First publ. in: Journal on Multimodal User Interfaces 3 (2010), 3, pp. 197-213

DOI: 10.1007/s12193-010-0044-2

Interactive design of multimodal user interfaces

Reducing technical and visual complexity

Werner A. König · Roman Rädle · Harald Reiterer

Abstract In contrast to the pioneers of multimodal interaction, e.g. Richard Bolt in the late seventies, today's researchers can benefit from various existing hardware devices and software toolkits. Although these development tools are available, using them is still a great challenge, particularly in terms of their usability and their appropriateness to the actual design and research process. We present a three-part approach to supporting interaction designers and researchers in designing, developing, and evaluating novel interaction modalities including multimodal interfaces. First, we present a software architecture that enables the unification of a great variety of very heterogeneous device drivers and special-purpose toolkits in a common interaction library named "Squidy". Second, we introduce a visual design environment that minimizes the threshold for its usage (ease-of-use) but scales well with increasing complexity (ceiling) by combining the concepts of semantic zooming with visual dataflow programming. Third, we not only support the interactive design and rapid prototyping of multimodal interfaces but also provide advanced development and debugging techniques to improve technical and conceptual solutions. In addition, we offer a test platform for controlled comparative evaluation studies as well as standard logging and analysis techniques for informing the subsequent design iteration. Squidy therefore supports the entire development lifecycle of multimodal interaction design, in both industry and research.

Keywords Multimodal user interfaces · Post-WIMP user interface · Natural interaction · Design environment · Zoomable user interface · Semantic zooming · Multimodal interaction · Squidy

W.A. König (✉) · R. Rädle · H. Reiterer
Human-Computer Interaction Group, University of Konstanz,
Box D-73, Constance, Germany
e-mail: Werner.Koenig@uni-konstanz.de
url: http://hci.uni-konstanz.de

R. Rädle
e-mail: Roman.Raedle@uni-konstanz.de

H. Reiterer
e-mail: Harald.Reiterer@uni-konstanz.de

1 Introduction

With recent advances in computer vision, signal processing, and sensor technology today's researchers and interaction designers have great opportunities to go far beyond the traditional user interface concepts and input devices. More natural and expressive interaction techniques, such as tangible user interfaces, interactive surfaces, digital augmented pens, speech input, and gestural interaction are available and technologically ready to be incorporated into the multimodal interface of the future (see some examples in Fig. 1). However, the actual utilization of these techniques for the design and development of multimodal interfaces entails various critical challenges that interaction designers and researchers have to face.

In contrast to the design of traditional graphical user interfaces, the development of multimodal interfaces involves both software and hardware components [12]. However, conventional development environments (e.g. MS Visual Studio/.Net, Adobe Flash, Eclipse IDE) fall short of supporting uncommon interaction modalities and appropriate data processing (e.g. computer vision), not to mention the handling of multipoint and multi-user applications (e.g. for multi-touch interaction). As a consequence a broad variety of very heterogeneous and specialized toolkits and frameworks have evolved over the last few years such as

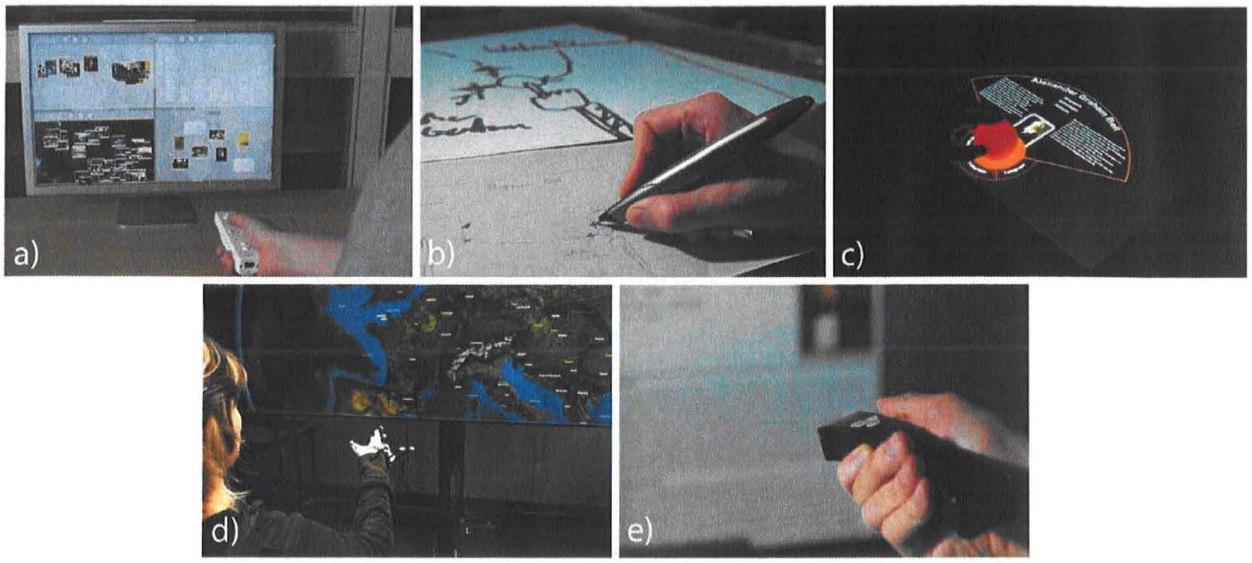

Fig. 1 Diverse input devices for single-modality or multimodal interfaces: (**a**) Physical game controller offer absolute pointing, motion sensing and gesture-recognition to the end-user. (**b**) Digital pens build upon users' pre-existing knowledge and thus offer a very natural mode of interaction e.g. for digital sketching and prototyping. (**c**) Multi-touch surfaces augmented with physical tokens reduce the gap between real-world and digital-world interaction. (**d**) Finger gestures provide a very expressive and direct mode of interaction. (**e**) Well-known devices such as an omnipresent laser pointer provide flexible input from any distance

Table 1 Interaction designers have to cope with very different environments for the same interaction modality, touch input

Hardware platform	Microsoft Surface	Custom-build table	Apple iPhone	HTC Hero
Form factor	Table	Table	Mobile	Mobile
Operating system	Microsoft Windows	Linux/Windows	Mac OS X	Android OS
Programming language	C#	C++	Objective-C	Java
Software framework	Surface SDK	Touchlib	iPhone SDK	Android SDK

Apple iPhone SDK[1], Microsoft Surface SDK[2], GlovePIE[3], Processing[4], NUI Group Touchlib[5]. They provide support for specific interaction modalities, but are mostly restricted to a dedicated hardware environment and entail further requirements and dependencies. When using touch as input for instance, the interaction designer has to cope with different hardware platforms, operating systems, programming languages, and software frameworks (see Table 1). When developing single-modality interfaces, this diversity can be bypassed—at least in the short-run—by focusing on just one specific device. But the combination of multiple devices, e.g. for multimodal interaction involves further platforms, devices, and frameworks, resulting in an unmanageable technical and mental complexity.

There are development environments that support at least some of the more uncommon input devices and modalities (e.g. physical turntables, mixing desks, multi-touch surfaces and simple vision tracking). Two examples are Max/MSP[6] and vvvv[7]. Both are graphical development environments for music and video synthesis and are widely used by artists to implement interactive installations. Their popularity in the design and art community arises in particular from their graphical user interface concepts. Both are based on the concept of visual dataflow programming and utilize a cable-patching metaphor to lower the implementation threshold [24] for interactive prototyping. Users arrange desired components spatially and route the dataflow between the components by visually connecting pins instead

[1] Apple iPhone SDK, http://developer.apple.com/iphone/.

[2] Microsoft Surface SDK, http://www.microsoft.com/surface/.

[3] GlovePIE, http://carl.kenner.googlepages.com/glovepie/.

[4] Processing, http://processing.org/.

[5] NUIGroup Touchlib, http://nuigroup.com/touchlib/.

[6] Max/MSP, http://cycling74.com/products/maxmspjitter/.

[7] vvvv, http://vvvv.org/.

of textual programming. However, the visual representation of each primitive variable, parameter, connection, and low-level instruction (e.g. matrix multiplication) leads to complex and scattered user interfaces, even for small projects. vvvv offers the possibility of encapsulating consecutive instructions in so-called "subpatches". This approach helps to reduce the size of the visual dataflow graph, but the hierarchical organization introduces additional complexity. In contrast to the visual encapsulation in vvvv, the "external" mechanism of Max/MSP supports the visual and technical encapsulation of certain functionality in an external object as a "black-box". This mechanism offers high flexibility and abstraction but requires low level programming in C. This results in a higher threshold and lower interactivity of the design and development process, since changes have to be textually written and compiled in an external development environment before the external object can be used in Max/MSP.

