reconfigure` function is relevant only for widgets that are expected to manage some children (i.e. mostly layout widgets).

The remaining `widget_t` function that needs to be explained is `repaint` handler. The handler is called in a top-bottom manner on all widgets in the widget tree as a reaction to `ET_WINDOW_REFRESH` event that was described further above. Therefore, it is necessary for all widgets that have some children to pass the call further down. As suggested by the function name, widget is expected to redraw itself without changing anything else. Hence, `repaint` function could be regarded as a special case of the `rearrange` function – most of the widgets will indeed consider `repaint` function only as a wrapper. However, the `repaint` function gives the widget an opportunity to optimize the drawing when there is no need to change the size. As an example, consider a buffered drawing canvas widget that must allocate a new buffer and rebuild the entire scene in the `rearrange` function, while the `repaint` function can contain just an optimized copy operation.

Apart from the already described mechanisms, widgets also have a generic way to communicate with each other through a signal-slot mechanism. The important property of the signal-slot mechanism when compared to an ordinary function call is the decoupling of the caller and the target of the call, possibly in an asynchronous manner. In other words, widget can broadcast the change of its internal state by emitting the signal, but does not care who and when will receive it – there could be zero, one or even multiple targets subscribed to the signal. The connection between a widget that provides the signal and a widget that provides the slot is created by a third party, which is usually the application code or some other widget.

The signal-slot mechanism in the initial implementation of HelenOS widget toolkit is not as fast, powerful and comfortable to use as the mechanisms provided by the mainstream widget toolkits, partly due to the limited implementation scope, partly because of the usage of C language (signal-slot mechanism can greatly benefit from meta-programming and object-oriented features of more advanced languages). However, it still delivers the two basic features mentioned above – decoupling and asynchronicity.

```
typedef sysarg_t signal_t;
typedef void (*slot_t)(widget_t *, void *args);
```

The signal is represented by an address of the integral data type `signal_t` that is expected to be stored as a data member of a widget object. This way, it is possible to uniquely identify signals without implementing any sort of dedicated identification generator. Slots are ordinary function pointers that are intended to be used as one of the methods within a widget object. More specifically, slot is uniquely represented by a function pointer and a pointer to the widget instance on which it shall be executed. Note that the widget pointer can be null, so the slot can be just a normal static function in the application code, not associated with any widget. For simplicity, slots are currently limited to a single void pointer argument, which means that widgets are expected to (un)pack and cast the actual arguments for themselves.

```
typedef struct {
   link_t link;
   widget_t *widget;
   slot_t slot;
} slot_node_t;

typedef struct {
   link_t link;
   signal_t *signal;
   list_t slots;
} signal_node_t;
```

```
void sig_connect(signal_t *, widget_t *, slot_t);
void sig_disconnect(signal_t *, widget_t *, slot_t);
```

The signal-slot mechanism is maintained as a list of active signals, each of which contains a list of connected slots. This list of lists is managed via `sig_connect` and `sig_disconnect` functions. Whenever a new connection is made, it is checked whether the signal list already contains such signal – if so, the slot is appended to the end of the signal's slot list, otherwise new entry is created in the signal list. Similarly, if the signal and slot become disconnected, the slot is removed from the slot list of the signal, and if it is the last slot in the list, the signal is removed from the overall signal list as well.

```
void sig_send(signal_t *, void *args);
void sig_post(signal_t *, void *args, size_t args_size);
```

