Rapid Development of User Interfaces on Cluster-Driven Wall Displays with jBricks

Emmanuel Pietriga, Stéphane Huot, Mathieu Nancel, Romain Primet

Rapid Development of User Interfaces on Cluster-Driven Wall Displays with jBricks

Emmanuel Pietriga[1,2] **Stéphane Huot**[2,1] **Mathieu Nancel**[2,1] **Romain Primet**[1]

[1]INRIA
F-91405 Orsay, France

[2]LRI - Univ Paris-Sud & CNRS
F-91405 Orsay, France

ABSTRACT

Research on cluster-driven wall displays has mostly focused on techniques for parallel rendering of complex 3D models. There has been comparatively little research effort dedicated to other types of graphics and to the software engineering issues that arise when prototyping novel interaction techniques or developing full-featured applications for such displays. We present jBricks, a Java toolkit that integrates a high-quality 2D graphics rendering engine and a versatile input configuration module into a coherent framework, enabling the exploratory prototyping of interaction techniques and rapid development of post-WIMP applications running on cluster-driven interactive visualization platforms.

Keywords

Wall Displays, Clusters, Interaction, Toolkit, Prototyping

INTRODUCTION

Over the last decade, wall-sized displays have evolved from experimental, CRT monitor-based setups to sophisticated arrays of tiled projectors or LCD panels. The latter are often called *ultra-high-resolution* displays to emphasize their significantly higher display capacity compared to projector-based *very-high-resolution* displays. They typically accommodate several hundred megapixels, and are driven by clusters of computers. As an example, the setup depicted in Figure 1 uses 32+1 graphic processing units in 16+1 computers to display $20480 \times 6400 \simeq 131$ megapixels on a $5.5m \times 1.8m$ surface ($\simeq 100dpi$). These displays enable the visualization of truly massive datasets. They can represent the data with a high level of detail while retaining context [4, 16], and enable the juxtaposition of data in various forms [1]. To make them interactive, wall-sized displays are increasingly coupled with advanced input devices, e.g., motion-tracking systems, wireless multitouch devices, in order to enable multi-device and/or multi-user interaction with the displayed data [16, 17]. These interactive ultra-high-resolution displays can be used in many application domains, including command and control centers, geospatial imagery, scientific visualization, collaborative design and public information displays.

E. Pietriga, S. Huot, M. Nancel, R. Primet.
OntoTrix: A Hybrid Visualization for Populated Ontologies.
In WWW '11: Proceedings of the 20th international conference companion on World wide web, ACM, March 2011.
Authors Version

These new environments pose new research challenges. From a *computer graphics perspective*: how to render complex graphics at high frame rates, taking advantage of the cluster's computing and rendering power. From a *human-computer interaction perspective*: how to design effective visualizations that take advantage of the specific characteristics of large, ultra-high-resolution surfaces; how to design interaction techniques that are well-adapted to this particular context of use, and how to handle the multiple and heterogeneous input devices and modalities typically used in this context. Finally, from a *software engineering perspective*: how to enable the rapid prototyping, development, testing and debugging of interactive applications running on clusters of computers, providing the right abstractions.

In this paper, we focus on the latter research question, that we consider essential to foster more research and development from the HCI perspective. We present jBricks, a Java toolkit for the development of post-WIMP applications executed on cluster-driven wall displays, that extends and integrates a high-quality 2D graphics rendering engine and a versatile input management module into a coherent framework hiding low-level details from the develeoper. The goal of this framework is to ease the development, testing and debugging of interactive visualization applications. It also offers an environment for the rapid prototyping of novel interaction techniques and their evaluation through controlled experiments, such as the one we recently conducted about mid-air pan-and-zoom techniques for wall-sized displays [16].

Background and Motivation

The parallel-rendering techniques developed over the last ten years enable the efficient display of 3D graphics on tiled displays driven by clusters of computers. This is usually done by sending already rendered images to the cluster nodes, or by sending geometry and performing compositing operations to produce the final wall-sized image. Different techniques exist, including *sort-first* and *sort-last* pipelines as well as various hybrid solutions. Well-known frameworks include Chromium [11], Equalizer [10] and SAGE [13]. See Ni *et al.* [17] for a comprehensive survey.

