A Cost-effective Approach for Developing Application-control GUIs for Virtual Environments

Carlos Andujar* Marta Fairén† Ferran Argelaguet‡

Modeling, Visualization, Interaction and Virtual Reality Group
Universitat Politècnica de Catalunya

ABSTRACT

In this paper we present a new approach for fast development of application-control User Interfaces (UIs) for Virtual Environments (VEs). Our approach allows developers to build sophisticated UIs containing both simple widgets (such as windows, buttons, menus and sliders) and advanced widgets (such as hierarchical views and web browsers) with minimum effort. Rather than providing a new API for defining and managing the interface components, we propose to extend current 2D toolkits such as Qt so that its full range of widgets can be displayed and manipulated either as 2D shapes on the desktop or as textured 3D objects within the virtual world. This approach allows 3D UI developers to take advantage of the increasing number of components, layout managers and graphical design tools provided by 2D UI toolkits. Resulting programs can run on platforms ranging from fully immersive systems to generic desktop workstations with little or no modification. The design of the system and the key features required on the host UI toolkit are presented and discussed. A prototype system has been implemented above Qt and evaluated on a 4-sided CAVE. The results indicate that this approach provides an efficient and cost-effective way for porting and developing application-control GUIs on VEs and thus it can greatly enhance the possibilities of many VE applications.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimension Graphics and Realism—Virtual Reality; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques

Keywords: virtual environment, interaction techniques, graphical user interfaces, 3D window manager

1 INTRODUCTION

Users of modern computer applications have become intimately familiar with a specific set of UI components, including input devices such as the mouse and keyboard, output devices such as the monitor, interaction techniques such as drag-and-drop, interface widgets such as pop-up menus and interface metaphors such as the desktop metaphor [5]. Some of these interface components are appropriate for virtual environment applications but require some kind of adaptation. For example, interface widgets must be accommodated so that they can be perceived correctly within a 3D world displayed on a stereoscopic device. On the other hand, specific characteristics of VEs make some interface components completely inappropriate. For example, a CAVE user may be moving constantly inside the CAVE, making the use of mice and physical keyboards impractical.

The specific characteristics of VEs has led to substantial research on designing, implementing and evaluating new user interface components and on adapting or extending existing components to 3D applications.

User-interaction tasks on a VE are often divided into four categories: navigation, selection, manipulation and application control [2]. In this paper we will focus on application control tasks. Application control refers to the user task of issuing commands to request the system to accomplish a particular function or change its internal state [2]. In 2D UI, application control is often performed using a graphical user interface (GUI) following a WIMP (Windows, Icons, Menus and Pointers) metaphor. For immersive VE applications, a number of interaction techniques have been proposed ranging from voice recognition to gesture-based interaction through specialized physical devices. However, graphical menus and visual tools are still the prevalent technique for application control in VE.

Despite the broad diversity of system control techniques for 3D UIs, only a limited amount of attention has been paid to the development of standard APIs and re-usable software components allowing for 3D UI development in a cost-effective manner.

Several approaches for incorporating GUIs in virtual environments have been proposed. Two different approaches can be observed:

- GUI outside the virtual world: a 2D GUI is displayed and operated in a separate device.
- GUI inside the virtual world: a GUI is displayed in 3D space as a virtual object.

Some VE applications follow the GUI-outside approach. These applications have their user interface split into two parts. Some of the functionality is accessible through a 3D user interface (usually navigation and manipulation tasks) and the rest can only be controlled through a conventional GUI running on a console outside the VE. This approach simplifies UI development but presents obvious usability problems. Solutions using handheld computers [38] and specialized hardware follow a similar approach.

On the other hand, placing GUI components inside the virtual world offers much more possibilities and interaction styles, and thus this is often the most suitable option. Developers willing to put a 3D GUI inside a VE application are faced with two options. They could implement the full functionality of the 3D GUI inside the application (a non cost-effective approach) or they could adopt any of the few available APIs for 3D GUI creation [21, 29, 15]. Specific APIs for handling 3D interaction in VE are undoubtly a reasonable solution but in their current state of development they present some limitations:

- Specific APIs for 3D widget creation force developers to learn a new API. The lack of standard APIs and mature, well-established implementations make this problem worse. One of the few standardization efforts is UsiXML (USer Interface eXtensible Markup Language) [24], a language for describing interactive applications with different types of interaction techniques. Although some promising tools exist for automatic generation of VRML97 or Extensible 3D (X3D) GUIs

*e-mail: andujar@lsi.upc.edu
†e-mail: mfairen@lsi.upc.edu
‡e-mail:fargelag@iri.upc.edu

IEEE Symposium on 3D User Interfaces 2006
March 25 - 26, Alexandria, Virginia, USA
1-4244-0225-5/06/$20.00 ©2006 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

from UsiXML specifications [25, 26], these tools are still in an early stage of development.

- Most 3D widget libraries implement the functionality of each user interface component (widget), including drawing and handling. As a consequence, readily available 3D toolkits offer a limited set of widgets and make it difficult to plug-in third-party components (e.g. media players and web browsers).

- A commonly accepted opinion is that GUI interfaces should be optimized for each platform. For example, a desktop interface using 2D interaction for system control is certainly appropriate for a desktop system whereas an immersive, stereoscopic-compatible interface should be used in an immersive system. Most widget libraries for VEs target only the second kind of platforms and thus two or more different versions of the applications are often required.

These problems suggest that traditional 3D UI tools are not always appropriate for fast development of complex UIs and particularly for fast migration of existing desktop-based 3D applications to immersive VEs. For example, an existing desktop-based 3D application can be modified to display its contents on a stereoscopic workbench with minor effort, but the migration of its GUI can be a much more difficult task.

