Software Architecture for Multimodal User Input – FLUID

Tommi Ilmonen and Janne Kontkanen

Helsinki University of Technology
Telecommunications Software and Multimedia Laboratory
Konemiehentie 2, Espoo, Finland
Tommi.Ilmonen@hut.fi, Janne.Kontkanen@hut.fi

Abstract. Traditional ways to handle user input in software are uncomfortable when an application wishes to use novel input devices. This is especially the case in gesture based user interfaces. In this paper we describe these problems and as a solution we present an architecture and an implementation of a user input toolkit. We show that the higher level processing of user input such as gesture recognition requires a whole new kind of paradigm. The system we designed and implemented - FLexible User Input Design (FLUID) - is a lightweight library that can be used in different kinds of software. The potential application areas include all system where novel input devices are in use: virtual reality, entertainment systems and embedded systems.

1 Introduction

Input devices used by most of the computer software are a mouse and a keyboard. Still there are many applications and platforms in which using these standard devices is awkward or impossible. Currently, interest in alternative input methods is increasing, because lots of new kinds of devices that cannot use the conventional input methods are emerging into the market. These devices include information appliances such as mobile phones and hand-held computers as well as embedded systems. Embedded systems, such as those in modern washing machines, have been around for long, but their programming is still done on relatively low level without sophisticated toolkits for user interaction.

This paper introduces a new paradigm and a toolkit for managing input devices. This architecture is suitable for any application where novel input devices are in use. The system is scalable from embedded systems to ordinary computers. The design takes into account the needs of higher-level application development – support for input data processing (gesture detectors etc.) and ease of programming. While the system is generic in nature we have developed and used it primarily in virtual reality (VR) applications.

The novelty of our approach is in the new architecture to handle multimodal user input. While our approach shares common features with some previous systems, the overall structure is unique. Also it seems that the FLUID architecture is the first to emphasize the need to design the low-level input API and data processing layers at the same time. In addition to restructuring ideas from previous research our system introduces the concept of device-specific *history buffer*. This paper focuses on presenting the

architecture, but also introduces our implementation of the architecture and examples of how we have used it.

When designing the architecture we have taken into account the need to collect data from various devices and the need to further process the data. It also enables efficient sharing of input processors (gesture detectors etc.) between possibly very different applications. The architecture is composed of two layers: the input layer and the data processing layer (figure 1). The input layer handles the devices and maintains a buffer of history data for each device. The data processing layer is used to process the data – detect gestures, generate events and calculate features. The purpose of these layers is to offer a simple and universal method for application developers to access the devices and to refine the data.

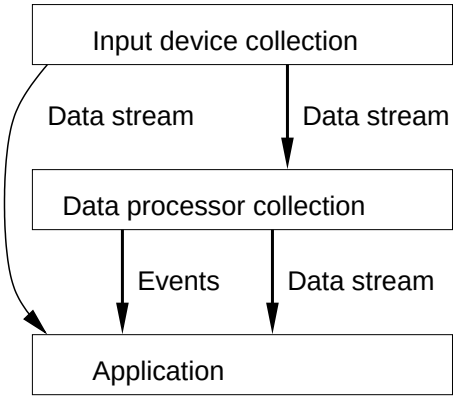

Fig. 1. Overview of the FLUID architecture.

The architecture we designed satisfies the following requirements:

- Manage arbitrary input devices for any kind of application
- Offer a good infrastructure for data processing
- Offer a way to share data processing elements (gesture detectors etc.) between applications
- Specify a simple architecture for these tasks

In the end we hope to make multi-modal input management easy for the application developer. In the ideal case a developer would select the desired input devices and data processing modules (gestures detectors, signal converters etc.), add the necessary callbacks to pass the event and signal information from the input toolkit to the application and then concentrate on the application development.

Our test-bed for these experiments is a virtual reality system. Since normal input devices – mouse and keyboard – function badly in VR applications we must employ novel devices and combine data from those. The FLUID project was started to enable faster and more cost-effective application development in VR environments. At the same time

we wanted to create a system that is not VR-centric. Instead these new tools should be re-usable in other contexts – desktop computing, entertainment systems and even in embedded (or ubiquitous) systems.

This paper first describes the FLUID architecture and toolkit for collecting and refining input data. We then introduce a way in which it can be used in virtual reality software and give examples of applications that we have built with FLUID.

FLUID is an attempt to create a framework that fits to the needs of applications that need user input and that need to process that data. It's design supports any number of concurrent input devices and fulfills the needs of data processing algorithms. It is easy to extend – a programmer can add new device drivers, device types and data processing algorithms. FLUID offers a design framework that enables developers to avoid application-specific custom-solutions. Thus FLUID promotes software re-usability.

This work is heavily influenced by our earlier work on full-body gesture recognition and gesture-based user interfaces [1][2]. These earlier systems were not VR-driven, instead they were built for musical goals. As we kept working with multimodal gesture-based interfaces it became clear that working with various kinds of non-conventional input devices is anything but straightforward. We think that it is necessary to try to attack this problem and try to make the application development for multimodal environments easier in this aspect. During our previous research we created highly customized pieces of software for collecting and processing the input data. Unfortunately it is difficult to reuse these components in any other application due to lack of well designed standard framework. We would like to avoid this situation in the future. FLUID project was started since we could not find toolkits or architectures that would offer the features that were needed. The FLUID toolkit will be released under an open-source license.

2 Multimodal Interaction

As long as people are forced to interact with computers using mice and keyboards important elements of human communication are lost. One cannot use gestures, speech or body motion with such clumsy devices. Instead we are forced to express ourselves with key-presses and mouse.

Our research is inspired by the needs for different interaction modalities. This need is caused by the fact that mouse and keyboard do not offer the best interaction method for all applications. Embedded applications (phones, PDAs) as well as immersive applications (virtual and augmented reality) cannot rely on the same interaction modalities as normal desktop computers.