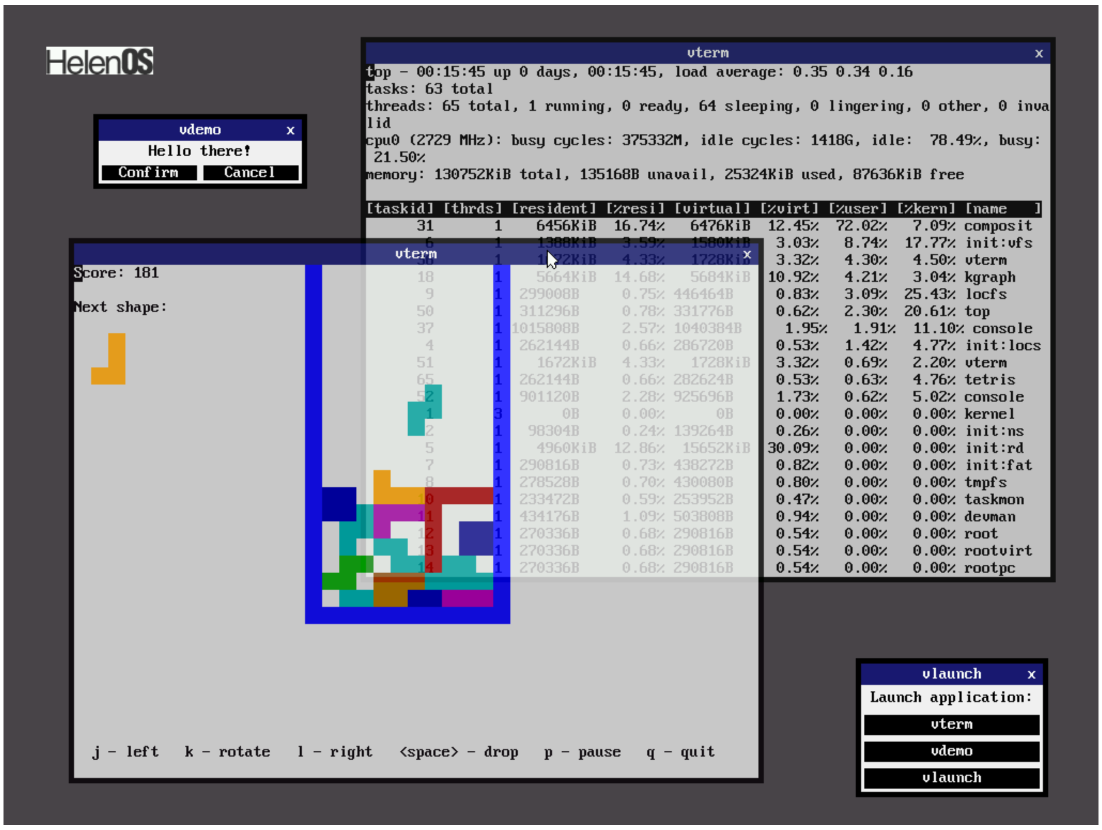
Signals can be emitted either synchronously via `sig_send` or asynchronously via `sig_post`. Synchronous invocation is handled like an ordinary function call – signal list is searched and if the signal is found, all of the connected slots are called with the given arguments. Asynchronous invocation is facilitated by posting `ET_SIGNAL_EVENT` into the event loop. Note that in such case, the arguments must be copied and carried as a part of the event because it is not certain whether they are allocated on the stack or on the heap. Once the event is popped from the event queue, the event handler works the same way as in the case of synchronous invocation – i.e. the asynchronicity is achieved by the event queue. Since the event queue is properly synchronized, it is possible to send asynchronous signal even from the fibril that is different from the one in which the event loop is executed, which makes a cross-fibril communication of widgets possible.

The initial implementation of the widget toolkit is equipped with grid layout widget, button widget, label widget and terminal emulator widget. Note that the terminal emulator widget is different in the feature branch and the mainline version of the graphics stack – while in the feature branch, terminal widget emulates the graphics driver, mainline version emulates the console server instead. Also because the feature branch version of the terminal can be optionally buffered, it demonstrates how to draw a buffered widget. The grid layout widget allows splitting the window surface into multiple equally sized rectangles. Each child widget can be then assigned a rectangular area consisting of one or more of these rectangles. The rest of the mentioned widgets is currently intended more for the demonstration purposes rather than serious usage in applications. Both label and button widgets have only very basic visual appearance. Moreover, the button widget needs more robust mouse event handling.

To conclude this chapter, it is suggested to review the code of the example application shown in the **Appendix B**. It demonstrates the overall usage of the widget toolkit from the application programmer's perspective, particularly the signal-slot mechanism and the extension of the label widget via inheritance.