Some authors have proposed the immersion of 2D applications into 3D worlds to make them available to VE and Augmented Reality (AR) applications. The basic idea is to project 2D image content onto texture-mapped planes on the 3D world, and to let the users interact with them through VR input devices such as gloves and tracking systems. Current systems for launching and/or sharing existing 2D applications into VE and 3D window managers are good examples of this technique. Although application-sharing systems and 3D workspaces can be used to access 2D GUIs within VEs, they suffer from some performance and flexibility limitations mainly because they are built upon framebuffer-oriented protocols such as VNC [30].

In this paper we explore a new approach for fast development of 3D GUIs, dubbed *get3d* (GUI extension toolkit). Rather than providing a new API for defining and managing application-control widgets, we propose to add new features to current extensible toolkits such as Qt so that its full range of 2D widgets (and even user-defined *true* 3D widgets) can be displayed and manipulated either as 2D shapes on the desktop or as textured 3D objects within the virtual world. We call *host toolkit* the 2D extensible toolkit whose widgets are accommodated to VE applications.

The basic idea is to extend the host toolkit so that widgets are not rendered directly to the framebuffer but to a virtual buffer which is then converted into a texture and drawn using an OpenGL texture-mapped rectangle. User actions captured by generic input devices such as wands and tracking devices are mapped into low-level mouse and keyboard events that are then sent to the host toolkit for processing.

Our approach allows 3D UI developers to take advantage of the increasing number of widgets, layout managers and graphical design tools provided by 2D UI toolkits. Resulting programs can run on platforms ranging from fully immersive systems such as CAVEs to generic desktop workstations with minor or no modifications.

The main contributions of this paper are:

- A new approach for fast, almost transparent accommodation of 2D GUIs to VEs.

- An analysis of the main requirements of a conventional 2D GUI toolkit to be extended to VEs.

- A discussion of the design of a Qt-based prototype implementation.

- An evaluation of the performance and usability of the system under different interaction techniques.

The rest of the paper is organized as follows. Section 2 reviews related work on interaction techniques for system control and previous approaches for immersing 2D applications into 3D worlds. Section 3 describes the architecture of *get3d*. Some implementation issues concerning our Qt-based implementation are described in Section 4. We present performance and usability results on Section 5, and provide concluding remarks in Section 6.

2 Related work

2.1 Interaction techniques for system control

Beside graphical menus, a number of interaction techniques have been proposed for accomplishing application control tasks within immersive VE applications. These interaction techniques range from voice-recognition [27] to gesture-based interaction [18] through specialized physical devices such as Tangible User Interfaces [17]. However, graphical menus and visual tools are still the prevalent technique for application control in VE.

The simplest and most popular menus are 2D widgets adapted to 3D space. These widgets are simple adaptations of their 2D counterparts and basically work in the same way as they do on the desktop. Ring menus [23] rely on the fact that menu selection is essentially a 1-DOF operation. These menus have their items arranged into a circular object; users can select the desired item by using their hand. When the number of menu items is reduced, VR gloves can be used to perform menu selection by attaching each menu item to a different finger [3]. Typically, graphical menus are operated through 2-DOF selection methods (such as ray-casting) for reducing the DOFs of the interaction and thus increasing user efficiency and minimizing errors.

2.2 Software tools for 3D UI authoring

Software tools for 3D UI authoring can be broadly subdivided into three categories: general frameworks for developing VE applications, specific modules for 3D UI development, and automatic generation tools from specifications.

A number of frameworks and APIs for developing device-independent VE applications have been proposed. Some well known examples are DIVERSE [19], VRCO's CAVElib[41] and Iowa State's VR Juggler [1]. These tools put the emphasis on abstracting common programming tasks on VE such as window creation, viewer-centered perspective calculations, stereoscopic viewing, displaying to multiple channels, cluster synchronization and accessing VR hardware. However, only a few provide specific modules for 3D UI creation. VR-Juggler's Tweek [15, 16] provide VE users with an extensible Java GUI that communicates with VR applications through a combination of technologies. All communication between the Java GUI and the C++ VR application is managed by CORBA, the Common Object Request Broker Architecture. The VEWL (Virtual Environment Windowing Library) [21] is an API providing a window manager that supports the use of menus, windows, buttons and other widgets within an immersive virtual environment. VEWL widgets use class names similar to Qt, although the actual rendering is done using OpenGL. VEWL provides device-independent input through DIVERSE [20]. The it3d (Interactive Toolkit Library for 3D Applications) [29] provides an input/output library for distributed devices, a 3D widget library for multimodal interaction and a gesture-recognition library. Despite these valuable tools, there is a lack of standard APIs and well-established tools for 3D UI development.

Proceedings of the 2006 IEEE Symposium on 3D User Interfaces (3DUI'06)
1-4244-0225-5/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

2.3 Immersing 2D applications into 3D worlds

A related and very active research topic is the immersion of 2D applications into 3D space. We distinguish two basic categories of tools: 3D window managers for desktop computers and tools for accessing remote applications from within VE and AR applications.

Tools in the first category aim at providing 3D workspaces designed for replacing 2D desktops [42, 32]. These tools rely on mouse and keyboard interaction and are intended for non-immersive display devices. A first example is MaW [22], a prototypal 3D Window Manager. MaW allows the application to create windows that are drawn in 3D space using OpenGL primitives. SphereXP [42] is a 3D desktop system designed to be a replacement for Microsoft's Windows XP. Task Gallery [32] is a 3D prototype user interface that expands the desktop into an entire office with an unlimited number of desktops. The screen becomes a long gallery with paintings on the walls that represent different tasks.