Basically, Max/MSP and vvvv show interesting user interface concepts but they are focused on real-time audio composing and 3D rendering and were not designed to support the development of multimodal interfaces in general. For that, interaction designers require not only a set of ready-to-use interaction techniques and input devices but also the possibility to physically develop and integrate new interaction modalities and hardware devices. Hardware toolkits such as Phidgets [11], Smart-Its [10] or iStuff [1] offer a set of compatible microcontrollers, sensor devices and software frameworks enabling rapid prototyping of physical input and output devices. However, the technical complexity of the software frameworks requires advanced programming and signal processing knowledge, in particular when multiple devices are used in parallel. iStuff mobile [2] combines the hardware toolkit iStuff with a visual programming environment based on Apple's Quartz Composer.[8] This was originally designed to support the visual development of interactive multimedia and 3D rendering. It shares the cable-patching metaphor with the already discussed development environments vvvv and Max/MSP. This combination of hardware toolkit and visual development environment facilitates fast iterations and rapid prototyping on multiple levels. However, it is restricted to the domain of mobile phone interaction and limited in its functionality and the type of input (e.g. no support for computer vision).

All of the aforementioned development environments and toolkits support diverse devices and modalities but they are not especially designed to support the design of multimodal interfaces. Here, multiple inputs have to be synchronized (e.g. hand-gesture and speech), processed and composed to a higher level command (e.g. moving an object). There are few frameworks that address these requirements. ICARE [5] is a conceptual component model and a software toolkit for the rapid development of multimodal interfaces. It provides two types of software components: the elementary components, consisting of *Device and Interaction Language* components used to develop a specific modality, and the *Composition* components that combine the diverse modalities. It was used for different use cases (e.g. design of a multimodal flight cockpit) but it became apparent that only a limited set of the defined components were really generic [27] and the toolkit was not easily extensible [22].

Based on the experiences gained with ICARE, the open source framework "OpenInterface" was developed by the OpenInterface Project[9] that is dedicated to multimodal interaction. The OpenInterface framework is composed of the OpenInterface Kernel, a component-based runtime platform, and the OpenInterface Interaction Development Environment (OIDE), a graphical development environment for the design of multimodal interfaces [27]. In order to integrate an existing input device as component into the OpenInterface Kernel the component interface has to be specified in an dedicated XML-based CIDL description language (Component Interface Description Language). This specification can be semi-automatically generated from the source code of the component by the OpenInterface platform. It also generates C++ code to encapsulate the external binary into a well defined programming interface [3]. Due to this explicit description of the interface the encapsulated component can be used in the graphical development environment OIDE. This uses a cable-patching metaphor similar to Max/MSP, vvvv, and Quartz Composer in order to define the dataflow by combining the selected components visually. Lawson et al. [22] identified diverse shortcomings of the OpenInterface OIDE and the introduced application design process. A major issue is the limited focus and inflexible design of the components. The developers rather focus on the design of their individual component than on the application as a whole. This leads to an inflexible design of the components and the application in general that hinders the reuse, extension and exchange of components as well as the entire application. This inflexibility also restricts interaction designers in exploring diverse alternatives, which then impedes rapid prototyping and limits epistemic production [18] of concrete prototypes. In order to address the identified issues Lawson et al. introduced an all-in-one prototyping workbench for multimodal application development [22]. It is a combination of the OpenInterface Kernel with an Eclipse Plugin as graphical editor that is named SKEMMI. The editor is also based on the cable-patching metaphor, but provides three levels of detail with respect to the displayed information. The low-detail "workflow" level reduces information and facilitates the initial sketching of

[8] Apple Quartz Composer, http://developer.apple.com/graphicsimaging/quartzcomposer/.

[9] OpenInterface Project, http://www.oi-project.org/.

the desired interaction techniques. In the "dataflow" level where all details for the routing of the dataflow are represented, the user selects, arranges and logically links the components without the need to route and connect every single pin. In a third-level, the "component" level, only a specific component with its input and output pins is visualized and the user is able to tailor the component's interface (e.g. changing the port attributes and parameters). SKEMMI provides also an alternative source code editor view that allows for changes of the component or its interface programmatically. The three-layer approach helps to control the visual and functional complexity of the components, but there is no higher-level abstraction concept (e.g. hierarchical pipelines or semantic zooming). If the designed multimodal interface incorporates multiple devices and various signal processing components, the SKEMMI user interface gets increasingly crowded. The geometric zoom of the user interface is not a solution for the complexity issue since it just changes the size of the displayed information but not the information representation itself.

To sum up, there are only very few frameworks that support the design of multimodal interfaces. However, they either provide a limited range of interaction modalities or are hardly extensible regarding the platform, the components or the visual user interface. The OIDE or the SKEMMI graphical editors seem very promising, but the complexity issue is critical in real world projects. Moreover, all of the discussed development environments focus mainly on rapid prototyping and the early steps of iterative design. None of them provide tool-support for the empirical evaluation of the designed interfaces (e.g. ISO 9241-9 tapping tasks and suitable data-logging). All of the graphical development environments utilize the cable-patching metaphor in a similar way in order to connect input and output pins. However, the dataflow programming could be more powerful without losing its simplicity. Furthermore, they still require a deep understanding of the underlying technology on behalf of the designers, since they have to understand and route each primitive variable/data item even when using "black-box" modules.

In the following, we present our Squidy Interaction Library, which contributes on different levels:

- The software architecture: Squidy enables the unification of heterogeneous devices and components in a common library. The architecture is designed to provide great flexibility, simple extension, high independency and fast parallel processing.
- The visual development environment: Squidy enables the interactive design and configuration of multimodal interfaces for interaction designers and researchers. The user interface concept is designed to provide a low threshold (ease-of-learn) and high ceiling (high functionality) and scales well with increasing complexity.
- Tool-support for the entire development lifecycle: Besides the visual design and configuration for rapid prototyping, Squidy also provides advanced development and evaluation techniques for iterative design.

After giving a short conceptual overview in the next section, we will discuss the software architecture in Sect. 2.1 and afterwards describe the user interface concept in detail in Sect. 2.2. In Sect. 3 we will show the appropriateness of our solution to the actual design and research process in the context of a variety of real world projects.

2 Squidy interaction library

We introduce the Squidy Interaction Library, which unifies a great variety of device toolkits and frameworks in a common library and provides an integrated user interface for visual dataflow management as well as device and data-filter configuration. Squidy thereby hides the complexity of the technical implementation from the user by providing a simple visual language and a collection of ready-to-use devices, filters and interaction techniques. This facilitates rapid prototyping and fast iterations for the design and development. However, if more functionality and profound customizations are required, the visual user interface reveals more detailed information and advanced operations on demand by using the concept of semantic zooming. Thus, users are able to adjust the complexity of the visual user interface to their current needs and knowledge (ease of learning).

The basic concept (see Sect. 2.2 for a more detailed discussion) that enables the visual definition of the dataflow between the input and output is based on a pipe-and-filter concept (see Fig. 2). By using this concept Squidy provides a very simple, yet powerful visual language for designing the interaction logic. Users can select an input device of choice as source, e.g. a laser pointer, which is represented by an input node in the visual user interface. They connect it successively with filter nodes for data processing, such as compensation for hand tremor or gesture recognition and route the refined data to an output node as sink. Basically, the user defines the logic of an interaction technique by choosing the desired nodes from a collection (knowledge base) and connecting them in an appropriate order assisted by a heuristic-based node suggestion. The filter nodes are independent components that can transmit, change, or delete data objects, and also generate additional ones (e.g. if a gesture is recognized). The source and sink are specific drivers that handle the input/output operations and map the individual data format of the devices to the generalized data types defined in Squidy (see Fig. 4). The pipe-and-filter concept provides also very technical advantages, since the encapsulation of functionality in independent "black-boxes" ensures information hiding, modifiability and high reuse by

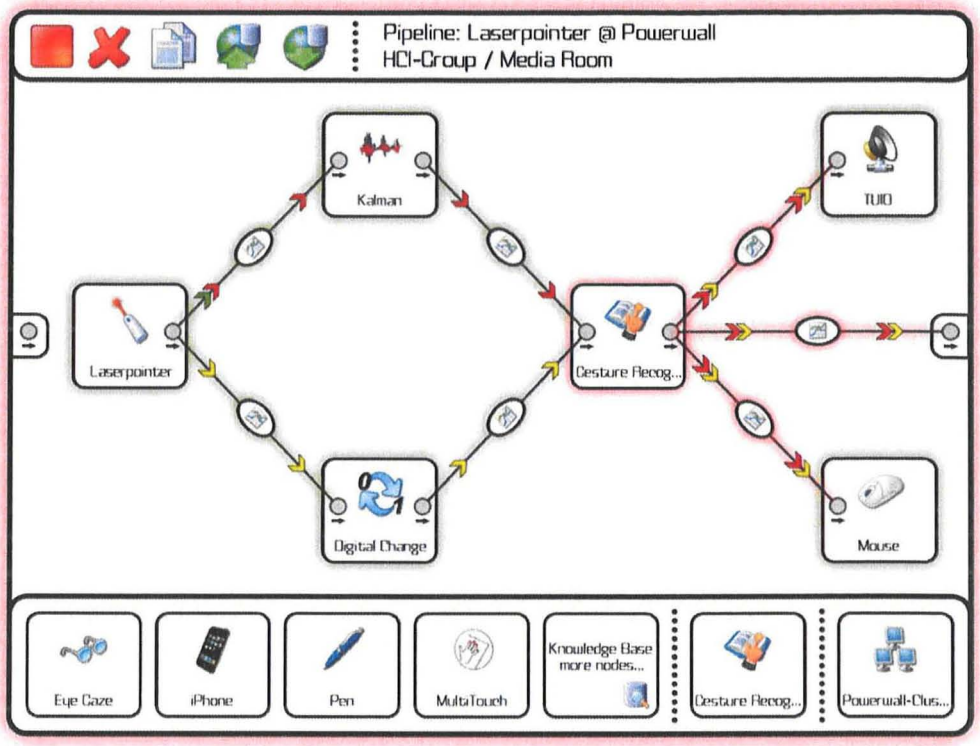

Fig. 2 View of a simple pipeline in Squidy. The pipeline receives position, button and inertial data from a laser pointer, applies a Kalman filter, a filter for change recognition and a filter for selection improvement and finally emulates a standard mouse for interacting with conventional applications. At the same time the data is sent via TUIO to listening applications. The pipeline-specific functions and breadcrumb navigation are positioned on *top*. The zoomable knowledge base, with a selection of recommended input devices, filters, and output devices, is located at the *bottom*