5 Conclusion

Presented thesis describes the design and implementation of a proper graphics stack for HelenOS. The thesis deals with all major parts of the stack – i.e. graphic driver interface, drawing library, graphics server and widget toolkit. As the topic is very extensive, the thesis includes the analysis of target priorities for both the design and the implementation phase, so the resulting product would be complete enough to be actually used and then gradually extended by the HelenOS developer community. The individual parts of the thesis correspond to the major parts of the created graphics stack. Each part is analyzed with the respect to existing solutions, target priorities and HelenOS architecture. For each part, the thesis also provides detailed description of the resulting design and prototype implementation.



```
top - 00:15:45 up 0 days, 00:15:45, load average: 0.35 0.34 0.16
tasks: 63 total
threads: 65 total, 1 running, 0 ready, 64 sleeping, 0 lingering, 0 other, 0 inva
lid
cpu0 (2729 MHz): busy cycles: 375332M, idle cycles: 1418G, idle:  78.49%, busy:
 21.50%
memory: 130752KiB total, 135168B unavail, 25324KiB used, 87636KiB free

[taskid] [thrds] [resident] [%resi] [virtual] [%virt] [%user] [%kern] [name    ]
      31       1    6456KiB  16.74%    6476KiB  12.45%  72.02%   7.09% composit
       6       1    1388KiB   3.59%    1580KiB   3.03%   8.74%  17.77% init:vfs
      58       1    1672KiB   4.33%    1728KiB   3.32%   4.30%   4.50% vterm
      18       1    5664KiB  14.68%    5684KiB  10.92%   4.21%   3.04% kgraph
       9       1  299008B    0.75% 446464B     0.83%   3.09%  25.43% locfs
      50       1  311296B    0.78% 331776B     0.62%   2.30%  20.61% top
      37       1 1015808B    2.57% 1040384B     1.95%   1.91%  11.10% console
       4       1  262144B    0.66% 286720B     0.53%   1.42%   4.77% init:locs
      51       1    1672KiB   4.33%    1728KiB   3.32%   0.69%   2.20% vterm
      65       1  262144B    0.66% 282624B     0.53%   0.63%   4.76% tetris
      52       1  901120B    2.28% 925696B     1.73%   0.62%   5.02% console
       1       3       0B    0.00%       0B    0.00%   0.00%   0.00% kernel
       2       1   98304B    0.24% 139264B     0.26%   0.00%   0.00% init:ns
       5       1    4960KiB  12.86%   15652KiB  30.09%   0.00%   0.00% init:rd
       7       1  290816B    0.73% 438272B     0.82%   0.00%   0.00% init:fat
       8       1  278528B    0.70% 430080B     0.80%   0.00%   0.00% tmpfs
      10       1  233472B    0.59% 253952B     0.47%   0.00%   0.00% taskmon
      11       1  434176B    1.09% 503808B     0.94%   0.00%   0.00% devman
      12       1  270336B    0.68% 290816B     0.54%   0.00%   0.00% root
      13       1  270336B    0.68% 290816B     0.54%   0.00%   0.00% rootvirt
      14       1  270336B    0.68% 290816B     0.54%   0.00%   0.00% rootpc
```

```
Score: 181

Next shape:

 j - left   k - rotate   l - right   <space> - drop   p - pause   q - quit
```

Figure 13 Visual appearance of user interface just before merging the project into mainline.

5.1 Accomplishments

All the goals set for this thesis were achieved. Bottom layer of HelenOS graphics stack is now more aligned with HelenOS device driver framework and equipped with a new library containing the graphic driver infrastructure. For now, the prototype implementation of graphic driver infrastructure contains support for multiple output screens and management of their frame buffers but lacks the support for direct rendering.

On top of the graphic driver infrastructure resides the graphics server that acts as a central communication hub between multiple output screens, input devices and application clients. In order to provide means to implement the graphics server and application clients, two auxiliary libraries were created – drawing library and widget toolkit. The cooperation between graphics server, drawing library and widget toolkit provides several features for blending the windows of individual clients onto the logically infinite desktop which parts are mapped onto the frame buffers. There can be arbitrary number of overlapping decorated windows on the desktop. Each window can be resized, moved, scaled, rotated or made partially transparent.