Tools on the second category aim at providing access to remote applications from within VE and AR applications. Two different approaches can be observed [6]. *Hardware oriented* approaches provide access to external applications through additional display devices such as PDAs and see-through-displays [38]. *Software oriented* approaches project image content onto planes in 3D. This can be further distinguished by the system component performing the 2D rendering (such as typesetting and linedrawing). The VE software can manage directly the actual 2D drawing (using geometric and text primitives) or it can access 2D display content generated by external applications and display them as texture-mapped rectangles.

Early work using application sharing in 3D environments is described by Dykstra [12], where texture-mapped rectangles are used to operate X applications on 3D virtual spaces. A similar approach for immersing X window applications into a 3D scene is described in [37]. This idea has been adopted and extended both in VE and AR applications.

VNC (Virtual Network Computing) [30] is a remote display system which allows viewing a computing desktop running elsewhere on a network. VNC provides a distribution mechanism for desktops on the lowest level by transmitting frame buffer contents to the remote client and receiving keyboard and pointing device events, inserting these into the server-side input queue. Each time a client interacts with the shared application, the VNC server broadcasts the image of the areas affected by updates on the remote interface. VNC is the foundation of a number of systems providing immersion of 2D applications into 3D space. VREng [8] is a distributed 3D application allowing navigation in virtual worlds connected over the Internet. VNC is used to immerse 2D interfaces in the 3D world. Soares *et al* [35] propose a VNC-based platform for increasing the sense of collaboration among co-workers sharing any standard single-user application. Sensing Surfaces [6] also follows the approach of mapping the contents of a GUI desktop to arbitrary textured geometry in the VE using VNC. ARWin [10] is an augmented reality desktop environment that allows users to operate X applications such as clocks within an augmented physical desktop. A custom VNC client connects to a virtual VNC-enabled X server and draws the window data to an OpenGL texture map which is applied to a polygon in the workspace. The Three-Dimensional Workspace Manager (3DWM) [13] is a software platform targeted at both research and development of 3D GUIs. The toolkit contains a basic set of 3D widgets such as buttons, text fields and sliders, and it provides access to external applications through VNC. All these approaches suffer from some performance and flexibility limitations mainly because they are build upon a protocol providing little control to the VE application over the properties and behavior of GUI components.

3 System description

3.1 Overview

The basic components of *get3d* are depicted in Figure 1. The *host toolkit* is a 2D GUI extensible toolkit (e.g. Qt) whose widgets are accommodated to *VE applications* by an *extension toolkit*. A key feature of our approach is that the additional features are provided by subclassing the 2D toolkit. One of the consequences is that existing applications with 2D GUIs can be adapted to a VE environment with minimum effort. All the application's source code for GUI creation and behaviour does not have to be modified.

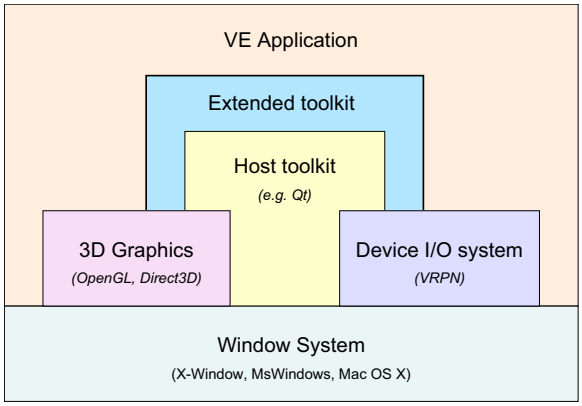

Figure 1: *get3d* overview

We now introduce some notation that will be used through-out the paper. The word *widget* is used to refer to user interface objects such as windows, buttons and sliders that are used as the basic constituents of GUIs. A widget receives mouse, keyboard and other events from the environment, and paints a representation of itself on the output device. Widgets are arranged into a hierarchical structure. A widget that is not embedded in a parent widget is called a *top-level widget*. Usually, top-level widgets are windows with decoration (a frame and a title bar). Non-top-level widgets are *child widgets*. We also distinguish between widget objects provided by the host toolkit (called *native widgets*) and the objects that represent the widget as a texture-mapped rectangle in 3D space (called *virtual widgets*).

Events of current 2D GUI toolkits can be roughly classified into two categories. *Application events* refer to GUI-related, high-level events such as show, hide, close, resize and paint events. Events produced by simple user actions on input devices such as tracking systems and wands will be referred to as *user events*. According to the originating source, we also distinguish between *native events* originating from the window system or the host toolkit, and *synthetic events* initiated by the extension classes. For example, when the user presses a button on a 3D wand with the aim of selecting a menu item, a user event will be created and then translated into a native, synthetic event (a mouse press event in this case) that will be send to the native widget representing the menu item.

3.2 System components

The core of the extension toolkit consists of the components depicted in Figure 2. The main responsibilities of each component are described below.

Application 3D provides a unified interface to the VE application, delegating client requests to appropriate subcomponents. This is the only component the application has to collaborate with, thus

