Ludwig-Maximilians-Universität München
Department "Institut für Informatik"
Lehr- und Forschungseinheit Medieninformatik
Prof. Dr. Heinrich Hußmann

Diplomarbeit

Survey and Review of Input Libraries, Frameworks, and Toolkits for Interactive Surfaces and Recommendations for the Squidy Interaction Library

Anton Zeitler
anton.zeitler@campus.lmu.de

Bearbeitungszeitraum:	15.04.2006 bis 14.10.2009
Betreuer:	Dipl.-Medieninf. Raphael Wimmer
Verantwortlicher Hochschullehrer:	Prof. Dr. Heinrich Hußmann

Acknowledgments

I would like to thank Roman Rädle and Werner König from the Human-Computer Interaction Group of the University of Konstanz. They supported me a lot and always found some time to answer many questions. Furthermore, Jan Wehrheim and Alexander Lang gave me valuable feedback.

Abstract

This diploma thesis presents a survey and review of 28 input libraries, frameworks, and toolkits. They originate from the domains multi-touch interaction, multi-modal interaction, tangible interaction and augmented and virtual reality. The main perspective is software engineering. Out of the set of software platforms the S*quidy Interaction Library* is chosen as basis for the *Curve* project, an interactive desk featuring a curved multi-touch surface. The software architecture of Squidy is discussed and compared with the architecture of *DirectShow*, a multimedia subsystem of Microsoft Windows. A number of improvements for further developments of Squidy are recommended and described in detail. One improvement has been exemplarily implemented.

Kurzzusammenfassung

Diese Diplomarbeit bietet eine Übersicht und eine Besprechung von 28 Bibliotheken, Frameworks und Toolkits zur Verarbeitung von Eingabedaten. Diese entstammen den Domänen Multi-Touch Interaction, Multi-Modal Interaction, Tangible Interaction und Augmented und Virtual Reality. Hauptperspektive ist die Softwaretechnik. Aus dem Bestand der Softwareplattformen wird die *Squidy Interaction Library* als Basis für das Projekt *Curve* ausgewählt, einen interaktiven Arbeitstisch mit einer gekrümmten Multi-Touch-Oberfläche. Die Softwarearchitektur von Squidy wird diskutiert und mit der Architektur von *DirectShow* verglichen, einem Multimedia-Teilsystem von Microsoft Windows. Für weiterführende Entwicklungen von Squidy wird eine Anzahl von Verbesserungen empfohlen und im Detail beschrieben. Eine Verbesserung wurde exemplarisch implementiert.

Aufgabenstellung

Im Rahmen der Diplomarbeit soll eine Recherche nach bestehenden Softwareplattformen durchgeführt werden, die sich zur Verarbeitung von Eingabedaten für das Projekt *Curve* eignen. Aus dem Bestand dieser Softwareplattformen soll eine fundierte und begründete Auswahl getroffen werden, die sowohl den qualitativen als auch funktionalen Ansprüchen von *Curve* gerecht wird. Aufbauend auf der gewählten Plattform sollen Vorschläge und Empfehlungen für notwendige Ergänzungen sowie sinnvolle Verbesserungen der Plattform diskutiert und ausgearbeitet werden. Weiterhin soll fehlende Kernfunktionalität der Plattform soweit wie möglich implementiert werden.

"Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.”

München, den 14.10.2009

____
Anton Zeitler

Table of Contents

1 Introduction

The *Curve* project [1] has been initiated at the University of Munich (LMU) and aims to create a curved multi-touch desk. In the long run, this desk should also be able to cooperate with tangible objects and other multi-modal interaction techniques. This desk will require a versatile software platform which operates a multi-touch surface, handles multi-modal user input, and suits all requirements of *human-computer interaction (HCI)* research. Furthermore, this software platform should be the basis for subsequent related research projects. Thus it has to be stable, flexible, extensible, and easy to use for multiple collaborating researchers. The creation of such a software platform is a challenge.

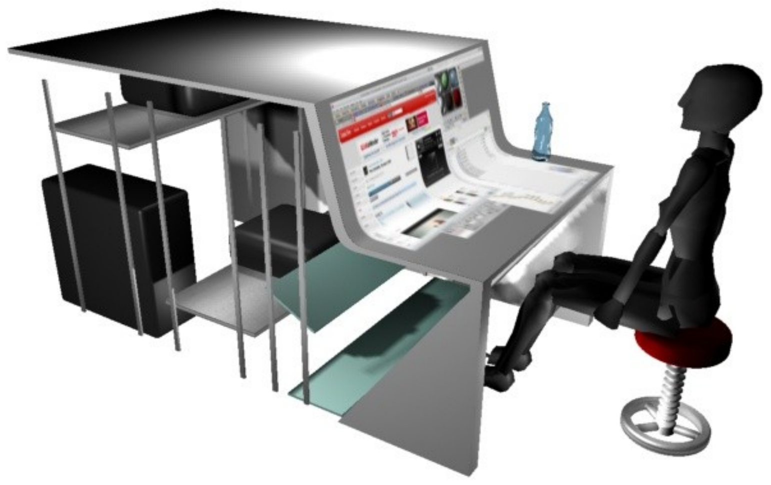


Illustration 1: The Curve desk

“Curve is a desktop-sized interactive surface with a horizontal and a slightly tilted vertical surface connected by a smooth curve.” The approach “seeks to combine a horizontal and a vertical surface into one large [seamless] interactive surface while preserving the unique properties of each.” [1]

1.1 Motivation of the Curve Project

The motivation of the Curve project reads as follows [1]:

“In today’s computer use at desks a dividing line exists between the physical work environment on the (horizontal) desktop and the virtual work environment on the (vertical) computer screen. Most human-computer interaction is done by means of keyboard and mouse combined with a vertical screen. For interacting with the physical world on our desktop hands, fingers and pens are the primary tools. Various research prototypes have tried to merge physical and virtual work environments on digital, horizontal screens. As horizontal and vertical surfaces have complementary advantages

and disadvantages they are not interchangeable. On the one hand, vertical surfaces are well suited for displaying information but less practical for manual (touch-)interaction. On the other hand, horizontal surfaces facilitate direct manual interaction with virtual and physical objects while making it difficult to work with different documents at the same time. In the following we describe the design of an interactive surface supporting those properties and our findings of its ideal size based on an initial exploratory study."

1.2 Status of the Curve Project

The ergonomic requirements of the Curve desk are currently evaluated within the scope of another diploma thesis. A user study was conducted to gain information about the optimal dimensions of the desk as well as the optimal radius of the integrated curve. [1]

Technical details of the hardware technologies used for the Curve desk are not yet finalized. Several multi-touch technologies [2] are available, most likely *FTIR (frustrated total internal reflection)* [3] and *modulated light* [4] will be used. However, the uncertainty regarding details of the hardware platform does not seriously affect the software platform. It is just one more reason why the software solution has to be able to handle various hardware configurations.

Regarding the software platform for the Curve desk, this work is the first in a row of further research activities.

1.3 Objective of this Work

This work treats technical aspects, especially technical aspects of a software platform for the processing of multi-touch and multi-modal user input. It focuses on aspects of software engineering and aims to provide a technical solution tailored to the requirements of the Curve project.

The author originally pursued the plan to develop a new software platform from scratch. However, during review of related work this plan was abandoned. Amount and quality of existing solutions lead to the decision that it does not make sense to initiate another entirely new software development project. Thus the objective changed to determining the most qualified existing solution and to analyzing the need and possibility of further improvements in the view of specific requirements of the Curve project. This solution is the *Squidy Interaction Library* [5-7]. A number of improvements is discussed and for one an implementation is given in this work.

Additionally, the review of related solutions lead to a survey of 28 software platforms for processing input from multi-touch, multi-modal, and tangible interaction as well as virtual and augmented reality. These platforms were characterized and compared and were also rated to find the most appropriate one for the Curve project.

1.4 Outline of this Work

This work starts in section 2 with a brief introduction to the field of software engineering and development to provide some basic knowledge required for comprehension of the rest of the work.

Section 3 continues with a specification of the requirements for a software platform for the Curve desk which handles multi-modal and especially multi-touch input using a curved surface.

The following section 4 is dedicated to the review and characterization of related work in terms of existing software projects for multi-touch and multi-modal input. Moreover, related fields of research including tangible interaction as well as virtual and augmented reality are examined.

The subsequent survey in tabular form in section 5 lists details and features of the projects that were presented in the preceding section and allows to compare them easily. Furthermore, the most qualified project for the Curve project is selected and the choice is justified.

Section 6 describes the internal structure and technical concepts of the selected software platform. It also compares these concepts with those of an established platform which is already long-term proven in practice.

Since the Curve desk demands for a versatile set of exceptional features, complements for the selected project in terms of improvements and missing functionality as well as concrete solutions for their realization are recommended in section 7.

In section 8 some remarkable unique features of some of the other reviewed projects are highlighted.

An implementation of one of the preceding proposals for improvements is provided in section 9 before this work closes with some final words in section 10.

1.5 Related Work

Related work, also looking into multiple software platforms, has been published by Christoph Endres et al. [8] in the field of ubiquitous computing and by Pablo Figueroa et al. [9] in the field of virtual reality. Bruno Dumas et al. [10] contribute their work in the field of tangible and multi-modal interaction. The *NUI Group* published a book [11] written by a community interested in multi-touch research which presents a survey of hardware and software technologies used for multi-touch input.

To characterize the requirements of user input, Scott R. Klemmer and James A. Landay [12] evaluated 24 applications working with physical user interfaces by four traits: input technology, input form factor, output form factor, and how tangible input and electronic output are coordinated.

2 Software Engineering Basics

Software engineering is a large field of research [13]. It covers many topics of the software development process such as project management which contains *project planning*, *risk analysis* and *quality assurance*. Further topics are *requirements engineering*, *architectural design*, *user interface design*, *software testing* and *object-oriented design*.

In this section, some basics of the design and implementation process of software engineering are introduced. Since no new software project is developed within this work, it is only a short introduction which helps to understand the ideas and terms used in the subsequent parts of this work.

However, the presented principles, concepts, and methods are worth nothing if they are not applied in an appropriate manner by talented and experienced software engineers and programmers. One can easily misunderstand or overuse these ideas or apply them in a wrong way. Software developers have to realize that "once more, the most elegant solution is the one which comes closest to the nature of the problem." [14] A crucial requirement for good software design is a correct understanding of the domain of the problem to solve, yet again the identification of the nature of the problem. Finally, a software benefits in every respect from developers who care keenly about its implementation [15].

2.1 Software Design Principles

Abstraction is a basic concept of software design. It permits to concentrate on a problem at some *level of abstraction* without regard to irrelevant details. "At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment." (p.342 in [13]) Successive steps of software design refine lower levels of abstraction down to the actual implementation in source code. This is the key for software engineers to have a chance to mentally cope with complex problems and projects.

Another elementary objective in software development is *reusablity* (p.121 in [13]). A common way to achieve reusabilty is to divide a software project into reusable *components* or *modules* which results into *modularity* (p.343 in [13]). In some contexts the terms *component* and *module* are used to denominate different elements or levels of abstraction of a software structure but in general they can be regarded as synonyms.

Creation and use of reusable components results in shorter development cycles and higher productivity. Modularity allows a program to become intellectually manageable and also allows to develop multiple modules independently from each other. In c*omponent-based software engineering*, the entire architecture of a software project is assembled from components (p.825 in [13]).

Cohesion is another important software design principle (p.353 in [13]). It demands that a module should concentrate on one specific task. *Strong cohesion* ensures clearly understandable modules and avoids unforeseen errors and side-effects caused by modules which do more than they are expected to do.

"*Coupling* is a measure of interconnection among module in a software structure." (p.354 in [13]) It tells how much a module relies on other modules. Coupling is also named *dependency* and is inversely related to cohesion. *Loose coupling* or *low dependency* results in strong cohesion. Ideally, a software structure should consist of loosely coupled modules which makes it easier to understand, assemble, modify, test, and reuse and avoids a possible "ripple-effect" of changes amongst modules.

The combination of abstraction, modularity, strong cohesion, and loose coupling is closely related to the principle of *separation of concerns*. "In the context of software evolution, a concern may be any criterion that allows us to separate parts of the software that exhibit different rates of change or that have a different impact on evolution." [16] More general, separation of concerns is the act of "focusing one's attention upon some aspect" [17] carried over to software development where it demands to treat and implement each concern separately. An example for two different concerns is the structure and the visual representation of a text document.

Last but not least, *Refactoring* is a technique to rework badly-structured source code into well-structured source code. "Refactoring it the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." (p.XVI in [18]) It is applied in modern software engineering methodology such as Extreme Programming (p.50 in [19]) but can also be used to improve existing projects [20].

2.2 Programming Paradigms

Programming paradigms are fundamental styles of computer programming. They differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints, etc.) and the steps that compose a computation (assignation, evaluation, continuations, data flows, etc.). Two programming paradigms are introduced in this section.

2.2.1 Object-Oriented Programming

Object-oriented programming (OOP) is the realization step of an object-oriented software engineering process. It uses *classes* as a method to form a software structure. Classes "encapsulate the data and procedural abstractions required to describe the content and behavior of some real world entity" (p.546 in [13]). Classes contain *attributes* (variables) which represent the current state of a class, and *methods* (functions) which operate on these attributes. An object is created as an *instance* of a particular class. A class can be regarded as a template which is used to *instantiate* new objects of the type of the class.

Important concepts of OOP are *encapsulation*, *inheritance*, and *polymorphism* (p.550 in [13]). Encapsulation supports cohesion and loose coupling by hiding attributes and methods of a class from other classes. Inheritance is a concept to realize levels of abstraction by *generalization* and *specialization* of classes. Polymorphism supports inheritance by providing a way to work with general methods of a class which actually execute specialized implementations of inherited classes. These three concepts are backed by the concept of *interfaces* which describe the external interface of a class separated from its actual implementation.

Object-oriented software engineering usually results in a hierarchical class structure with the most abstract functionality on the top level of the hierarchy encapsulating classes which fan out to lower levels of abstraction. All projects reviewed for this work which allow public access to their source code employ OOP.

2.2.2 Aspect-Oriented Programming

Compared to OOP, *aspect oriented programming (AOP)* [21] is rarely used. It is a powerful technique to realize separation of concerns but it may take a while for a programmer to internalize and master the ideas of AOP. However, AOP integrates with OOP and complements it with additional functionality.