We believe that the interaction style has direct impact on how people perceive technology. There is a difference between entering text by handwriting, typing and talking. Even tough people seldom use computers just for the sake of interacting with them the method of interaction needs to be considered carefully. For this reason we believe that it is necessary to offer alternative interaction modalities when appropriate.

Having alternative interaction methods is also a way to find new target groups for technology. For example children or illiterate people cannot use text-based communication with a computer. By enabling multimodal interaction we can make information technology more accessible for these people thus leading to more universal access of

computers. Since FLUID can be used to enable different input strategies for a single application it is a useful tool for building applications with universal access in mind.

In this part we share view with Cohen who argues that voice/gesture -interaction can offer significant advantages over classical interaction modalities[3]. Cohen also gives examples of how a multimodal interface has been found to make applications more productive. Although Cohen is primarily concerned with immersive applications we feel that multimodal interaction is important in other environments as well.

An interesting view to human-computer interaction is given by Schoemaker et al. who have studied the levels of observation[4]. Their work classifies four levels of observation – physical/physiological, information theoretical, cognitive and intentional. Many user input toolkits work on the information theoretical level of this model – they are only concerned with raw input data or simple manipulation of the input data. For real applications the cognitive level is usually more important since this is where the data gets its meaning.

The word "multimodal" is widely used to describe interaction systems. Unfortunately it is a word with many meanings. Term multimodal can be used to describe a system with multiple communication devices (mice, keyboards, cameras). The term can also be used to mean communication that uses different modalities (writing, drawing, gesturing, talking). The first definition is device-centric while the second is more human-centric.

To be able to utilize different communication modalities computers must also have different input devices. This is where our research is targeted. In this paper we use term "multimodal input" to refer to systems with multiple novel input devices. Of course any multimodal application is likely to have multiple output devices as well.

2.1 Software for Multimodal Input

Multimodal software is difficult to create. There are several obstacles – novel input and output devices and the need for diverse special software (rendering, animation, audio processing). In our own work we have found that there are few if any toolkits that would make it easier to handle multimodal user input.

The first task for an application is to collect the input data. This is a difficult task when one considers all the possible goals that should be satisfied. The system should not consume excessive amount of resources, it should be portable, it should accommodate different versions of the same device class (devices from different manufacturers), it should be extendible and it must fulfill the needs of the data processing algorithms.

An application seldom uses the input data directly. Instead of using raw input data an application needs refined data – information about what the user is doing. To bridge this gap we utilize gesture detectors and feature extractors. These algorithms turn the low-level numeric signals into more descriptive form, often compressing a multichannel signal to just a few events. An algorithm can be very simple – for example it is easy to create ad-hoc algorithms to detect hand claps, provided that the user has tracker sensors attached to both hands. A more complex algorithm might be used to interpret sign language.

All the data processor algorithms have one thing in common; they need data that is precisely in specific form. Most time-based gesture analysis algorithms work best

with constant-rate signals. That is, the input device generates samples at fixed intervals and the analysis algorithm is designed to work with such constant frequency signal. For example all digital filtering algorithms rely on constant sampling rate (see for example the algorithms in common DSP books[5]). The same is true for artificial neural networks that use the time-delay approach.

These considerations lead us to set the following requirements for the input layer:

- Data should be collected at constant sampling rate
- The system should know when a given sample was sampled
- It must be possible to utilize signals of different sampling rates
- The application must be allowed to access the input devices at arbitrary rate
- The user may instantiate several devices of the same type

The data processor layer in turn must have the following properties:

- Ability to turn input data into events – for example motion signal can be used to detect gestures
- Ability to transform signals to other kinds of signals – we might be only interested in the velocity of a sensor, or the mean velocity of a sensor
- Support re-use of data processors – we want to re-use the analysis tools in many applications

3 Related Interaction Research

In interactions research our topic is the design of input toolkits. While there are several competing toolkits for graphical 2D user interfaces (GUI) we have not been able to find general-purpose toolkits that would be designed to manage multiple novel input devices and support the input data processing.

The other trends in interactions research are not directly related to this work. For example Nigay's and others' work on the design spaces is directed towards the classification of different interaction modes and modalities[6]. The authors also propose an architecture for complex multimodal systems, but their architecture is more concerned with application logic and application interaction design. Thus it has little to say about how the user input is collected and processed. While our work is not directly connected to their's it is worth noting that these approaches are not conflicting.

Salber has published a "The Context Toolkit" for sensing the presence of the user acting upon that information[7]. Their approach is to gather data from environmental sensors and create widgets and turn the information into events. The context toolkit has been used in another project by Mankoff where it was combined with speech recognition engine to collect and process ambiguous user input data[8]. FLUID differs from the context toolkit by being aimed at a wider audience – while the context toolkit is targeted at sensing the presence of the user FLUID is intended for any kind of work. The example applications described by Salber and Mankoff do not apparently stress low latency, high performance or quality of the input data or the easy programming interface that are the basic requirements of the FLUID architecture. The context toolkit could be implemented with FLUID by creating the desired device drivers and coding

the processor objects that correspond to the widgets in the context toolkit. The ambiguity management described by Mankoff has no direct equivalence in FLUID although it seems it could be implemented on top of the generic FLUID framework.

The need to extract higher-level information from low-level data is shared between many kinds of applications. Often such applications separate the information retrieval (or gesture detection) to separate layer. This is the case with applications that use computer vision for user input and gesture-based interaction systems. For example Landay has used such approach in creating the SILK-library for handling 2D-sketches[9]. While this approach resembles the FLUID is structured it does not implement some of the key features that a multimodal input system needs: inclusion of arbitrary input devices and accommodation of devices with different sampling rate.

