A Reusable Library of 3D Interaction Techniques

Pablo Figueroa*
Universidad de los Andes, Colombia

David Castro†
Universidad de los Andes, Colombia

Abstract

We present a library for reusable, abstract, low granularity components for the development of novel interaction techniques. Based on the InTml language and through an iterative process, we have designed 7 selection and 5 travel techniques from [5] as dataflows of reusable components. The result is a compact set of 30 components that represent interactive content and useful behavior for interaction. We added a library of 20 components for device handling, in order to create complete, portable applications. By design, we achieved a 68% of component reusability, measured as the number of components used in more than one technique, over the total number of used components. As a reusability test, we used this library to describe some interaction techniques in [1], a task that required only 2% of new components.

Index Terms: D.2.2 [Software Engineering]: Design Tools and Techniques—Software libraries; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality D.3.2 [Programming Languages]: Language Classifications—Data-flow languages

1 Introduction

We have accumulated a wealth of information about 3D interactivity, in particular in the form of 3D interaction techniques. They can be classified in several ways, from hardware or domain dependency, to generic categories such as travel, selection, or control. Numerous studies have been done in order to understand how interaction techniques affect users performance and how they fit the particular requirements of a user in a particular domain.

Despite this progress, it is still difficult to develop applications with complex 3D interfaces. Novel techniques are created in novel scenarios, but it is difficult in any new development to benefit from previous work. We believe part of the problem is the set of tools we use to build new interaction techniques. Current languages and frameworks do not facilitate the development of reusable components for interaction. We also count with several libraries and toolkits to handle devices, audio, or 3D graphics in certain programming languages, but a counterpart for interaction and models for their integration to the rest of the application are still missing.

We propose a library for the development of interaction techniques that is independent from traditional programming languages. It is based on the InTml, the Interaction Techniques Markup Language [10], that allows the description of VR applications as a dataflow of components that represent devices, content, and behavior. Since reuse might happen at a lower level of granularity that a whole technique, we want interaction techniques to be composed by simple, independent components, which could be reused in other techniques. In this way, novel techniques can benefit from previous developments and reuse interesting parts, while adding new components as required. We also want to be independent of particular implementation APIs and programming languages, so this descriptive library could be implemented on top of a wide variety of technologies. In this way, VR applications described in this language have more opportunities to be ported to new technologies during their active lifetime.

This paper is divided as follows: First we present previous work. We then describe the components of the library, its divisions, and some examples of interaction techniques we have developed, embedded in entire applications on a hardware environment based mostly on a tracker and a generic display. Later we present some reuse metrics about our current implementation. Finally we present some conclusions and future work.

2 Related Work

This work is related to the topics of high level programming languages and libraries, in the particular field of VR applications and in the generic space of Computer Human Interfaces (CHI). We describe first some important results in high level languages for CHI and in particular VR, and later we describe some results related libraries for interaction. An assumption in our work is that previous results in CHI, mostly related to 2D interfaces, are either too generic or too hard to accommodate to the specific representational needs of the wealth of 3D interaction techniques our community has developed.

Our work is inspired in early efforts around the concept of User Interface Management Systems, or UIMS [17, 12], which provided a semantically rich metaphor for the development of user interfaces and a set of tools that supported several stages in the software life-cycle. Although there are some examples related to 3D interaction representation (i.e. the daisy menu in the PMIW language [12]), these works mostly concentrated in other elements important to UIMS, such as graphic feedback, device management, IDE support, or debug support.

Some high level languages for interface development in the field of WIMP [2, 16] and post-WIMP [12] interfaces used formal specification mechanisms such as Petri Nets, simplified programming languages, or state machines, which allow automatic checking but they are usually difficult to comprehend to non-expert users. Some abstract models and their implementations have been developed for the support of certain application types (i.e. multimodal applications in [4]). We are more interested in systems that use a dataflow abstraction, as ours, and the libraries of interaction techniques they may have created. The Input Configurator and MaggLite in [9, 11] present a three layered dataflow that connect physical input devices to responses in post WIMP applications, an approach similar to ours that we generalize to more VR devices and more complex, extensible, and non-layered dataflows that represent generic 3D interaction techniques. Squidy [14] presents a simplified dataflow with just one input and output port per node that allows the integration of several types of novel devices through standard protocols and a Java-based core. Our approach hides such particular protocols and it is language independent, which presents a cleaner abstract model. In [15], a dataflow based, UIMS-like system for the development of multimodal interfaces is presented, with very interesting features such as design by example and group development, but still a more comprehensive library of interaction techniques is missing, as a result of an analysis of a space design instead of a result of representing available technologies.
There have been several attempts to create high level languages for the description of VR applications. For example, [20] presents a formalism based on Petri Nets, [22] proposes a dual control and data flow architecture, and [24] proposes modules for interaction design, just to mention some recent results. However, there are few proposals in these languages related to libraries of reusable and independent components, specifically for interaction techniques. Finally, novel interaction techniques development methodologies such as [7] do not provide clear proposals for libraries of interaction.