AOP introduces *aspects* which operate orthogonal to the levels of abstraction of a software project. An aspect cross-cuts the software structure to allow detached implementations of components which work simultaneously on multiple levels of abstraction. The implementation of an aspect is automatically *weaved* into class methods during the code compilation process. The weaving process is controlled by *pointcuts* which are assigned to a particular aspect [22]. A *pointcut* allows a flexible definition of rules defining which classes and methods are connected with an aspect.

An aspect modifies the normal program flow of object-oriented programs. It can be applied for example to the entry point of each method of a class to verify the availability of a required resource and to provide error handling.

2.3 Patterns

Patterns were generally defined by Christopher Alexander: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." [23]

Two specific forms of patterns relevant for software development are *design patterns* and *architectural patterns* which are briefly introduced in this section.

2.3.1 Design Patterns

A specialized form of patterns applied to object-oriented software engineering are *design patterns*. They are defined as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." [24] In other words, design patterns provide proven solutions for common tasks in OOP. They help a software developer to build a well-designed software structure.

Only a small selection of existing design patterns [24] can be listed here:

- *Observer*: reduction of coupling by *Inversion of Control* [25]
- *Composite*: abstraction by a unique interface for individual and compositions of objects
- *Abstract Factory*: encapsulation of object instantiation
- *Proxy*: hiding of actual implementation to control access and encapsulate additional required operations
- *Visitor*: representation and extension of operations on other objects
- *Object Pool*: specialization of Abstract Factory including caching of objects [26]
- *Dependency Injection*: generalization of Abstract Factory including Inversion of Control [25]

2.3.2 Architectural Patterns

In contrast to the relatively small context of a design pattern, "an *architectural pattern* expresses a fundamental structural organization schema for software systems." (p.25 in [27]). It is a "template for [a] concrete software architecture" and is thereby an essential decision at the beginning of a software development project as it determines the basis for all subsequent development processes.

Layers

Layers can be found in nearly every software project as they are a typical instrument to realize abstraction (p.31 in [27], [28]). An example for a layered architecture is the *OSI model for communication protocols* [29]. It defines a stack of 7 layers of abstraction from the application layer which actually operates on domain-specific data types, over transport protocols for data packets, down to the physical medium for data transport.

Layers can for example be realized by inheritance of classes or by creating a hierarchy of modules which implement different levels of abstraction. Modules embedded in or encapsulating other architectural patterns than layers are often using a layered structure themselves. This also became evident during the review of existing software projects for this work.

Pipes and Filters

The *Pipes and Filters* pattern is appropriate for systems which process a continuous stream of data (p.53 in [27], [28]). Its architecture consists of a chain of processing nodes called *filters*. These are connected by *pipes* and thereby grouped to a *filter chain* or *processing chain*, also called the *pipeline*. Pipes can be designed as separate active components or as a logical construct only. The primary data flow within this processing chain is commonly unidirectional, the direction of data flow is called the *downstream* direction in opposite to the *upstream* direction [30; 31].

The interfaces of the pipe and filter components as well as the data transport format between both components should be defined identically for all components to allow flexible rearrangement of the filters in a pipeline.

Model-View-Controller

The *Model-View-Controller (MVC)* pattern is an architectural solution for applications with a user interface. It defines the *model* which is responsible for operations related to the application domain, the *view* which displays the state of the application, and the *controller* which executes user interaction with the model and the view (p.125 in [27], [32]). MVC separates the concerns *processing*, *output*, and *input* of an application.

2.3.3 Anti-Patterns

In contrast to a pattern which demonstrates a good solution for a problem, an *anti-pattern* denotes an explicitly bad solution for a problem. “An *anti-pattern* is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.” [33] Anti-patterns help to identify and name bad software design and suggest approaches how to improve it. They also try to raise the awareness to anti-patterns of software developers and educate them by giving negative examples.

Some examples of anti-patterns are:

- *The Blob* or *God Class*: one class monopolizes processing and is responsible for “everything” (p.42 in [33])
- *Spaghetti Code*: very little software structure, large method implementations (p.64)
- *Cut-And-Paste Programming*: several similar or duplicated segments of code (p.75)

A software project which features multiple anti-patterns tends to develop itself into a *Big Ball of Mud*. A Big Ball of Mud is a “haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.” [34] For a software developer, a Big Ball of Mud is hard to understand and expensive to maintain.

2.4 Concurrency, Multi-Threading, and Parallel Processing

Subsequent parts of this work sometimes refer to *multi-threading* and *parallel processing*. Both are aspects of *concurrency* which is ”concerned with the fundamental aspects of systems of multiple, simultaneously active computing agents that interact with one another.” [35]

Multi-threading is the concept of using multiple *threads* within a computer program. A thread contains an implementation of a sequential program flow which can be executed independently from other threads. Threads can be regarded as lightweight processes, in terms of instances of computer programs, which run within another computer program.

Processes and threads are commonly managed by the used platform. Multiple simultaneously executed processes or threads perform *parallel processing.*

If the underlying hardware platform incorporates less processing units than the number of simultaneously executed threads, an illusion of parallel processing is commonly achieved by *multi-tasking:* Multiple threads share a processing unit and are executed sequentially and interleaved based on a schedule (*time-division multiplexing*).

Contemporary (2009) hardware platforms often provide multiple processing units [36]. Concurrency enables a computer program to utilize the full processing power of such hardware. But it also requires that software developers care for possible issues and challenges arising from parallel execution of operations, such as synchronization of data access and adherence of the correct order of operations.

2.5 Libraries, Frameworks, and Toolkits

Since software should be designed in a reusable way (see section 2.1), it is obvious that the resulting reusable elements of a software project have to be organized in some way. A common way to do this is to create various packages containing reusable software modules. These packages can be used as a basis for other projects and can be provided to other developers either for free or against payment. Particular variants of such software packages are *libraries*, *frameworks*, and *toolkits.*

A *library* in terms of software is a generalized set of related algorithms which focuses exclusively on code reuse [37]. It provides a set of independent functions which can be called by another software using the library. Examples for such functions can be a mathematical operation, loading of a file into memory, or a sorting algorithm.

A *framework* does not solely include reusable code but incorporates a software architecture. There is no unique definition but possible definitions for a framework are “a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact” or “the skeleton of an application that can be customized by an application developer.” [38] Object-oriented frameworks make use of proven patterns and provide stable interfaces as well as generic components. They reduce the required effort for development of customized applications, they are extensible and they improve quality, reliability, and interoperability of software projects [39].

A *toolkit* goes one step further. It is an “integrated toolset to enable users to create and test designs for custom products or services that can then be produced 'as is' by manufacturers.” [40] Toolkits are easy to use, contain libraries of reusable modules commonly used in custom designs, and allow a user to create final products which do not have to be revised by additional experts before they are published. In terms of software, this can be a framework which includes additional tools for testing and evaluation, or GUI (Graphical User Interface) elements which can be used together with the framework [37].

3 Requirements for an Input Framework

When a software project has to be realized, the first step in the software engineering process is *requirements engineering*. It describes the procedure of specifying requirements for a project with respect to the demands of its various *stakeholders* (p.255 in [13]). Even if no new software development project is initiated in this work, a list of requirements which define selection and comparison criteria for the review of existing projects is needed.

The author acquainted himself with the domains of multi-touch and multi-modal input and created a list of requirements. He was also able to refine his list of requirements together with stakeholders which were familiar with these domains. The concept of an object-oriented framework (see section 2.5) is selected because it complies with the needs of the project and the stakeholders and can be extended to a toolkit if necessary.

In an abstract and generalized form, the focus of the input framework is management and control of the employed hardware devices such as cameras and sensors. It is also responsible for further processing and abstraction of input data produced by these devices. The framework must provide the resulting data in real-time in a unified format.

To get a more precise specification of the requirements for the framework, the perspectives of the relevant stakeholders for the Curve project are discussed separately in the following sections.

3.1 Software Developer's Perspective

Software developers and especially successors of software developers prefer to work with clearly structured software and source code which is easy to understand. Moreover, a software developer's life is easy if he can change a feature of the software without being afraid of disrupting any other part of it, or if he can add a feature without the need to modify existing source code.

Even if some of these requirements are expected from a framework (see section 2.5), experience of the author shows that they are not self-evident for every software developer. This results in the following list of requirements regarding the people who participate on the software development of the framework:

- modular, flexible, and easily extensible software design
- elaborate software architecture based on proven design patterns (see section 2.3)
- carefully written source code which follows clear conventions [15]
- documentation of architecture, modules and application programming API (application programming interface)

3.2 Application Developer's Perspective

An application developer who integrates the framework in his application is pleased to have a simple interface to the framework which directly conforms to the semantics of his application. Ideally, he can perform the integration once and then forget about it because the interface and the structure of data transferred over it will never change and universally matches any input device and interaction modality.

Therefore, the following points are important for application developers:

- output of input data in a unified data format through a unified and stable software interface
- abstraction and interpretation of input data including mapping to application semantics

3.3 End-User's Perspective

Experience shows that the end-user, regarded as the union of the diversity of many end-users, typically likes to have many features and possibilities. This means he would be happy to be able to use the framework on multiple platforms with many different input devices, especially exactly with the devices he likes or owns, maybe simultaneously with other users. He also wants to control all settings of these devices in a convenient way.

Additionally, multi-touch applications can suffer from high latency. That is the time which lies between the actual (inter)action of the user and the feedback of the application. From practice it is known that recognizable latency can be annoying for the user and should be reduced to a minimum. Therefore, the portion of latency contributed by the framework should be as small as possible. As a reference example, a popular multi-touch framework introduces about 30 ms latency on a contemporary (2009) hardware platform [2].

These findings result in the following list of requirements:

- support of multiple input devices and users
- support of multiple types of input devices and input data
- support of multiple underlying hardware and software platforms
- configuration and control of input devices by the user's application
- possibility to customize and configure all data processing steps
- low latency of multi-touch input processing (less than 30 ms)

3.4 Scientist's Perspective

A scientist often incorporates the previously named stakeholders in one single person and has some additional demands to support his research tasks. He ideally needs to have multiple easy and quick ways to evaluate and compare multiple hardware and software configurations. He also needs reproducible, comprehensible, and accurate input data and often collaborates with other scientists to work together using bleeding edge technology.

The subsequent list summarizes the individual requirements of a scientist employing the framework:

- *support for multiple rapid prototyping methods* to allow quick execution of research tasks with little effort
- *interface for scripting languages* as a particular method of rapid prototyping and simplified development
- *interface for logging output* for diagnosis of problems and recording input activities
- *access to raw and output data of every processing step on demand* to quickly hook up custom processing or diagnosis steps
- *distributed processing of input data* to support collaboration of multiple persons working with multiple potentially distributed platforms, and also to allow research on complex distributed system setups
- *synchronization of multiple input devices and input data* to ensure meaningful and accurate correlations between all devices and their data
- *synchronization interface which allows to synchronize an application with input devices* for synchronization with output devices and user interaction
- *parallel processing of data processing steps* as a technical instrument to fully utilize contemporary (2009) hardware platforms and also to keep latency low

3.5 Curve's Perspective

The design of the Curve desk (see section 1.1) implies some special requirements which have to be itemized more precisely. The following requirements originate from the demand to get a handsome, planar, and undistorted view of the entire curved surface which is captured by multiple cameras from a short distance with wide-angle lenses:

- support for simultaneous input of images from multiple video cameras
- distortion correction and seamless stitching of images in real-time
- geometric surface calibration of curved surfaces
- precise camera calibration including distortion correction of fish-eye lenses

3.6 Technical Specifications

In addition to the unspecific requirement listed above, some specific technical requirements for the framework were collected. The following listing does not claim to be comprehensive and emerges from previous knowledge, practical experience, and plans for future research.

3.6.1 Supported Communication Protocols

- *OSC including TUIO*, which are popular protocols in the field of multi-touch research

- *position and event protocols of libtisch* which is a project of researchers cooperating with the Curve team

3.6.2 Supported Interaction Devices and Interfaces

- multi-touch surfaces based on *FTIR*, *DI*, and capacitive sensing
- *DiamondTouch*, *Second Light*, *Thin Sight* and comparable multi-touch systems
- mobile displays which interact with the multi-touch surface using overlay techniques
- fiducial and other types of markers
- surface of objects, e.g. printed text documents or photographs
- shapes and shadows of objects including gestures
- keyboard, mouse, and similar input devices
- mobile phone or notebook via suitable interfaces such as *WLAN* or *Bluetooth*
- various types of sensors or tags communicating by *NFC*, *RFID*, or similar techniques

3.6.3 Supported Types of Data

- planar and spatial coordinates as well as motion and acceleration data of objects
- supplementary data of multi-touch events such as covered area and pressure
- camera and surface calibration data, especially for multi-touch surfaces
- images from video and photo cameras
- ID or URL of users, devices, and gestures
- abstracted interaction commands

4 Survey of Input Frameworks, Libraries, and Toolkits

The original intention of this work was to create a new input framework which meets all the requirements for Curve. Yet extensive investigation revealed way more related work than expected. Overall 28 frameworks, libraries, and toolkits either in implemented form or as a theoretical description were found. Thus the original plan was changed from creating a new framework to adopting and extending an existing one.

In order to come to a qualified decision which piece of work to take as basis for Curve, all frameworks, libraries, and toolkits have been inspected and rated. They were compared by basic characteristics, their features and extensibility, and compliance to the requirements for Curve. Detailed studies or even evaluating each of them in practice would have gone beyond the scope of this work.

Related work in terms of software projects covers primarily the field of *multi-touch interaction* which is the focus of Curve. It also considered related work done in other fields of research, in particular *augmented and virtual reality*, *multi-modal interaction*, and *tangible interaction.* These fields have comparable demands on input data processing. This section presents each related project in terms of framework, library, and toolkit with a short abstract.

Due to constraints in terms of extent and expenditure of time regarding this work, the survey covers the projects considered most relevant for the Curve project. It does not include the projects *ACTIF, Bespoke 3DUI, DIVERSE, DWARF, EVI3d, FLUID, MORGAN (DEVAL), Studierstube (OpenTracker), Tinmith-evo5, VENSA, VPRN,* and *VR Juggler (Gadgeteer)* from the research field of augmented and virtual reality. Moreover, *Concerto, EMF, FAME, Flippo, GlovePIE, HephaisTK, ICARE, Icon, iMap, iStuff, iStuff Mobile, Mengine, Pure Data, QuickSet, Santos, Service Counter System, STARS, W3C Multimodal Interaction Framework, GestureTek, IntuiKit, MAX/MSP,* and *trackd* from the research field of multi-modal interaction are not included.