However, not all wall display applications use 3D graphics. With the introduction of ultra-high resolution, high-quality 2D graphics open wall-sized displays to new applications, e.g., in astronomy, geospatial intelligence and visual analytics at large, to give a few examples. These applications essentially combine very large bitmap images, high-quality text and 2D vector graphics, e.g., satellite imagery aug-

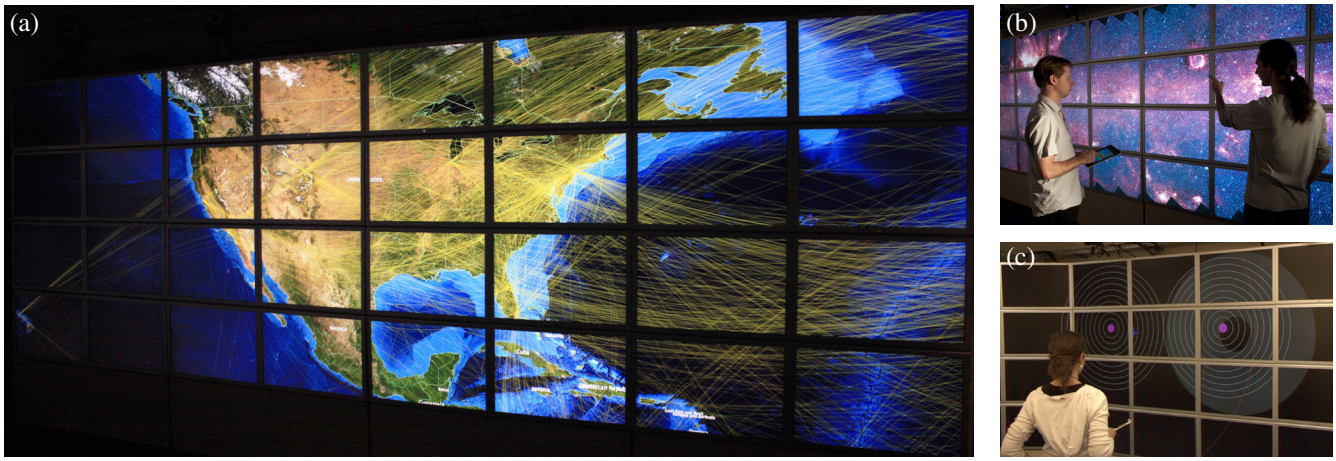

Figure 1. jBricks applications running on the WILD platform (32 tiles for a total resolution of 20 480 × 6 400 pixels). (a) Zoomed-in visualization of the North-American part of the world-wide air traffic network (1 200 airports, 5 700 connections) overlaid on NASA's Blue Marble Next Generation images (86 400 × 43 200 pixels) augmented with country borders ESRI shapefiles. (b) Panning and zooming in Spitzer's Infrared Milky Way (396 032 × 12 000 pixels). (c) Controlled laboratory experiment for the evaluation of mid-air multi-scale navigation techniques [17].

mented with data layers, or information visualization techniques for the display of large datasets, e.g., for the visual exploration of large networks (Figure 1-a). However, there is currently no good solution for the distributed rendering of high-quality 2D graphics on cluster-driven wall displays.

Low-level 3D graphics APIs such as OpenGL are currently the main solution for developing cluster-driven visualizations. They work well for the high-performance visualization of textured 3D scenes, but are ill-suited to programming high-quality 2D graphics interfaces, lacking appropriate support for the management and efficient rendering of text, line styles, arbitrary 2D shapes and WIMP widgets. This was already observed for desktop application programming [6], and remains true for cluster-driven wall-displays. Pixel streaming approaches à la SAGE work well when combining different windows of relatively limited size from different applications, potentially running on different machines. They would however not work for full-screen, highly-dynamic visualizations on ultra-high-resolution displays: updating hundreds of megapixels forming a single coherent image at an interactive refresh rate would require significantly more network bandwidth than is commonly available and would put an extremely heavy load on the node in charge of rendering the image to be streamed.

