Lumina-DE:

Redefining the Desktop Environment for Modern Hardware

Author:

Ken Moore
ken@pcbsd.org
PC-BSD/iXsystems

Lumina Desktop source repository:
https://github.com/pcbsd/lumina

Date:
Nov 2014

Abstract:

As computers continue to advance into every aspect of our daily lives through the pervasiveness of cell phones and tablets, the traditional “desktop computer” is gradually being shifted to a smaller subset of the total systems in use. This presents a problem for open source operating systems, as the available open source graphical environments are increasingly designed for systems with powerful hardware or traditional mouse/keyboard inputs – resulting in a much lower percentage of devices that are physically capable of utilitizing the OS. The open-source Lumina desktop environment is designed to solve these problems by meeting its goals of being a highly flexible and scalable interface that runs with relatively little hardware requirements. The project also provides a simple framework for integrating OS-specific functionality directly into the interface for ease-of-use without causing conflict with the underlying system or affecting portability. This paper will take a top-level view of the Lumina desktop project, breaking it down to its components, explaining the framework and methodology, and listing the work that is still yet to be completed to achieve its goals.

Please note: for all intents and purposes, there is no distinction between laptops and box-based desktop computers when it comes to the capabilities and distinctions of a graphical interface, so for the purposes of this paper they will both be considered “desktop” systems.

The Problem:

Smartphones, tablets, laptops and desktop computers all utilitize graphical interfaces to provide the user access to the capabilities of the device, but laptops and desktop computers are the only ones with fully open source desktop environments available. While there are many reasons for this, this paper will focus on the challenge of providing a single graphical interface that will function across all types of devices.

The most obvious difference that users will notice is that of the screen size. Desktops typically have screens that are at least 14+ inches along the diagonal (20+ for box-based systems) and can be easily connected to television screens that are much larger or be connected to multiple screens at once. In contrast, cell phones screens are typically less than 5 inches along the diagonal and tablets are usually 7-12 inches along the diagonal, both of them with almost no ability to support additional screens. Where this difference usually causes problems with a graphical interface is with managing the running applications. If you have a smaller screen (tablet/phone), you generally want to run every application in full-screen mode so that the screen is utilized to its maximum capabilities. In contrast, desktop systems traditionally show multiple applications on the screen at a single time, either through a tiling algorithm or by stacking the windows in layers with the currently active window on top. This difference in the basic nature of application management will need to be resolved for a graphical environment to function on all the various sizes of systems.

The next major difference between systems is the method of user input. Desktop computers rely on the traditional mouse and keyboard combination (or some variation), while smartphones and tablets usually rely on a touchscreen interface (with the occasional use of a keyboard/mouse on tablets as well). This is a fairly serious distinction that will result in major differences in the layout and design of the interface itself. For example, look at the changes in the GNOME desktop environment between versions two and three. GNOME2 was designed for the traditional mouse/keyboard desktop while GNOME3 was redesigned as a touch-based interface (primarily) and it stirred up controversy among desktop users and resulted in two forks of the GNOME source code (the MATE and CINNAMON desktop environments) with the primary purpose of maintaining the traditional desktop layout. This is only one example of how much the interface layout and design can make or break a system's usability.

The final differences are less about the interface and more about the available hardware. Smartphones and tablets generally have lower speed processors (if it has more than one) in order to reduce power consumption, while desktops are generally much more powerful. This difference is also mirrored in the available system memory, available disk space, and electrical power requirements with the desktop systems easily having more of everything. This means that if a graphical environment is developed primarily for a desktop system, it will generally be too “heavy” to use on smartphones or tablets on a daily basis as it places much greater limitations on the available hardware that applications need to use to function.

The Solution:

To address these issues with scaling a graphical environment across all the different types of systems, a multi-pronged approach must be taken which divides up the system software loadout into a number of individual pieces which can each be optimized on a per-device basis: the operating system, the graphical interface, and the default applications. With a BSD operating system, the first piece is extremely easy because the OS is already designed to be kept completely seperate from any system packages and is already managed/optimized seperately. Any required applications are also easily separated out and can be left up to the choice of the system distributor since they will know what types of devices the interface is going to be used on. For example, if the system includes a built-in camera

(such as for a cell phone or a tablet), the distributor will want to include an application for taking and saving pictures that works with that particular device. This gives the system distributors the freedom to create particular applications or utilities for very targeted usage without having the graphical interface come bundled with a bunch of applications that either do not function properly with the hardware or are pointless for that particular type of system.