4.1 Augmented and Virtual Reality

A multitude of input devices is used for HCI in *augmented and virtual reality* applications. Additionally, projects from this field of research are often combined with interfaces for output of three-dimensional computer graphics.

4.1.1 Unit

The *Unit*[1] framework [41-43] focuses on an abstraction layer for *interaction techniques.* "Interaction techniques involve the mapping of data from input devices to application

1 http://www.csc.kth.se/~alx/

semantics." [41] The framework comes with a visual programming interface implemented in Java3D.

It has a Pipes and Filters architecture comparable to *OpenInterface* and *vvvv* (see sections 4.2.1, 4.2.3) which introduces *units* as the equivalent of filters. Units are combined to interaction technologies in a *Unit Graph*. A unit has a number of *properties* of a specific data type which can be routed to properties of other units with matching data types. Thus different particular units vary in their input and output interface and cannot be connected in arbitrary order because suitable properties must be found for connection.

Unit allows distributed processing using Java RMI and already includes a number of units for input devices, input transformation, and input interpretation. Units can be replaced, remapped, and reconfigured at runtime. The authors of Unit are currently working on porting the implementation from Java and *Java3D*[2] to C++ and *OpenGL*[3].

4.1.2 ViSTA (DataLaViSTA)

ViSTA[4] [44] is a large toolkit for the development of virtual reality applications and includes an abstraction layer for input devices. It is possible to control the toolkit using a scripting language, an editor for visual programming is planned. The toolkit has not been available to the public for many years but went open source in September 2009. There was not enough time to evaluate the very large code base of ViSTA. Thus this work concentrates on *DataLaViSTA* (see section 8.6), a framework which is part of ViSTA.

DataLaViSTA [30] is an implementation of the Pipes and Filters architectural pattern. It does not form an entire input framework but concentrates on data transport and processing. The framework has been successfully used in research projects of its authors. They address multiple challenges which have to be solved for efficient data transport and processing in a distributed environment that is used for real-time interaction applications. These are for example multi-threading, synchronization, and inter-process communication.

The framework includes a layer structure for its pipes and filters. The *base layer* contains basic implementations for pipes and filters and data transportation. Specific filters and pipes are implemented in the *common layer*. The *construction layer* contains algorithms to construct the filter chain.

4.2 Multi-Modal Interaction

A *modality* in terms of input processing of the computer in HCI describes a class of devices or sensors which is being used by the human to provide input to the computer. *Multi-modal interaction* allows to use or combine several modalities for this task.

2 http://java.sun.com/javase/technologies/desktop/java3d/
3 http://www.opengl.org/
4 http://www.rz.rwth-aachen.de/ca/c/piz/

4.2.1 OpenInterface

OpenInterface[5] [45] is a very ambitious project for multi-modal input processing and interaction. It has been planned since 2004, developed since 2006, and continues until end of 2010. The project is being realized by a consortium of multiple academic, research, and commercial organizations. OpenInterface implements a framework consisting of a runtime platform and graphical editors for visual programming, combined to a workbench with additional tools and components. The project aims to "handle a rich and extensible set of modalities, enable a focus on new modalities or forms of multimodality, support dynamic selection and combination of modalities to fit the ongoing context of use, and enable iterative user-centered design."[6] OpenInterface currently includes 12 example projects, and interfaces for more than 20 devices and protocols which are also available from an online repository.

The OpenInterface platform uses a Pipes and Filters architecture for data processing and is designed to be platform and language independent. The runtime platform includes the *OpenInterface Kernel. OpenInterface Components* implement interfaces to devices, process data, etc. and are defined as "reusable and independent software units with exported and imported input/output interfaces." They communicate through the Kernel through an interface defined using *OpenInterface CIDL (Component Interface Definition Language)* and can be developed using C, C++, Java, MATLAB, or C#. CIDL code is generated from source code and is used to build proxy objects which connect in between Components and the Kernel.

OpenInterface Pipelines are described using *OpenInterface PDCL (Pipeline Description and Configuration Language).* A Pipeline defines a set of interacting Components and is assembled and executed by the Kernel. The particular variant of the employed Pipes and Filters architecture is comparable to *Unit* and *vvvv* (see sections 4.1.1, 4.2.3), it uses multiple input and output *ports* with specific data types which can connect to compatible types.

Two visual progamming environments are available, *OIDE* and *SKEMMI.* OIDE is a standalone application and is not being further developed. SKEMMI provides more functionality, is realized as a plug-in for the Eclipse platform, and supersedes OIDE. The goals of the SKEMMI environment can be compared to the *Squidy Interaction Library* (see section 4.2.2). These are integrated component development, multi-level design (compare to semantic zooming), reusability of components, integrated documentation, and integrated debugging at runtime.

The binary distribution package of OpenInterface is huge compared to most of the other projects (Kernel setup file for Windows 43 MB, installed 139 MB, SKEMMI installed 91 MB). It runs on the Linux and the Windows XP 32-bit platform (execution failed on Windows Vista 64-bit with version 0.3.5.5c, 0.3.6 only tested on Windows XP 32-bit). During testing, OpenInterface behaved ponderous, it was difficult to construct a working Pipeline with SKEMMI, and not all included examples were working. The

5 http://www.openinterface.org/platform
6 http://www.oi-project.org/

Kernel sometimes locked up, or failed to assemble or execute a Pipeline while displaying various error messages. Overall, development with SKEMMI and OpenInterface felt rather uncomfortable, intricate and error-prone to the author. This is very likely due to the not yet completed beta status of the project and the very complex interaction of the multitude of elements of OpenInterface. Moreover, SKEMMI and the OpenInterface Kernel have been and are still developed solely by one single author, Jean-Yves Lionel Lawson, who has to handle this large development task on his own.

4.2.2 Squidy Interaction Library

The *Squidy Interaction Library*[7] [5-7] is a platform independent framework implemented on the Java platform and consists of three logical parts. *Squidy Manager* provides a flexible infrastructure for data processing, transport, management, and persistence. *Squidy Designe*r offers a comfortable GUI for interactive visual programming and visual feedback. *Squidy Client Implementations* are external applications which are executed on a client platform and provide input data to Squidy Manager.

Squidy Manager is implemented using a multi-threaded Pipes and Filters architecture. It manages *Nodes* for data processing which are connected via *Pipes* to form *Pipelines*. In contrast to other comparable projects based on the Pipes and Filters architecture, for example *Unit* (see section 4.1.1), the framework does not directly integrate the internal properties of a Node into its data processing Pipeline and provides just a single pair of input and output ports per Node, including automatic data type management. Connection to external data sources and data targets is established using *Bridges*. Squidy Designer uses *semantic zooming* for visualization of Pipelines and its contents. A library of Nodes for multiple interaction modalities is already included.

The Squidy Interaction Library aims at being a platform for rapid prototyping of multi-modal interfaces and iterative development of multi-modal applications. Many of its objectives are similar to *OpenInterface* and *SKEMMI* (see section 4.2.1), are already realized, and have been successfully applied in practice. More details and advantages of the Squidy Interaction Library are presented and discussed in the sections 5.2, 6 and all subsequent sections.

4.2.3 vvvv

With the *vvvv*[8] toolkit, a developer combines a set of *Nodes* into a *Patch* by using a visual programming interface. A Node is a processing element with a number of input and output *pins* which can be connected and work in the same fashion as the *properties* of *Unit* and *ports* of *OpenInterface* (see sections 4.1.1, 4.2.1). A Patch can be used in the same way as a Node, multiple Patches can again be combined to another

7 http://www.squidy-lib.de/
8 http://vvvv.org/

Patch. vvvv provides a minimal GUI in terms of usability and conformity to established user interface guidelines.

The designated audience of vvvv are developers of multimedia applications and installations with physical interfaces. The current version of vvvv is shipped with 788 individual Nodes for 2D and 3D graphics and effects, animation, audio processing, mathematical and logical operations, debugging, input and output devices, and other purposes. The toolkit focuses on output and visual effects but it includes a video image source Node based on DirectShow (see section 6.7), an image undistortion filter, and some image trackers based on the *freeframe*[9] plug-in system, including a fiducial marker tracker based on the *fidtrack* library from the *reacTIVision* project (see section 4.5.2).

The pin configuration of Nodes and the structure of Patches is stored in XML files which internally refer to native code implementations or to other Nodes or Patches. They form a Pipes and Filters architecture and integrate multiple third-party libraries which are included with the toolkit. Patches can only be executed within the runtime environment of vvvv. A developer can also implement and integrate Nodes based on native code into vvvv on himself, a Node for example for *TUIO* [46] has been created by the Internet community[10].

4.3 Multi-Touch Interaction

Multi-touch interaction focuses on input created through touches on a surface. These touches commonly originate from multiple human fingers.

During detailed examination of the following projects it turned out that some of them focus on GUIs for multi-touch input instead of providing the input data itself. However, the author decided to keep them in the listing to provide a more comprehensive view especially regarding multi-touch development.

4.3.1 Bespoke Multi-Touch

The *Bespoke Multi-Touch*[11] framework is clearly structured and integrates with the *Microsoft XNA*[12] game development framework. Bespoke Multi-Touch comes with a user interface component for XNA, and its architecture is comparable to the one of XNA, too. The framework uses a hierarchical class structure covered by a class which represents a multi-touch input surface and which acts comparable to a service. This class signals multi-touch input events which can be handled by an application by using C# delegates.

The employed image filters are taken from the *AForge*[13] framework, DirectShow is used for the interface to video cameras. The configuration of a multi-touch surface is stored

9 http://freeframe.sourceforge.net/
10 http://vvvv.org/tiki-view_forum_thread.php?comments_parentId=20637&forumId=22
11 http://www.bespokesoftware.org/multi-touch
12 http://www.xna.com/
13 http://code.google.com/p/aforge/

in an XML configuration file. Since the framework is written using C#, scripting can be done by Microsoft PowerShell which is able to access any .NET object.

4.3.2 Community Core Vision

Community Core Vision[14] *(CCV)* is a newly developed multi-touch framework by the *NUI Group* as a cross-platform alternative to *Touchlib* (see section 4.3.15). It includes a GUI which allows to create and test standard multi-touch setups and stores the created image processing configuration in an editable XML file. The framework comes with a set of image filters based on OpenGL pixel shaders which are executed on the GPU (Graphics Processing Unit), including a blob detector.

CCV is based on *OpenFrameworks*[15], a software development library which provides platform independent functionality for multi-media development in C++ including support for image capture from video cameras. CCV provides a C++ interface which can be implemented by a class to handle multi-touch input events. It is compatible with the Windows, Mac OS X and Linux platforms but it uses three separate and redundant sets of source code.

4.3.3 EquisFTIR

The *EquisFTIR*[16] library [47] is designed to be very compact and fast and integrates with the Microsoft XNA game development framework. It connects to cameras using DirectShow and includes some optimized image filters which use compiler intrinsics to utilize the SSE processing unit of modern CPUs. Multi-touch input events have to be fetched by the application by polling them from in internal queue of the library.

4.3.4 libavg

libavg[17] is a versatile library for input, processing, and output of audio, video, and image data. It includes support for image capture from video cameras via multiple interfaces for different hardware subsystems (*libdc1394*[18], *CMU1394*[19], *Video4Linux*[20], *DirectShow*). Furthermore, it comes with a large set of image filters including some with GPU support, and contains blob detection and tracking functions. libavg also includes support for scripting using *Python*[21].

Since it is a library for software development, libavg does not provide ready-to-use functionality for multi-touch systems but only functions to build such a system. A user can use the provided functionality for image capture, image processing and graphical output to develop a multi-touch input system. He needs skills in a programming language or the Python scripting language to solve this task.

14 http://ccv.nuigroup.com/
15 http://www.openframeworks.cc/
16 http://research.cs.queensu.ca/~wolfe/equisftir/
17 http://www.libavg.de/
18 http://damien.douxchamps.net/ieee1394/libdc1394/
19 http://www.cs.cmu.edu/~iwan/1394/
20 http://linux.bytesex.org/v4l2/
21 http://www.python.org/

4.3.5 libtisch

The *libtisch*[22] framework [48] supports multiple types of input devices and includes input interpretation and presentation. It has a lightweight and easy to understand software architecture and does not rely on external dependencies except an interface for image capture from video cameras (*libdc1394*, *Video4Linux*, or *DirectShow*) and some included small libraries for mathematical calculations. libtisch supports shadow tracking [49] and comes with a Sudoku game as example application.

Moreover, it allows distributed processing and has a strict separation of layers which are implemented as separate processes connected by sockets using proprietary, compact, and text-based communication protocols. An application can connect to one or in between two layers and process input data by parsing or generating commands using the communication protocols of libtisch.

The *hardware abstraction* layer manages and unifies hardware and includes image processing, blob recognition, and tracking. It outputs 2D coordinate data to the *transformation* layer which applies surface calibration transformations to the data. The *interpretation* layer detects gestures on the transformed data and signals them to the *widget* layer. The latter one is designated for integration in a user application and comes with two example widgets which use OpenGL for presentation.

4.3.6 mu3

mu3[23] [50] is a new framework which was released in alpha development stage in September 2009. It focuses on analysis and abstraction of multi-touch and tangible input data. The framework aims to identify and track multiple objects and users by applying advanced methods for segmentation of input data and trajectory detection. It does not yet support video cameras as a data source but it works on multi-touch input data supplied by TUIO.

mu3 distinguishes itself by elaborate software architecture and design. The framework wraps input data intro stream classes, which are transparently wrapped by calibration classes to transform input data according to the calibration. Input streams are managed by input providers which can be registered in an environment class. Input devices are abstracted by a touch device class which supports registration of listener classes for touch input notification. The design separates concerns and allows to develop and extend each concern without affecting others.