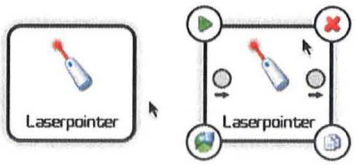

Fig. 3 Input node in Squidy representing an interactive laser pointer. In order to reduce visual complexity the node-specific functions (active/inactive, delete, duplicate, publish to knowledge base) and the unconnected in and out ports are only shown if the pointer is within the node

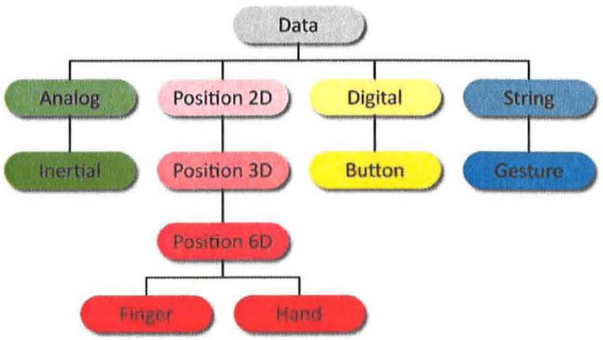

Fig. 4 Data type hierarchy in Squidy based on primitive virtual devices [30]. Any data processed in Squidy consists of single or combined instances of these basic data types

abstraction. The possibility for multiple input and output connections offers a high degree of flexibility and the potential for massive parallel execution of concurrent nodes. In our implementation each node generates its own thread and processes its data independently as soon as it arrives. This effectively reduces the processing delay that could have a negative effect on the interaction performance.

The sink can be any output technique such as a vibrating motor for tactile stimulation or LEDs for visual feedback. Squidy also provides a mouse emulator as an output node to offer the possibility of controlling standard WIMP-applications with unconventional input devices. Multipoint applications (e.g. for multi-touch surfaces or multi-user environments) and remote connections between multiple Squidy instances are supported by an input/output node that transmits the interaction data either as TUIO messages [17] or as basic OSC messages over the network. TUIO is a

widely used protocol for multipoint interaction based on the more general OpenSound Control protocol (OSC), which is a successor to the MIDI standard. By providing these standard interfaces for both input and output connections Squidy supports the majority of multi-touch applications that have recently become very popular in both research and industry. Above these basic network interfaces Squidy also supports and integrates more complex frameworks such as the Apple iPhone SDK, the Android SDK, the NUIGroup Touchlib, and the Microsoft Surface SDK. Users therefore benefit from the particular functionalities and specific hardware of all these techniques. Inside Squidy, however, they are also able to define, control, evaluate, and reuse interaction techniques independently from the hardware or the specific framework. This flexibility results from the architecture utilized and the generalized data types which will be explained in more detail in the following section.

2.1 Software architecture

There are several frameworks and toolkits that provide ready-to-use components for input devices and signal processing. Instead of connecting the components to pipelines programmatically, most of these frameworks and toolkits offer a basic language for controlling the dataflow visually (for example Max/MSP, vvvv, OIDE or SKEMMI). Such a visual programming language reduces the technical threshold and complexity and aids users with little or no programming experience. Also, the integration of new modalities requires a fine grasp of the underlying technology and thus is still a highly demanding task. Although, extending a framework with new components is only offered by a few of today's common frameworks such as ICARE [4] or the open source framework OpenInterface (www.oi-project.org). However, integrating new components into the frameworks requires either an additional programming effort or a dedicated definition of the interface by a specific mark-up language. Basically this means that a developer has to switch between different applications and programming languages while developing a new interaction technique, increasing the mental workload.

2.1.1 Generic data types

In order to unify very heterogeneous devices, toolkits and frameworks, we generalized the various kinds of input and output data to a hierarchy of well-defined generic data types (see Fig. 4) based on the primitive virtual devices introduced by Wallace [30] and adapted to the work of Buxton [6] and Card et al. [7]. Each generic data type consists of a type-specific aggregation of atomic data types such as numbers, strings or Boolean values bundled by their semantic dependency. Simply adding a single connection between two nodes in the visual user interface performs routing of dataflow based on these generic data types.

This is quite a different approach when compared to some of the aforementioned frameworks such as the ICARE [5] and vvvv. These frameworks use atomic data types defined in the particular programming language and assign them visually by connecting result values with function arguments in their specific user interfaces. In order to use the functionality of a module in these frameworks, the user has to route each of these low-level data types. Each x-, y-, and z-value of a three-dimensional data type has to be routed separately, for example. This is a procedure that needs additional effort and can be error-prone, in particular when designing complex interaction techniques. Furthermore, this approach requires detailed knowledge about the functionality of each node and its arguments. Routing low-level data types therefore puts high cognitive load on the user and leads to visually scattered user interfaces, particularly as the number of connected nodes increases.

Squidy, on the other hand, does not require the designer to visually define every value and programming step manually. The interaction data is grouped in semantically bundled generic data types as mentioned before. Squidy therefore offers the abstraction and simplicity of a higher-level dataflow management and reduces the complexity for the interaction designer without limiting the required functionality.

2.1.2 Squidy bridge

In order to achieve high extensibility and to simplify the integration of new devices and applications, we provide the Squidy Bridges as common interfaces that support widely used network protocols and also offer a specific native API if high-performance data transmission is needed. For the purpose of unifying data produced by different hardware devices or applications (especially relevant for incorporating multiple interaction modalities), the Squidy Bridges map the diverse data originating from heterogeneous sources into the generic data types. Thus, the internal data processing is harmonized and completely separated from the diversity of the external world. These bridges are able to handle data transformations in both directions (e.g. from Apple iPhone into the Squidy Core and from the Squidy Core to the application running on the panoramic display and vice versa in order to close the feedback loop e.g. activation of the vibrator on the iPhone as tactile feedback of the application's status (see Fig. 5)). The interaction library already comes with an OSC Bridge and a Native Interface Bridge that can be used out-of-the-box. The OSC Bridge offers the possibility of directly connecting the various available devices and toolkits using this communication protocol. Since OSC is based on standard network protocols such as UDP or TCP,