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

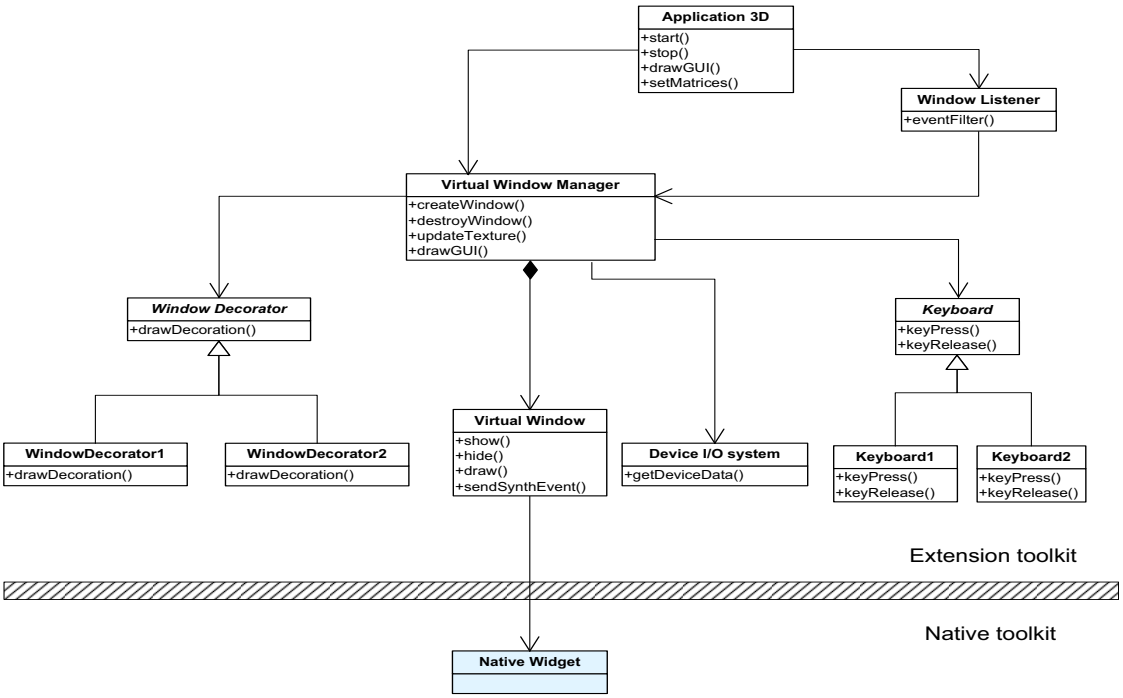

Figure 2: Conceptual design of the extension toolkit using UML notation. Only the most relevant components and operations are shown.

making the toolkit easier to use. This component monitorizes the creation of new native widgets by the application and asks the Virtual Window Manager (see below) to create a new virtual window everytime the application instantiates a new top-level widget. Application 3D manages the drawing of the GUI windows in 3D space as texture-mapped objects (forwarding the request to other components) and configuration options.

The *Virtual Window Manager* manages the behavior of virtual windows associated with top-level native widgets. This component handles basic window operations such as creation, destruction, and show, hide and resize operations. This component also manages user events. This includes getting VR device data through the Device I/O System, managing interaction with the virtual window decoration through a decorator subclass, and translating user events into synthetic events that will be sent to the appropriate virtual window.

The *Virtual Window* keeps attributes concerning the 3D version of a top-level native widget such as rectangle size, texture size and transformation matrix. This component checks for intersection between the virtual window's plane and a selection primitive (e.g. a ray). In response to user events, Virtual Window objects send synthetic mouse events (to the child widget at a given 2D position) and keyboard events (to the child widget having the keyboard focus).

The *Event Listener* acts basically as an application-event filter. This component monitorizes the application events related with a widget's life cycle: show, hide, close and resize events originated from within the application. It also monitorizes changes on the image content of any visible widget and asks the virtual window manager to update the texture rectangle of the virtual window containing the native widget.

The rest of the components have simple responsibilities. The *Virtual Keyboard* defines an interface for a virtual keyboard that sends synthetic keyboard events to the application in response to user actions. The *Window Decorator* defines an interface for drawing the decoration of a virtual window. Decoration includes the frame, and buttons to iconify, deiconify and close the virtual window. Finally, the *Device I/O system* is used to read data from an extensible set of generic input devices.

3.3 Widget creation and placement

Each time a native window is created the Event Listener receives a notification. If the widget is top-level, a virtual window is created along with a texture map capturing the contents of the widget's area. A rectangular portion of this texture will be updated everytime a child widget changes its appearance.

An important issue is the initial window placement inside the 3D world. The placement strongly influences the user's ability and accuracy. A situation which arises in the use of multiple windows is the need to constantly arrange windows to get access to the particular window which houses the task or information needed at a given instant (*window thrashing*) [22]. According to the spatial reference, we can consider windows that are world-referenced, object-referenced, head-referenced and device-referenced [5, 14]. Head-referenced menus provide an appropriate spatial reference frame as they take the most of the user's proprioceptive sense, allowing users to accomplish some application control tasks without having to look at the menu. Since the extension toolkit knows everything about the widget hierarchy, any of these strategies can be plugged in. In the CAVE usability tests described in Section 5 we used device-referenced windows, i.e. windows remain in a fixed position (initially facing the user) with respect to the CAVE, unless the user moves them.

3.4 Input handling

User events such as hand movements are captured by the Device I/O Module and converted into synthetic events that are sent to native widgets. We will describe the main steps involved in this process with a concrete example. Suppose a CAVE user wearing a virtual wand (the wand has a six-DOF sensor, a two-DOF joystick and

Proceedings of the 2006 IEEE Symposium on 3D User Interfaces (3DUI'06)
1-4244-0225-5/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

three buttons). Each time the user presses the left button a new user event is generated. This event contains a ray and a button state as parameters. This event is forwarded to the Virtual Window Manager which searches for the nearest virtual window containing the ray intersection. Finally, the intersected virtual window creates a synthetic event (a mouse event) and posts it to the native widget.

The main advantage of this solution is that the management of the GUI elements and their behavior is delegated completely to the host toolkit, thus simplifying the migration of existing GUIs. Moreover, this approach allows the system to provide several interaction techniques for object selection such as ray-casting and arm-extension.