4.3.7 Multi-Touch Vista

Multi-Touch Vista[24] operates on pre-processed multi-touch input data such as 2D coordinates of a finger contact. It provides an abstraction layer for multiple input data sources and transmits the input data to an application using *WCF*[25] *(Windows Communication Foundation)*. It also includes input interpretation logic and and a set

22 http://tisch.sourceforge.net/
23 http://code.google.com/p/mu3/
24 http://www.codeplex.com/MultiTouchVista
25 http://msdn.microsoft.com/en-us/netframework/aa663324.aspx

of user interface controls for use with *WPF*[26] *(Windows Presentation Foundation)*, and a driver to provide input to Windows 7 via the framework (see section 4.4.2). Multi-Touch Vista installs its core module as a Windows service which can be configured through a configuration interface by a dedicated application.

The framework is well designed and flexible. It makes use of multiple technologies (WPF, WCF, contracts, unmanaged code, native Windows API, Windows services, etc.), design patterns (e.g. dependency injection), and consists of several separate modules which are barely documented. Thus for programmers with few experience it is hard to understand the internals of the framework. However, it is very easy to use as an application programmer because it integrates into Microsoft Visual Studio, can be configured using *XAML*[27], and can be controlled using few lines of source code.

4.3.8 multitouch

The *multitouch*[28] framework appears to be in an incomplete testing state, which is also stated on the project website. Its focus is graphical output of the acquired input data and it is able use Apple *QuickTime*[29] for image capture from video cameras. Most of the functionality of multitouch is hard-coded in the included examples.

4.3.9 MultiTouch.framework SDK

The *MultiTouch.framework SDK*[30] [51] aims to be a standard solution for multi-touch enabled applications on the Mac OS X platform. It focuses on connection to external multi-touch devices and provides a simple and unified interface for application developers. Input devices are connected by *Input Modules* which are realized as plug-ins for the framework. It currently supports the Apple iPhone and iPod as input devices as well as camera-based input.

The framework is currently redesigned by its authors and extended for use with tangible widgets [52]. It is planned to release the redesigned version and the source code to the public but currently neither a release schedule nor a license model have been determined.

4.3.10 multitouchframework

The scope of *multitouchframework*[31] is to provide a set of WPF controls for a multi-touch GUI. It takes input data from a data source which sends TUIO commands and translates them for its WPF controls. The framework recognizes a predefined set of gestures. Since it is written using C#, scripting can be done by Microsoft PowerShell which is able to access any .NET object.

No source code is provided with the framework except some usage examples. It integrates into Microsoft Visual Studio and Microsoft Expression Blend for

26 http://windowsclient.net/wpf/
27 http://msdn.microsoft.com/en-us/library/ms752059.aspx
28 http://code.google.com/p/multitouch/
29 http://www.apple.com/en/quicktime/
30 http://hci.rwth-aachen.de/multitouch
31 http://code.google.com/p/multitouchframework/

development. An application developer can use the provided user interface controls and a multi-touch input processing class to create own applications.

4.3.11 OpenTouch

The *OpenTouch*[32] framework was created for Google Summer of Code 2007, a programming competition for students. The framework is very small and has few features and only rudimentary functionality. It seems to be in an extremely experimental state and does not include useful examples. Therefore, it is hard to determine which functionality and interfaces were planned to be realized. It uses parts of *reactiVision* (see section 4.5.2), the *SDL*[33] library, and *OpenCV*[34] for video camera access and image processing.

4.3.12 pyMT

The *pyMT*[35] framework concentrates on rapid development of multi-touch user interfaces. It is written with the Python scripting language, comes with a set of over 30 widgets and 32 examples which is a lot compared to other projects. The framework supports OpenGL and is well documented.

PyMT can process input data from any data source providing multi-touch input data but the only implemented data source is TUIO. Usage of pyMT is very simple. A multi-touch application using two widgets which visibly reacts on a touch event can be written with less than 10 lines of source code[36].

4.3.13 Sparsh UI

Sparsh UI[37] [53] aims at support of multiple different multi-touch input devices and has its focus on gesture recognition. Its implementation is separated into a stable and an experimental branch.

The stable branch consists of a gesture server implemented in Java which supports nine gestures and includes some example applications. The gesture server receives multi-touch input coordinates and events via a socket connection using a proprietary protocol and recognizes gestures from the input data. The gesture server sends its results to an application via another socket connection using another proprietary protocol.

The experimental branch provides interfaces to some multi-touch input data sources such as *Touchlib* (see section 4.3.15). Additionally, a source code skeleton for a new implementation of the gesture server in C++ is included. Overall, Sparsh UI leaves a subjective impression of a not yet completed patchwork project which is in a process of continuous change and is mixed from different programming languages and styles.

32 http://code.google.com/p/opentouch/
33 http://www.libsdl.org/
34 http://opencv.willowgarage.com/wiki/
35 http://pymt.txzone.net/
36 http://pymt.txzone.net/docs/api/tutorial-introduction-tut2.html
37 http://code.google.com/p/sparsh-ui/

4.3.14 TouchKit

TouchKit[38] from *NOR_/D Labs* has been designed for the multi-touch hardware products of the company. The lightweight framework is realized as an extension for OpenFrameworks and extends the application class of OpenFrameworks to provide multi-touch input events. It uses OpenCV for image processing and supports OSC through another included extension for OpenFrameworks. Moreover, TouchKit captures images from a video camera or uses a video file as image source, both features are provided by OpenFrameworks.

4.3.15 Touchlib

Touchlib[39] is one of the few frameworks which use a Pipes and Filters architecture. The surrounding filter graph management is done by some classes which have a layer architecture as well as the filters itself. Touchlib stores and restores the entire filter graph configuration in and from an XML file by iterating over all filters and their set of parameters. The framework contains filters which use *DSVideoLib* or *VideoWrapper* to capture images from a video camera. OpenCV is used for image processing and *OSCPack* is used to send multi-touch input data with TUIO.

Touchlib comes with some sample applications and a configuration application which assists the user in calibrating a multi-touch surface. Additionally, some flash applications controlled by TUIO input data are included.

An application which uses Touchlib can be created by using one of the included sample applications and manual modification of the XML configuration file. An application developer can use the API to create and control one or multiple filter graphs, too. The filter graph provides multi-touch input events either by explicit polling from an internal queue or through a listener interface which can be registered by the application which uses Touchlib.

4.3.16 touchpy

touchpy[40] is a very small framework for the creation of multi-touch user interfaces and is written with the Python scripting language. It does not provide much more functionality than mapping TUIO input data to an event signaling mechanism which can be used to control a Python application. A remarkable supplementary feature of touchpy is the included plug-in for the *Compiz* window manager which enables multi-touch user input for the Linux desktop.

4.3.17 Touché

Touché[41] can be used on the Mac OS X platform only because it is based on the Apple Cocoa framework. It offers a comfortable configuration GUI and a calibration assistant for multi-touch surfaces. Video images can be captured from cameras supported by

38 http://labs.nortd.com/touchkit/
39 http://www.touchlib.com/
40 http://code.google.com/p/touchpy/
41 http://gkaindl.com/software/touche

Apple QuickTime and *libdc1394*. Besides video cameras, the Nintendo Wiimote is supported as input device. Moreover, gesture recognition is supported and three sample applications are included.

Touché consists of a considerable amount of source code which has been carefully written. It combines multiple software architectures and is clearly structured. The framework uses a filter chain for image data processing, a layered class hierarchy for all other tasks, and acts as a stand-alone server application to which clients can connect to receive multi-touch input data. Application clients can connect to the server either using client classes from the Touché framework or using TUIO. Flash applications can connect using a Flash XML protocol or the Flash Local Connection interface.

For fast image filtering, Touché comes with a set of 14 image filters written in GLSL (OpenGL Shading Language) which are executed on the GPU. Blob detection is done using OpenCV. The framework uses multi-threading and executes blob detection as well as data transmission to clients in separate threads.

4.3.18 xTouch

The *xTouch*[42] framework has not been released under this name yet. Its current name is *BBTouch* which is still available as a part of the *OpenTouch* source code distribution (see section 4.3.11). It is implemented using the Apple Cocoa framework and supports the Mac OS X platform only.

BBTouch uses a model-view-controller architecture and includes a small demo application. The framework captures video images using QuickTime and contains a fixed image filtering procedure which is based on OpenCV. Tracked blob coordinates can be sent to client applications using TUIO. BBTouch also includes classes for customized OSC encoding and decoding.

xTouch shall support multiple cameras and fiducial markers. The author plans to integrate faster image filters, too.

4.4 Multi-Touch Interaction (Commercial)

In contrast to the projects presented in the preceding section, this section describes commercial products or parts of commercial products.

4.4.1 Microsoft Surface SDK

The *Microsoft Surface SDK*[43] is designed to work with the *Microsoft Surface* multi-touch table only and is well documented. It supports two gestures (tap, press and hold) and six manipulations resulting from these gestures (drag, pan, flick, move, resize, rotate). 24 multi-touch-enabled user interface controls are included.

A developer of an application based on the Surface SDK can choose to use the *Core layer* where he has to handle multi-touch events and implement user interface controls

42 http://benbritten.com/category/multitouch/
43 http://www.surface.com/

on his own. The WPF layer comes with multi-touch-enabled user interface controls and allows development using XAML and Microsoft Expression Blend.

4.4.2 Microsoft Windows Touch SDK

Microsoft Windows Touch[44] in integrated into the Windows 7 operating system, the associated SDK is provided for free to Windows developers as part of the Windows SDK. Windows Touch supports a slightly extended set of gestures compared to the Surface SDK and does not include multi-touch-enabled user interface controls.

Applications which want to use multi-touch input can receive Windows messages over a standard Windows message queue. These messages signal multi-touch events (*WM_TOUCH*) and detected gestures (*WM_GESTURE*). The pan, press and hold, and zoom gestures are additionally mapped to standard Windows messages to allow scrolling and simulated mouse interaction for applications which are not aware of multi-touch input. Furthermore, the SDK offers a set of interfaces which enable a more comfortable way for a developer to process the multi-touch input messages.

Windows Touch does not do image processing, it requires pre-processed touch coordinate data and contact events. To provide multi-touch input data to Windows Touch, an input device must support the Windows driver interface for *human input devices (HID)*. A driver which simulates a HID must be implemented using the Windows Driver SDK to feed in data without a hardware device but by using software. The *MultiTouch Vista* framework includes such a driver (see section 4.3.7).

4.5 Tangible Interaction

Tangible interaction refers to tangible objects for HCI which are commonly placed on a surface monitored by the computer. Therefore, it is closely related to multi-touch interaction. Depending on their type, tangible objects can optionally operate independently from a surface. Thus tangible interaction is also close to multi-modal interaction.

4.5.1 Papier-Mâché

The *Papier-Mâché* [12; 54; 55] toolkit is primarily designed to handle input directly generated by physical objects but it also includes interfaces to photo and video cameras for image input. Physical objects and objects detected on images are abstracted to multiple hierarchically organized types of *Phobs*. The interface to video cameras is provided by JMF (Java Media Framework) and image processing is done with JAI (Java Advanced Imaging). The toolkit includes implementations of Phobs for barcodes and RFID tags.

Papier-Mâché[45] uses a complex system for input abstraction. *Association Factories* match Phobs with *Classifiers* and produce different types of *Association Elements*

44 http://msdn.microsoft.com/en-us/library/dd562197(VS.85).aspx
45 http://hci.stanford.edu/research/papier-mache/

when a Classifier matches a Phob. Classifiers define decision criteria and can be applied to Phobs. They determine if they can operate on a particular Phob type and if their criteria match a particular Phob instance. Association Elements split into *Association Nouns* and *Association Actions* which form the application interface for mapping physical input to application logic. Nouns act as object selectors and Actions are applied to selected Nouns. Partially, the implementation of this architecture does not clearly separate concerns and is a little difficult to comprehend.

The integrated image-based object recognition process is based on background subtraction, edge detection, and connection of components. When this process is used together with a common camera-based multi-touch setup, it produces a result comparable to simple blob detection. The image processing is based on Java and does not focus on performance. Thus Papier-Mâché might be used for multi-touch input processing without larger modifications when processing performance is no crucial objective.

4.5.2 reacTIVision

The *reactTIVision*[46] framework [56] recognizes and tracks fiducial markers. It is part of the *reactable* project, a collaborative electronic music instrument with a tabletop tangible multi-touch interface comparable to *Xenakis* (see section 4.5.3). Its authors are also the inventors of TUIO [46]. The framework offers robust tracking of fiducial markers and includes rudimentary support for multi-touch input. It supports input from a single camera and comes with a set of fiducial markers. Recognition results are published by the framework using the TUIO protocol.

The reacTIVision framework incorporates four selectable tracking engines, each specialized on a particular type of fiducials. Image acquisition from video cameras is done by *PortVideo*, a platform independent library. The fiducial recognition algorithms and the TUIO implementation are located in separate libraries, too. TUIO client implementations for ten platforms are included.

4.5.3 TWING

The *TWING*[47] framework [57] has been developed for *Xenakis*[48] [58], a music creating tangible interface comparable to reactable (see section 4.5.2). It supports tangible and multi-touch input of 2D coordinate data and object identifiers via a tracker interface. TWING comes with implementations for two proprietary trackers (*Jammt* and *MatraX*). A key feature of TWING is the interpretation of input data and issuing of explicit user commands. The framework provides an interface where applications can connect to receive such user commands. Four demo applications are included. Moreover, the framework contains implementations of graphical controls for Windows GDI, the Microsoft XNA framework, and the open source 3D rendering engine *Horde3D*.

46 http://reactivision.sourceforge.net/
47 http://xenakis.origo.ethz.ch/
48 http://xenakis.3-n.de/

TWING uses a MVC architecture (see section 2.3.2). Its source code has exceptional good quality in terms of software design, structuring, readability, consistency, and documentation. TWING also distinguishes itself by an elaborate input data interpretation architecture (see section 8.2) which supports users in extending the framework with new interaction techniques. Since TWING is written using C#, scripting can be done by Microsoft PowerShell.

5 Compact Comparison of Frameworks, Libraries, and Toolkits

The preceding section introduces each related project but makes it difficult to oversee and compare all projects. Therefore, this section presents a compact survey for comfortable comparison including more details of each project.

Subsequent to this comparison, one of these projects is selected as basis for Curve research (see section 1.1) and reasons for the selection are given.