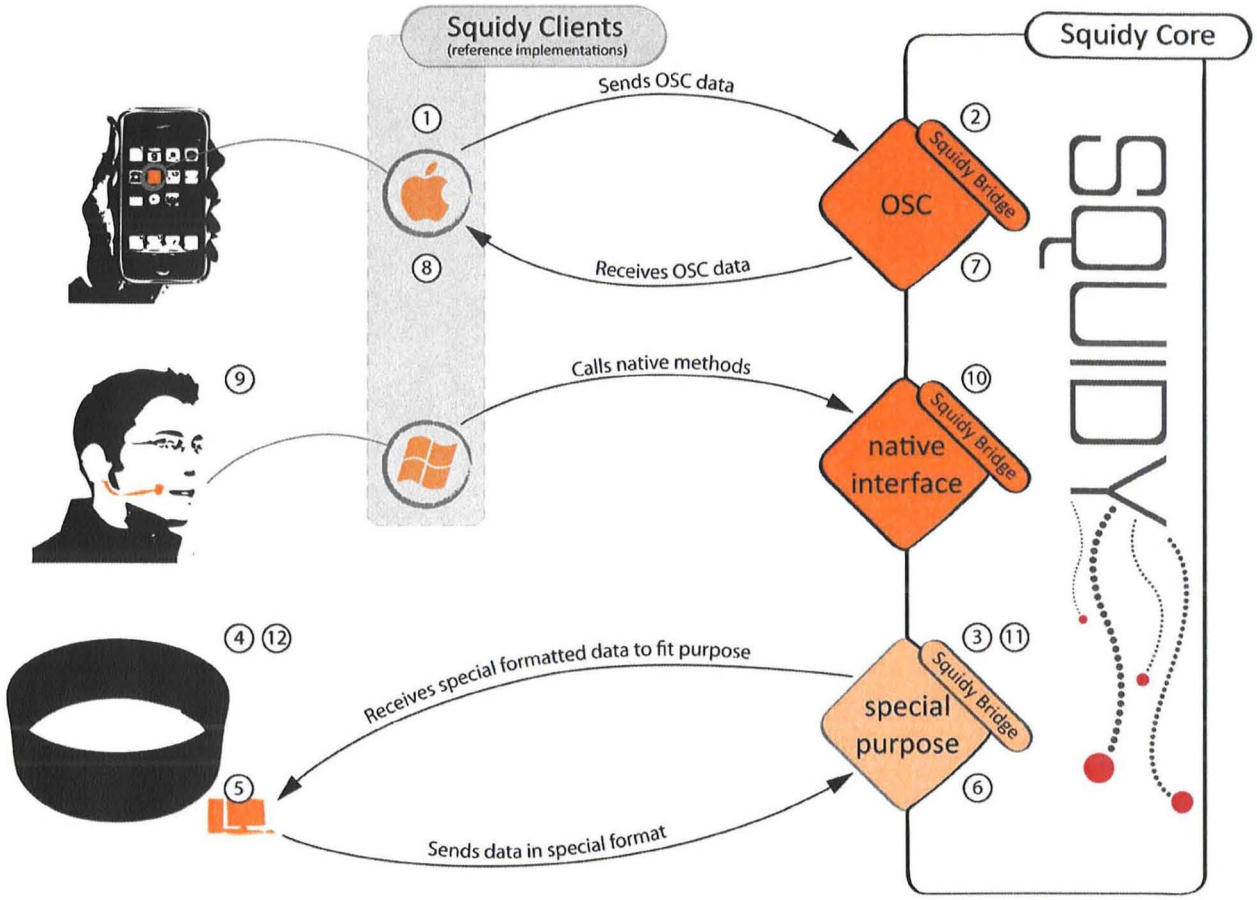

Fig. 5 This figure shows the usage scenario of an interactive and multimodal environment to control an application running on a 360° panorama screen by using touch gestures and speech. The user interacts with his fingers by touching the display of an Apple iPhone (1). All recognized touches will be sent from an iPhone Client application (OSC reference implementation running on the iPhone) to the OSC Bridge of Squidy (2). The Squidy Core will process the incoming data appropriately and sent it via the "special purpose bridge" (3) to the 360° application (4) to control a cursor object, which visually highlights the users current finger position. If the user has selected an interactive element with such a touching gesture the application (5) sends a tactile feedback back to its connected bridge (6). The tactile feedback coming from the application will be forwarded through the OSC Bridge (7) to the iPhone (8) where the vibration motor will be activated to inform the user that he is hovering above an interactive element. After the user has realized the tactile feedback and thus the interactive element (9), he will use a spoken command to invoke an action on the selected object. Therefore, the spoken command will be recognized by the operating system's speech recognition and then will be sent to the "native interface bridge" (10). The appropriate spoken command will have been processed by the Squidy Core (11) and transformed into an action, which will be sent to the application to trigger object activation/manipulation (12). This multimodal scenario can be implemented with Squidy using pluggable Squidy Bridges to receive data from different devices and a simple arrangement of nodes to process that incoming data

it is highly flexible and widely applicable, in particular for mobile or ubiquitous computing. An OSC message consists of several arguments such as the class name of the generic data type, a unique identifier and data-type-specific parameters. For instance, a message for a two-dimensional position that may be sent from an Apple iPhone would contain the Position2D data type as first argument, IMEI number as second argument, x- and y-value as third and fourth argument (Listing 1).

The flexibility gained from this network approach (e.g. hardware and software independence, high scalability by distributed computing (see Fig. 15)) entails a certain delay that can have a negative effect on user input performance [23]. Thus, for those purposes when performance is more important than flexibility, the Native Interface Bridge provides a straightforward Java and C/C++ API to map data from individual devices to the generic data types in Squidy programmatically. In contrast to the OSC Bridge, this technique increases throughput and reduces the delay to a minimum.

For devices that support neither the OSC protocol nor the Native Interface Bridge by default, Squidy provides client

Listing 1 OSC Message sent from an Apple iPhone contains four arguments (finger touch)

```
/**
* 1. generic data type
* 2. IMEI as identifier
* 3. x-position
* 4. y-position
*/
String: de.ukn.hci.squidy.core.data.
    Position2D
String: 49 015420 323751 8
double: 0.25
double: 0.17
```

Listing 2 Methods to insert new or changed data objects into the dataflow

```
/**
* Publishes 1...n data objects to enhance the
* dataflow semantics.
*/
public void publish(IData... data);

/**
* Publishes a data container that consists of
* an array of data objects and a timestamp on
* which the data container has been released.
*/
public void publish(IDataContainer
    dataContainer);
```

reference implementations (e.g. Squidy Client for iPhone OS[10] and for Android OS[11] that can be deployed on these devices, minimizing the effort and threshold of device integration. However, if the hardware is not able to communicate via existing bridges natively, or if deployment of proprietary software is not desired or is not possible due to hardware restrictions, then users can add further bridges to allow communication, for instance through special-purpose protocol bridges such as the Virtual-Reality Peripheral Network [29].

The support of multiple bridges as interfaces in combination with the device-independent generic data types enables a separation of the data sources and the signal processing in the Squidy Core. This offers a simple but flexible integration of new interaction techniques and modalities without touching existing core functionality. As with ICARE [4] or OpenInterface (www.oi-project.org), interaction techniques designed with the user interface are completely decoupled from the individual hardware or the connected applications. Replacing devices (e.g. switching from the Apple iPhone to the Microsoft Surface) therefore does not affect the applied interaction techniques (e.g. "selection by dwelling") or the concrete application also connected to a Squidy Bridge. The independent-bridge approach in combination with the generalization of data types enables the integration of very heterogeneous devices and toolkits in Squidy. Interaction techniques that have been defined once can be reused multiple times. Squidy thus reduces complexity by abstraction, offers high flexibility and enables rapid prototyping.

2.1.3 Squidy core

All data resulting from (multimodal) user interaction is bridged from devices to the Squidy Core. The core processes this data automatically and in parallel without any programming effort or further customizations. Users can define a filter chain (processing chain) using visual dataflow programming provided by the visual user interface of the Squidy Interaction Library. In order to process the interaction data, the Squidy Core provides a flexible API for manipulating (CRUD – Create/Read/Update/Delete) the dataflow. To insert new or changed data objects into the dataflow, the publish-method (Listing 2) of the API can be called at the desired place in the pipeline. For instance, a gesture recognizer that has detected a pre-defined gesture will publish a new gesture object into the dataflow. These methods accept $1 \ldots n$ instances of data objects or a data container that consists of an array of data objects as well as a release timestamp. The interface 'IData' ensures the compatibility of the published data objects with the generic data types defined in Squidy and specifies common methods and enumerations.

Furthermore, the Squidy Interaction Library comes with diverse off-the-shelf filters for signal processing, data fusion, filtering and synchronization that provide the essential functionalities for developing multimodal interfaces. Compared to OIDE [27] or SKEMMI [22], Squidy incorporates the facility to add new filters (including properties, algorithms, logic and descriptions) without the need for switching to a different development environment. Therefore, the source code is embedded and can be manipulated by users directly. Changes made to the source code will be compiled and integrated on-the-fly and the new or changed functionality is thus instantly available to users. Each implementation of a filter owns a data queue and a processing thread without any effort on the developer's part. The incoming data will be enqueued until the processing thread dequeues data to perform custom data processing automatically [5]. Thus, the interaction library runs in a multi-threaded environment that allows concurrent data processing by each filter without blocking the complete process chain (e.g. a filter that is currently waiting for a system resource does not block other

[10]Squidy Client for iPhone OS: http://itunes.apple.com/app/squidy-client/id329335928.

[11]Squidy Client for Android OS: http://sourceforge.net/projects/squidy-lib/files/Components/Squidy-Client-for-Android-OS.

Listing 3 The "preProcess" stub grants access to all data of a data container

```
/**
* Diverse collection of data accessible by
* this method stub before individual
* processing.
*/
public IDataContainer preProcess(
    IDataContainer dataContainer);
```

Listing 4 Processing single data objects of a specified type at a time

```
/**
* Processes data of particular generic data
* type (DATA_TYPE is a placeholder for
* those generic data types)
*/
public IData process(DATA_TYPE data);
```

filters during that time). This system of self-contained filter components prevents side effects on the signal processing and thus aids users to design consistent and reliable interaction techniques. Users can intercept a filter's internal processing by implementing simple pre-defined method stubs similar to the concept of "Method Call Interception". The following method stubs reflect different points of entry that differ in the quantity and type of dequeued data provided. The processing thread determines in a certain sequence whether a stub is implemented and then invokes this stub using reflection.

In the "preProcess" stub (Listing 3), the collections of data types grouped within a data container are passed to the method's implementation. This is an easy way to access all data at a glance or iterate through the data collection manually, e.g. to search for interaction patterns consisting of a diverse set of data types concerning multimodal interaction. Whenever it is sufficient to process one particular data instance at a time, the 'process' method stub is appropriate. The code fragment in Listing 4 is a generic representation of such a process method stub.

In the case of the "process" stub (Listing 4), the Squidy Core iterates through the collection automatically. It therefore does not have to be done programmatically as in the "preProcess" stub. Here, DATA_TYPE is the placeholder for a generic data type (Sect. 2.1.1), offering a simple data-type filter for the dataflow. The Squidy Core only passes instances of that generic type to that method implementation.