We should also mention the state of the art in game engines, that provides a good starting foundation for several types of VR applications. However, in general, their architectures do not consider changes in the interaction techniques, and they are fixed to the most popular techniques with standard devices, mostly keyboard and mouse.

There are several libraries that focus on the visualization part of an interface, both in WIMP and 3D applications, such as [3], where the interaction is implicit or limited to the standard metaphors and devices in WIMP. In terms of libraries of interaction techniques and novel devices, there are some attempts in the field of WIMP [18] and post-WIMP interfaces ([11]). Several systems provide fixed and implicit sets, such as the standard X3D [23], so their purpose so far is limited for the creation of novel interaction techniques in novel hardware environments. Some extensions such as Contigra [8] provide interesting implementations of common interaction techniques, but due to the programming style in X3D, it is difficult to identify components for reuse in novel interaction techniques. Several systems declare the existence of a library of devices or reusable modules (i.e. [9]), although there are not enough details about its support to generic interaction techniques of design process. Squidy [14] introduces the concept of a searchable knowledge base of filters, although few details are available about the design rationale of such collection of filters. Commercial systems such as 3D Via [22] or Vizard [25] provide a wealth of functionality, although their components do not particularly address interaction, their component’s communication and encapsulation characteristics can be improved, their design has not taken reusability and shared code from the beginning, and they are tight to a particular language and vendor.

Current libraries and toolkits include some support for interaction techniques. Device libraries such as VRPN and OpenTracker provide a common, programming language independent interface to devices, although they just offer simple ways to connect the rest of the application code. Complex scene graphs such as Java3D provide some support for interaction modules, but they have limited components and have not evolved despite their time in the market.

3 The Library

The library was defined by means of an iterative process, that added at a time one interaction technique from [5] and a sample application that use it. Each time we added a new interaction technique, we took into account the components available in the library. If possible, we either reused or slightly modified an existing component. If it was not possible, new components were created and added. All these components and libraries were created with the aid of our IDE, a set of plug-ins for eclipse that allows us to create libraries, components inside a library, applications, and generate code stubs in a particular runtime. Figure 1 shows our IDE with the object library and a ray casting application. Although details are too small, it can be seen the canvas for visual programming, the set of tools that each canvas provides, the outline view that shows the entire application, and the list of properties for the selected component in the application.

The following subsections describe key components in the library, divided in three main groups: devices, objects, and behaviors.

First we describe the basics of the dataflow model we are using, and the basic types that all components are based upon.

3.1 Basic Model Elements

This library is defined on top of the InTml [10], which describes VR applications as a dataflow of components, which can then be translated to a particular runtime environment, provided that such a target environment has been implemented. Each component has a type, that describes a set of input and output ports that can be used, as the only method for intercomponent communication. Not all ports have to be connected in an application, and zero or more events can be received by each port at any particular frame. Ports have types, which are an abstraction of basic types in low level programming languages. A port type can also be a component, in order to allow objects to be sent to other behavior components. For example, components for collision detection may require the geometry of a pointer object, which could be sent to this module through an input port. Entire applications can be defined as dataflows of components, some of them generic, some of them application specific. It is also possible to encapsulate entire applications in a component, although this mechanism precludes reusability.

3.2 Basic Types

As part of our requirements we want to make this description as programming language independent as possible, so it may be feasible to implement these modules in several execution environments, especially in strongly typed, general purpose programming languages. For this reason we define the types in Table 1. These types can be easily translated to the language of each runtime environment, as part of the basic implementation support. From this table, it is important to notice the Signal type that informs the occurrence of an event with no associated information, and the Id* types that contain an identifier and information of a certain type. The latter allow ports to send events from several sources, as we will mention in the following sections.