4 Related Virtual Reality Research

In VR applications one is always confronted by non-conventional input hardware. As a result VR toolkits usually offer a way to access input devices. A practical example of such a system is the VR Juggler[10]. VR Juggler offers an abstraction for a few input device types – motion trackers, data gloves and analog inputs. It also includes a few utilities that can process the data further. VR Juggler includes simple finger gesture detector code and coordinate transformation code for the motion trackers. Also the older CAVELib[tm] toolkit can manage motion trackers[11].

There are also VR toolkits for input device management. OpenTracker is an example of such an approach [12]. It is a toolkit that is aimed at making motion tracker management and configuration easy and flexible. The VRPN (virtual reality peripheral network) system is another toolkit for managing input devices[13]. While OpenTracker is an effort at high-quality tracker management VRPN is a more general-purpose system – it can be easily extended to handle any kind of input devices. The VRPN shares many features with FLUID. The main difference is that FLUID includes an architecture for processing the input data.

Cohen has created a *QuickSet* -system for multimodal interaction with distributed immersive application[3]. QuickSet is directed towards commanding 2D and 3D environments and it supports gesture and voice interaction. It covers all areas of multimodal application development – input, application logic and output. It is created with distributed processing in mind. Our approach differs in that FLUID architecture is more simple, it is not targeted only at detecting commands and it is does not address the distribution of processing elements. FLUID is also intended to be a small component that can be added to any application – not an application framework that would require specific programming approach.

Bimber has published a multi-layered architecture for sketch-based interaction within virtual environments[14]. Although that work is directed at sketching applications the software architecture could probably be used for other purposes as well.

5 The Fluid Architecture

At present there is no common way to handle the novel input devices. If one builds a 2D GUI then there are several toolkits available. All of these toolkits include similar structure – a collection of graphical elements and user input via call-back functions. This contrasts the way one handles non-standard devices. Each application has its own special way of handling input devices and -data. For this reason we propose a new architecture for handling multi-modal user input.

The FLUID architecture contains 1) input layer, 2) data processor layer and 3) application (see figure 1). The application executes in its own main loop and refreshes the fluid layers frequently. All the input devices are managed by a central object – the input device collection. The application may use one or more data processor collections to refine the input data into more usable form.

The main purpose of the input layer is to collect data from various devices and present it to the application and data processors with a simple, monolithic API. Although this process is simple there are still pitfalls that must be taken care of. If we think about multimodal interaction this layer corresponds to the device-oriented definition – it is responsible for handling multiple different devices.

The data processor layer refines the input data in a way that application can better utilize it. The purpose of this layer is to extract semantic information from the raw input data. If we follow Schoemaker's terminology then we can say that this layer tries to obtain cognitive information from the user input.

If necessary the processor layer can be used to hide the input devices from the application. This way the input devices can be changed with minimal changes to application structure. For example hand claps can be detected with different input devices – camera, microphone or motion tracker. If the application only wants know that the user clapped his hands together then it does not make difference how this information is obtained.

6 Input Layer

All the input devices share the same abstract base class. To create a device type one needs to inherit this base class and add the device-specific data structures to the new class. This new class is effectively an API for that device. The FLUID library contains definitions for a limited number of input device types, but users can add new device types without modifying the core library. In practice all device APIs should be defined in the base library. If they are not, then people may create different and conflicting APIs for the same device types. At any rate we feel that is it necessary to offer users the possibility to add device types of their own.

The type of the data that is stored into the buffers depends on the type of the device. Thus motion tracker samples are stored as an array of rotation matrices and location vectors, data glove samples are stored as an array of finger joint angles etc.. Each data element is timestamped with its measurement time. The time-stamping is necessary since the higher-level components may need to fuse signals of different sampling rates together (for example magnetic tracker at 68 Hz and data glove at 50 Hz). Without timestamping there would be no way to tell which samples coming from different sources took place simultaneously.

6.1 Threading and Buffering

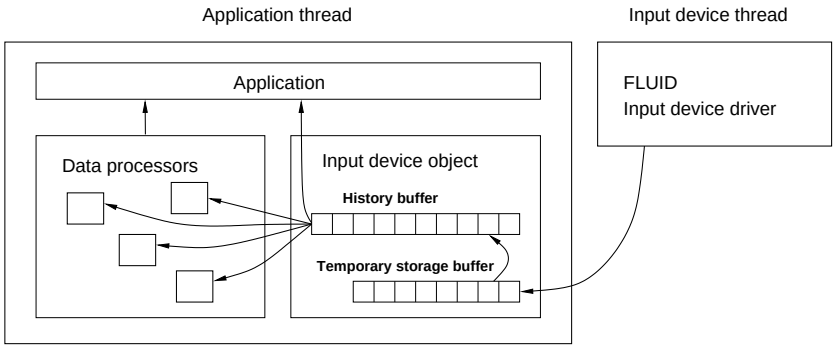

Fig. 2. The input thread, buffers, and data transfer paths

The input layer contains objects that take care of all the input devices. Figure 2 outlines the way input devices work. Input objects are threaded – they collect data in the background and move the data to a temporary buffer. This means that each device object contains an internal thread that reads the input from the device's native API. The data is then moved to a history buffer when requested. This makes the data available for the application.

Double buffering is necessary since it is the only way to guarantee that every input sample becomes available to the application and data processors. If this was not done then the application would have to update the input devices at such a rate that no sample can escape. In practice this is a difficult requirement – the application main loop would have to check the devices at fixed sampling rate. With our approach the application simply needs to re-size the history buffer and temporary buffer to be large enough to contain the necessary amount of data. While the size of the history buffer determines how much history data is available for higher level analysis the temporary buffer sets the upper limit for the input layer update interval. In any case the history buffer needs to be at least as large as the temporary buffer.

As a result the application can run its main loop in variable frame rate and update the input layer only when necessary. Even tough the input layer is updated at random intervals it will read the input data at fixed rate and store the data in the internal buffers. This threading approach is similar to the approach used by VRPN[13].