Before the data collection is published to the next filter of the processing chain or bridged back to any device or application, the data collection can be accessed through the "postProcess" stub (Listing 5). An example of using this post processing is the functionality to remove duplicate data from the dataflow to reduce data-processing overhead.

Listing 5 All data objects of a data container are accessible through the "postProcess" stub after individual data processing

```
/**
* Diverse collection of data accessible by
* this method stub after individual
* processing.
*/
public IDataContainer postProcess(
    IDataContainer dataContainer);
```

The Squidy Core uses the Java Reflection mechanism to determine if a filter has implemented such a data interception and passes inquired data to the implementation automatically. Therefore, no additional effort is required for interface declaration, generation and compilation such as is needed for the CIDL used by the OpenInterface framework (www.oi-project.org). This flexibility of the Squidy Core to quickly integrate or modify filter techniques provides the capability often needed to rapidly and iteratively prototype interactive and multimodal interfaces.

Heterogeneous devices and toolkits can be easily tied to the Squidy Interaction Library using existing Squidy Bridges (OSC Bridge, Native Interface Bridge) or custom bridge implementations (e.g. to integrate devices or toolkits communicating via special protocols). The Squidy Core provides a multi-threaded environment to perform concurrent data processing and thus increases data throughput, minimizes lag and enhances user's experience while using multimodal interaction. A suitable API supports developers to quickly implement new filters or change existing filters without the need for recompilation or repackaging. The three-tier architecture covers usage by both interaction designers and developers, assists them with appropriate tools and thus reduces mental activity to a minimum.

Currently we run applications based on Microsoft .Net, Windows Presentation Foundation and Surface SDK, Adobe Flash and Flex, OpenGL for C++ or JOGL as well as standard Java technology. The Squidy Bridges combined with Squidy Client reference implementations provide various external and integrated drivers and toolkits. Currently, Squidy supports the NUIGroup Touchlib, the Apple iPhone SDK, the Android SDK and Microsoft Surface SDK for multi-touch interaction, the ART DTrack and the NaturalPoint OptiTrack for finger gestures [9] and body-tracking, the libGaze for mobile eye-tracking [14], the iPaper framework for pen and paper-based interaction [28], the Microsoft Touchless SDK for mid-air object tracking, the Phidgets API for physical prototyping and self-developed components for laser pointer interaction [19], GPU-accelerated low-latency multi-touch tracking (SquidyVision), Nintendo Wii Remote and tangible user interface (TUI) interaction.

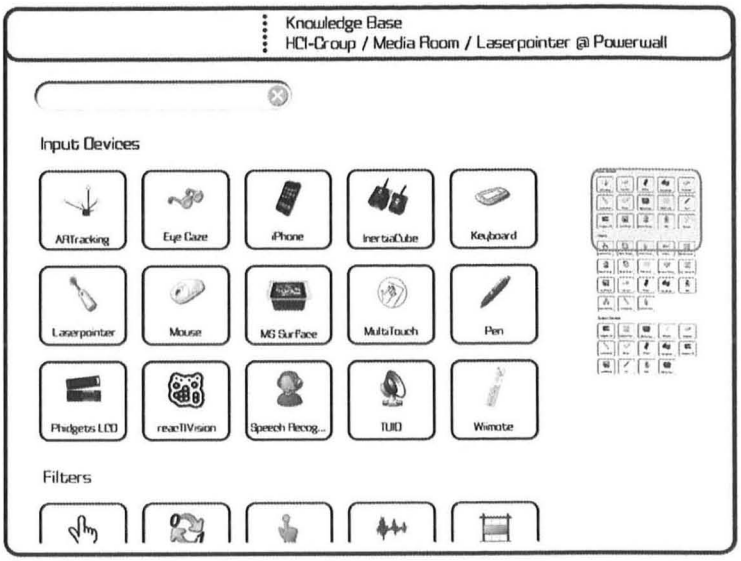

Fig. 6 The Squidy Knowledge Base is a searchable interface for accessing all implemented input device and filter nodes

2.2 User interface concept

The Squidy user interface concept is based on the concept of zoomable user interfaces. It is aimed at providing different levels of details and integrating different levels of abstraction so that frequent switching of applications can be avoided. In the following subsections we will provide more details about the different user interface concepts.

2.2.1 Knowledge base

Squidy provides a wide range of ready-to-use devices and filter nodes stored in an online knowledge base that is accessible within the Squidy user interface. An assortment is directly offered at the bottom of the pipeline view (see Fig. 2). The selection and arrangement of the nodes are based on statistics of previous usage and thus give a hint of suitable partners for the currently focused device or filter. This dynamic suggestion may lead to a higher efficiency and also helps novice users to limit the otherwise overwhelming number of available nodes to a relevant subset. The user can directly drag a desired node from the selection (bottom) to the design space for the pipeline (centre). If the desired node is not part of the suggested subset, the user has the possibility of accessing all nodes of the knowledge base by zooming into the corresponding view at the bottom. Therein, dynamic queries support the exploration (see Fig. 6). These are based both on automatically generated metadata about each node as well as user-generated tags.

2.2.2 Semantic zooming

In accordance with the assumption that navigation in information spaces is best supported by tapping into our natural spatial and geographic ways of thinking [25], we use a zoomable user-interface concept to navigate inside the Squidy visual user interface. When zooming into a node, additional information and corresponding functionalities appear, depending on the screen space available (semantic zooming). Thus, the user is able to gradually define the level of detail (complexity) according to the current need for information and functionality.

2.2.3 Interactive configuration & evaluation

In contrast to the related work, the user does not have to leave the visual interface and switch to additional applications and programming environments in order to get additional information, to change properties, or to generate, change or just access the source code of device drivers and filters. In Squidy, zooming into a node reveals all parameters and enables the user to interactively adjust the values at run-time (see Fig. 7). The changes take place immediately without any need for a restart, providing a direct relationship between user interaction and application feedback and thereby maintaining causality, as Card et al. puts it [8]. This is especially beneficial for empirically testing a number of different parameters (e.g. adjusting the noise levels of a Kalman filter) because of the possibility of directly comparing these settings without introducing any (e.g. temporal) side effects. This process of interactive configuration and evaluation is much needed during the design of multimodal interaction, especially when using uncommon interaction techniques and user interfaces. Squidy therefore facilitates fast development iterations.

2.2.4 Details on demand

Going beyond the access and manipulation of parameters, Squidy provides illustrated information about the function-

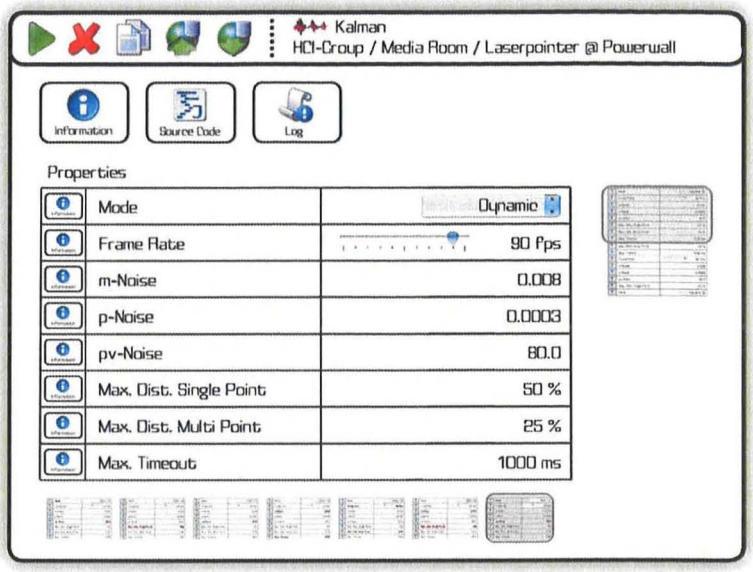

Fig. 7 View of a zoomed Kalman filter node with table of parameters. Parameter changes are applied immediately. Spatial scrolling with overview window (*right*) and temporal scrolling of last changes (*bottom*) is provided visually. Via automatic zooming, the user can access further information about the node (Fig. 8) and the filter source code (Fig. 9)

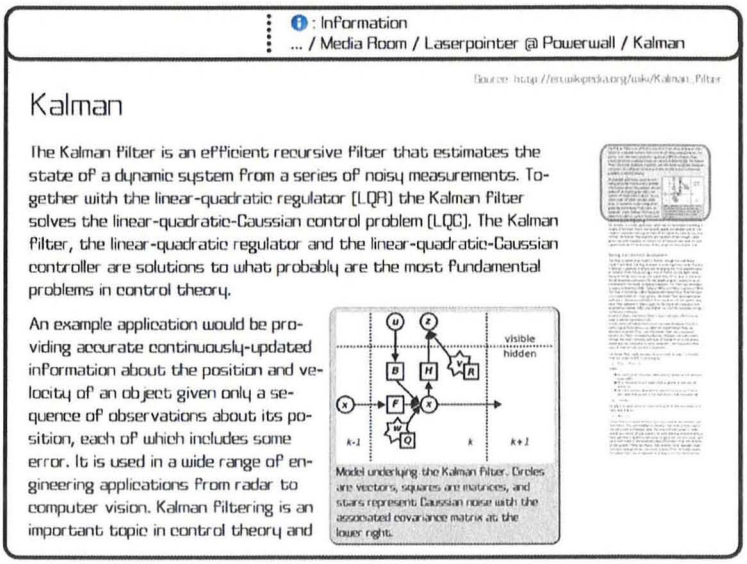

Fig. 8 Information view of the Kalman filter node providing illustrated descriptions about its functionality

ality, usage and context of the node, and this information is directly embedded in the node. By zooming into the information view marked by a white "i" on a blue background (see Fig. 7), the information is shown without losing the context of the node. This information view (see Fig. 8) may contain code documentation (e.g. automatically generated by javadoc), user-generated content (e.g. from online resources such as wikipedia.org or the Squidy-Wiki) or specifically assembled documentation such as a product specification consisting of textual descriptions, images or videos. The interaction designer using Squidy does not need to open a web browser and has to search for online documentations in order to get the relevant information. Due to the semantic zooming concept the user specifies her information need implicitly by navigating in the zoomable user interface and spatially filtering the information space.