Spatial types are also of particular interest in 3D applications. For example Pos3D model a 3D position in space, both Orientation and Quaternion model an orientation in space, and Matrix4 can model both. While we have required this redundancy in certain components, in general we prefer to have abstract types that can be implemented in several ways. In that sense, we prefer Orientation than Quaternion, and pairs of instances of Pos3D and Orientation events instead of one Matrix4 event.
3.3 Devices

The most easily understood component from the designer’s point of view is a device. A device in this library is an abstraction of a family of physical devices (i.e., gamepads, joysticks, and trackers), as independent as possible from particular APIs or low-level programming languages, which share form factor and functionality. Our purpose is that each device component can be directly related to a physical device, therefore facilitating its identification during design and maintenance. Although they are technically speaking outside of the scope of interaction techniques, it is important to us to be able to describe an entire application in the same formalism. Therefore, devices (as well as content objects, as it is mentioned in the following Section) were required. Devices get reused when several applications use the same hardware in their solution. We have also defined components for one of a kind devices such as the SpaceMouse, the Wii Remote Control, the Phantom Omni by Sensable, the Falcon by Novint, the P5 Glove, and more specific devices can be defined in the future by designers. These one of a kind devices may be generalized (i.e., a generic haptic device, or a generic glove), once more devices are included in the library and commonalities between them are found. In this way, a designer will be able to use either a generic component that describes common functionality in its particular device, or a particular (but maybe not so reusable) component that gives access to all of a device’s functionality. We also included a component for one of our custom-built devices in order to show the library’s extension possibilities.

Generic devices provide a common interface for a family of physical devices, which have the same form factor, expected outputs, and expected inputs. For example, Figure 2 shows a generic joystick. Since joysticks can vary in their number of elements (i.e. buttons or analog sensors), we define the following ports per element of interaction: one that outputs the number of elements, and ports that send streams of each type of event such elements can produce. For example, the joystick component has the numButtons port in order to output the number of buttons a particular device has, and a set of ports (i.e. btnsPressed, btnsReleased, btnsClicked) that outputs the events such devices can produce, each type of event in its own port. In this case, since these events do not have attached information, we model each event as an integer that informs the identification (a number) of the particular button that generated such an event. In a similar manner, joystick has the numAnalogs port that informs how many analog sensors the device has, and analogValues that outputs a stream of analog events. Each one of these events is modeled with the IdFloat basic type, which is a tuple that indicates the id of the sensor that produces the event and its float value. The last output port is devName, which allows us to identify particular devices, when such information is provided by the subjacent driver. Finally, we added a vrpnConnInfo input port, that allows us to define the configuration string that our subjacent driver requires in order to receive events from the physical device — in this case, VRPN. This port can be generalized in order to receive any particular configuration string that may be required by inner implementations.

Device families in our case are defined not only in terms of functionality, which could be seen in the available input and output ports, but also in their form factor and ergonomics. For example, Figure 3 shows a generic gamepad. Although we can notice it has the same input and output ports than a Joystick, it is used in a different way, it has a different form factor, and therefore we believe it is important to separately describe it.

Special purpose or one of a kind devices are represented in the same manner, although their ports are directly related to what a particular driver implementation provides. For example, Figure 5 shows a component for the Wii Remote Control. Its ports represent available input and output through the wiiuse library 1. In this case, it is up to the designer to provide as many ports as possible, which may preclude implementation in several programming languages.

1Details of the wiiuse functionality can be found at http://www.wiiuse.net/.
due to variations in support, or provide minimum functionality that could be implemented in all targeted environments. In this case, the WiiDevice is modeled for completeness. It provides events for all its buttons, analog values for sensors like its joystick and accelerometer, the number of infrared dots that are detected at any time and their positions in a projected 2D space, and the current battery level. As inputs, it may receive a vibration value, a sound value, and a combination of a connection string and an integer that identifies it, if more than one Wii device is present.