We chose to store the data to buffers since this makes the history data directly accessible to the processor layer. Thus if a processor in the higher level needs to access the history data (as gesture detectors frequently do) then the data is available with no extra cost. A gesture detector may require several seconds of input data. It is natural to use the input data buffer to store this data so that the gesture detectors do not need to keep separate input history buffers. In the general case the input device object cannot know how much history is required by high level analysis. For this purpose the processor objects request the input device to enlarge its buffer to be large enough for the needs of

the processor. This leads to minimal memory consumption as all data is buffered only once (in the input device object).

The buffering can also increase performance: If the samples were handed out one at a time (via call-back as in VRPN) then each new sample would have to be separately handled. This is not a problem with devices with low sampling rate, but if we consider audio input at 44.1 kHz then this approach takes lots of computational resources. In these cases the most efficient approach is to handle the data as a buffer of samples and process many samples whenever the application main loop executes.

The buffering approach is also useful when different kinds of data are used together. If there is one object receiving data from several sources it is usually best to update this object once all the source devices have been updated. Then the receiver can process all the new data at once. If we used call-back functions to deliver each new sample to the high-level processors then a processor might need to first wait until it gets all the necessary data from various sources via the call-backs, store the data internally and eventually process the data.

6.2 Device Management

Even tough the input layer is highly threaded, this is invisible to the application programmer; the history buffers are guaranteed to change only when they are explicitly updated. Thus the application programmer does not need to take threading issues into account.

The input device drivers are hidden from the application. This is necessary since they are used to abstract the exact device brand and model from the user. The drivers are designed to be very simple – they simply output one sample at a time.

The driver can be used in one computer, its data is sent over the network to the application running FLUID and received by the corresponding network driver. This distribution of device drivers over a network is necessary since VR installations often have several computers with one computer handling one physical input device. For example in our installation we have an SGI computer for graphics, but the data glove is connected to a Linux PC. The speech recognition software also runs on the Linux PC. The only way to cope with such complex hardware/software dependencies is to run the device-specific servers in the machines that can run them and transfer the data to the computer that is running the actual application (like VRPN).

6.3 Input Device Collection

The input devices are managed by a central input device collection -object. This is a singleton object that is globally accessible[15]. The device drivers are plug-ins that are loaded into the application as the input device layer is initialized. The user can configure the devices via a text file. Thus there is no need to recompile the application to get access to new devices or to change the devices.

When an application needs a particular device it requests the device from the input device collection. If the device is already initialized it is returned, but if not, the system tries to initialize it and then returns it. This allows applications the ease to ask for any

device at any time. Since the input collection keeps track of devices the programmer does not need to worry about how to start or shut down the devices.

If the application needs to do complex operations on the input devices then this approach may not fit the needs. The most problematic part is a case where an application would like to reconfigure the input devices after they have been initialized. As this is a rare case we have not created very elaborate system for these cases. In these cases the application can how-ever stop the desired device, reconfigure it and restart the device.

7 Data Processor Layer

Typically applications cannot use the input data directly. Instead the input data needs to be refined to be useful. For this purpose FLUID has a data processor layer. The objects in the data processing layer transform the data into a form that is more usable for the application.

It is possible for the application to transfer parts of the application logic to the processor objects. Although we make clear distinction between input data processing and application it should be noted that these are not at all independent components. This separation is only intended to serve as borderline between reusable software components and application-specific code. A developer can freely use minimal data processor layer and keep the application monolithic. An extreme alternative is to put as many application components as possible to the data processor layer.

One reason why one might put application logic into the data processing layer is that it can be used as an abstraction layer between the input devices and and the application. For example an application might be able to operate with mouse or camera input. If the camera- and mouse-specific parts of the application can be isolated to the processor layer, then the application logic does not need to know how the input data was collected.

Another element further confuses the separation of input processing and application: Situation-specific tuning of the data processors. This means that the behaviour of the data processors may need to be adjusted to match the current program state. For example we might need to instruct some gesture detector that some of the potential gestures are not accepted when the application is in some state. This has already been the case in our previous research where the gesture detectors and semantic analyzers formed feedback-cycles[16]. With FLUID this is possible, but one must build the data processors to offer the necessary application-specific functionality.

7.1 Processor Collections

The data processing layer is a collection of data-processing objects. Each object performs some operation on either the input data or data coming from other processor objects. The processor objects fall into roughly two categories: gesture detectors and data converters. The gesture detectors serve the application by detecting gestures (or events) as they take place. Data converters do some operations on the input data, but do not try to detect explicit gestures. In some cases high-bandwidth signals can be compressed into simple events. In some others the processor objects simply change the representations of the data – for example from 3D location data to 3D acceleration data. Figure 3 shows how data might flow from the input devices to the application.

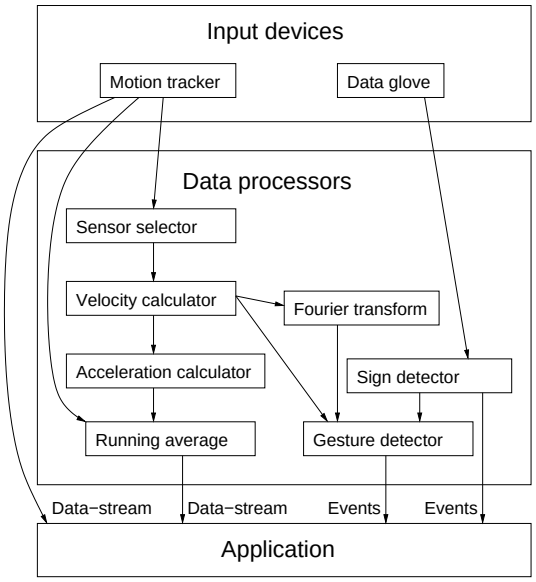

Fig. 3. An example of how data can flow from input devices to the application.

In the data processing layer we have adopted a design principle that algorithms are broken down into parts when possible. The advantages of this approach are that processor objects can rely on other objects to perform some routine calculations. This enables different processor algorithms to share parts, resulting in less coding work due to code re-use.