Rich interactive 2D desktop applications, usually termed post-WIMP applications, are typically developed with structured graphics toolkits [2, 7, 12, 18] that provide useful abstractions on top of low-level APIs. They enable rapid prototyping and development of advanced interactive visualizations. Our goal is to offer a structured graphics toolkit capable of running transparently on cluster-driven wall displays and capable of handling a wide range of input devices and modalities. From a graphics perspective, this requires hiding the complexity entailed by having to distribute rendering on multiple computers. While our focus is on expressiveness and ease-of-use, we also pay attention to scalability issues, adapting ideas originally developed for efficient distributed 3D rendering to our context, such as the use of a multicast protocol to transmit updates to cluster nodes, and a culling algorithm adapted to zoomable user interfaces. From an input management perspective, this requires going beyond the basic redirection mechanisms found in existing distributed rendering frameworks that only support conventional input devices, i.e., mouse and keyboard operated from the master computer. For now, support for other devices is mostly achieved *via* ad hoc solutions (drivers or libraries) that are strongly integrated and statically linked within applications. This approach is not generic and flexible enough when exploring and prototyping novel interaction techniques [9]. An alternative approach consists in providing high-level abstractions of input modalities that enable association and runtime substitution of devices. It has proven successful in other domains, including physical ubiquitous computing [5], virtual reality (Gadgeteer for VR Juggler [8]) and in the more general context of post-WIMP applications (ICon [9], Squidy [14]), and we adapt it to interactive wall displays.

jBricks FRAMEWORK ARCHITECTURE

The framework is essentially composed of two independent modules: one for managing all graphical operations, and one for handling input. The two modules are loosely coupled. They communicate via a dynamic plugin architecture and network sockets using high-level protocols such as OSC. This makes the framework highly flexible: modules can be instantiated multiple times and can run on different nodes.

Structured Graphics

Our goal is to provide an API and feature-set similar to those of desktop structured graphics toolkits [2, 7, 12, 18] while i) hiding the complexity entailed by distributed rendering, ii) promoting ease of learning and ease of use, and iii) enabling code reuse: visualization components initially developed for desktop computers should run on cluster-driven wall displays with minimal changes to the original application code. With these high-level objectives in mind, we chose to extend an existing structured graphics toolkit rather than start developing a new one from scratch.

We used the open-source ZVTM toolkit [18], that supports most Java2D drawing primitives but offers higher-level abstractions that ease the management and manipulation of graphical objects: rendering is handled in retained mode, meaning that the toolkit retains a complete model of the objects to be rendered. ZVTM follows a monolithic approach, as opposed to a polylithic one[1]. Experience has shown that monolithic approaches are conceptually easier to handle by developers, generate less lines of code and require managing a smaller number of objects [7]; properties that we consider of high importance for rapid UI development.

Featured types of graphical objects include polygons of arbitrary shape, splines, Swing widgets, bitmap images and high-quality text, with support for advanced stroke and fill patterns. Those objects (*Glyphs*) are placed on infinite drawing surfaces (*Virtual Spaces*) that are observed through one or more *Cameras*. A camera renders the objects that lie in its viewing frustum in a *View*, that corresponds to a window on the screen. The toolkit makes it easy to create zoomable user interfaces (cameras can be smoothly panned and zoomed). It supports multiple independent views, as well as *Portals* (views within views) [6], multiple layers within a view (each corresponding to a different camera), as well as a variety of built-in focus+context visualization techniques. Cameras and glyphs can be animated using various pacing functions.

Cluster-based Structured Graphics Rendering

jBricks' extension of ZVTM to render graphics on cluster-driven tiled displays is conceptually straightforward. It takes an approach similar to what *sort-first* algorithms do for parallel rendering of 3D graphics in retained mode: as ZVTM already enables multiple cameras to observe a given virtual space, implementing tiled rendering basically consists in sharing that virtual space between all cluster nodes and setting one camera per display tile. Each camera's viewing frustum is configured so that their juxtaposition forms an overall coherent image from the user's perspective, according to the physical layout of display tiles.