As a result, the graphical environment is left with only a few simple tasks: provide an interface for launching applications that is configurable for various screen sizes, a window manager for keeping track of the number of screens and any running applications, a backend system for applications to be able to launch files/applications as necessary, and perform it all with as little system overhead as possible so that it can function on small devices. Now let us go through each of these different tasks, and see how the Lumina desktop environment is designed to satisfy these requirements.

Desktop Interface Scaling:

To accomplish scalability of the interface itself, the Lumina desktop environment is designed at its core to simply be a blank canvas for the system with all the interface elements able to be added/removed/moved at any time. This is accomplished by placing all the non-graphical system administration functionality within the Lumina Session class (LSession.[cpp/h]), while creating a single graphical desktop (LDesktop.[cpp/h]) for each screen upon which can be placed a wallpaper image as well as any interface plugins. However, since the desktop often gets covered or hidden by applications, it is understandable that a particular screen edge may need to be reserved for a canvas that remains visible at all times, and the Lumina Panel class (LPanel.[cpp/h]) was created for this purpose. The only other differences between a panel and the desktop are that a panel is colored instead of having a wallpaper image, and that the panel plugins are arranged in a 1-dimensional line instead of a two-dimensional area (resulting in a slightly different base plugin template). The result of this framework is that, by creating various types of interface plugins with the same basic functionality, it will be possible to easily support various types of systems since either the distributor (for initial settings) or the user (for customization later) can simply change the plugins that are used on their particular system.

For an example of this approach, observe the traditional “task manager” functionality of a desktop. This feature typically provides individual buttons for each application window (usually in some kind of reserved screen space), allowing the user to quickly see what applications are running and perform tasks such as activate or close each application. For smartphones or tablets, there is not enough screen space for individual buttons for each application window, nor will the user want the additional system overhead for maintaining the status of every application window at all times. In Lumina, a few different plugins are provided for task manager functionality. For the traditional desktop, there is the standard button-based panel interface plugin, while alternate plugins could be created to simply provide a button which, when triggered, generates a list of all the running applications in either a popup menu or a full screen slideshow form. This plugin decreases much of the system overhead for managing applications and turns it into a service that is only activated on demand, satisfying both requirements for small-screen devices such as smartphones, while also providing various interface options for users with more powerful systems.

Please note that with the current version of the Lumina desktop environment (0.7.2), the available plugins are generally desktop-oriented since that is the primary development platform for the project. However, the framework is already in place for additional plugins that are focused on smaller form-factor devices as well, even if those plugins have not been created at the present time.

Application Window Management:

A fully-scalable window manager (WM) has only a few requirements for functionality. The first thing it needs is to be able to support the various standards set in place by the FreeDesktop organization (the ICCCM and EWMH) since those standards are widely supported by almost the entire open-source and closed-source communities on *nix operating systems at the present time. These standards provide widely-used methods for applications to be able to register important information about themselves with the graphical subsystem so that users can quickly determine which applications are running and to distinguish between different instances of the same application. While adherence to these particular standards might not always be mandatory, it is highly recommended for any open source desktop environment to be able to communicate some basic information with any running applications in a standardized format.

Second, the WM should be able to support various modes of operation, with the ability to change modes without much (if any) downtime. The first mode of operation that should be supported is a single-application mode for small screens where only a single application window will be visible at a given time and uses all available screen space (excluding any reserved space for a panel). This is probably the simplest mode to implement, as it removes most of the variability of showing multiple applications on a single screen. The second mode of operation is that of the traditional “stacking” window manager, where each window has a frame with simple management controls on it and allows the windows to exist anywhere within the available screen space. This mode is so named because it includes the ability to stack the most recently used applications on top of older applications, partially hiding some windows from view as necessary. The final mode of operation is that of a “tiled” window manager, where each window may have the surrounding frame like the “stacking” mode, but the windows are organized into tiles with the active window accupying the majority of the screen space, and the inactive windows shrunken down to tiny icons or tiles surrounding the active window. This mode shares many capabilities with the stacking mode, and I think it can be accomodated by simply creating tiling algorithms for the stacking mode since the only major difference is the location and size of the windows on the screen. These tiling algorithms may then be used whenever a new application window is created, or on-demand as the user preferences indicate.

The Lumina desktop environment currently relies on the Fluxbox window manager to provide window management functionality, but there are a few compelling reasons to create a new WM to replace Fluxbox in the near future. First, Fluxbox only operates as a “stacking” manager and does not provide any type of single-window functionality, preventing the current version of Lumina from being used on small screen devices. Second, Fluxbox utilizes its own internal graphics engine for all window frames, causing a sharp disconnect with the visuals of Qt and the preventing any kind of coherence with the Lumina themes. Third, Fluxbox operates a few other background services (toolbar, system tray, task manager) that are superceded by the Lumina operations and runtime resources could be saved by not including these services in the WM at all. Finally, the current state of Fluxbox development appears to have stalled, with reported but unrepaired bugs, ineffective window placement algorithms, and unimplemented user-submitted patches inhibiting the project's usability. At this point in time, I think a compelling argument can be made for the Lumina project to create its own Qt-based window manager as it will result in less runtime overhead and can better fulfill the requirements listed above.