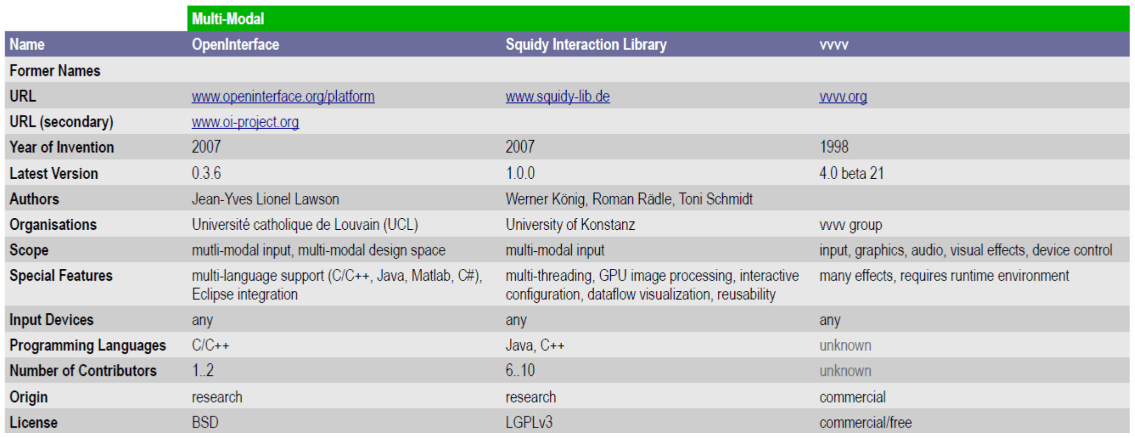
5.1 Tabular Survey

The following tabular survey lists a selection criteria which have been chosen based on the requirements for an input framework (see section 3). The survey also gives generic information about each project and includes information about the project's extent as well as it rates the quality of its implementation from a subjective point of view.

Many of the criteria are assigned scores depending on the particular value chosen for a criteria. The scores are chosen to correspond to the weighting the author of this work wants to give to a specific criteria to reflect its importance. Thus the used scoring is targeted to the requirements of the Curve project and includes a portion of subjectivity.

The individual scores are summed up to an overall score. This overall score gives a clue which project fits best for the the requirements of the Curve project. The electronic version of this work is being accompanied by a worksheet which allows one to adapt the individual scores to his requirements. It also recalculates the overall score.

The complete survey can be found in the Appendix of this work.



	Multi-Modal		
Name	**OpenInterface**	**Squidy Interaction Library**	**vvvv**
Former Names			
URL	www.openinterface.org/platform	www.squidy-lib.de	vvvv.org
URL (secondary)	www.oi-project.org		
Year of Invention	2007	2007	1998
Latest Version	0.3.6	1.0.0	4.0 beta 21
Authors	Jean-Yves Lionel Lawson	Werner König, Roman Rädle, Toni Schmidt	
Organisations	Université catholique de Louvain (UCL)	University of Konstanz	vvvv group
Scope	mutli-modal input, multi-modal design space	multi-modal input	input, graphics, audio, visual effects, device control
Special Features	multi-language support (C/C++, Java, Matlab, C#), Eclipse integration	multi-threading, GPU image processing, interactive configuration, dataflow visualization, reusability	many effects, requires runtime environment
Input Devices	any	any	any
Programming Languages	C/C++	Java, C++	unknown
Number of Contributors	1..2	6..10	unknown
Origin	research	research	commercial
License	BSD	LGPLv3	commercial/free

Illustration 2: Extract from the tabular Survey of Input Libraries, Frameworks, and Toolkits

5.2 Pleading for the Squidy Interaction Library

The *Squidy Interaction Library* (in the following only *Squidy*) reaches the highest score of all examined projects. Thus it is a potential candidate to choose as basis for Curve research. Squidy is indeed by far the most appropriate candidate for this task. This assertion has to be justified because it is not obvious.

Since the survey cannot disclose all details and features of a project, a selection of the advantages of Squidy is listed in the following to prove Squidy's qualification.

- Squidy is platform independent because it is implemented on the Java platform. This means it can run on many different hardware and software platforms.
- Squidy includes a GUI for visual programming. This enables fast development and testing and makes it very easy for inexperienced users to create new input configurations.
- Squidy offers the unique feature of interactive graphical feedback of input data as well as interactive development and source code editing within the GUI even during active processing of input data. Thereby it allows quick iterative development which is ideal for research and rapid prototyping.
- The software architecture of Squidy allows to build reusable modules and to employ them in a very flexible way. The level of abstraction of these modules is chosen to match well with the requirements of HCI research. It obviates the need to care for small details for beginners but still allows versatile and in-depth development for experts.
- The software architecture of Squidy is well designed and its source code has been carefully written by experienced programmers. This makes Squidy easy to maintain and extend.
- Squidy already includes a set of 46 modules for input devices, data filtering, data conversion, remote connection, etc. This set of modules already covers many use cases and reduces the need for new development efforts.
- Squidy has successfully proven itself in multiple research projects of its authors.
- Last but not least, Squidy is very actively maintained and cooperation regarding Squidy between its authors and the Curve research team showed to be very beneficial and pleasant.

As mentioned before, this is only a selection of the characteristics of Squidy. None of the other projects reviewed for this work can keep up with Squidy's amount of advantages. They either provide none or only few of them.

For these reasons, Squidy is selected as basis for the Curve project. All subsequent parts of this work present and discuss mainly technical aspects and possible further improvements to Squidy.

6 The Squidy Interaction Library

The *Squidy Interaction Library* is a framework for the design, development, and evaluation of multi-modal data input processing tasks. The term *Library* does not refer to the software architecture of Squidy because in terms of software development it is not a library but a framework. Squidy provides a set of ready-to-use device interfaces, filters and interaction techniques for HCI. The term *Library* refers to this set.

Squidy has been designed with a focus on usability, rapid prototyping, iterative design, employs visual data flow programming and allows visual debugging. Additionally, the framework is built using a modular software design, makes use of loosely coupled components, allows parallel data processing, and its implementation is platform independent. Thus it is flexible and extensible and can take advantage of modern multi-processing hardware architectures. The various benefits, influences, design principles and new inventions of Squidy are described in detail by the inventors of Squidy [5-7].

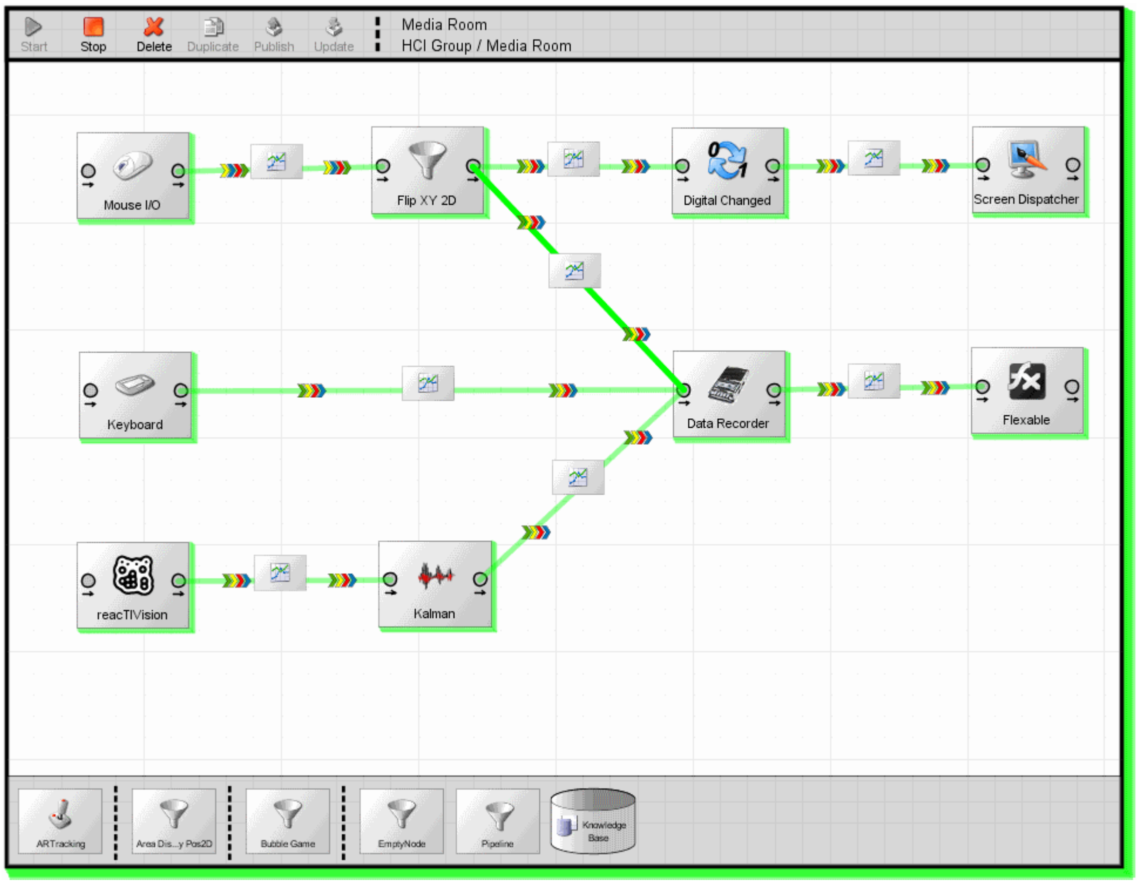


Illustration 3: Squidy Designer, the GUI of Squidy for visual programming

The following discussion concentrates on technical details and internals of the framework which have not yet or only partially been described by its authors. It mainly characterizes concepts and implementation details which are relevant for understanding the framework itself rather than for a user of the framework. Moreover, by comparison with a long-term proven implementation of the employed Pipes and Filters architectural pattern (see section 2.3.2), several approaches for further improvements are pointed out.

At the time of writing (summer of 2009), all vital parts of Squidy were implemented and running and the framework has already been used for various research projects at the University of Konstanz in Germany, where it has been invented. It was not yet completed, several technical solutions have been subject to change and improvement, and work on Squidy is continuing to realize all plans and ideas. An overview of some of these plans are presented in section 6.6.

On September 17 of 2009 the Squidy Interaction Library has been published as an open source project licensed under the *GNU LGPLv3*[49] *(Lesser General Public License 3.0).* It had been a closed source project of the Human-Computer Interaction Group at the University of Konstanz before.

Since one of the basic requirements for the Curve project is the support of multi-touch input (see section 3), it is important to mention that Squidy already contains a module for multi-touch input. The module has proven itself in practice but lacks in multiple important features compared to the dedicated multi-touch frameworks introduced before. This issue is discussed in detail in the sections 7.5, 7.6, and 7.7.

Squidy is implemented in the Java programming language and has been successfully tested on Microsoft Windows, Mac OS X, and Linux platforms by its authors. The preferred development environment is the *Eclipse*[50] development platform but it loads and compiles successfully in *NetBeans*[51], too. The Maven project management tool is used for automatic building. Squidy makes use of several external libraries and technologies such as *Java Architecture for XML Binding*[52] *(JAXB)* for storage, or the Piccolo framework for 2D visualization and zooming.

The Squidy Interaction Library is divided into three logical parts which are described in this chapter: *Squidy Core*, *Squidy Bridges*, and *Squidy Client Implementations*.

6.1 Squidy Core

Squidy Core consists of two integral parts, *Squidy Manager* and *Squidy Designer*. Squidy Manager is responsible for all kinds of data and process management as well as serialization. Squidy Designer is used to create, modify, control, visualize and debug the data processing configuration of Squidy.

Squidy's data processing uses a Pipes and Filters architecture (see section 2.3.2). Its filters are called *Nodes* which can be connected using *Pipes* and thereby grouped to the *Pipeline*. The data flow within the Pipeline chain is unidirectional.

49 http://www.gnu.org/licenses/lgpl.html
50 http://eclipse.org/
51 http://www.netbeans.org/
52 http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

6.1.1 Squidy Manager

Squidy Manager provides the platform for all tasks performed by Squidy. When Squidy Manager is executed, it runs invisibly to the user.

Squidy Manager is designed to work independently. This means it can run without Squidy Designer and can process pre-configured Pipelines. Currently, the stand-alone feature needs to be revised, see plans in section 6.6. Moreover, Squidy allows multiple instances of the Squidy Manager to communicate with each other. This is realized by a component called *Squidy Remote* which is explained in section 6.2.11.

6.1.2 Squidy Designer

Squidy Designer provides a GUI to create, modify, control, and debug Pipelines. It uses visual representations of the *Workspace*, *Pipelines*, *Nodes*, and *Pipes* which are called *Shapes*. These items are introduced in the next section 6.2.

Squidy defines only one single global *Workspace* for the user which is also reflected in the user interface of Squidy Designer. The GUI employs the concept of *semantic zooming* for all visual representations [59; 60]. According to Squidy's authors, "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)." (p.4 in [5])

Squidy Designer requires Squidy Manager to be executed because it directly relies on its functionality and only provides a visual representation of the data managed by Squidy Manager.

6.2 Concepts in Squidy Core

6.2.1 Processable

Pipes, Nodes, Pipelines and the Workspace are individual classes. They are derived from the class *Processable* which incorporates methods to do any kind of processing. Processing itself is implemented using threads to permit asynchronous and independent execution of each Pipeline or Pipeline segment. The object hierarchy of Processables looks as follows:

```
Processable
  Pipe
  Piping
    Node
       Pipeline
       Workspace
```

A hierarchical structure between Processables is created by a collection of any number of *sub-Processables* which can be assigned to a Processable. Sub-Processables are controlled by their parent Processable. Thereby processing of sub-Processables starts and stops automatically if processing of the parent Processable starts or stops.

The Processables and sub-Processables of a Squidy Workspace are structured by the following basic hierarchy:

```
Workspace
  Pipeline[s]
    Pipeline[s]
    Node[s]
    Pipe[s]
```

6.2.2 Pipeline

The *Pipeline* class acts as a container for the processing chain which normally consists of Nodes and Pipes. Multiple instances of Pipelines can be created. These can be connected by Pipes, too. Since a Pipeline is also a Processable, it inherits the management of sub-Processables. And it can contain other Pipelines which can be integrated into the processing chain as well.

This principle allows to group functionality hierarchically. A Pipeline behaves and can be used like a Node if it is observed from the group perspective. Squidy Designer visualizes the contents of a Pipeline or its Node representation by semantic zooming, depending on the zoom level. A similar concept called *Patches* can be found in *vvvv* (see section 4.2.3).