Distributed Virtual Spaces. jBricks adopts a client-server model [17]: as shown in Figure 2, a single instance of the application runs on a *client node*, generating the geometry (populating virtual spaces with glyphs) and distributing it to *render servers* running on *cluster nodes*. Virtual spaces and glyphs contained therein are broadcast to all cluster nodes. They are replicated and kept synchronized as glyphs are added, removed, or have their properties changed. Parallel rendering frameworks for 3D graphics have mainly focused on the visualization of static-geometry models where only the camera(s) are manipulated interactively. The applications that jBricks aims to support typically manage much more dynamic objects, both in terms of geometry and visual appearance (color, stroke, font, etc.), potentially requiring a lot of network bandwidth. Multicast communication can greatly decrease bandwidth requirements for those updates [15]. We use JGroups (`http://www.jgroups.org`) as our group communication layer, that provides reliable messaging over IP multicast. Over this layer, we exchange atomic changes called `Delta`, which are serialized Java objects representing a new value for a given glyph attribute, propagated to the corresponding glyphs on render servers.

[1] *Monolithic* toolkits primarily use compile time inheritance to extend functionality, while *polylithic* toolkits primarily use run-time composition to do so, typically using a scene graph [7].

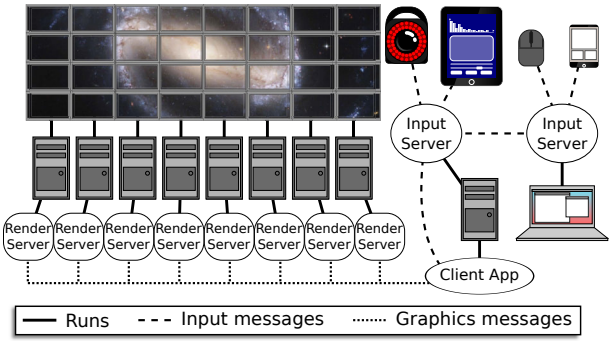

Figure 2. Example jBricks configuration: wall's graphics client and input server for motion tracker and tablet run on client node; input server for mouse, keyboard and smartphone run on user's laptop.

Performance. As noted by Bederson and Meyer [6] about zoomable user interfaces: *"Smooth real-time interaction is crucial. If the system becomes slow and jerky, the metaphor dies"*. The use of a multicast protocol for updating glyphs enables us to smoothly animate several hundred property changes simultaneously and independently of the number of render servers. Camera animations do not require significant bandwidth, as moving a camera only requires updating a maximum of three double-precision floating point values per frame. A more serious bottleneck when panning and zooming is the frame rate achieved by render servers. ZVTM already implements efficient culling algorithms for zoomable user interfaces. Glyphs get projected and rendered for a given camera only if they lie in that camera's viewing frustum. jBricks benefits from this directly: each server renders only the glyphs that will eventually be visible in the associated tile, which significantly decreases the computational and rendering load for scenes with high object counts.

Preliminary tests have shown that visualizations containing up to 200,000 objects could be rendered at interactive frame rates on the platform depicted in Figure 1. jBricks also benefits from Java2D's OpenGL pipeline, and from ZVTM's spatial indexing and dynamic external resource (un-)loading mechanisms. These were developed to support multi-scale navigation in very large datasets, such as gigapixel bitmap images decomposed recursively as a region quadtree. We adapted these mechanisms in jBricks to work in a distributed context, enabling the interactive visualization of very large images. Example images that have been visualized include the 26 gigapixel panorama of Paris (354 048 75 520 pixels) and Spitzer's Infrared Milky Way (Figure 1-b), that can be freely panned and zoomed on a wall display.