While the graphics server can be considered as the most evolved part of the stack, both drawing library and widget toolkit contain only as much functionality as was required for the implementation of the graphics server and the initial set of applications. The drawing library provides the pixel transfers supporting affine transformations, masking, clipping and alpha blending but lacks the capability to draw any graphic primitives apart from the basic bitmap font. Similarly, the widget toolkit contains the necessary infrastructure (i.e. event queue, widget object hierarchy, scene graph, signal-slot mechanism) but offers only a very limited set of widgets – grid layout, label, button and most importantly terminal emulator. In order to demonstrate abilities of the prototype implementation, three basic applications were created – application launcher, signal-slot demo and console terminal.

The prototype implementation was positively accepted by the HelenOS developer community and after some modifications, improvements and optimizations it is now merged into the project main repository. The look and feel of the resulting product just before the merging into mainline can be seen in the tutorial video available either online [31] or on the attached medium. Alternatively, you can refer to **Figure 13**.

5.2 Added Value

Addition of the graphics stack into the system means that HelenOS is one more step closer to its overall strategic goal to become general-purpose operating system. Generally speaking, replacement of the temporary implementations of subsystems by the properly designed ones enables HelenOS to further grow and cover more usage scenarios. As for the new graphics stack, there are numerous occurrences of this phenomenon. First of all, the graphic driver infrastructure simplifies future addition of new graphic drivers as it provides the common functionality that would be otherwise implemented by each driver on its own. The drawing library and the widget toolkit have the same effect on the addition of new applications. Continuing, the graphics stack enables HelenOS users, and more importantly its current developers, to see more information on the screen as it is now possible to see output of multiple console terminals simultaneously. Since the current implementation of the graphics stack covers more breadth than depth, its addition also creates numerous opportunities for the current and especially for the new developers to accomplish their ideas and contribute to the project. Finally, the improved visual appearance of the system can attract more attention from both enthusiasts and potential new contributors, therefore accelerating the project development in a long term.

5.3 Future Work

With respect to the topic extent, there are numerous directions in which the current prototype implementation could be broadened. The following list enumerates the most obvious of them:

- Support for direct rendering in graphic driver infrastructure.
- Proper graphic driver able to leverage acceleration potential of the underlying device.
- More drawing operations in drawing library (lines, curves, polygons, circles, etc.).
- Retained scene description in drawing library.
- Hardware acceleration of drawing library.
- Better quality of drawing library output (advanced filtering, antialiasing).
- Better precision of drawing library output (floating point coordinates and pixels).
- Addition of trigonometric operations into HelenOS C library.
- Drawing library and graphics server support for arbitrary-angle rotation of windows.

- Support for more image and font formats in drawing library.
- More widgets in widget toolkit (menu bars, scrolls bars, edit boxes, list boxes, etc.).
- Support to minimize and maximize windows.
- Desktop environment (task bar, icons).
- Improvements of terminal emulator (resizing, scrolling, history).
- Console applications awareness of resizing and termination of their terminal emulator.
- Further (platform-specific) optimizations and performance improvements.

Clearly, some of the listed extensions are more challenging than the others. It could be also beneficial to port some established drawing library or widget toolkit to HelenOS in order to accommodate existing applications from other operating systems.

Bibliography

[1] *HelenOS 0.2.0 Design Documentation*. Available online, http://www.helenos.org/doc/design.pdf, 2006.

[2] Jeřmář, J. *HelenOS IPC for Dummies*. Available online, http://trac.helenos.org/wiki/IPC, 2009.

[3] Svoboda, J. *Async Sessions*. Available online, http://trac.helenos.org/wiki/AsyncSessions, 2010.

[4] *Compiling HelenOS From Source*. Available online, http://trac.helenos.org/wiki/UsersGuide/CompilingFromSource, 2013.

[5] Trochtová, L. *Device drivers interface in HelenOS system*. Available online, http://www.helenos.org/doc/theses/lt-thesis.pdf, 2010.

[6] Microsoft Corp. *Windows Vista Display Driver Model Design Guide*. Available online, http://msdn.microsoft.com/en-us/library/windows/hardware/ff570593, 2012.

[7] Microsoft Corp. *DXGI Overview*. Available online, http://msdn.microsoft.com/en-us/library/bb205075, 2012.

[8] Martin, K.E., Faith, R. E., Owen, J., Akin, A. *Direct Rendering Infrastructure, Low-Level Design Document*. Available online, http://dri.sourceforge.net/doc/design_low_level.html, 1999.