This approach can also result in better computing performance. The reason for this is that if two or more algorithms use the same feature that is extracted from input data, this feature can be calculated only once and the results are shared by all the algorithms utilizing it. For example in figure 3 there are several gesture detectors that need velocity information. With this approach the velocity can be computed only once and the data is then available to all interested objects. While this design promotes modularity it does not rule out big monolithic data processors.

Originally the data processing layer was split into four parts. The purpose of this split was to separate different parts of the signal processing to different layers with one layer following another. Later we realized that the layering was artificial and any layering would be totally application-specific. In practice one can design different layering structures with each layer performing some small operation on the data. For example Bimber's architecture contains eight layers[14]. Of these eight six correspond to the single data processing layer in FLUID. Such layering can be useful for separating tasks, but it also requires application programmers to be aware of all the layers and their interaction. Once we take into account the fact that modules that operate in lower level may need to be controlled by modules from higher level we end up with feedback-cycles that essentially break down the layering approach. For these reasons we selected a single monolithic data processing layer. The users can create arbitrary data processing

networks within this layer. Since the FLUID data passing mechanisms are very flexible it is possible to create any kind of layering within data processing layer.

7.2 Data Flow

FLUID has a data-flow architecture that allows arbitrary data to be passed from one object to another. There are two ways to pass data: data streaming via IO-nodes and event passing.

Each processor object can have an arbitrary number of named output nodes. When other nodes need to access the data they typically need to perform two tasks. First they get access to the node that contains the data. At this phase they must also check that the node is of correct type. Typically a processor object stores pointers to its input data nodes and only performs this operation once during its life-time. Once the processor has access to the IO-node they can read data from it. Since the IO-node is of known type the processor object can access its data directly, with minimal overhead. This data-flow architecture causes minimal run-time performance penalty. The user can introduce new IO-node types by inheriting the virtual base class and adding the data structures for the new type. In practice this scheme is similar to OpenTracker's data-flow architecture[12]. The primary differences are that in FLUID the users can create new processors and IO-node types and FLUID does not (yet) support XML-based processor-graph creation. Additionally the FLUID data-flow architecture is based on polling – data is not pushed from processor to another. In fact the OpenTracker framework could be implemented on top of FLUID's input-device and data-flow components.

While the data-flow architecture is good for dealing with fixed-rate signals it is not ideal for passing events that take place seldom. For these situations we have augmented the system with message-passing interface. Each processor can send events to other processors. Events are delivered with push-approach. FLUID has definitions for the most common event types (integer- and floating point numbers and character strings) and users can introduce new event types when needed.

7.3 Processor Creation

The processor objects are recursively created as needed. For example the application might request for an object that detects hand claps. In this case the application passes a request-object to the processor collection[15](page 233). This request object first checks if the requested object type (with matching parameters etc.) already exists. If the object does not exist then the request-object tries to create one. This may lead to new requests since the gesture detector would need to know the acceleration of the hands. This causes a request for an acceleration object. As the acceleration calculator is created it needs a velocity calculator. The velocity calculator in turn needs a motion tracker which it requests from the input layer.

If the gesture detector programmer had been very clever there might even be a possibility that if there is no way to detect hand claps with motion trackers (they might be missing) then the request-object could try to create a clap detector that relies on microphone input or accelerometer input. At any case the request tries to create the processor

object and all necessary objects recursively. If the process is successful then it returns an object that outputs events as user claps hands.

This infrastructure enables the application to ask for a particular data processor without knowing what is the exact method by which the data processing detector works (or even the needed input devices). This system also enables different algorithms to share common parts without knowing much else than the output node types of the relevant objects. While this infrastructure provides a way to share algorithms and algorithm parts between applications it is heavy if one only needs to create specific processor object. To accommodate these cases there is a possibility to directly add a data processing detector to the collection, bypassing the request approach.

The system includes dependency management that tries to optimize the call-order of the processors. Thus the system first calls the nodes that are closest to the input and once they are updated it goes on to the higher-level nodes.

There can be multiple data processor collections in one application. This makes it easy for an application to shut down one processor section if it is not needed. For example when application changes its state and user interaction type it might switch over to a totally different set of data processors.

7.4 Example

An example of how the nodes behave is in figure 3. The left side of the figure shows how a stream of data is transformed as it passes thru the system. The motion tracker object has an output node called “location”. This node contains a ring-buffer of motion samples. The sensor selector reads data from the motion tracker and stores data from one sensor to two output nodes (velocity and rotation). The velocity calculator reads data from this node, calculates the velocity of the tracker sensor and places the data to its own output node. The acceleration calculator is in fact identical to the velocity calculator. The only difference is that it takes its input from the output of velocity calculator. The running average calculator in turn uses the acceleration data it obtains from acceleration calculator and calculates the average acceleration over a period of time. The application in turn can use this as parameter according to the application logic.

In the right hand side there is a sign detector that relies detects different finger signs. As the sign changes the information is passed to the application in the form of an event.

In the center there is a network that combines data from two sources. The Fourier transform calculator performs Fourier separately on each three dimensions of the velocity vector. The gesture detector then uses information coming from the Fourier transformation, sign detector and velocity calculation to trigger an event as the user performs some gesture.

8 Implementation

Above we have outlined the FLUID architecture. This architecture could be implemented in nearly any language or platform. In this section we outline our proof-of-concept implementation. By proof-of-concept implementation we mean that the current

FLUID toolkit does not have support for a wide range of input devices, device types or data processors. It has been used in pilot applications to test the architecture in practice.

We have implemented FLUID with C++. This choice was made since we already use C++ and it offers high performance, reasonable portability and support for object-oriented programming. At the moment the FLUID core libraries work on IRIX and Linux operating systems. The drivers in turn are rather platform-specific, so some of them work on IRIX, some on Linux and some on both. The FLUID library is very compact and it can be easily ported to any platform that offers support for ANSI C&C++ and POSIX threads. FLUID does not have any other external dependencies so porting it to different platforms should be fairly easy.