6.2.3 Nodes

Nodes are elementary components of the architecture. They can be combined by the user to provide the desired input functionality. A Node can for example represent an interface to a device, a *Bridge* to a communication protocol or a data filter. Particular examples for existing Nodes in Squidy are Wiimote, iPhone, TUIO protocol or Kalman filter.

Squidy comes with a set of basic Nodes. However, a user can easily implement his own Nodes based on the class *AbstractNode*. He does not need to know much about the internals of the framework but mainly needs to implement the method *process()* within his new Node class derived from *AbstractNode*. Multiple overloaded implementations of the *process()* method can be provided by the user to handle different data types. These overloaded methods are detected by the reflection mechanisms of Java and automatically invoked by the *AbstractNode* class.

In Squidy Designer, the functional state of a Node is visually represented by color coding: gray means stopped, green means active, and red indicates a malfunction.

6.2.4 Pipes

Pipes are the connections between Nodes. A Pipe holds a list of supported incoming and a list of supported outgoing data types. Transmission of certain types of data between Nodes can be restricted by data type filtering which is done by removing data types from the list of supported types.

Processing of the Pipe's data occurs in between the data type filtering. A Pipe does not modify data but provides an interface for graphical real-time data feedback in Squidy Designer which allows visual debugging.

In Squidy Designer, the functional state of a Pipe is visually represented similar to a Node. The visual representation also contains an icon for access to the visual debugging view.

6.2.5 Piping

The *Piping* class is an abstract base class for the actual data processing classes. It contains two collections, one for incoming and one for outgoing Pipes. These collections are propagated to the derived classes. Thus Nodes, Pipelines and the Workspace can manage incoming and outgoing connections through their attached Pipes.

The connection points of a Piping object are called *ports*. Each Piping object has one incoming and one outgoing port where Pipes can connect.

6.2.6 Data Representation

Within an instance of Squidy, data is transmitted within native Java objects. These objects share the common interface *IData*. The properties of this interface consist of a pointer to the object's source Node, a time stamp and any number of attributes of any Java supported object type. The attributes are type safe as their mapping between type and value is managed by the dedicated class *DataConstant*. The time stamp is set on instantiation of a data object.

The class *AbstractData* implements *IData* and the common functionality of all data types such as management of attributes, time stamp and basic serialization. All specialized data classes are derived from *AbstractData*, for example *DataPosition2D* which stores 2D coordinates.

The data types of Squidy form a hierarchical structure which is designed to support all kinds of input devices. This hierarchy provides generic types such as a string and highly specialized types for example for an entire hand. Data types are grouped into semantic contexts to assist interpretation and processing of data.

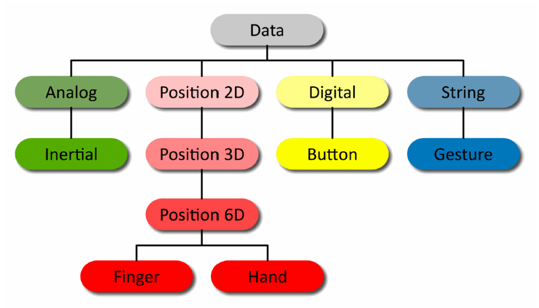


Illustration 4: Data type hierarchy of Squidy

The idea behind this hierarchy is to provide a simple structure of basic data types which are easy to handle and to understand for a user of Squidy. Complex custom data types can also be integrated but a developer should place them in a separate branch in the hierarchy to clarify that these types are for special purposes and are not required for most tasks in Squidy.

If a type of data is needed which is not yet implemented, one can create a new class derived from *AbstractData*. The new data type can be used and transmitted within the entire framework without any further modifications. Of course, suitable data processing must be implemented if a Node should process a new type of data.

6.2.7 Data Processing

Java Reflection is used to determine which data types a Node is able to process. Therefore, all overloaded implementations of the method *process()* which processes incoming data are enumerated by the *AbstractNode* class on itself. Thereby user-defined data processing methods of a derived class are found, which can process the current data object. The *AbstractNode* class stores these methods in a map for caching to accelerate further invocations.

If a Node needs to publish multiple types of data at once, a data container is required to wrap the individual data objects. *IDataContainer* and its implementation *DefaultDataContainer* define such a container for multiple data objects.

The following pseudo-code illustrates the internal data processing procedure of a Node:

```
publishing data from external Node:
  invoke process(container)
    put container in data queue
      notify processing thread
processing thread:
  while processing enabled
    wait for new container in data queue
      poll container from queue
    invoke beforeDataContainerProcessing(container)
    for each data in container
      look for matching process(data) overload of derived class
        invoke matching process(data) of derived class
      store processed data in container
    invoke afterDataContainerProcessing(container)
    publish container
      for each Pipe in outgoing Pipes
        invoke Pipe.process(container)
        get target of Pipe
          if number of Pipes > 1 then clone container
          invoke target.process(container)
```

6.2.8 Persistence

All objects within and including the Workspace are serializable. Squidy uses JAXB for automatic serialization of class attributes. This allows to store and restore the entire object hierarchy to and from an XML file with only a few lines of additional code: An annotation containing the tag name for its XML representation needs to be added to each class attribute which should be serialized.

6.2.9 Dynamic Reconnection

In HCI research and evaluation it is often necessary to modify the input chain to test new devices and different setups. Most input frameworks require data processing to be offline or in halted state to modify essential parts of their data processing. This can be time-consuming during daily use. Squidy Designer allows to change data processing on-the-fly while data input is active and running. Not only configuration parameters of modules can be modified but the entire data processing chain can be rearranged, modules can be added or removed. Data processing is only interrupted as long as it takes to accomplish the desired modification by the user and continues automatically as soon as the user has reconnected all involved modules.

This feature is implemented in the Processable class by two collections of incoming and outgoing Pipes. During the reconnection process, these collections are updated accordingly. An existing Pipe can be deleted and new pipes can be created. Squidy Designer does currently not support to reconnect an existing Pipe to a different source or target. Since the publishing process of a Node enumerates all items of the collection of outgoing Pipes every time it publishes a data object, all modifications of this collection become effective immediately. The collection of incoming Pipes is currently not used during the reconnection and publishing processes.

6.2.10 Dynamic Compilation

A *direct manipulation* [61] interface allows the visual interactive configuration of properties in the user interface by using control elements which results in immediate feedback for the user. Squidy allows this kind of direct manipulation of Nodes. An extension to this functionality is enabled by the integrated feature of Java source code editing and *dynamic compilation*. Squidy Designer provides a user interface to edit each Node's source code directly within its GUI and compiles and integrates the modifications, also when input processing is active. Thus smaller code corrections can be applied very fast without the need to switch to a development environment and to interrupt the entire workflow.

To enable dynamic compilation, Squidy requires a Java compiler which is commonly included in a Java Development Kit (JDK).

Dynamic compilation is implemented using a repository for dynamically compiled code which is managed by two custom implementations of Java class loaders. When a class is requested by the virtual machine, the *DynamicCodeClassLoader* class checks if the source code for this class has been modified compared to the compiled version in the code repository. If necessary, the class is recompiled by the installed Java compiler before it is loaded. Alternatively, the *HotDeployClassLoader* class can be used which does not need a Java compiler to be installed. It checks the repository for newer versions of a class and reloads it if it detects a newer version.

6.2.11 Squidy Remote

Several instances of Squidy Manager can connect to each other using a Node called *Squidy Remote*. This Node represents a Bridge (see section 6.3) which translates between Squidy data objects and the OSC protocol. The OSC protocol consists of plain ASCII text and can be transmitted as a data stream via network interfaces. This concept enables distributed processing of input data.

6.2.12 Data Recorder

The *Data Recorder* Node included with Squidy allows to serialize and deserialize a stream of data objects. Each data object contains adequate methods for serialization and deserialization to and from a string representation. These methods are invoked by the Data Recorder. Data is stored in a file and can be replayed on demand while retaining the original time span between data objects. Using this Node, specific interaction tasks can be recorded for analysis and can be replayed for development, testing, debugging and simulation purposes.

6.3 Squidy Bridges

A *Bridge* connects Squidy to any kind of data transmission protocol or provides external connection to any kind of data interface different to its native internal interfaces. The purpose of a Bridge is to translate between an external data representation and the generalized internal data representation of Squidy. Squidy Bridges follow the common design pattern of a *bridge*.

Bridges are a separate logical part of Squidy because they differ from the elements in the processing chain. A Bridge has no universal purpose but only the purpose of data mapping. To integrate a Bridge into a Pipeline, a Node has to be created which represents the Bridge. The current implementations of Bridges are entirely integrated into Node implementations.

A Bridge Node does not solely operate on generalized internal data types. Thus it is normally located at the beginning or at the end of a Pipeline to perform data mapping operations. If a Bridge Node is integrated in the middle of a Pipeline, incoming data objects are also routed to the Node's output port besides processing by the Bridge itself.

6.4 Squidy Client Implementations

A *Squidy Client Implementation* is an application or part of an application which runs on a user client platform and communicates with Squidy. Communication is realized by a communication protocol suitable for the application. Translation between this protocol and Squidy data objects is done by a matching Squidy Bridge.

An example for a Squidy Client Implementation could be an application running on the Apple iPhone which captures multi-touch input data generated by the user and transmits it to a PC running Squidy via WLAN using the TUIO protocol. The TUIO input data can be processed by Squidy's TUIO Bridge.

6.5 Performance Considerations

If some kind of input processing includes computationally intensive tasks, the Java programming language and its virtual machine might not be the ideal choice. However, multiple benchmarks[53,54,55] prove that Java virtual machines can reach the performance of implementations in native code depending on the evaluated algorithm or set of functions. These benchmarks compare a Java implementation to functionally equal code written in a lower-level programming language such as C or C++. This code is compiled to native code for the underlying platform by an optimizing compiler. If translation of the evaluated code into this language from Java is done by a skilled programmer, the benchmarks show that the native code performs equally or outperforms the Java code. The difference in performance depends on the skills of the programmer and on the evaluated algorithm or set of functions.

Small tasks such as data flow control can be handled easily in Java and do not have noticeable performance impacts. Image processing might require a fast implementation in a lower-level programming language to reduce latency in a processing chain as much as possible. Squidy can make use of the *Java Native Interface*[56] *(JNI)* to delegate such tasks to modules running in native code on the employed hardware platform.

Today's computers (2009) provide even more processing power on their graphics adapters compared to the processing power of their CPU. Graphics adapters are driven by a GPU which is designed for fast processing of 3D vertices and pixel data. A GPU consists of several hundred logical processing units (*streaming processors*) which can process large numbers of image pixels in parallel. This hardware architecture also qualifies for conventional 2D image processing where many image pixels have to be analyzed or modified: GPUs can take advantage of their distinct parallel processing compared to the single or few processing cores of a conventional CPU. Consequently, Squidy includes a multi-touch input module which utilizes GPU computation for blob tracking.

Using a GPU for purposes other than their originally intended usage is labeled *GPGPU*[57] *(General Purpose Computation on Graphics Processing Unit).*

6.6 Planned Features

The following list outlines features, ideas, and goals of the authors of Squidy which should be integrated in future versions of the framework. This list is not considered to be complete and cannot be documented by references as it evolved from communication and discussion with the authors of Squidy.

53 http://bruscy.republika.pl/pages/przemek/java_not_really_faster_than_cpp.html
54 http://www.freewebs.com/godaves/javabench_revisited/
55 http://www.stefankrause.net/wp/?p=9
56 http://java.sun.com/javase/6/docs/technotes/guides/jni/
57 http://gpgpu.org/

- Documentation of Squidy is only minimal. Since detailed documentation is an essential part of a software development project, missing documentation for source code and individual Nodes included in the Squidy Interaction Library is planned to be provided soon.
- Due to foregoing refactoring processes, the standalone operation mode of Squidy Manager has been disabled and has to be re-enabled. Currently, Squidy Manager can operate only in conjunction with Squidy Designer.
- A Node publishes its data objects directly to its target Nodes by requesting them from its Pipes. Publishing data to the Pipe which contacts its target on its own would encapsulate the publishing process.
- Logging of messages is realized with the Apache *log4j*[58] framework. It shall be replaced or at least wrapped by *SLF4J*[59] *(Simple Logging Facade for Java)*. Besides abstraction from the actually used logging framework, SLF4J brings a feature called parameterized logging which reduces processing costs when logging is disabled.
- Data objects are created by the *new* operator on demand. When a data object is not needed anymore, the garbage collector of Java frees the memory allocated for it. An object pool (see section 2.3.1) shall avoid this continuous process of creation and destruction and reduce processing costs.
- In Squidy's current software architecture, a *Shape* is the visual representation of a Processable. The Shape is directly connected to the Processable itself. This violates the programming paradigm of *separation of concerns* which forbids to blend different functionality within one module. A Shape controls modifications of its Processable. For this reason the concern *visualization* could not be removed or modified without losing or influencing the concern *control*. In this case, the model-view-controller architectural pattern shall be applied. A controller class shall be put in between the Processable (model) and its Shape (view) to encapsulate the concerns and to remove their dependency.
- The JAXB framework should be replaced by the *Hibernate*[60] persistence framework, which relies on a relational database and therefore enables centralized and distributed data storage (see section 6.6). JAXB should not be replaced entirely but should be retained for data backup purposes and as data exchange format.
- It is planned to integrate version management of Nodes directly into the framework. Source code of Nodes shall be managed directly by the *SVN*[61] version control system. Versioned properties of Nodes should be managed within the database on which the Hibernate framework operates. Possible implementation approaches consider the Hibernate interceptor mechanism which allows intervention in the serialization process or use of a version property which Hibernate provides for each serialized attribute.