[9] Fonseca, J. *Gallium3D: Introduction*. Available online, http://jrfonseca.blogspot.de/2008/04/gallium3d-introduction.html, 2008.

[10] Apple Computer, Inc. *Imaging With QuickDraw*. Available online, http://developer.apple.com/legacy/mac/library/documentation/mac/pdf/ImagingWithQuickDraw.pdf, 1994.

[11] Apple Computer, Inc. *Quartz 2D Programming Guide*. Available online, https://developer.apple.com/library/mac/#documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html, 2012.

[12] Gettys, J., Scheifler, R. W. *Xlib − C Language X Interface*. Available online, http://www.x.org/docs/X11/xlib.pdf, 2002.

[13] Microsoft Corp. *Graphics and Multimedia*. Available online, http://msdn.microsoft.com/en-us/library/aa969176.aspx, 2010.

[14] *Cairo Documentation*. Available online, http://cairographics.org/documentation/, 2012.

[15] *Skia Documentation*. Available online, https://sites.google.com/site/skiadocs/, 2012.

[16] Scheifler, R. W., Gettys, J. *The X Window System*. ACM Transactions on Graphics, Vol. 5, No. 2, pp. 79-109, 1986.

[17] Microsoft Corp. *Desktop Window Manager*. Available online, http://msdn.microsoft.com/en-us/library/aa969540.aspx, 2012.

[18] Russel, M. *What Is Quartz (or Why Can't Windows Do That)*. Available online, http://oreilly.com/pub/a/mac/2005/10/11/what-is-quartz.html, 2005.

[19] Høgsberg, K. *The Wayland display server*. Available online, http://wayland.freedesktop.org/docs/html/, 2012.

[20] Porter, T., Duff, T. *Compositing Digital Images*. Computer Graphics, Vol. 8, pp. 253–259, 1984.

[21] Chapuis, O., Roussel, N. *Metisse is not a 3D desktop!*. Proceedings of UIST'05, pp. 13-22, 2005.

[22] *Metisse on GNU/Linux demo*. Available online, http://www.youtube.com/watch?v=dxsUKX6xXyE, 2006.

[23] Feske, N., Helmuth, Ch. *A Nitpicker's guide to a minimal-complexity secure GUI*. Proceedings of ACSAC'05, pp. 85-94, 2005.

[24] Starynkevitch, B. *Description of NeWS*. Available online, http://starynkevitch.net/Basile/NeWS_descr_oct_1993.html, 1993.

[25] Samek, M. *Who Moved My State?*. Available online, http://www.drdobbs.com/who-moved-my-state/184401643, 2003.

[26] *Qt Documentation*. Available online, http://qt-project.org/doc/, 2012.

[27] *GTK+ Documentation*. Available online, http://www.gtk.org/documentation.php, 2012.

[28] *wxWidgets Documentation*. Available online, http://wxwidgets.org/docs/, 2012.

[29] *IUP Documentation*. Available online, http://www.tecgraf.puc-rio.br/iup/ , 2012.

[30] *FLTK Documentation*. Available online, http://www.fltk.org/documentation.php, 2012.

[31] *HelenOS GUI (experimental demonstration)*. Available online, http://www.youtube.com/watch?v=ZjqYRv2xOSw, 2012.

Appendix A

Attached CD-ROM structure

`/binaries/`

Compiled bootable images of HelenOS for 32-bit Intel platform of both the feature branch and the project mainline. The images are mainly intended to be run within a virtual machine environment provided either by VirtualBox (https://www.virtualbox.org) or qemu (http://www.qemu.org). However, it is also possible to use a physical machine if it has a processor compatible with Pentium 4. Make sure that the (virtual) machine has at least 128MB of system memory. Both images are configured to use the new graphics stack at 1024×768 resolution with 16-bit color depth. For convenience, the directory further contains the portable version of qemu that allows trying HelenOS from within Windows operating system without installing anything. For example, to boot the mainline image, just execute the `run-helenos-mainline-1756M-ia32.bat` script. Note that the performance of this convenient solution is fairly limited due to the lack of any emulation acceleration. For the full performance, use either VirtualBox or qemu KVM[23].