Programming. jBricks adds cluster support to ZVTM by monkey-patching the original toolkit using AspectJ, without altering its source code. This makes the cluster extension module small (3 000 lines of code vs. 39 000 for ZVTM) and facilitates forward compatibility. This also keeps API changes to a minimum: virtual spaces, glyphs, animations and most other constructs are managed through

the original ZVTM API; low-level mechanisms for distribution to render servers are hidden from the developer. Only cameras and views get created and managed in a slightly different manner. The tiled display's geometry has to be declared: number of rows and columns, size of each screen (pixels), options such as whether to paint pixels behind the bezels separating the tiles (overlay approach) or ignore them (offset approach). *Clustered Views* replace regular ZVTM views: a clustered view is divided into blocks, each block corresponding to a tile and render server. Render servers can be instantiated multiple times on a single node if that node drives multiple tiles. ZVTM-based desktop applications, originally written to run on single hosts, can be adapted to run on a cluster-driven large displays by changing as little as four lines of code. Render servers are instances of a generic display program that is part of jBricks, meaning that developers only have to modify the client application and do not have to run application-specific code on cluster nodes. This enables a quick development and deployment lifecycle. It is also interesting to note that the client application and render servers can run anywhere, including on the same computer, which facilitates development outside the cluster platform.

Advanced Input & Interaction

Wall-sized displays are often augmented with a complex interactive environment, made of heterogeneous input modalities ranging from actual input devices (e.g., mouse, 6-DOF devices, tablets), to the output of interactive systems used for input (e.g., motion-tracking system software, multi-touch table tracker, mobile device sensors interpreter). jBricks's cluster extension to ZVTM handles all aspects related to graphics distribution and rendering, but supports little beyond basic input redirection for conventional devices. An input management system is required to handle the multiple input channels and to ease their fusion so as to eventually deliver high-level input events to applications, that make the description of complex interaction techniques easier [12].

We identified three main requirements for such an input management system. The system should be able to handle various kinds of distributed input in a *generic* way to allow easy substitution of input modalities, and should provide generic output to several distributed applications, no matter whether they were specifically developed for this platform or not. The system should be *extensible*, making it easy to support new devices and functionalities with re-usable processing functions or interaction techniques. Finally, the system should be *adaptable*, enabling runtime addition of new devices and changes to the input configuration.

With these objectives in mind, we developed the jBricks Input Server (jBIS), the distributed input and interaction management system of jBricks. jBIS is built on top of the FlowStates toolkit [3], that combines the ICon [9] and SwingStates [2] libraries. ICon's dataflow model can handle multiple devices and describe advanced interactions efficiently [12]. Its visual editor makes it simple to connect them to application input endpoints (Figure 3). SwingStates extends the Java language with state machines and provides a simple yet powerful programming language that simplifies the description of interaction logics on the application side. FlowStates integrates these two models seamlessly: state machines are instantiated as dataflow processing devices that can be graphically connected to input devices or to other state machines in the dataflow configuration.

Input handling. Thanks to the ICon library, the jBricks Input Server has built-in support for various regular and advanced input devices: mouse, keyboard, various tablets, Nintendo Wii remotes, VICON motion-trackers, interactive pens, etc. These input devices are instantiated as dataflow processing devices that can be connected to adapters or application devices through the dataflow editor (see the mouse device in Figure 3). These dataflow components are high-level structured representations of input devices (or classes of input devices) with typed output slots mapped to the various channels of the input device they handle.

We extended ICon to support generic devices through various protocols with specific dataflow devices that can receive and send OSC, Ivy or TUIO messages. This approach provides an implicit way of performing automatic device registration thanks to the addressing mechanism of these protocols: each input source that sends a message addressed to a specific receiving device in a running configuration is implicitly considered. For instance, a jBIS' OSC receiver device can listen to messages addressed to `/jBIS/position` with two arguments, x and y. This device will then externalize the corresponding output slots. These will be updated each time that a new `/jBIS/position` message is received, wherever it comes from: a smartphone running an application that sends OSC messages from touchscreen events, the tracking software of an interactive table, mouse movements from a laptop running another instance of the jBIS, etc.