2.2.5 *Embedded code and on-the-fly compilation*

The user even has the ability to access the source code (see Fig. 9) of the node by semantic zooming. Thus, code changes can be made directly inside the design environment. Assistants such as syntax highlighting or code completion support the user even further. If the user zooms out, the code will be compiled and integrated on the fly, again without needing to restart the system. Users may also generate new input and output devices or filters by adding an

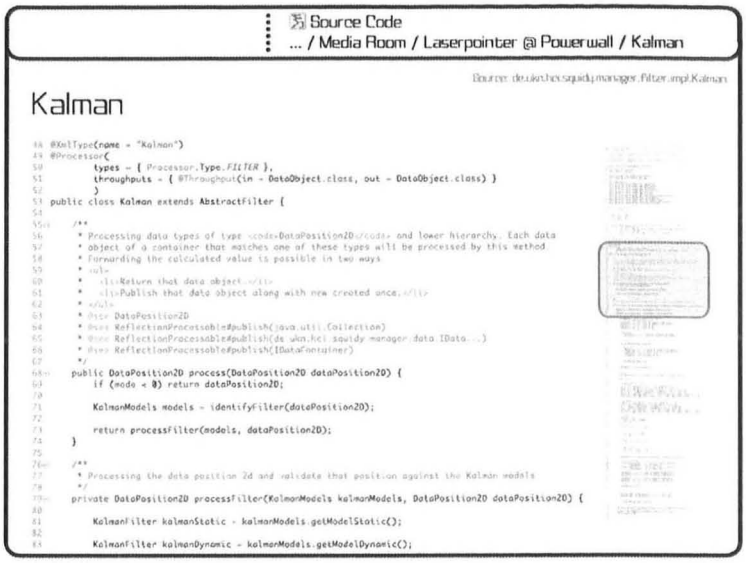

Fig. 9 Source Code of the corresponding device or filter node is directly accessible by semantic zooming. Zooming-out leads to runtime compilation of the source code and live integration into the current pipeline

empty node and augmenting it with applicable code. In order to minimize the threshold for the first steps and to reduce the writing effort, the empty node already contains all relevant method definitions for data handling and processing. Therefore, only the desired algorithm has to be filled in the suitable method body of the node. By zooming out the new node is compiled and it is than immediately ready for usage. In order to share the new node with the community the user can publish it into the knowledge base (see Publish-button in Figs. 3 and 2). The design rationale is not to replace the classical development environments such as Microsoft Visual Studio or Eclipse, but rather to integrate some of their functionality directly into Squidy. Thereby, we provide a unified environment that seamlessly integrates the most relevant tools and functionalities for the visual design and interactive development of multimodal interfaces.

2.2.6 *Dataflow visualization—visual debugging*

The visual design of an interaction technique requires a profound understanding of the data flow and the semantics of the designed pipeline. For instance, to detect and analyze interaction patterns such as gestures or multimodal input, researchers or interaction designers should be able to quickly get an overview of the interaction data flow during a particular time span. In compliance with the pipe-and-filter metaphor, we integrate a data-flow visualization at the centre of each pipe (see Fig. 2). This simple yet powerful view (see Fig. 10) visualizes the data flow through its corresponding pipe with respect to its temporal and spatial attributes. At a glance, users are able to inspect a massive amount of data, as well as data occurring in parallel, according to its spatial location and chronological order.

Direct manipulation of the time span allows the user to adjust the range to their current need. The visual representation of data depends on the type that the data belongs to (e.g. representation of a position in 2D differs from the representation of a gesture being recognized—see Fig. 10). Thus, users benefit from the insight into the interaction data flow by getting a better understanding of the effect of different parameter settings.

Every node in Squidy operates strictly within its own thread and therefore implies multi-threading and concurrent data processing without any additional effort. This allows a higher bandwidth and enhances the data throughput. Nevertheless, users may produce errors while implementing nodes or use incorrect parameter settings. This can cause side effects (e.g. array index out of bounds) that in consequence may lead to an inaccurate interaction technique or a local error (other nodes run in separate threads and are therefore unaffected). Thus, the design environment supplies each project, pipeline, node and pipe (in the following we call these shapes) with a visual colour-coded outer-glow effect (see Fig. 2) that represents the node's current status. Three distinct colours (green, red, grey) are uniquely mapped to a class of conditions. A green glowing shape indicates a properly operating node implementation running underneath. Additionally, pipes possess a green illumination when interaction data is flowing or has recently been flowing through them. The red glow indicates that an error has occurred during execution of node implementation (e.g. unhandled exception—NullPointerException). Then, all connected outgoing pipes to a defective pipeline or node are given the same error colour-coding status to enhance error detection and allow faster error correction. Shapes that are not set as activated (not running) and pipes that currently do

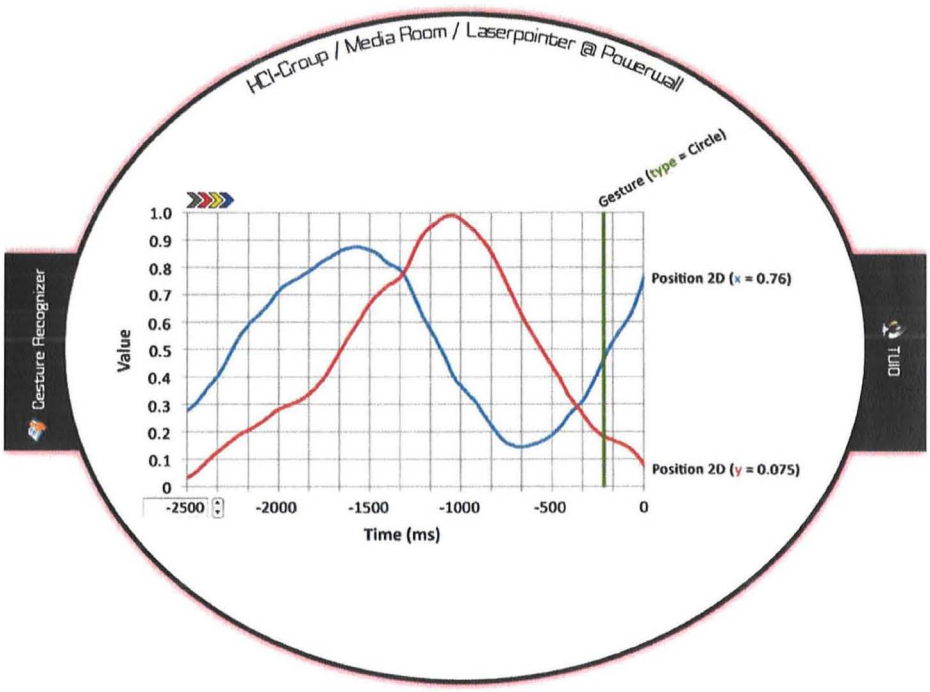

Fig. 10 Dataflow visualization showing the values of all forwarded data objects of a pipe within a defined time span

not have interaction data flowing through receive a grey illumination. Thereby, without any need for interaction, the user can perceive the status of the data flow between the nodes of the pipeline.

Occasionally, researchers or interaction designers require the capability to preclude parts of interaction data from being propagated to nodes (e.g. disabling the buttons pressed on a laser pointer and instead using gestures to control an application). Thus, a pipe provides two opportunities to narrow the set of interaction data flowing through it. The first possibility for reducing the set of interaction data is before data is visualized by the dataflow visualization. This allows the user to visually debug designated types of data. The second possibility is immediately after the data comes out of the dataflow visualization. The user can visually debug the data but nevertheless prevent it from being forwarded to nodes connected downstream. Users are able to zoom into the data-type hierarchy view (see Fig. 4) and select (which means this data is forwarded) or deselect a data type by clicking on it. In Fig. 4 all data types are selected and therefore have a coloured background. A deselected data type would just have a coloured border.

3 Squidy use cases

Over the last two years, we iteratively developed, used and enhanced Squidy during the course of applying it in several diverse projects. The starting point was the need for an infrastructure that facilitates the design and the evaluation of novel input devices and interaction techniques in multimodal and ubiquitous environments.