`/sources/`

Exported source trees for both the feature branch and the project mainline. While the feature branch presents the graphics stack as was originally intended just before the integration, project mainline reflects also the feedback of HelenOS developer community and contains some additional modifications, improvements and optimizations.

`/media/`

Video file containing the screencast demonstration of the feature branch as it appeared just before the integration of the project into the mainline. The video shows the capabilities and features of the graphics stack. It also shows how to control the user interface via keyboard and mouse.

[23] Kernel-based Virtual Machine

Appendix B

Widget toolkit usage example

```
#include <window.h>
#include <grid.h>
#include <button.h>
#include <label.h>

#define NAME "vdemo"

typedef struct my_label {
  label_t label;
  slot_t confirm;
  slot_t cancel;
} my_label_t;

static void deinit_my_label(my_label_t *lbl) {
  deinit_label(&lbl->label);
}

static void my_label_destroy(widget_t *widget) {
  my_label_t *lbl = (my_label_t *) widget;
  deinit_my_label(lbl);
  free(lbl);
}

static void on_confirm(widget_t *widget, void *data) {
  my_label_t *lbl = (my_label_t *) widget;
  const char *confirmed = "Confirmed";
  lbl->label.rewrite(&lbl->label.widget, (void *) confirmed);
}

static void on_cancel(widget_t *widget, void *data) {
  my_label_t *lbl = (my_label_t *) widget;
  const char *cancelled = "Cancelled";
  lbl->label.rewrite(&lbl->label.widget, (void *) cancelled);
}

static bool init_my_label(my_label_t *lbl, widget_t *parent,
    const char *caption, uint16_t points,
    pixel_t background, pixel_t foreground) {

  lbl->confirm = on_confirm;
  lbl->cancel = on_cancel;
  bool initialized = init_label(
      &lbl->label, parent, caption, points, background, foreground);
  lbl->label.widget.destroy = my_label_destroy;
  return initialized;
}

static my_label_t *create_my_label(widget_t *parent,
    const char *caption, uint16_t points,
    pixel_t background, pixel_t foreground) {

  my_label_t *lbl = (my_label_t *) malloc(sizeof(my_label_t));
```

```
    if (!lbl) {
        return NULL;
    }

    bool initialized = init_my_label(
        lbl, parent, caption, points, background, foreground);
    if (initialized) {
        return lbl;
    } else {
        free(lbl);
        return NULL;
    }
}

int main(int argc, char *argv[]) {
    window_t *main_window =
        window_open(argv[1], true, true, "vdemo", 0, 0);
    if (!main_window) {
        printf("Cannot open main window.\n");
        return 1;
    }

    pixel_t grd_bg = PIXEL(255, 240, 240, 240);
    pixel_t btn_bg = PIXEL(255, 0, 0, 0);
    pixel_t btn_fg = PIXEL(255, 240, 240, 240);
    pixel_t lbl_bg = PIXEL(255, 240, 240, 240);
    pixel_t lbl_fg = PIXEL(255, 0, 0, 0);

    my_label_t *lbl_action =
        create_my_label(NULL, "Hello there!", 16, lbl_bg, lbl_fg);
    button_t *btn_confirm =
        create_button(NULL, "Confirm", 16, btn_bg, btn_fg);
    button_t *btn_cancel =
        create_button(NULL, "Cancel", 16, btn_bg, btn_fg);
    grid_t *grid = create_grid(window_root(main_window), 2, 2, grd_bg);
    if (!lbl_action || !btn_confirm || !btn_cancel || !grid) {
        window_close(main_window);
        printf("Cannot create widgets.\n");
        return 1;
    }

    sig_connect(
        &btn_confirm->clicked,
        &lbl_action->label.widget,
        lbl_action->confirm);
    sig_connect(
        &btn_cancel->clicked,
        &lbl_action->label.widget,
        lbl_action->cancel);

    grid->add(grid, &lbl_action->label.widget, 0, 0, 1, 2);
    grid->add(grid, &btn_confirm->widget, 1, 0, 1, 1);
    grid->add(grid, &btn_cancel->widget, 1, 1, 1, 1);
    window_resize(main_window, 200, 70);

    window_exec(main_window);
    task_retval(0);
    async_manager();
    return 1;
}
```