3.3.1 Utility Components for Devices

In order to facilitate the use of device components, we designed a set of utility components with the following functionality:

- **Selectors.** Output ports of types such as IdFloat or IdPos3D provide a stream of values of the same type from several sources. Selectors filter information from a particular source, each one identified by an integer. A selector receives the stream of tuples (id, value) and a particular integer that identifies the source to filter, and they output just the information that such a source provides. One example of these components is shown in Figure 6, that filters position and orientation events from a particular tracker or marker. Note that the type for the output ports do not require the source id that is embedded in the input events.

- **Echo.** A component for showing any type of events in the console.

- **Type translators.** Several input events can be combined in order to produce an event of another type. For example, two Floats can be combined into a Pos2D, or a Pos3D that represents a vector can be converted into an Orientation. Utility components can do this job.

- **ComposedEventsButtons** produces events for buttons clicked, double clicked, held, just pressed, or just released.

- **SemanticButtonEvent** produces events that require knowledge about devices. For example, shootEvent is produced once the shoot button in a particular device is pressed, which depends on the particular ergonomics of a device.

3.4 Objects

The object library solves common requirements that interaction techniques impose to objects in the scene, and common objects that are required in such techniques. The basic interactive object is shown in Figure 7. It allows loading its geometry from a file, basic transformations (i.e. translation, rotation, scaling), appearance changes (i.e. to show a bounding box, make it visible, change its material), and parenting changes, so it can be dependant from other object. Since several changes can be applied at each frame, an object can inform at the end which is its final state.

Other objects in this library represent basic shapes (Ray, Cone, Cylinder, Plane, Sphere, Cube, Disk), a loader from a file that separates several objects on it(Scene), and a component for a Material. In the case of basic shapes, although they can be created as a SimpleObject, their particular types allows specific modifications and functionality for interaction. For example, Figure 8 shows a ray as we found it useful for interaction. We can load its geometry and transform it as a generic object, but we can also define it in terms of a position, orientation, length, and radius, which allows us to provide feedback about the direction of selection or gaze in a particular interaction technique. Special output ports are also related to these features, such as length and radius.

3.5 Behaviors

These components provide the foundational algorithms for interaction techniques. Currently, we have divided them into selection behaviors and travel behaviors, but they could be rearranged in the future, once more interaction techniques are represented and more
commonalities are found (i.e. components for travel techniques that can be used for selection).

The selection components consist on the basic algorithms for object selection (i.e. RayCaster, ConeCaster, ObjectCollisioner), and utility components that allow the implementation of interaction techniques, some of them presented in Section 4. Figure 9 presents the ConeCaster. It receives the selection cone and the selectable objects. As output, it computes the new set of selected objects (i.e. the ones inside the cone). Figures 10 and 11 present the RayCaster and ObjectCollisioner, respectively. Notice how similar their structures are, in particular the expected outcome (one or more selected objects) from an input set. Other selection techniques can be added in the future, such as fast solutions based on GPU computations or OpenGL state. In these cases we foresee more basic types in order to represent the input information these techniques will require, and explicit output ports in display devices in order to produce such information.

The travel library contains 6 technique-dependant components, among them a way to draw a path in the floor (PlaneDrawer), an interpolator between positions (PositionInterpolator), and a way to identify the exact collision point between a ray and a floor plane (FloorRayCaster). Most of them are used in the Drawing a Path technique [5, p.207]. We show here the IncrementalSpeedPositioner (Figure 12), which receives a current position, current speed, and orientation of movement, and outputs the expected new position. Notice that the component does not actually change the view position. Instead, this or other component can affect the View instance of an application and therefore move the user, as we will see in the examples below.

4 Examples

As we have said, we have implemented several interaction techniques from [5]. These interaction techniques are represented as dataflows of instanced components, and we show some of them here in the context of applications that use a generic display (not visible in the diagrams and therefore implicit), trackers, and Wii controls as devices. Devices can be changed at will if necessary, although it is future work to identify which interaction techniques make more sense in particular hardware scenarios.

Figure 13 shows an application that uses one of the most common selection techniques, ray casting [6]. The core of this technique is the rayCaster component, that receives a Ray that is moved and oriented by means of a hand tracker, and the selectable set of objects in the scene. Objects are loaded from the vrmlScene.wrl file by means of the Scene component. Objects are loaded by the Scene, and copied by an ObjectCopier. From those copies we select which copies will be selectable and which not at objectsForSelection. Both sets of copies are handled as ObjectSets and scaled. Selectable copies are given as the input set of objects for a rayCaster, which will inform which object is selected by a ray 4. Finally, objectSelById will output the objects whose copies were selected.