The first input device that we implemented with Squidy was an interactive laser pointer. This enabled a flexible interaction with a large, high-resolution display such as the Powerwall located at the University of Konstanz (221 inches, 8.9 megapixels) from any point and distance [19]. A major issue of this interaction technique was the pointing imprecision introduced by the natural hand tremor of the user and the limited human hand-eye coordination [21]. Squidy improved the design and research process by providing the opportunity to interactively implement, change and empirically test diverse smoothing techniques without introducing side effects. In an iterative approach the dataflow was visualized, the filter logic was adapted, the filter parameters were optimized, and the resulting interaction technique was finally evaluated based on a Fitts' Law Tapping Test (ISO 9241-9), which is also provided in Squidy as a ready-to-use component (see Fig. 12). Thus, Squidy supported the entire development lifecycle, resulting in a very efficient and effective project progress.

In a follow-up project we specifically made use of the separation of the three layers in Squidy since we could easily apply the laser pointer interaction to an artistic installation. This scenario utilized the laser pointer for interaction but came with a very different display and visualization technique. Surrounded by 360°-satellite images of the earth, visitors to the "Globorama" installation explored the

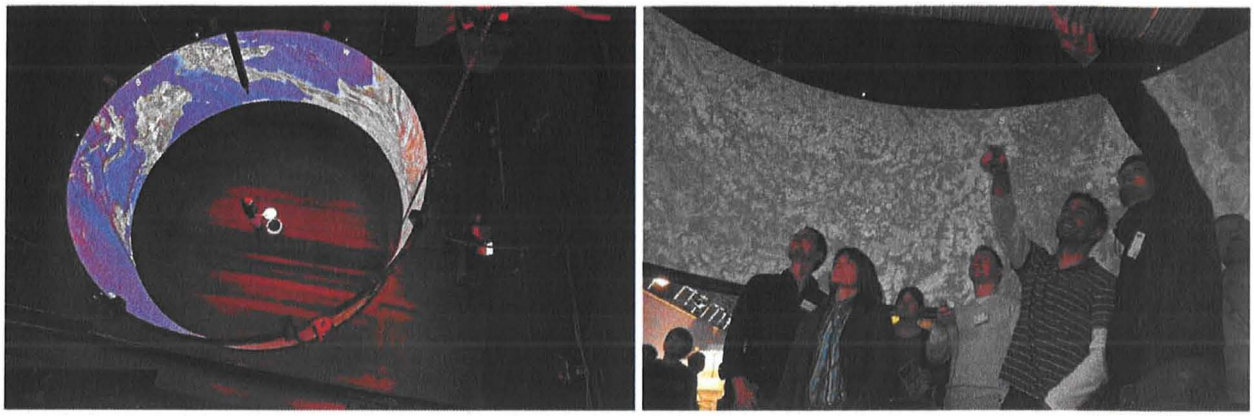

Fig. 11 Globorama installation on a 360° panoramic screen. On the *left*: top view of the panorama screen with 8 m in diameter. On the *right*: Visitors exploring the Globorama installation with an interactive laser pointer

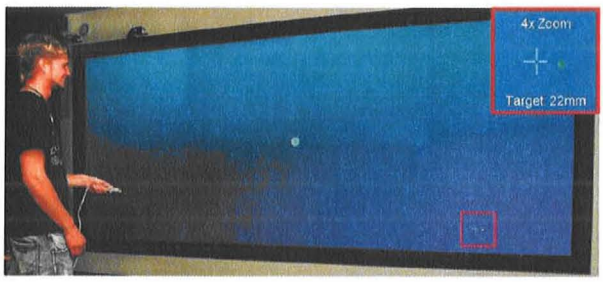

Fig. 12 Laser Pointer Interaction in front of a large high-resolution display. Squidy facilitated the integration and evaluation of precision enhancing and smoothing techniques, allowing the precise selection of targets as small as 22 mm in diameter from a 3 m distance [21]

entire globe with the laser pointer and submerged at selected points in geo-referenced pano-ramic photographs or web-cam images of the respective location (see Fig. 11). The visitors were able to physically move inside the 360° panoramic screen (8 m in diameter, 8192 × 928 px) while dynamically controlling zooming and panning and selecting interesting areas [20]. We also augmented the laser pointer with a vibrator for tactile feedback and multiple RGB-LEDs for visual feedback on the device. The tactile feedback was given whenever the visitor moved the cursor over an active element such as a geo-referenced photograph and the color LEDs visualized the current status of the system. The "Globorama" installation was exhibited at the ZKM Center for Art and Media in Karlsruhe (2007) and at the ThyssenKrupp Ideenpark 2008 in Stuttgart.

Squidy was also used to design functional prototypes for personal information management with interactive television sets [15]. For this application domain, the Nintendo Wii, in its role as a standard input device for home entertainment, was integrated into Squidy. Although the device, the application, and the display were completely different to the previous projects, the smoothing filters implemented for the laser pointer could be applied to the Wii and proved to be very beneficial, since both the Nintendo Wii and the laser pointer share an important similarity in being absolute pointing devices. Furthermore, the wiigee gesture recognition toolkit [26] was integrated into Squidy to enable three-dimensional gestures with the Nintendo Wii. Although the toolkit was originally designed for the Nintendo Wii, the laser pointer can be used interchangeably, since Squidy unifies the individual data types of the devices with the generic data types commonly defined in Squidy.

In the context of Surface Computing, we conceptually and technically combined multiple touch-sensitive displays aiming to provide a more ubiquitous user experience based on the naturalness and directness of touch interaction [16]. In this scenario, we integrated mobile handhelds (Apple iPhone) as personal devices as well as shared multi-touch tables (Microsoft Surface) and large high-resolution walls (eyevis Cubes) for collaborative design and visual information organization. In order to facilitate multimodal input and context-aware applications, we integrated speech recognition, mobile eye tracking [14] and freehand gestures [9]. To further close the gap between the digital and the physical world, we enhanced this environment with digital pens for interactive sketching and the possibility of interacting with physical tokens on the diverse multi-touch displays (see Fig. 14). All of these techniques were integrated in, and driven by, Squidy and installed in our interaction lab known as the Media Room (see Fig. 13). This physical infrastructure in combination with Squidy as the common software infrastructure gives an ideal design and development environment for researchers and interaction designers developing the user interfaces of the future.

4 Conclusion and future work

"Creating interactive systems is not simply the activity of translating a pre-existing specification into code; there is significant value in the epistemic experience of exploring alternatives" (Hartmann et al. [13]).

This statement is especially true for the design of multimodal interfaces, since there is no well established body of knowledge and no ready-to-use solution for multimodal interfaces the designer can take advantage of. Interaction designers need to physically explore and prototype new interaction modalities and therefore require development environments that especially support the interactivity and the dynamic of this creative development process. We presented the Squidy Interaction Library that supports the interactive design of multimodal user interfaces with a three-part contribution. First, it provides a software architecture that offers the flexibility needed for rapid prototyping and the possi-

Fig. 13 The Media Room is a lab environment which provides a variety of input and output devices with different modalities and form factors. Squidy serves as the basic technology to integrate these different devices as well as to configure and evaluate them

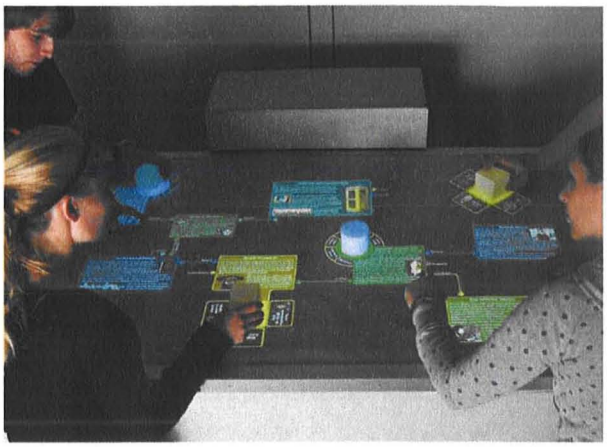

Fig. 14 Multi-touch surfaces augmented with physical tokens used in the context of blended museum