3.5 Collaboration

An additional benefit of *get3d* is that it can be easily extended to support remote collaboration. The collaborative architecture presented in [36] can be used to allow remote users to share the same virtual environment and interface.

Regarding the interface sharing, users can decide if they want to make their virtual windows visible and/or accessible to other remote users connected to the same session. Each time a GUI-related event is initiated on a virtual window visible to other remote users, a synthetic event will be generated and broadcasted to the remote users connected to the virtual session. If the windows are also usable by remote users, the broker [36] will be the responsible of maintaining a list of blocked windows, so when a user operates on a certain window that is not already blocked by another user, the broker will block it and post the event asked by the user. The window will be unblocked after all the event's effects end. In case the window is blocked by another user, the action is not executed and the originating user is notified about this.

3.6 Toolkit requirements

We now summarize the main features required for an extensible 2D GUI toolkit to be used as a host toolkit for the 3D extension:

- A hook to monitorize the creation and destruction of top-level widgets. In addition to this, the toolkit has to provide a mechanism to intercept widget-related events such as paint, show, hide, close and resize. For example, if a widget is repainted in response to a paint event, the application has to intercept the event to know that the associated texture requires an update. This feature is required by the Event Listener component.

- An operation to send a synthetic event (keyboard/mouse) to any widget (required by the Virtual Window component).

- A function to capture the image of a native widget into a bitmap. The image will be used to update the texture of the virtual window containing the widget (required by the Event Listener). If this function is supported also on widgets hidden by other windows, the native GUI can share the desktop space with the OpenGL window where the VE application renders the virtual world.

- Functions to traverse the widget hierarchy (access to parent and children widgets) and operations to obtain the visible child widget at a given pixel position.

4 Implementation

4.1 Qt-based implementation

We have implemented a prototype version of *get3d* over Qt [40], a cross-platform GUI development toolkit (see Figure 3). The source code can be downloaded from `http://www.lsi.upc.edu/~virtual/Qt3D`.

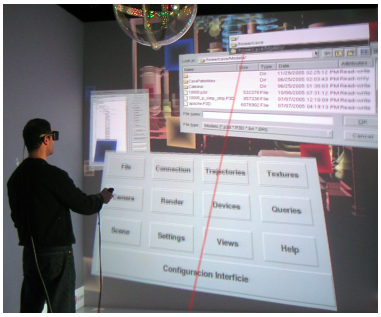

Figure 3: User interacting with the test application

Qt fulfills all the requirements listed in the previous section. Creation of top-level widget can be monitorized by overriding QApplication's virtual function *polish()*. This function is called by every Qt widget before it is first shown. Some 2D GUI toolkits provide an equivalent function for doing style-based central customization of widgets. Application events can be intercepted through QObject's *installEventFilter()*. The filter can either stop the event or forward it to other objects. Moreover, Qt allows multiple event filters to be installed on a single object, thus avoiding conflicts with application-defined event filters. Synthetic events (including predefined and user-defined events) can be sent to any object through QApplication's *postEvent()* and *sendEvent()* methods. The former adds a synthetic event to the event queue with a given object as the receiver of the event; the later sends the event directly to the object. The QPixmap class provides two methods to grab the contents of a widget: *grabWindow()* grabs pixels directly from the screen, whereas *grabWidget()* asks the widget to paint itself by calling *paintEvent()* with output redirected to a bitmap. Although a bit slower, the later is more suitable because it works with hidden widgets. Finally, the toolkit provides operations for traversing the object hierarchy and for returning the visible child widget at a given pixel position in the widget's own coordinate system.

A concrete example of collaboration is shown in Figure 4. When a widget detects that it should repaint itself, it sends an event to the QApplication indicating which part of the widget should be repainted (1). When appropriate, QApplication tells the widget it has to be painted (2). This event is captured by the event filter (3) and produces a grab widget call to the QPixmap object (4), which sends a repaint command to the QWidget (5). Finally, the window listener asks the virtual window manager to update the texture (6).

4.2 Prototype details

Our current implementation only supports ray-casting selection. Input devices are handled either through a VRPN (Virtual Reality Peripheral Network) client [39] or through custom events that must be sent by the VE application to the extension toolkit. We use a magnetic tracking system and a virtual wand. The associations for wand buttons is similar to [21].

The prototype has been tested with two different applications: a scene viewer specialized for shipbuilding design and a volume rendering application. Both applications had a Qt-based GUI and were extended to display stereoscopic images on either a CAVE or a stereo workbench.

The source code modifications required to accommodate the GUI to the CAVE were minimum. The main changes are summarized in the code below. Including minor changes, less that 100 lines of source code were modified.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

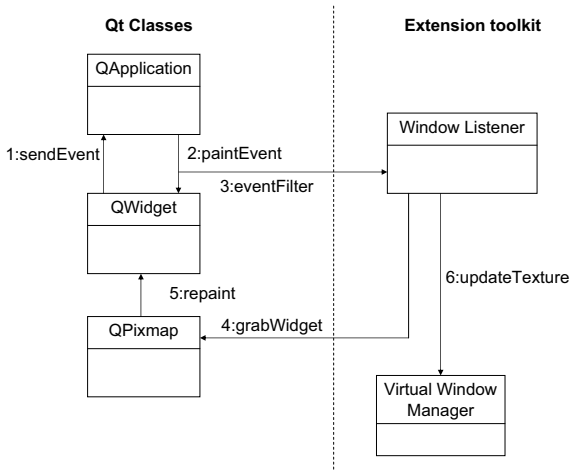

Figure 4: Collaboration for processing a paint event.

```
void main()
{
  // The VE application simply creates an instance of
  // QApplication3D instead of usual QApplication.
  QApplication3D qApp3d(argc,argv);

  qApp3d.setGLWidget(w); // OpenGL window
  qApp3d.start(); // activates the extension toolkit
  ...
}

void paintGL()
{
  // setup modelview and projection matrices
  qApp3d.drawGUI();
  ...
}
```