Any application can use the components of FLUID – it does not force the application into certain framework (internal main-loops etc.). As such it can be added to nearly any software with ease.

The input layer and processor layer are in separate libraries. It is therefore possible to use only the input layer in projects where the data processors are not needed.

FLUID library is internally multithreaded, but it hides the complexity of multithreaded programming from application developer. However, the system is not thread safe in a sense that if the application developer utilizes the FLUID input API from multiple threads the results are be undefined. It should be noted that this is a limitation of the current implementation and as there are only a couple of places where a conflict might occur, it should not require much effort to make the system fully thread safe.

The current version has an API and input drivers for mice, motion trackers, data gloves and speech recognition. The speech recognition system is based on the commercial software package ViaVoice by IBM[17]. The speech recognition API is independent of the ViaVoice package however.

There is also possibility to write data onto the disk and read it later (as with VRPN). This enables us to simulate and debug application behaviour without actually using the physical devices. This cuts down costs as one can test VR applications with realistic input data without using the expensive VR facilities. It also helps in debugging since we can use identical input data sequences between runs.

All of the device drivers have option for network-transparent operation – the physical device and the application can be in different computers. The device data is transmitted over a TCP/IP connection from the physical device to the application. This network operation is encapsulated within the FLUID device drivers so that application developers do not need to know about such details. This feature was necessary since some of the devices we use can only be attached to one kind of computer (Linux PC) while the application runs in other kind of machine (IRIX workstation). While network transparency has not been a primary goal for us it is a positive side-effect of our implementation strategy. This only applies to the input drivers, we have not tried to make FLUID processor collection a distributed system like QuickSet[3]. A programmer creating a new data processor can of course distribute the processors to multiple CPUs with multithreading or to multiple computers via network interface.

The FLUID device drivers are implemented as plugins that are loaded as the application starts. Thus there is no need to modify the core libraries to add new device

drivers. This also guarantees that the device APIs do not depend on any particular device manufacturer's proprietary APIs or protocols.

An important detail we only realized when implementing the input layer is that the input threads must have a possibility run often enough. The problem is that a multitasking operating system may well give plenty of CPU-time to the main thread of the application, but fail to give enough CPU-time to the input threads. As a result the input data buffers do not get new data even tough there would be new data available. This problem occurs when the main thread of the application is very busy (many multimedia application – games and VR systems – do just this). The way to overcome this problem is by increasing the priorities of the input threads so that they can run as fast as they need to run. This also reduces the latency caused by threading.

We have also built a small library of data processors. This library offers a few gesture detectors (simple hand clap- and finger sign detectors) and some feature extractors (velocity and acceleration calculators and finger flexure calculator).

8.1 Performance Issues

The FLUID architecture has been designed with performance issues in mind. Depending on the application there are two alternate bottle-necks.

The first cause for overhead is the input layer. The threading and buffering of input data cause extra overhead for the application. In normal circumstances this is hardly a problem. As a benchmark we created a minimal application that reads data from motion tracker, mouse and two data gloves – all at 33 Hertz sampling rate. This application consumes less than 3 percent of the available CPU time on low-end hardware (SGI O2 with 195MHz R10k processor). This reflects the fact the the input driver threads do not have much to do. Most of the time they wait for new data to come. This figure does not tell the actual overhead of the input layer, but even if the load of 3 percent was caused solely by FLUID overhead this is seldom harmful for the application. A situation where such overhead might become significant is in the realm of ubiquitous computing. In these cases the host computer may have the computing power of an old 386 or 486 -processor. In any case the computer running FLUID must be powerful enough to run a multitasking operating system. Obviously many embedded systems do not fulfill this criterion.

The other potential bottle-neck is the data-processing layer. Even tough the data processors may do heavy computation this layer should not cause significant overhead. The data is passed from one processor object to another directly without any generalization mechanisms. In theory the only source of overhead compared to a dedicated solution should be the single virtual function call per data processor.

8.2 Latency

Some multimodal applications require minimal latency between input data measurement and the moment when the data is used. For example in immersive virtual reality systems it is necessary to update the projection with data that is as new as possible. Thus the toolkit should not induce extra latency in the data transfer path.

In the FLUID architecture the device driver threads are run at high "real-time" priority that guarantees that the drivers threads can always operate when new data becomes available from the physical data source (device/network). As a result the device threads can offer the data immediately to the application thread. In practice this approach minimizes the latency caused by FLUID to the short time that the operating systems spends when switching between threads.

9 Fluid And Other Toolkits

It is sometimes the case that the application is using another toolkit that depends on user input. This might impose a problem, since it is rare that input device APIs have support for accessing the input from multiple toolkits at the time. Typical case like this arises in VR systems since virtual reality toolkits must utilize some input devices to be successful. The most common reason for this is the projection calculations that are done to compensate user movements. As a consequence many toolkits (VR Juggler, DIVE) have integrated motion tracker support. While this makes life easy for the toolkit it poses a problem for a programmer who wishes to use FLUID – the tracker device is managed by the other toolkit with it's internal API. This makes it impossible for FLUID to connect to the device.

We have solved this problem with VR Juggler by creating new VR Juggler device drivers that actually run on top of FLUID input layer. In this way VR Juggler works perfectly while the actual data is coming from FLUID. One might also do the reverse – use VR Juggler native device drivers and transmit data from those over to FLUID. This latter alternative would have the problem that VR Juggler does not maintain history of samples in the low-level drivers. As a result the FLUID drivers would have to re-sample the VR Juggler input devices with some frequency hoping that no samples would be lost. This would certainly lead to loss of data quality.

With our current approach one has the benefits of both systems: VR Juggler's integrated projection management and FLUID's high quality input data and data processing libraries.