Application Launching Framework:

One of the final pieces necessary for a graphical system is a method for applications to communicate that the system needs to open a file/dir/URL with the appropriate application. For example, if the user just downloaded a file with a web browser, they generally just want to click on the download notification to open up that file immediately, instead of exiting the browser, moving to the download directory, and then opening the file. The way this has traditionally been performed on open

source desktops is through the use of the *xdg-open* utility. It is just a little shell script that detects which desktop environment is currently running, and then forwards the request on to the launching utility for that desktop. For Lumina, this meant that it needed to have a designated utility for launching applications and it was named *lumina-open*. The *lumina-open* utility is designed to parse the input string, determine what type of input it might be, see if there is a default application registered for that type of input (using the FreeDesktop MIME-type specifications whenever possible) and either launch the application if one is registered or prompt the user to select which application to use for that type of input (giving recommendations if possible). To support keyboard shortcuts (or other types of buttons depending on the hardware) for common tasks, the *lumina-open* utility also has support for running specific OS interactions. Specifically, it can change the audio volume or screen brightness with the current version. These types of OS interactions are governed by a very simple group of functions within the Lumina library, the "LuminaOS" class.

This class of functions is unique within the project, and it is designed to provide the ability for the graphical interface to provide status updates or notifications about various OS-specific functionality. For example, the way that a battery might be detected, or how to determine the amount of power left in a battery would be two functions within the LuminaOS class. Other interactions contained in this class provide the ability to provide shortcuts to the application "store" and the control panel for the OS, detect available external devices, control audio volume or launch the audio mixer, detect battery status, and perform shutdown/reboot operations on the system. Remember, Lumina is just an interface to OS operations, and should try to make them available to the user whenever possible. It does not try to create new features or functionality for the OS, but simply leverage them or provide user access to them. In order to make Lumina aware of different OS capabilities or to port Lumina to an OS, there is just a single source file in the library that needs to be created/maintained – possibly making Lumina one of the easiest graphical environments to port to various operating systems.

Minimizing system overhead:

There are a number of ways that the Lumina project tries to reduce system usage, but we will quickly list a few of the major ones. First, Lumina relies on the Qt runtime framework almost exclusively for standard operations, and only needs access to the X11 Libraries for some of the more window-focused operations. This prevents massive dependency chains or the requirements of multiple programming toolkits or modules on the system. Second, Lumina has absolutely no dependancy on any long-lived or hidden background processes (unless you count *xscreensaver*) – so there is nothing in the background that is constantly using CPU cycles or memory unless the OS requires it. Third, the number of 3rd party utilities is kept to a bare minimum as Lumina currently only requires *xscreensaver* and *numlockx*, with *xbrightness* an OS requirement for FreeBSD. This not only saves hard drive space for a basic installation, but also limits the number of (possibly) exploitable utilities installed on the system. One of the future plans for reducing this list even further is to implement a custom Qt-based screensaver to remove *xscreensaver* and all of its dependencies as well, although there is no official schedule for when that will happen.

Current state of the project:

Version 0.7.2 of the Lumina desktop was tagged in the source repository on November 19th, 2014, and this version is considered to be a beta-quality release for desktop enthusiasts or users familiar with the command line. It is highly stable and usable on a daily basis, but is still lacking a variety of interface plugins as well as facing the possibility for backend changes to require that the user's custom settings be reset to defaults occasionally. As such, this is only recommended to be used for traditional

desktop systems at the present time. Version 0.8.0 is currently in development, and is scheduled to include the updates necessary to move from the Qt4 toolkit to the newer Qt5 toolkit.

Conclusions:

The Lumina desktop project is making significant progress to providing a fully scalable, open source, graphical interface that should function on almost every type of device regardless of screen size or input format. The final pieces of the project necessary to accomplish this goal are the creation of a new window manager, the expansion of the variety of plugins for different types of devices, the continued streamlining and optimizing of the Lumina system itself, and the replacement of the few remaining third-party application dependencies. The Lumina project, when fully realized, can help the open-source community breach the divide between the personal computer and mobile device markets, allowing secure and transparent operating systems to compete with the various closed-source operating systems in wide use today.