5 RESULTS

5.1 Usability Evaluation

We conducted an informal usability evaluation to measure the performance and effectiveness of the system. Seventeen external and internal users in the age range of 24 to 58 participated in the study. All participants had desktop PC experience; only two subjects had hands-on experience with the CAVE. Before the evaluation began we gave a demonstration of how the wand works and described the functionality of each button. Subjects used the interface to open, move and close windows for about 2-3 minutes to allow them to get used to the CAVE and overcome the initial learning curve associated with using the wand. After this exploratory phase, subjects were ready to begin a more focused task-based evaluation. It is worth mentioning the GUI design of the VE test application has been maintained from the original application; the only change has been the replacement of the menu bar by a floating window menu. Of course, better results could have been obtained by adapting the GUI to the target system (e.g. using larger buttons) but the comparison with the 2D version of the system would have been more difficult. Therefore, we decided to preserve the original GUI and thus evaluate the effectiveness of our approach for fast migration of existing applications to the CAVE.

5.1.1 Evaluation test

The evaluation exercise each user performed consisted on three phases, repeating on each phase the same group of tasks over different platforms with different input devices. In the first phase subjects used a *desktop PC with a classic mouse* as input device. In the second phase subjects used a *notebook PC with the touchpad* as input device. In the third phase subjects used a *4-wall CAVE system with a wand* as input device. We designed this evaluation test in order to have comparable results (in terms of time to complete each task) that allow us to fairly evaluate the operative effectiveness of the interaction.

5.1.2 Tasks

The first task the subject had to perform consisted in opening a new model. The task involved interaction with several menus and a file browser dialog. The second task consisted in adjusting some rendering parameters such as the background color. The task involved using checkboxes, sliders and buttons on several windows. Users were instructed to get slider values as close to the target value as possible (we accepted a 4% error). The third task consisted in changing the navigation mode, which involved interaction with buttons and combo boxes on several windows.

On the CAVE, the tasks were performed by each user four times, one for each of the combinations of the following variations:

- Using a straight line for the ray vs. a curved ray snapped to the nearest item (adapted from [9]).
- Positioning the virtual windows several meters away from the user vs. only 1.5 meters away, preserving in both cases the projected area over the screen.

5.1.3 Experimental results

The results of these tests are reflected in Tables 1 and 2.

		PC-mouse	PC-touchpad	CAVE-wand
Task 1:	$\bar{x}$	9.68	16.29	12.81
	σ	2.26	4.14	2.96
Task 2:	$\bar{x}$	16.12	27.20	21.24
	σ	3.72	6.59	4.96
Task 3:	$\bar{x}$	6.34	10.99	8.50
	σ	1.49	2.91	2.12

Table 1: Times (seconds) users spent to perform the different tasks on the different platforms.

Table 1 shows the time spent by the users on each task (mean and standard deviation) in the different platforms. For the CAVE test we show the best of the four different tests (Table 2). Although the best times are achieved over the PC system with a classic mouse, the time spent by the users in the CAVE system with the wand is lower than the time spent in a notebook with a touchpad. This result shows that the proposed 3D interface is at least as much effective as a very common input device.

Table 2 shows the results on the CAVE system. We can see in this table that the use of a curved ray helps the user on his task (times are lower even if the windows are far). The best case for all three tasks is the use of a curved ray with the widgets close to the user. This is because interaction with nearby windows is less sensitive to hand orientation changes.

Proceedings of the 2006 IEEE Symposium on 3D User Interfaces (3DUI'06)
1-4244-0225-5/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

	Near		Far	
Straight line	$\bar{x}$	σ	$\bar{x}$	σ
Task 1	14.49	4.36	15.71	4.43
Task 2	24.15	7.26	26.18	7.17
Task 3	9.66	2.91	10.47	2.76
Curve	$\bar{x}$	σ	$\bar{x}$	σ
Task 1	12.81	2.96	14.22	4.04
Task 2	21.24	4.96	23.70	6.74
Task 3	8.5	2.12	9.48	2.70

Table 2: Times of each task in the CAVE platform by using straight line or curved ray and widgets near or far from the user.

5.1.4 Survey

After using the interface, subjects completed a small survey that asked them to compare the Qt-based interface with similar desktop interfaces that they are familiar with. They rated the ease of using the menus, closing a window, selecting a button, moving a window and pointing the mouse. Questions used a 7-point Likert scale, where 1 mean "near impossible" and 7 mean "as easy as a desktop computer". We also asked subjects about what they found most difficult and what was the easiest for them.

The results were very similar from all users. All of them agreed on having no problems on selecting buttons, moving or closing windows. When asked, most of the subjects said that these tasks were what they found to be the easiest and this is mirrored in their average usability rating of 6.4. The most difficult task was to achieve a certain value on the sliders, because the free movement of the wand on their hand makes difficult to maintain the value when the finger is moved to release the button. This task got an average usability rating of 2.5 from the subjects. Because of the same problem, the average rating of using the popup menus was 4.9, which suggests that, while they were not as easy to use as their desktop counterparts, they were still quite usable.