10 Building Applications with FLUID

The FLUID libraries has been designed to fit easily into many kinds of applications. To outline how one can use FLUID in a new application we give an example of how one can use FLUID in a multimodal application. Although this example is expressed in general terms it matches the AnimaLand application that we have build (explained in section 11).

A typical multimodal application collects input data from several devices and delivers output to the user via multiple media. The application has a main loop that is synchronized to one of the devices – for example the application may draw a new graphics frame each time the main loop is executed (common approach in games). In each loop iteration the application collects input data from the devices and uses application logic to control the output devices (graphics, sound, etc.). The loop iteration rate can vary as the application runs depending on how heavily the computer is loaded.

The threaded and buffered input device layer of FLUID fits this scheme well – the application can run at nearly any iteration rate and the input devices will not lose data. The application can use the gesture detector layer to extract information from the input data. The input data is turned into fixed-rate data streams or events that the application receives via call-backs functions (as in many GUI toolkits). The application builds one or more processor collections to match its needs. While one collection might fit to the needs of a particular application there are cases where the ability to remove parts of the processing is necessary. For example the application might require special processing when it enters a given state. In these situations the application can build new gesture detector collections on demand and erase them as they are no longer needed. Alternatively the application can create the detectors in the beginning and later on simply use the relevant processor collections.

There can be special output and input devices that need to be controlled separately from the application main loop. Often the reason for this separation is that are strict latency limits that some input/output -operations must meet (force-feedback and audio systems being common examples). The processing for these special devices often happens in a separate high-priority thread. If the application needs such high-priority threads to process data at rate that differs fro the main loop rate these threads must have processor collections of their own. All the threads can how-ever access the same input devices as long as the application makes sure that the different application threads do not update the input devices while another thread is reading data from them.

11 Examples

We have used FLUID in three cases. These cases illustrate how building multimodal applications is easier with FLUID and how it can be used as a small component to introduce novel input devices to any application. The first two applications also demonstrate user interaction that is very different from the traditional computer usage. Such new interaction styles could potentially be used to enable more universal access to information technology and information networks. Compared to our previous experience with handling novel input devices [1][2] these new applications were easier to create.

In the AnimaLand project we built an application where user can control computer animated particle system in real-time[18]. The control mechanisms are gesture-based. For interaction we selected gestures that are easy to detect – hand claps and finger gestures. We also included some generic parameters to be used as control – average velocity of and the average joint angle (“fistiness”) of user’s left hand. Figure 4 shows the application in use. The processor layer with its gesture detectors simplified the application development significantly. Instead of building the data-processing blocks inside the application we coded the gesture detectors into the FLUID library. As a result the gesture detectors are usable in other applications as well. The application architecture became more modular since we could separate input processing from the animation engine. We could also take advantage of FLUID’s ability to store the input data to a file. We used this feature for debugging and also as a way to store the control information for post-processing the animations later on.

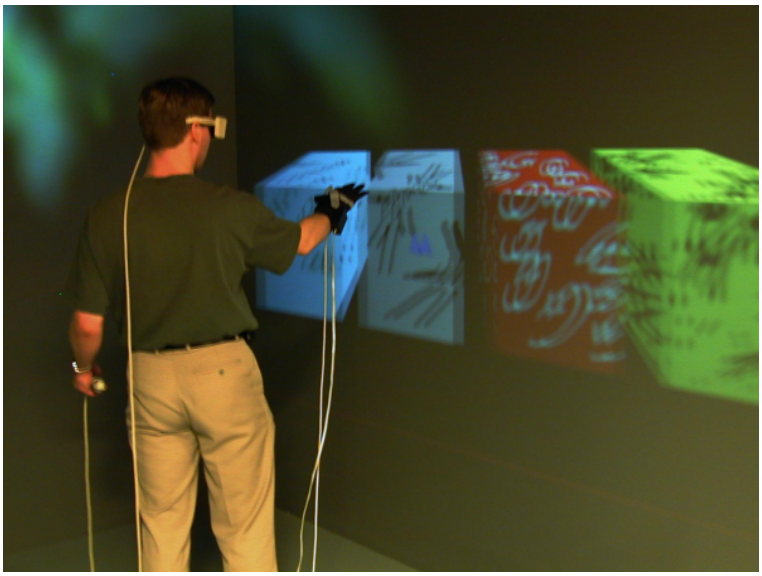

Fig. 4. The user is working in the AnimaLand environment.

In another project undergraduate students of our university created a virtual reality sculpting application “Antrum” (figure 5). The user can “draw” lines and surfaces in 3D space by moving hands. In this case the ability to collect data at constant sampling rate is very important. The application must collect motion data at constant rate even if the rendering process slows down significantly. In practice artists always create models that eventually choke the computer. With FLUID the application can refresh the graphics at any rate (be it 6 or 60 Hz) and we can still guarantee that all details of the user’s motion will be stored at the specified sampling rate (be it 33 or 133 Hz). If one only got the motion samples as the application main loop executes once then we would lose data as the graphics frame rate goes down. Although Antrum does not use the FLUID processor layer the input layer is used since it offers a simple API to access the devices and handle the buffering issues.

Our third project was a desktop-application that needed to get input data from cheap a motion tracker. The application was a sound-processing engine Mustajuuri that is running the signal processing at very low latency – less than 10 milliseconds[19]. The motion tracker access was a cause for random latency – it took some time to read each new sample from the device. To move this cause of latency to another thread we used the FLUID input layer. As a result the sound-processing thread can execute at the required rate and the data from the motion tracker is made available to it when the data is read from the device. In this case FLUID was only a small component within a large pre-existing application. Since FLUID does not enforce any particular application framework it was easily integrated in this case.