Interaction configuration. Input configuration and the lower-level description of interaction techniques (typically the connection to inputs) get specified in jBIS with an ICon dataflow configuration. ICon provides an extensive library of adapter devices, e.g., math or logic operators, control structures, flow control. These can be used to manipulate and transform the raw values of input channels into higher-level data structures (e.g., the mult device in Figure 3). The jBIS built-in library also extends the basic processing devices of ICon with platform-specific ones, adapted to interactive wall-sized displays: for instance, the *pointed tile* dataflow component returns the display tile that is intersected by a 3D vector received as input (typically modeling the user's arm). More than simple low-level processing components, these higher-level devices are close to the re-usable interaction techniques of [12], offering several levels of granularity to the user when building an input configuration.

The jBricks Input Server also includes a plug-in mechanism for the creation of custom dataflow devices with FlowStates [3]: state machines are instantiated as dataflow components, and their transitions are triggered by the connected inputs (pointing and pan-zoom in Figure 3). Programmers can use this descriptive and straightforward approach to extend the jBIS library and to describe some parts of the interaction logic of an application, or even more generic libraries that can be used with multiple applications running on the platform.

Link with application/visualization software. In jBricks, the higher-level interaction logic (manipulation of objects, graphical feedback) is encoded in the client application (Figure 2) developed with ZVTM. The link between the jBIS and this application can be established in two ways. The first solution consists in using specific dataflow devices in the input configuration to deliver high-level interaction events to the application through a networking protocol such as OSC; the client application interprets these messages and reacts accordingly. The other solution consists in using the plug-in mechanism of jBIS to implement application-specific devices that will be instantiated as endpoints of the dataflow. These plugins can define their own protocol to communicate with the client application, or even encapsulate it, enabling direct communication as the client node is running in the same process (same Java Virtual Machine) than jBIS.

Finally, jBIS can be controlled remotely, so that applications can trigger commands (start/stop/change the input configuration) or dynamically install a plugin. Several jBIS instances can run simultaneously, communicating through networking dataflow devices (Figure 2). This modularization, based on the description of partial input configurations, reinforces the flexibility and adaptability of the platform as partial configurations can easily be substituted.

The architecture of jBricks and the resulting development and configuration tools make it possible to develop applications outside the platform, i.e., on a simple laptop, and then deploy and run them on an actual cluster-driven wall display. On the graphics side, changes to the client application are minimal (four lines of code) and can easily be managed using, e.g., command line options or Maven profiles. On the interaction side, the jBricks Input Server makes it easy to dynamically reconfigure and adjust inputs according to available devices and modalities. In the following section, we illustrate these principles with a short scenario showing how jBricks can be used for the prototyping and implementation of interaction techniques for a controlled experiment on a wall-sized display.

jBricks IN ACTION

Abelard and Eloïse need to prepare an experiment to compare one-handed mid-air interaction techniques for selection of very small targets on wall-sized displays. They consider two techniques: a very precise bi-modal pointing technique, and a cursor-centered pan & zoom technique.

They first describe the two techniques with state machines (Figure 3) and plan to implement and configure them as follows. The pointing technique will be operated with a gyroscopic mouse and will feature a coarse mode – i.e., ray-casting – and a precise mode – i.e., relative pointing with a low CD gain. Precise mode will be triggered using the right mouse button; target selection using the left button (Figure 3-a). The pan & zoom technique is operated with an iPod Touch. Vertical thumb movements control the zoom factor, ray-casting of the user's arm controls the cursor's position. Two small areas at the bottom of the iPod's screen trigger panning and target selection, respectively (Figure 3-b).

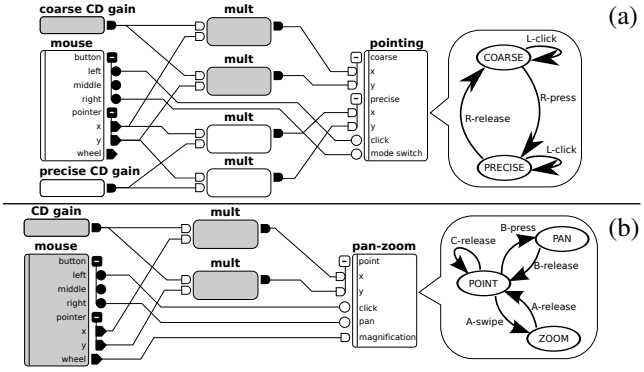

Figure 3. jBIS configurations of the pointing (a) and pan & zoom (b) techniques and their corresponding state machines. A mouse is used to control the techniques and simulate unavailable devices.