5.2 Performance

We have conducted a simple experiment to evaluate the overhead involved in the immersion of the 3D GUI. This overhead is due basically to the event filter, the widget grabbing, the texture update and the rendering of the texture-mapped rectangle. We run the tests on Pentium IV PCs at 2.8 GHz with 1GB of RAM equipped with 128 MB Nvidia GeForce FX 5200 cards. We compared the frames-per-second at several stages of the interaction with the extension toolkit enabled and disabled. In both cases the VE application was rendering a 50.000 polygons scene. On the CAVE we got a constant 60 fps, due to the framelock and vertical sync required by active stereo glasses. On a passive stereo workbench we disabled the vertical syncronization. The application run at 175 fps without 3D GUI. With 3D GUI enabled, the average performance was 150 fps, increased to 175 fps when virtual windows were shown minimized. The maximum overhead was produced when several windows covered the whole viewport, achieving 128 fps. These results show that overhead of the system is small enough to provide smooth interaction and feedback.

5.3 Discussion

Our approach has several advantages over previously-reported systems. Applications adopting this approach can provide an interface optimized for each platform (e.g. a desktop interface can be used on a desktop system, and an immersive interface can be used in an immersive system, without needing to modify the application.

Moreover, importing 2D interfaces into the VE has some considerable advantages:

- Transparent use of all the functionality provided by the host toolkit, enabling the use of a large number of widgets.
- Delegation of functional interface handling to an independent component.
- Users can work with familiar user interfaces.
- Accommodation of different 3D selection techniques (such as ray-casting).

From the point of view of software development, some of the advantages of our approach are:

- An important part of the UI can be developed and tested in a desktop workstation.
- Access to UI graphical design tools (e.g. QtDesigner).
- Fast porting of existing applications to VEs with minimum changes.
- Transparent support to cluster-based systems and collaboration.

Our approach has several advantages over VNC-based methods for immersing 2D GUIs. *get3d* is not framebuffer-oriented but widget-oriented. That implies that the VE application knows everything about the immersed GUI, not only framebuffer updates. That allows for much more flexibility for placing and sizing the widgets: host widgets can be automatically resized so that width and height are appropriate for fast texture conversion (power of two), widgets can be independently resized and moved from within the VE application, pop-up menus and pop-up lists can be true pop-up components, i.e. they can be drawn at a certain offset from the parent widget; the look-and-feel is completely customizable from within the VE application. Moreover, *get3d* does not have any network nor image encoding overhead. For example, when running on a cluster, texture updates are not broadcasted to all clients because the native widgets and the virtual windows are local to each process.

6 Concluding remarks and future work

The development of 3D UIs for VE applications has many challenges, including the lack of standardization, the multiple hardware platforms, and the lack of reusable 3D user interface toolkits. In this paper a simple-to-implement system to accommodate GUIs to VE systems have been presented and evaluated. Our approach provides a fast and cost-effective way of developing new GUIs and porting existing GUIs to VE systems. Of course *get3d* is not aimed at replacing true 3D widgets, but rather to simplify the GUI development. The results of our work done so far indicate that the seamless integration of conventional GUIs with VE can greatly enhance the possibilities of many VE applications. We plan to use *get3d* to migrate existing 3D applications to the CAVE. We plan also to implement automatic accommodation to different screen resolutions by modifying the widget's layout and size (specially text font size).

7 Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Technology under grant TIN2004-08065-C02-01.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.

REFERENCES

[1] A. Bierbaum, C. Just, C. Hartling, K. Meinert, A. Baker, and Carolina Cruz-Neira. VR Juggler: A virtual platform for virtual reality application development. In *IEEE VR'2001*, Yokohama, Japan, March 2001.

[2] D. Bowman, E. Kruijff, J. LaViola, and I. Poupyrev. An introduction to 3D user interface design. *Presence: Teleoperators and Virtual Environments*, 10(1):96–108, 2001.

[3] D. Bowman and C. Wingrave. Design and evaluation of menu systems for immersive virtual environments. In *Proc. of IEEE Virtual Reality*, pages 149–156, Yokohama, Japan, 2001.

[4] Doug Bowman. Interaction techniques for immersive virtual environments: Design, evaluation, and application. In *Human-Computer Interaction Consortium Conference (HCIC)*, 1998.

[5] Doug A. Bowman, Ernst Kruijff, Joseph J. LaViola, and Ivan Poupyrev. *3D User Interfaces : Theory and Practice*. Addison Wesley, 2004. ISBN: 0-2017-5867-9.

[6] Matthias Bues, Roland Blach, and Frank Haselberger. Sensing surfaces: bringing the desktop into virtual environments. In *EGVE '03: Proceedings of the 9th Eurographics Workshop on Virtual environments 2003*, pages 303–304, 2003.

[7] Erwin Cuppens, Chris Raymaekers, and Karin Coninx. VRIXML: A user interface description language for virtual environments. In *Developing User Interfaces with XML: Advances on User Interface Description Languages*, pages 111–117, 2004.

[8] Philippe Dax. VREng - virtual reality engine. http://vreng.enst.fr/net/vreng/.

[9] G. de Haan, M. Koutek, and F.H. Post. IntenSelect: Using Dynamic Object Rating for Assisting 3D Object Selection. In Erik Kjems and Roland Blach, editors, *Proc. of the 9th IPT and 11th Eurographics VE Workshop (EGVE) '05*, pages 201–209, 2005.

[10] Stephen DiVerdi, Daniel Nurmi, and Tobias Hollerer. ARWin - a desktop augmented reality window manager. In *Second International Symposium on Mixed and Augmented Reality (ISMAR'03)*, page 298. IEEE Computer Society, 2003.

[11] Stephen DiVerdi, Daniel Nurmi, and Tobias Hollerer. A framework for generic inter-application interaction for 3D AR environments. In *Augmented Reality Toolkit Workshop, 2003*, pages 86 – 93. IEEE Computer Society, 2003.

[12] Phillip Dykstra. X11 in virtual environments: combining computer interaction methodologies. *The X Resource*, (9):195–204, 1994.