58 http://logging.apache.org/log4j/
59 http://www.slf4j.org/
60 http://www.hibernate.org/
61 http://subversion.tigris.org/

- To support a quick and iterative development process directly in the user interface of Squidy Designer, it shall be possible to duplicate Nodes. Duplication includes the values of properties of the Node as well as its source code. Modification of the duplicate shall not affect the original Node. A Node shall be able to coexist in different binary versions compiled from different versions of its associated source code. Version management shall be provided transparently for the user by Squidy. This shall be realized using the integrated version control system interface and version information contained in the name of the source code file.
- Squidy is typically distributed to the user in a Java archive file. This packaged file format requires additional effort and processing power to integrate dynamically generated code into the existing code base. The Java archive file must be updated with the newly compiled classes. As a possible approach to solve this problem, an external code repository outside the distribution package is planned.
- Multi-user support of Squidy is currently reduced to multi-device support. Each data object owns an attribute which identifies its source Node. This allows to determine the physical data source but not the user of this source. A dedicated user identification mechanism shall be integrated. For multi-touch interaction, research of identification of different person's hands is in progress.
- Integration of Squidy Manager and Squidy Designer in Squidy Core is realized using native Java interfaces. The two parts should be decoupled to connect via IP. This enhances Squidy Designer's functionality as a graphical management interface which can remotely connect to instances of Squidy Manager. It allows to distribute Squidy Manager to multiple different execution environments while the user can retain control over all Squidy Manager instances from a central workstation.
- All Shapes in Squidy Designer have a fixed visual size except the possibility to make use of semantic zooming. It shall be possible for the user to resize at least the Shape representing a Pipeline. There is no concrete concept how to realize this feature yet.

6.7 Comparison of Squidy with DirectShow

DirectShow[62] [31] is the subsystem for streaming media of Microsoft Windows. It was part of *DirectX*[63], a set of multimedia APIs for Windows, but was moved by Microsoft in 2005 to become part of the common Windows APIs. DirectShow uses a Pipes and Filters architecture and processes *media samples* within its filter chain. This architecture has proven itself for many years of practical application in the Windows operating system. DirectShow is primarily designed for media output instead of data input but it contains many concepts similar to those in Squidy and can provide ideas for concepts which are not yet included in Squidy. These ideas and concepts are explained in section 7 and refer to the introduction to DirectShow given in the following sections.

62 http://msdn.microsoft.com/en-us/library/dd375454(VS.85).aspx
63 http://www.microsoft.com/windows/directx/

DirectX includes a subsystem called *DirectInput*[64] for managing input devices, too. It seems to be obvious to compare Squidy to DirectInput because it is also focused on data input, but the two approaches differ substantially. Squidy incorporates a versatile architecture to handle and process any kind of input devices and data. In contrast, DirectInput just concentrates on providing a unified API for established devices including the mouse, keyboard, joystick, and other game controllers, as well as for force-feedback (input/output) devices. Therefore, DirectInput could be used to implement Nodes connecting to physical devices and providing input data to Squidy, of course limited to the Windows platform.

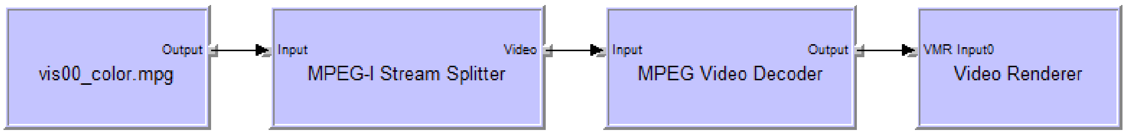


Illustration 5: A Direct Show graph for replay of an MPEG video file

An additional API from Microsoft related to DirectInput is *XInput*[65] on the Windows platform. Compared to DirectInput it is simplified and specialized for the game controller device of the Microsoft Xbox 360 game console.

In this section, general aspects of Squidy and DirectShow are compared. The following sections sometimes refer to further details of DirectShow, too.

6.7.1 Elements of DirectShow

The equivalent to Squidy's Nodes in DirectShow are *filters*, according to the name of the applied software architecture. In DirectShow, the Pipeline equivalent is called the *filter graph*. DirectShow is a very modular and extensible system which is entirely based on *COM*[66] *(Component Object Model).*

The Squidy Core contains Squidy Manager which provides the necessary infrastructure for Nodes and data processing in the Pipeline. In DirectShow, the *Filter Graph Manager* holds the same position, controlling all filters contained in the filter graph. When talking about tasks accomplished by Squidy or DirectShow, this normally refers to these management instances.

DirectShow includes a tool comparable to Squidy Designer, called *GraphEdit*. The tool allows to build, manipulate, control and visualize DirectShow filter graphs. It comes with the DirectShow SDK and is primarily targeted on application developers to test and debug their applications. Therefore, the user interface of GraphEdit neither is designed for end users nor provides a large set of features. A more advanced open source version of a filter graph editor called *GraphStudio*[67] is available, too. Contrary to these tools, Squidy Designer is made for end users and focuses on usability and comprehensive manipulation opportunities.

64 http://msdn.microsoft.com/en-us/library/ee416842(VS.85).aspx
65 http://msdn.microsoft.com/en-us/library/ee416996(VS.85).aspx
66 http://msdn.microsoft.com/en-us/library/ms877981.aspx
67 http://www.monogrammultimedia.com/graphstudio.html

A DirectShow *media sample* is similar to a Squidy data object but it offers only one general interface which is identical for all types of data (*IMediaSample* or the extended version *IMediaSample2*). This differs from Squidy which provides only a minimal common interface (*IData*). All Squidy data types derive from this interface and form a class hierarchy with an extended interface for each class depending on the individual data type.

Squidy represents different data types by different individual classes. In contrast to Squidy, DirectShow data types are named *media types* and are described by a data structure (*AM_MEDIA_TYPE*) which contains meta data only. It consists of a fixed set of properties and an optional format block of variable size depending on the media type specified in the fixed property set. DirectShow media types identify themselves by a pair of GUIDs for *majortype* (e.g. for video) and *subtype* (e.g. for a specific color format). The media type data structure is not connected or transmitted with a media sample, it is used as accompanying information during the filter connection process described below.

DirectShow filters can own none or multiple input and output *pins* which can be compared to the single input and output ports of a Squidy Node. One pin can connect exactly to one other pin and can provide only one media type which is negotiated by the participating filters during the connection process between two pins. Unlike DirectShow, Squidy ports accept all kinds of data types and the attached Node can dynamically consider at runtime if it processes a specific type of data. Furthermore, a Squidy port can connect to multiple other ports simultaneously.

Squidy's ports allow to multiply data and to distribute it on multiple paths. This is not possible in DirectShow, it needs a dedicated filter for this purpose. Due to the concept of media samples and media types, only one implementation of this kind of filter is required which fits for all media types. It needs to accept any media type on its input pin and passes the same media type to all its output pins. The number of output pins needs to adapt dynamically to the required number of outgoing connections. This functionality is implemented in the *Infinite Pin Tee Filter* coming with DirectShow.

6.7.2 Connection Process

The filter connection procedure of DirectShow consists of a complex sequence of requests and confirmations between two filters. This procedure is directed by the Filter Graph Manager in order to find the ideal media type between two pins (output and input) based on their capabilities and preferences. DirectShow also considers and automatically inserts intermediate media type conversion filters which can be taken from an inventory of installed filters. This inventory consists of all DirectShow filters installed on the underlying Windows platform. Each filter has assigned a *merit* which determines if or at what priority it should be considered as an intermediate filter during the connection process.

The filter connection mechanism of DirectShow is also used for automatic *rendering*. Automatic rendering means that the Filter Graph Manager builds a filter graph which renders all media types, provided by a *source filter*, to their default output devices. These devices are represented by *renderer filters*. In case of a video stream this might be the computer display, represented for example by the *Video Mixing Renderer* filter. Other examples for renderer filters are the *DirectSound Renderer* for audio signals or the *File Writer* which stores media samples in a file.

Advantages of Squidy

Connection of Nodes in Squidy does not require a negotiation process because each Node connects with all input data types. Therefore, its entire connection process is less complex than in DirectShow and it is easier to realize dynamic reconnection of Nodes during runtime. Moreover, any number of different data types can be processed by Squidy Nodes without additional costly connection processes.

Disadvantages of Squidy

Squidy offers no automatic Node connection mechanism equivalent to DirectShow. The user has to build a Pipeline manually or construct it by programming a procedure which creates instances of Nodes and connects them. Both approaches are possible with DirectShow, too.

Without a negotiation process in Squidy, automatic construction of a Pipeline is impossible. The user has to know which Nodes can handle which data type. Additionally a small processing overhead is introduced by handling all types of data. A Squidy Node has to inspect the incoming data type and distinguish further processing for each data sample again and again whereby DirectShow can complete this task in advance.

6.7.3 Filter Types

It already became clear that DirectShow distinguishes between categories of filters. A specific filter implementation is assigned to a *filter category* by its function, it can also belong to multiple categories. The filter categories do not have to be specified in a filter implementation, they just help to organize filters:

- *Source filters* introduce data into the filter graph and construct new media samples. Typically they have no input pins but only output pins.
- *Transform filters* process incoming media samples and produce output media samples. They can reduce, multiply or convert data to a different media type. Transform filters have input and output pins.
    - *Splitter filters* are a special shape of transform filters. They split an input data stream into two or more output data streams, typically including a transformation of media types.

  - *Mux (multiplexer) filters* are the opposite of splitter filters, they take multiple input data streams and combine them into a single output data stream, typically including a transformation of media types.
- *Renderer filters* receive media samples and apply final processing such as presentation to the user, serialization or data transmission out of the filter graph. Typically they have no output pins but only input pins.

In contrast to DirectShow, Squidy does not distinguish between types of Nodes.

6.7.4 Data Processing

Each DirectShow filter operates in its own thread, which is the same with Nodes in Squidy. Additionally, each Squidy Node operates a data input queue which buffers incoming data objects until they have been processed. A Node pushes or publishes data objects downstream to the next Node.

The data processing model of DirectShow is more complex and more flexible. An input pin in DirectShow can buffer incoming media samples but can also block an incoming media sample until its filter is ready to process it. Moreover, it distinguishes between a *push* and a *pull transfer model*, depending on the filter initiating data transfer: Looking at a pair of filters, the upstream filter can emit (push) media samples, and the downstream filter can request (pull) media samples.

To transport data between filters, pins have to implement a *transport interface* which determines what *transfer protocol* and what transfer model is used by the specific pin to transport data. This transfer protocol typically defines local memory transport by media samples. DirectShow also supports transport of meta data while actual data transport is done within a hardware component.

6.7.5 Allocators

An *allocator* in DirectShow, particularly a *memory allocator*, is an implementation of the *object pool* design pattern (see section 2.3.1). It creates and allocates a pool of reference counted media samples. A source filter requests a new media sample from the allocator when it introduces data into the filter graph. For transport, a reference to a media sample is transferred to the next filter and the media sample travels through the filter graph. When the last reference is released from the media sample, it returns itself to the allocator and can be reused by the source filter.

The same concept can be found in the C++ STL[68] (Standard Template Library), it has been invented by Alexander Stepanov. Allocators using a memory pool eliminate the need of continuous costly allocation of new memory and reduces the amount of data to be transferred between filters to a minimum. It also takes the liability of creating and managing media samples away from the filter itself, it encapsulates memory management. An allocator can hold multiple media samples to allow parallel processing which is discussed in the sections 6.8 and 7.4.1.

68 http://www.sgi.com/tech/stl/

An allocator in DirectShow holds media samples of one type only. Furthermore, a filter graph can contain multiple instances of allocators. This can become necessary if a filter transforms media samples and the resulting media samples differ in size from the input. An allocator is normally managed by the filter which fills its media samples with data.

Squidy does not have comparable mechanisms, it simply creates and destroys data objects on demand.

6.8 Parallel Processing in Squidy

A primary demand on input processing is to keep latency as low as possible (see section 3.3). Use of parallel processing can be a way to satisfy this demand. For this reason, further details regarding parallel processing are explained. Squidy as well as DirectShow make extensive use of parallel processing. Each Squidy Node and each DirectShow filter operate a thread for their processing task. Thus, the details discussed in this section apply to Squidy and DirectShow and they also subject of the subsequently following proposals of the author of this work (see section 7).

6.8.1 Stages

A *stage* is one processing element of the processing chain, in other words, it is the task of one filter or one Node. Each Node operates its own worker thread. Incoming data from an input Pipe is placed into a queue which is monitored by the worker thread. The thread is notified when new items arrive in the queue. To clarify the benefit of this approach, some characteristics of this design incorporated in a processing chain are collected below:

- The minimal time tc_{min} to push a particular data object through the entire processing chain (Pipeline) equals the sum of the individual processing times tf of each filter (Node) which the data object passes.
- The rate of data objects introduced into the processing chain, expressed as time span td between two data objects, may not fall below the time tf_{max} which is required by the slowest filter in the processing chain to process a data object. Otherwise data objects accumulate in the input queue of the filter and the chain becomes congested. To resolve such a congestion, data objects have to be dropped from the input queue of the filter.
- Optimal utilization of the processing chain with tc_{min} for each item of a continuous stream of data objects is reached only if the input queue of any filter in the processing chain never contains a new data object before the filter has completed processing of the foregoing data object.
- Maximum utilization of the processing chain is reached if td equals tf_{max}.
- Maximum utilization of parallel processing takes place if tf of all filters in the processing chain is equal.

- Optimal and maximal utilization of the processing chain and of parallel processing takes place if *td* equals tf_{max} and *tf* of all filters in the processing chain is equal. This also means tf_{max} equals *tf*.

6.8.2 Benefit of Parallel Processing

Parallel processing is typically applied to accelerate processing of tasks. Acceleration of processing of stages can be gained by processing different data objects in parallel within different threads in different filters in the processing chain. Therefore, parallel processing of stages can solely accelerate processing of multiple (sequentially introduced) data objects. Additionally, this applies only if tc_{min} is larger than *td*.

Minimum latency of input processing is equal to tc_{min}. Due to the preceding characteristics, minimum latency cannot be reduced by parallel processing of stages. Parallel processing is able to keep up tc_{min} when processing a continuous stream of data objects if *td* goes below tc_{min} until it reaches tf_{max}. Linear processing also keeps tc_{min} but *td* may not go below tc_{min} otherwise the linear processing chain becomes congested and data objects have to be dropped.

This shows the benefit of parallel processing of stages with the Pipes and Filters architecture: It is possible to operate at higher data rates than with linear processing if tc_{min} of the entire chain is larger than tf_{max} of the slowest filter in the processing chain. This is most likely true for all processing chains with more than one filter. It is obvious that parallel processing has to be executed on a platform which provides multiple processing units to take effect.