Prototyping

As jBricks' graphics and input modules are loosely-coupled, Abelard can work on the experiment's graphics while Eloïse implements and configures the two interaction techniques.

Abelard is working on the graphics part of the experiment. Using ZVTM, he creates an application that displays the targets, cursor appearance and textual instructions on his personal computer without having to worry about the specifics of the cluster-based wall display environment. He just needs to consider the actual dimensions of his graphical scene (in this case, a 20000 7000 pixel area). To make the entire scene visible on his screen, he sets the zoom factor higher than it will eventually be in the real experiment (a straightforward operation in a zoomable user interface).

Meanwhile, Eloïse implements each technique as a FlowStates state machine and encapsulates them in a jBricks Input Server plugin, making them available as dataflow processing devices. During this early prototyping stage, Eloïse focuses on developing the interaction logic, using a basic version of the graphics interface provided by Abelard. She does not need to work on the actual hardware platform either. She runs jBIS on her laptop and uses a regular mouse to simulate the actual input devices that will be used eventually (motion-capture system, gyroscopic mouse, iPod Touch). In this testing configuration, ray-casting with the motion-capture system and gyroscopic mouse are replaced by mouse coordinates; the mouse wheel and buttons are used in lieu of touch events. The output ports of the mouse device are connected to the technique devices, pan-zoom and pointing (Figure 3), the two modes of the pointing technique being simulated by applying constant multipliers to the mouse coordinates (the mul and CD gain processing devices). Later, these configurations will be slightly modified to handle the actual input devices to be used in the experiment.

Porting to the Wall Display Hardware Platform

On the input side, Eloïse substitutes the devices used for prototyping on her laptop with the platform's actual devices, as shown in Figure 4. The regular mouse can be directly substituted with the gyroscopic mouse, with only a CD gain adjustment (changing the value of precise CD gain, Figure 4-a). jBIS has built-in support for the 10-camera motion tracking sys-

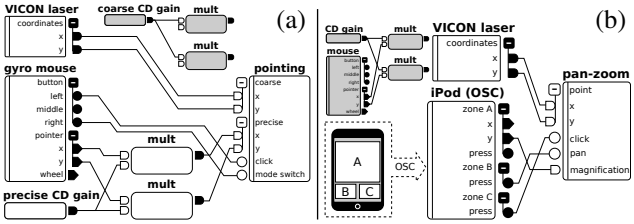

Figure 4. jBIS configurations of the final pointing (a) and pan & zoom (b) techniques. The simulation inputs (in grey) can be reused at any time simply by changing the connections.

tem in the room (the VICON laser device). For the iPod Touch, Eloïse uses a built-in OSC receiver device in her input configuration to receive touch events from a freely-available application running on the handheld (Figure 4-b). To deploy the client application on the actual hardware, Abelard only needs to add a few jBricks instructions describing the *Clustered View*. He then embeds the application into the jBIS plugin made by Eloïse. The client application is launched by jBIS; it has access to the state machines' output and will react according to the chosen interaction technique.

Further iterations, switching back and forth between the simplified configuration running on personal computers and the one for the actual wall display hardware is straightforward. Abelard and Eloïse can also easily add new techniques by implementing new state machines and test several input configurations for each of them.

CONCLUSION

The jBricks framework extends and integrates state-of-the-art structured graphics and input management toolkits to enable the rapid development of post-WIMP applications for cluster-based wall displays equipped with advanced input devices and modalities. Its architecture and features enable easy deployment and reconfiguration, allowing developers to partially implement and debug their applications on conventional hardware such as a single laptop or workstation.