[13] Niklas Elmqvist. 3Dwm: A platform for research and development of three-dimensional user interfaces. Technical Report CS:2003-04, Chalmers Department of Computing Science, 2003.

[14] Steven Feiner, Blair Macintyre, and Doree Seligmann. Knowledge-based augmented reality. *Communications of the ACM*, 36(7):53–62, 1993.

[15] Patrick Hartling, Allen Bierbaum, and Carolina Cruz-Neira. Tweek: Merging 2D and 3D interaction in immersive environments. In *6th World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, Florida, July 2002.

[16] Patrick Hartling and Carolina Cruz-Neira. Tweek: A framework for cross-display graphical user interfaces. In O. Gervasi et al., editor, *ICCSA 2005, LNCS 3482*, pages 1070–1079. Springer-Verlag, 2005.

[17] Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241, 1997.

[18] Joseph J. LaViola Jr. A survey of hand posture and gesture recognition techniques and technology. Technical Report CS-99-11, Brown University. Computer Science Department, 1999.

[19] John Kelso, Lance E. Arsenault, Ronald D. Kriz, and Steven G. Satterfield. Diverse: A framework for building extensible and reconfigurable device independent virtual environments. In *VR'02: Proceedings of the IEEE Virtual Reality Conference 2002*, page 183, Washington, DC, USA, 2002. IEEE Computer Society.

[20] John Kelso, Steven G. Satterfield, Lance E. Arsenault, Peter M. Ketchan, and Ronald D. Kriz. DIVERSE: a framework for building extensible and reconfigurable device-independent virtual environments and distributed asynchronous simulations. *Presence: Teleoperators and Virtual Environments. Special issue: IEEE virtual reality 2002 conference*, 12(1):19–36, 2003.

[21] Daniel Larimer and Doug Bowman. VEWL: A framework for building a windowing interface in a virtual environment. In *Proceedings of INTERACT: IFIP International Conference on Human-Computer Interaction*, pages 809–812, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Geoff Leach, Ghassan al Qaimari, Mark Grieve, Noel Jinks, and Cameron McKay. Elements of a three-dimensional graphical user interface. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 69–76, London, UK, UK, 1997. Chapman & Hall, Ltd.

[23] J. Liang and M. Green. JDCAD: A highly interactive 3d modeling system. *Computers & Graphics*, 18(4):499–506, 1994.

[24] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. Lopez. USIXML: A language supporting multi-path development of user interfaces. In *9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS 2004*, Hamburg, July 2004.

[25] Valerie Maquil. Automatic generation of graphical user interfaces in studierstube. *B. Sc thesis, Institute for Software Technology and Interactive Systems, Vienna University of Technology*, 2004.

[26] Jose Pascual Molina Masso, Jean Vanderdonckt, Francisco Montero Simarro, and Pascual Gonzalez Lopez. Towards virtualization of user interfaces based on UsiXML. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, pages 169–178, 2005.

[27] Michael F. McTear. Spoken dialogue technology: enabling the conversational user interface. *ACM Computing Surveys*, 34(1):90–169, 2002.

[28] Sun Microsystems. The looking glass project. http://www.sun.com/software/looking_glass.

[29] Noritaka Osawa, Kikuo Asai, and Fumihiko Saito. An interactive toolkit library for 3D applications: it3d. In *EGVE'02: Proc. of the Eighth workshop on Virtual environments*, pages 149–157, 2002.

[30] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[31] Dave Roberts. RealPlaces, 3D interfaces for office applications. In *International Workshop on Tools for Working With Guidelines TFWWG'00*, London, 2000. Springer-Verlag.

[32] George Robertson, Maarten van Dantzich, Daniel Robbins, Mary Czerwinski, Ken Hinckley, Kirsten Risden, David Thiel, and Vadim Gorokhovsky. The Task Gallery: a 3D window manager. In *CHI '00: Proceeding of the SIGCHI conference on Human factors in computing systems*, pages 494–501, New York, NY, USA, 2000. ACM Press.

[33] Benjamin Schaeffer and Camille Goudeseune. Syzygy: Native PC cluster VR. In *VR '03: Proceedings of the IEEE Virtual Reality 2003*, page 15, Washington, DC, USA, 2003. IEEE Computer Society.

[34] D. A. Smith, A. Raab, D. P. Reed, and A. Kay. Croquet: A menagerie of new user interfaces, 2004. Whitepaper, http://www.opencroquet.org/.

[35] J. Marques Soares, P. Horain, and A. Bideau. Sharing and immersing applications in a 3D virtual inhabited world. In *Proceedings of Laval Virtual, 5th virtual reality international conference (VRIC 2003), Laval, France*, pages 27–31, 2003.

[36] Víctor Theoktisto and Marta Fairén. Enhancing collaboration in virtual reality applications. *Computers & Graphics*, 29(5), 2005.

[37] Alexander Topol. Immersion of Xwindow applications into a 3D workbench. In *Conference on Human Factors in Computing Systems (CHI'00) Student Poster*, pages 355–356, 2000.

[38] K. Watsen, R. Darken, and M. Capps. A handheld computer as an interaction device to a virtual environment. In *Proceedings of the International Projection Technologies Workshop*, pages 303–304, 1999.

[39] VRPN, Virtual-Reality Peripheral Network. http://www.cs.unc.edu/Research/vrpn/.

[40] Qt whitepaper, 2005. Trolltech, http://www.trolltech.com/.

[41] VRCO's CAVELib. http://www.vrco.com.

[42] SphereXP. http://www.hamar.sk/sphere.

1-4244-0225-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: UNIVERSITAT POLITÈCNICA DE CATALUNYA. Downloaded on February 9, 2010 at 05:01 from IEEE Xplore. Restrictions apply.