As an example of traveling, Figure 16 shows a gaze directed steering technique [13]. A tracker’ orientation is selected (by headSelector) and transformed into world coordinates (by means of headOffset). The positioner component computes a new position by means of such an orientation plus the current view position and a speed factor. In this example, the speed is defined by means of two buttons of a WiiDevice, that increment or turn to zero the current user’s speed.

4.1 Implementation

We have implemented most components on top of our C++ runtime environment, which is built on top of VRPN, OpenSG, xerces, and VR Juggler. We have measured an overhead of 10% in our current implementation as the average time of each cycle vs. the sum of averages for each component in a simple application. More work has to be done to compare the performance of our dataflow vs. the performance of a similar application in other formalisms.

5 Metrics

We have performed two analysis of the components and applications we have designed so far. First, we want to show that the-de-
signed components are indeed reusable, from the point of view of the designed applications. Second, we want to test reusability of these components with new applications and interaction techniques.

5.1 Level of Reusability in Examples

Table 2 shows a summary of components used in the 12 applications we have designed, which implements the following interaction techniques: Ray Casting (RC), Two-Handed Pointing (TP), Flashlight (FL), Aperture (AP), Virtual Hand (VH), Go-Go (GG), World in Miniature (WM), Walking Camera in hand (WH), Gaze Directed Steering (GS), Pointing Torso (PT), Drawing a Path (DP), and Manipulating User Representation (MR).

We use 28 components in these applications, of a total of roughly 50. Non used components are either devices or utility components we designed for completeness. Of this 28, only 9 are used in just one interaction technique. So far, we have reused more than 35% of the components we designed, which is a good percentage according to common practices. If we take into account just the used components, we have achieved a 68% of reusability.

Most of the applications use between 10 and 15 components, which we believe shows a good granularity level: fewer components might be difficult to reuse, while more components might be too many for these size of programs. Nevertheless we could also notice some repetitive patterns due to common functionality in order to handle trackers, their translation to world coordinates, view handling, and feedback. If we omit these common components, we can find that interaction techniques are implemented in terms of 3 to 8 components. Of particular interest is the low percentage of components that are unique to an application: in average, only 6% of the components are unique in these applications, which seems to indicate that the design space for interaction techniques is rather similar, with very few variations between techniques.

5.2 Extended Reusability

As a test for component reusability, we tried to design other applications with novel interaction techniques, apart from the ones we covered here. For that purpose, we took a small sample of 4 techniques from [1] (Object Picking, Identifiers, First Person, and Scripted Navigation). They are similar to some of the techniques we use, and we only needed minor changes (modifications in some input ports or changes in devices) in order to design them. In total, we made changes in 2% of the library.

6 Conclusions and Future Work

We have presented a library of reusable components for the development of interaction techniques. The interaction components are combined in a dataflow with components that represent content and behavior, in order to create complex applications. These
components have shown to be highly reusable among the selected interaction techniques from [5] and [1]. This library, the eclipse-based IDE for InTml, and the InTml runtimes are available at http://intml.sourceforge.net.

There are several issues we plan to address in future work. We plan to continue our description of more interaction techniques, newer than the ones included in the books we used as reference. We want to include components that could address more complex applications, in order to show reusability when application-specific components are included. We plan to complement previous usability studies we have performed. We have shown that non-programmers could understand InTml and do simple applications after few hours of introduction [10]. We want now to do more studies on the issue of library reusability, and how we can compare productivity with InTml to other formalisms. We also plan to include in our IDE implementation other abstraction mechanisms in our formal description such as composite components, which could help in the understanding of applications with reused parts.

ACKNOWLEDGEMENTS

Thanks to anonymous reviewers, for their very valuable comments.

REFERENCES

[7] J. Chen and D. A. Bowman. Domain-specific design of 3d interaction techniques: An approach for designing useful virtual environment ap-
Table 2: Used Components in Sample Applications. Numbers Indicate How Many Instances of a Particular Component an Interaction Technique Has. Abbreviations are defined in Section 5.1.