We have successfully used jBricks for the rapid prototyping of novel interaction and visualization techniques, and to run controlled experiments for their evaluation [16]. It is also used for the development of various applications for the visualization of large datasets in other disciplines: astrophysics, social network analysis, geospatial intelligence. The Java-based platform makes it easy to use existing libraries in client applications. In addition, ZVTM features several extension modules that enable, e.g., the layout of large networks using JUNG or GraphViz, the visualization of treemaps, native high-quality PDF rendering using ICEpdf, FITS astronomy image display using JSky, interactive navigation in OpenStreetMap, from world overview down to street level. Future work will focus on improving the Java2D/OpenGL rendering pipeline by optimizing the stream of instructions. The implementation of a higher-level communication protocol, based on HID definitions on top of OSC, will improve dynamic input device registration and configuration. jBricks will be made available under an open-source software license (`http://insitu.lri.fr/JBricks`).

ACKNOWLEDGEMENTS

We wish to thank Caroline Appert and Olivier Chapuis for helpful comments on early drafts of this paper. This work is supported by a Région Île-de- France / Digiteo grant.

REFERENCES

1. C. Andrews, A. Endert, and C. North. Space to think: large high-resolution displays for sensemaking. In *Proc. CHI '10*, 55–64. ACM, 2010.
2. C. Appert and M. Beaudouin-Lafon. SwingStates: Adding state machines to Java and the Swing toolkit. *SP&E*, 38(11):1149 – 1182, 2008.
3. C. Appert, S. Huot, P. Dragicevic, and M. Beaudouin-Lafon. FlowStates: Prototypage d'applications interactives avec des flots de données et des machines à états. In *Proc. IHM '09*, 119–128. ACM, 2009.
4. R. Ball, C. North, and D. Bowman. Move to improve: promoting physical navigation to increase user performance with large displays. In *Proc. CHI '07*, 191–200. ACM, 2007.
5. R. Ballagas, M. Ringel, M. Stone, and J. Borchers. istuff: a physical user interface toolkit for ubiquitous computing environments. In *Proc. CHI '03*, 537–544. ACM, 2003.
6. B. Bederson and J. Meyer. Implementing a zooming user interface: experience building pad++. *SP&E*, 28:1101–1135, August 1998.
7. B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit Design for Interactive Structured Graphics. *IEEE Trans. Software Eng.*, 30(8):535–546, 2004.
8. A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proc. VR '01*, 89. IEEE, 2001.
9. P. Dragicevic and J.-D. Fekete. Support for input adaptability in the icon toolkit. In *Proc. ICMI*, 212–219. ACM, 2004.
10. S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A Scalable Parallel Rendering Framework. *IEEE TVCG*, 15(3):436–452, 2009.
11. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002.
12. S. Huot, C. Dumas, P. Dragicevic, J.-D. Fekete, and G. Hégron. The MaggLite post-WIMP toolkit: draw it, connect it and run it. In *Proc. UIST '04*, 257–266. ACM, 2004.
13. B. Jeong, L. Renambot, R. Jagodic, R. Singh, J. Aguilera, A. Johnson, and J. Leigh. High-performance dynamic graphics streaming for scalable adaptive graphics environment. In *Proc. SuperComputing*. ACM, 2006.
14. W. König, R. Rädle, and H. Reiterer. Interactive design of multimodal user interfaces. *J Multimod. UI*, 3:197–213, 2010.
15. M. Lorenz, G. Brunnett, and M. Heinz. Driving tiled displays with an extended chromium system based on stream cached multicast communication. *Parallel Comput.*, 33(6):438–466, 2007.
16. M. Nancel, J. Wagner, E. Pietriga, O. Chapuis, and W. Mackay. Mid-air pan-and-zoom on wall-sized displays. In *Proc. CHI '11*. ACM, 2011. In press.
17. T. Ni, G. S. Schmidt, O. G. Staadt, M. A. Livingston, R. Ball, and R. May. A Survey of Large High-Resolution Display Technologies, Techniques, and Applications. In *Proc. VR '06*, 223–236. IEEE, 2006.
18. E. Pietriga. A Toolkit for Addressing HCI Issues in Visual Language Environments. In *Proc. VL/HCC'05*, 145–152. IEEE, 2005.