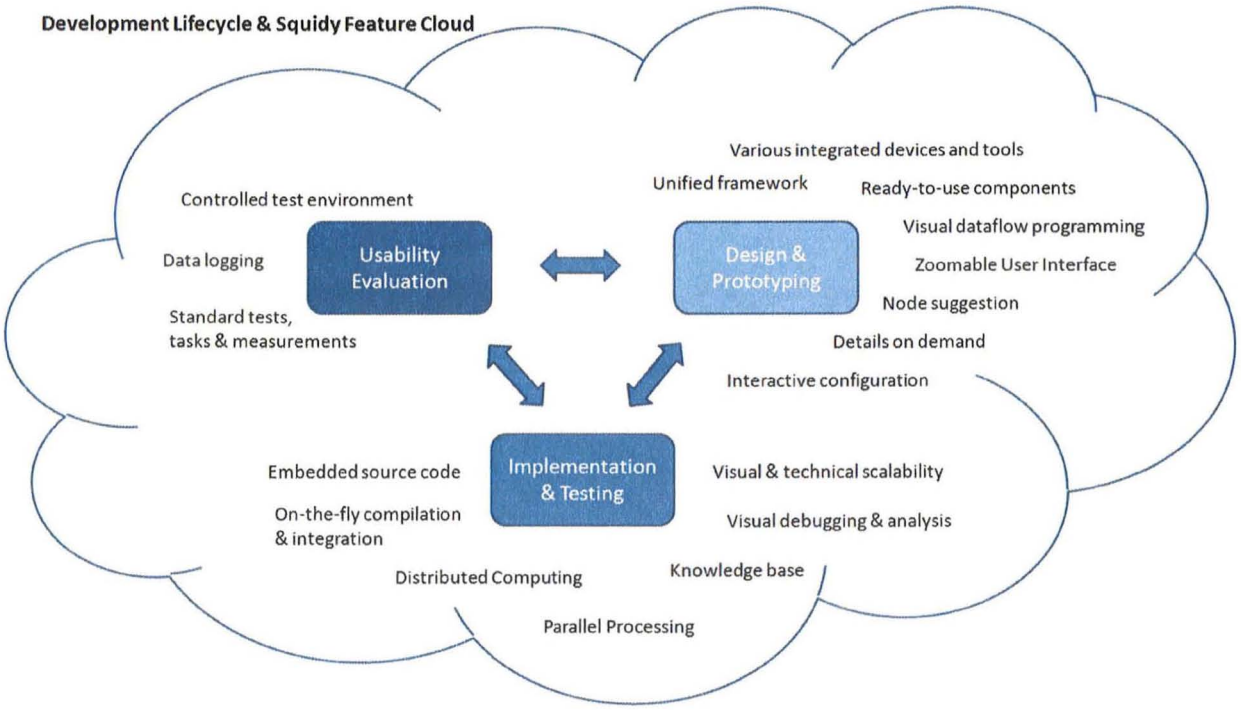

Fig. 15 This cloud shows how Squidy contributes to the development lifecycle of multimodal interaction techniques. Each phase in the lifecycle, whether it is the design and prototyping, the implementation and testing, or the usability evaluation phase is surrounded by a variety of Squidy features that support the interaction designer or developer during this activity

bility to integrate a vast variety of heterogeneous input devices and signal processing filters. Second, the Squidy visual user interface introduces a new user interface concept that combines visual dataflow programming with semantic zooming in order to reduce the visual and technical complexity. This visual approach enables also a high degree of interactivity that is further supported by the fluid integration of code views, filter mechanisms and visualization tools. Third, the Squidy Interaction Library does not only focus on the rapid prototyping, but also provides advanced development techniques and tool-support for empirical evaluation of the developed interfaces. Figure 15 shows a high-level feature cloud of the Squidy Interaction Library with respect to the different development phases. The appropriateness of Squidy to the actual design and research process was practically shown by the presented use cases. Additionally, we will conduct qualitative usability tests in order to validate and inform the design of the Squidy user interface concept. Up to now, the Squidy Interaction Library has not provided multi-user support. This, and the integration of version controlling, will be future work.

The Squidy Interaction Library is free software and published at http://hci.uni-konstanz.de/squidy/ under the licence of the LGPL.

References

1. Ballagas R, Ringel M, Stone M, Borchers J (2003) Istuff: a physical user interface toolkit for ubiquitous computing environments. In: CHI '03: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, New York, pp 537–544
2. Ballagas R, Memon F, Reiners R, Borchers J (2007) Istuff mobile: rapidly prototyping new mobile phone interfaces for ubiquitous computing. In: CHI '07: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, New York, pp 1107–1116
3. Benoit A, Bonnaud L, Caplier A, Jourde F, Nigay L, Serrano M, Damousis I, Tzovaras D, Lawson J-YL (2007) Multimodal signal processing and interaction for a driving simulator: Component-based architecture. J Multimodal User Interfaces 1(1):49–58
4. Bouchet J, Nigay L (2004) Icare: a component-based approach for the design and development of multimodal interfaces. In: CHI '04: CHI '04 extended abstracts on human factors in computing systems. ACM, New York, pp 1325–1328
5. Bouchet J, Nigay L, Ganille T (2004) Icare software components for rapidly developing multimodal interfaces. In: ICMI '04: Proceedings of the 6th international conference on multimodal interfaces. ACM, New York, pp 251–258
6. Buxton W (1983) Lexical and pragmatic considerations of input structures. SIGGRAPH Comput Graph 17(1):31–37
7. Card SK, Mackinlay JD, Robertson GG (1991) A morphological analysis of the design space of input devices. ACM Trans Inf Syst 9(2):99–122
8. Card SK, Newell A, Moran TP (1983) The psychology of human-computer interaction. Erlbaum Associates, Hillsdale
9. Foehrenbach S, König WA, Gerken J, Reiterer H (2008) Natural interaction with hand gestures and tactile feedback for large, high-res displays. In: MITH 08: Workshop on multimodal interaction through haptic feedback, held in conjunction with AVI 08: international working conference on advanced visual interfaces
10. Gellersen H, Kortuem G, Schmidt A, Beigl M (2004) Physical prototyping with smart-its. IEEE Pervasive Comput 3(3):74–82
11. Greenberg S, Fitchett C (2001) Phidgets: easy development of physical interfaces through physical widgets. In: UIST '01: Proceedings of the 14th annual ACM symposium on user interface software and technology. ACM, New York, pp 209–218
12. Harper R, Rodden T, Rogers Y, Sellen A (2008) Being human: human-computer interaction in the year 2020. Microsoft Research, Cambridge
13. Hartmann B, Abdulla L, Mittal M, Klemmer SR (2007) Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. In: CHI '07: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, New York, pp 145–154
14. Herholz S, Chuang LL, Tanner TG, Blthoff HH, Fleming R (2008) Libgaze: Real-time gaze-tracking of freely moving observers for wall-sized displays. In: VMV '08: Proceedings of the 13th international workshop on vision, modeling, and visualization
15. Jetter H-C, Engl A, Schubert S, Reiterer H (2008) Zooming not zapping: Demonstrating the ZOIL user interface paradigm for itv applications. In: Adjunct proceedings of European interactive TV conference, Salzburg, Austria, July 3–4, 2008. Demonstration Session
16. Jetter H-C, König WA, Reiterer H (2009) Understanding and designing surface computing with ZOIL and squidy. In: CHI 2009 workshop—multitouch and surface computing
17. Kaltenbrunner M, Bovermann T, Bencina R, Costanza E (2005) Tuio—a protocol for table based tangible user interfaces. In: Proceedings of the 6th international workshop on gesture in human-computer interaction and simulation
18. Kirsh D, Maglio P (1994) On distinguishing epistemic from pragmatic action. Cognitive Sci 18(4):513–549
19. König WA, Bieg H-J, Schmidt T, Reiterer H (2007) Position-independent interaction for large high-resolution displays. In: IHCI'07: Proceedings of IADIS international conference on interfaces and human computer interaction 2007. IADIS Press, pp 117–125
20. König WA, Böttger J, Völzow N, Reiterer H (2008) Laserpointer-interaction between art and science. In: IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces. ACM, New York, pp 423–424
21. König WA, Gerken J, Dierdorf S, Reiterer H (2009) Adaptive pointing: Design and evaluation of a precision enhancing technique for absolute pointing devices. In: Interact 2009: Proceedings of the twelfth IFIP conference on human-computer interaction. Springer, Berlin, pp 658–671
22. Lawson J-YL, Al-Akkad A-A, Vanderdonckt J, Macq B (2009) An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components. In: EICS'09: Proceedings of the first ACM SIGCHI symposium on engineering interactive computing. ACM, New York
23. MacKenzie IS, Ware C (1993) Lag as a determinant of human performance in interactive systems. In: CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on human factors in computing systems. ACM, New York, pp 488–493
24. Myers B, Hudson SE, Pausch R (2000) Past, present, and future of user interface software tools. ACM Trans Comput-Hum Interact 7(1):3–28
25. Perlin K, Fox D (1993) Pad: an alternative approach to the computer interface. In: SIGGRAPH '93: Proceedings of the 20th annual conference on computer graphics and interactive techniques. ACM, New York, pp 57–64
26. Schlömer T, Poppinga B, Henze N, Boll S (2008) Gesture recognition with a Wii controller. In: TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction. ACM, New York, pp 11–14

27. Serrano M, Nigay L, Lawson J-YL, Ramsay A, Murray-Smith R, Denef S (2008) The openinterface framework: a tool for multimodal interaction. In: CHI '08: Extended abstracts on human factors in computing systems. ACM, New York, pp 3501–3506
28. Signer B, Norrie MC (2007) Paperpoint: a paper-based presentation and interactive paper prototyping tool. In: TEI '07: Proceedings of the 1st international conference on tangible and embedded interaction. ACM, New York, pp 57–64
29. Taylor RM, II, Hudson TC, Seeger A, Weber H, Juliano J, Helser AT (2001) VRPN: a device-independent, network-transparent VR peripheral system. In: VRST '01: Proceedings of the ACM symposium on virtual reality software and technology. ACM, New York, pp 55–61
30. Wallace VL (1976) The semantics of graphic input devices. SIGGRAPH Comput Graph 10(1):61–65