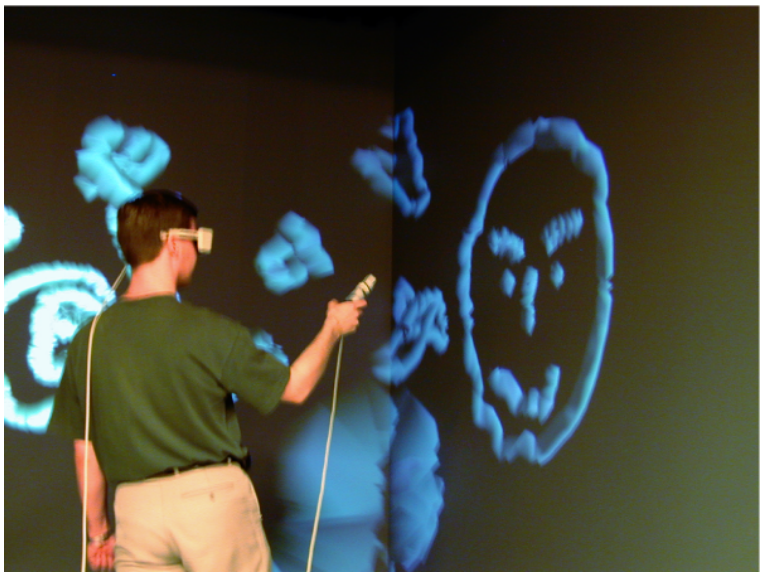

Fig. 5. Sculpting in virtual reality.

12 Conclusions And Future Work

We have presented an architecture for user input data management and outlined our implementation of the architecture.

This architecture incorporates support for arbitrary input devices and arbitrary input processing networks. It is intended to make programming of multimodal applications easier.

We have created a toolkit to handle user input. The toolkit is fit for different applications, but it has been tested and proved only in VR applications so far. We have found that FLUID makes application development easier. It offers a clear distinction between input data, input processing and application and offers a useful set of data processors.

The FLUID architecture has proven to be solid and thus there is no need for major adjustments. In future we expect that most of the work will be in adding new device drivers and device types (audio, video and MIDI input for example). We are also planning to test FLUID in a multimodal desktop application that relies on video and audio input.

References

1. Ilmonen, T., Jalkanen, J.: Accelerometer-based motion tracking for orchestra conductor following. In: Proceedings of the 6th Eurographics Workshop on Virtual Environments. (2000)
2. Ilmonen, T., Takala, T.: Conductor following with artificial neural networks. In: Proceedings of the International Computer Music Conference. (1999) 367–370 URL: http://www.tml.hut.fi/Research/DIVA/old/publications/1999/ilmonen_icmc99.ps.gz.

3. Cohen, P.R., McGee, D.R., Oviatt, S.L., Wu, L., Clow, J., King, R., Julier, S., Rosenblum, L.: Multimodal interactions for 2d and 3d environments. IEEE Computer Graphics and Applications (1999) 10–13
4. Schoemaker, L., Nijtmans, J., Camurri, A., Lavagetto, F., Morasso, P., ıt, C.B., Guiard-Marigny, T., Goff, B.L., Robert-Ribes, J., Adjoudani, A., Defée, I., Münch, S., Hartung, K., Blauert, J.: A taxonomy of multimodal interaction in the human information processing system. Technical report, ESPRIT BRA, No. 8579 (1995)
5. Proakis, J.G., Manolakis, D.G.: Digital Signal Processing. Macmillan Publishing Company, New York (1992)
6. Laurence, N., Joëlle, C.: A design space for multimodal systems: Concurrent processing and data fusion. In: The proceedings of InterCHI '93, joint conference of ACM SIG-CHI and INTERACT. (1993) 172–178
7. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: Aiding the development of context-enabled applications. In: Proceeding of the CHI 99 Conference on Human factors in Computing Systems, Pittsburgh, Pennsylvania, United States, ACM Press New York, NY, USA (1999) 434–441
8. Mankoff, J., Hudson, S.E., Abowd, G.D.: Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In: Proceedings of the CHI 2000 conference on Human factors in computing systems, The Hague, The Netherlands, ACM Press New York, NY, USA (2000) 368–375
9. Landay, J., Myers, B.: Sketching interfaces: Toward more human interface design. Computer **34** (2001) 56–64
10. Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., Cruz-Neira, C.: Vr juggler: A virtual platform for virtual reality application development. In: The Proceedings of IEEE VR Conference 2001. (2001)
11. CAVELib: Cavelib user's manual. WWW-page (Cited 24.6.2001) http://www.vrco.com/CAVE_USER/.
12. Reitmayr, G., Schmalstieg, D.: An open sotfware architecture for virtual reality interaction. In: Proceedings of the ACM symposium on Virtual reality software and technology, ACM Press New York, NY, USA (2001) 47–54
13. Taylor, R.M., Hudson, T.C., Seeger, A., Weber, H., Juliano, J., Helser, A.T.: Vrpn: a device-independent, network-transparent vr peripheral system. In: Proceedings of the ACM symposium on Virtual reality software and technology, ACM Press New York, NY, USA (2001) 55–61
14. Bimber, O., Encarnação, L.M., Stork, A.: A multi-layered architecture for sketch-based interaction within virtual environments. Computers & Graphics **24** (2000) 851–867
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: elements of reusable software. Addison Wesley Longman Inc. (1994)
16. Ilmonen, T.: Tracking conductor of an orchestra using artificial neural networks. Master's thesis, Helsinki University of Technology, Telecommunications Software and Multimedia Laboratory (1999)
17. IBM: Ibm voice systems. WWW-page (Cited 24.6.2002) http://www-3.ibm.com/software/speech/.
18. Ilmonen, T.: Immersive 3d user interface for computer animation control. In: The Proceedings of the International Conference on Computer Vision and Graphics 2002, Zakopane, Poland (2002 (to be published))
19. Ilmonen, T.: Mustajuuri - an application and toolkit for interactive audio processing. In: Proceedings of the 7th International Conference on Auditory Displays. (2001) 284–285