Thus, if a stage is divided into a number of sub-stages which are executed in parallel to reduce *tf*, the maximum processable data rate derived from *td* scales with tf_{max}. In other words: To process more data, the processing chain must be divided into more and smaller stages. In an optimally designed processing chain, *tf* for all stages is equal.

6.8.3 Limitations of Parallel Processing

Since parallel processing of stages does not help to reduce latency, the sole remaining approach to achieve this is to reduce *tf* of the individual stages. This could be achieved for instance by parallel processing within a stage, by improving algorithms, or by execution on a specialized hardware platform. The latter solution is applied by the multi-touch input Node of Squidy and is refined by the subsequent proposals regarding this Node. Latency could also be reduced by removing filters but this would modify the processing chain and would only make sense if the chain contained unnecessary filters.

The data queue mechanism of Squidy is implemented using Java synchronization mechanisms, particularly synchronized methods, *wait()*, and *notify()*. To place a data object in the input queue, the source Node identifies the target Node via the connecting pipe and invokes its *process()* method. The *process()* method adds the object to the queue and notifies the associated worker thread.

This procedure causes a small processing overhead because the involved synchronization objects have to be managed and the operating system needs to coordinate the participating threads, too. This overhead participates in the sum of tf and thereby tc_{min}, increasing the overall input latency. Especially with very long processing chains, this management overhead might sum up to an amount which affects overall performance in a way that is noticeable by the user[69] (about 75 ms [62]). To keep latency as low as possible, it makes sense to disable the data queue mechanism if parallel execution of sequential stages is impossible.

69 A C++ program to test the impact of synchronization was written in the course of this work (see Appendix). It operates on native synchronization functions and thereby omits any additional overhead which could be introduced by wrapper functions of synchronization libraries or the environment of a virtual machine. The test platform used the *Microsoft Windows Vista x64 SP2* operating system and an *Intel Core 2 Quad Q9550* CPU. Tests showed an interesting behavior: Independently from the workload of a thread, the resulting synchronization overhead of two threads alternately waiting for each other was either 0.006 ms or 0.024 ms between multiple test runs (equally distributed). The reason for this behavior could not be determined. Assuming the worst case of 0.024 ms, about 3000 synchronization events have to occur to introduce a latency of 75 ms. With a device which introduces data at a rate of 50 Hz this would require a processing chain with 60 filters. However, this does not consider the overhead omitted through the native implementation, overhead required for queue management, possible delays in thread scheduling due to multitasking, the possibility to use active (threaded) pipes with an additional processing queue, and the latency introduced by all other data processing tasks in the processing chain.

7 Proposed Improvements to Squidy

Squidy is a sophisticated input framework but it is also work in progress and has potential for improvements. The framework has been designed for HCI research purposes. It is extended as far as required for current research which is also constrained by time frames and availability of human resources. Therefore, neither all possible or planned features are implemented nor are all of the existing features entirely mature. To sum up, Squidy cannot be considered to have the status of a final product.

The Node for multi-touch input, one of the major concerns of this work, offers only basic functionality compared to other specialized frameworks. The author of this work addresses this and other issues and proposes concrete solutions to resolve them. These solutions differ in their level of detail depending on their overall complexity.

7.1 Node Types

For a user of Squidy it could be helpful to be able to distinguish between Node types similar to filter types in DirectShow (see section 6.7.3). Selection from a list of Nodes could be visually arranged by their purpose and Squidy Designer is enabled to display visual hits.

7.1.1 New Categorization

The following hierarchical categorization of Nodes is proposed. It is derived from the filter classification of DirectShow with respect to the concept of Bridges of Squidy:

- *Bridge Nodes* (any number of input and output data types)
    - *Source Nodes* (no input port, any number of output data types)
    - *Renderer Nodes* (any number of input data types, no output port)
- *Transformer Nodes* (any number of input and output data types)
    - *Splitter Nodes* (any number of input and more different output data types)
    - *Multiplexer Nodes* (any number of input and less different output data types)

When implemented, each Node is aware of its *category* and *subcategory* (none or one of the listed ones) and stores them in static attributes defined in *AbstractData.* To force the developer of a Node to specify the Node's category and subcategory, two abstract methods *getCategory()* and *getSubCategory()* are added to *AbstractData* and must be implemented in the concrete Node.

To keep complexity low, a Node cannot belong to multiple categories. Squidy Designer can make use of this information and provide matching visual representations to the user. For example, Source and Renderer Nodes can omit the representation of their input or output port. Moreover, Squidy Designer can present Nodes ordered by their type.

7.1.2 Bridge Nodes

A *Bridge Node* slightly widens the concept of Squidy Bridges compared to the original intention of the inventors of Squidy. To translate data to and from Squidy data objects, the data has not necessarily to be transmitted or communicated but it can also be produced or rendered. Examined more closely, this is only a detail of a specific Bridge how it internally gets or emits native data. This detail is transparent for all other software components.

Thus for example a Node representing a device such as a laser pointer also acts as a Bridge Node. It translates native data from the device to Squidy data objects. To distinguish unidirectional Bridge Nodes such as the one for a laser pointer from bidirectional Bridge Nodes, subcategories for Source and Renderer Nodes are introduced.

7.1.3 Transformer Nodes

A *Transformer Node* modifies the input data it receives and emits the modified data. The subcategories of Splitter and Multiplexer Nodes are based on the classification of DirectShow.

The primary purpose of the Splitter Node category is to visualize the process of splitting data objects into data objects of different types by a specific graphical representation in Squidy Designer. Multiplexer Nodes have to care in particular for synchronization of incoming data objects before they start to process them. However, data synchronization is an issue for all Nodes processing input data.

7.2 Data Synchronization

When a Node receives input data objects from multiple Pipes which should be processed together, all incoming data objects must be synchronized first. Only data objects which correspond temporally may be processed and published, otherwise no useful correlation of outgoing data objects can be guaranteed.

Squidy does not yet offer a solution to synchronize data. However, this feature can be added with little effort. The input data queue of a Node already buffers incoming data. This allows the Node to hold back data without temporal correspondence. The worker thread of a Node can now iterate over all items in the input data queue, compare its time stamps, and process and publish only complete sets of data from all incoming Pipes with temporal correspondence.

Temporal correspondence can be defined by a maximum time span by which the time stamps of two data objects differ. A useful basis to define the maximum time span could be the duration of one sample. Unfortunately, this duration is unknown but DirectShow offers solutions for this problem.

DirectShow defines the behavior of Renderer filters concerning time stamps, defines two different times (*stream time* and *media time*), and provides interfaces for Source filters

(*IAMPushSource* and *IAMLatency*) to expose time offset and latency information. In short, time management and synchronization in DirectShow is flexible but also complex. For a basic synchronization mechanism, two simple concepts of DirectShow related to time stamps shall be added to Squidy.

7.2.1 Start Time Stamp and Stop Time Stamp

In DirectShow the media type exchanged during the filter connection process can contain an average duration per media sample. Furthermore, each media sample carries two time stamps, a *start time* and a *stop time*. The latter information is the one which is more useful as it is much more detailed.

By adapting this idea for Squidy and replacing the single time stamp of object creation in the *AbstractData* class with a start time stamp and a stop time stamp, the basis for synchronization has been laid. Secondary, the two time stamps provide information about the data rate of the specific data objects which can be useful for other situations.

The current time stamp is set automatically on instantiation of the *AbstractData* class. By having start and stop time stamps, the liability to set these values correctly is delegated to the Source and Bridge Nodes. These Nodes have to be adapted and must set the start and stop time stamps before publishing a data object. This change fits very well to the concept of data object allocators in Squidy which is introduced in section 6.7.5.

When a data object passes a Transformer or Bridge Node, both time stamps must be carried over to the outgoing data objects as it is currently done with the single time stamp. This can be solved automatically by data allocators (see section 7.3.3).

7.2.2 Reference Time

A meaningful time stamp requires an accurate time base from which it can be created. If several time stamps should be comparable, they all have to be created from the same time base. Such a time base is called *reference time* in DirectShow.

The current implementation of Squidy uses the system time acquired by the Java method *System.currentTimeMillis()* as reference time. This Java method returns the current time in milliseconds which should be sufficient for most input tasks. However, the Java method *System.nanoTime()* provides access to the most precise system timer available on the current platform[70,71,72,73] and returns the current time in nanoseconds.

70 In the source code of Sun's JDK it can be verified if *System.nanoTime()* actually provides more precise time information by looking at its internal platform-specific implementation. The implementation for Microsoft Windows can be found in the function *os::javaTimeNanos()* (*hotspot\src\os\windows\vm\os_windows.cpp*). It relies on the functions *QueryPerformanceCounter()* and *QueryPerformanceFrequency()* of the Windows API. The implementation in the JDK appears weird, as it converts the integer time values to floating point values, calculates the time in seconds and then multiplies the result again to convert it to nanoseconds. This is likely to introduce rounding errors and requires more processing power compared to solely integer based calculation. Nevertheless it can be expected that the accuracy of the result is better than one millisecond. There is no detailed documentation from Microsoft how exactly the *QueryPerformance* functions work internally but this can obviously[71] depend on the hardware platform and is not necessarily reliable. However, Windows versions later than Windows XP make use of the reliable *HPET*[72] *(High Precision Event Timer)* when it is available on the hardware platform, according to Microsoft and other sources[72,73]. A numerical overflow of the time value occurs after approximately 292 years. Based on the assumption that Squidy is executed on current operating systems and hardware platforms which can make use of an HPET (High Precision Event Timer) and because a more precise time stamp guarantees more accurate synchronization, *System.nanoTime()* should be used as source for the reference time in Squidy.

71 http://www.virtualdub.org/blog/pivot/entry.php?id=106

72 http://en.wikipedia.org/wiki/High_Precision_Event_Timer

73 http://www.microsoft.com/whdc/system/sysinternals/mm-timer.mspx

7.2.3 Distributed Environment

Squidy can operate as a distributed system, too. Since each instance of Squidy Manager uses the system time of the platform on which it is executed to create time stamps, the system time of all these platforms has to be synchronized in a distributed environment if accurate synchronization is required.

Time synchronization of all participating distributed platforms can be accomplished by an external application or service. This is not an ideal solution as it delegates responsibility to the user. Therefore, the Squidy Remote Node which encapsulates communication between multiple instances of Squidy Manager should take care of this task. It can make use of a standardized time synchronization protocol such as *NTP*[74] to synchronize the time in downstream direction between all instances of Squidy in regular intervals. An example implementation of the NTP protocol in Java can be found on the website of the NTP project[75].

An implementation which synchronizes multiple instances of Squidy Manager does not have to modify system time. This would require administrative user rights for the underlying platform. Synchronization can operate on an internal *Squidy time* which is constructed from an offset value relative to the system time of each individual platform instead.

7.3 Data Allocator

Squidy neither makes use of an *object pool* nor of an *allocator*. The major advantages of allocators have been explained in section 6.7.5 and Squidy could also benefit from these advantages. An allocator also combines and encapsulates related tasks and makes them easier to use for a developer.

The idea of DirectShow to use allocators appears convenient. Therefore, a custom allocator should be integrated in Squidy. By using *Java Generics*[76], an allocator class can be applied to all Squidy data types.

Due to a detail of the implementation of Squidy, consequent use of data allocators similar to DirectShow offers one more advantage: Squidy creates and publishes duplicates of a data object when multiple target Nodes are connected, which is a costly and often needless process. This is done as a precaution to be sure that modifications of a data object in one partial downstream processing chain cannot affect another parallel partial downstream processing chain. Without the copying procedure, the parallel chain would operate on the same instance of the data object.

The need for data duplication originates from concepts of *dataflow programming.* They demand for *freedom from side effects* and define the *single assignment rule* which states to “disallow the reassignment of variables once their value has been assigned.” [63] However, it is desirable to eliminate needless data duplication to reduce required

74 http://www.ntp.org/
75 http://support.ntp.org/bin/view/Support/JavaSntpClient
76 http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

processing power and memory with respect to the rules of dataflow programming. When using data allocators, duplicates can be created on demand and only if they a required. A comparable strategy could also be implemented without data allocators but would still cause additional continuous data object instantiations within the processing chain.

Instantiation of a data object causes the creation of its time stamp which indicates when the data value held by the object has been produced. This works well if a new data object is instantiated for each new data value. When using an allocator, objects are instantiated only once. A solution for this problem comes with start and stop time stamps which are described in section 7.2.1.

7.3.1 Data Object Creation on Demand

To overcome the issue of continuous data object instantiation, each Node operates its own data allocator. The data allocator is used by Nodes introducing data which are Bridge and Source Nodes, other Nodes use it on demand only.

To understand what *on demand* means, it has to be defined when a data allocator is required: It is required if a Node needs to create duplicates of the incoming data objects, or if it creates a different type or a different number of data objects than the incoming ones. The latter two cases do not need to be inspected because they are self-explanatory.

A Node only needs to create duplicates of outgoing data objects if the following conditions are true:

- The Node has more than one outgoing connection, thereby an instance of a data object is published to multiple Nodes in parallel. These Nodes rely on unmodified data objects from their upstream processing stage.
- At least one of the following downstream Nodes modifies data objects. This globally modifies a single instance of a data object.
- The Node modifying data objects does not operate a data allocator. Thus it modifies the original data objects, not new data object instances from its data allocator.

When all of these conditions apply, a Node has to initialize and use its data allocator. It publishes duplicates of data objects created by its data allocator to all outgoing Pipes which announced modification. All other outgoing Pipes (or the last one if all Pipes modify the data objects) receive the original data object. Since the Node does not know about the behavior of its downstream Nodes, the connection procedure between Nodes has to be extended:

- The newly connected downstream Node announces upstream if it or one of its downstream successors modifies data objects. This happens every time a new connection is established. The announcement is propagated upstream and remembered by each passed Node until it reaches either the beginning of the chain, a

Node operating a data allocator, or a Node with more than one outgoing connection which requires to initialize its data allocator.

- When the connection between two Nodes is deleted, the upstream Node needs to start the announcement. It is propagated in the same way as on a new connection. If it reaches a Node with more than one outgoing connection and which operates a data allocator, the Node either needs to keep its data allocator or may destroy it. This has to be decided depending on the remaining requirement of duplicates. If the Node destroys its data allocator, the announcement must be propagated further upstream.

This procedure can be implemented in the class *AbstractNode*, transparently for inherited Node classes.

7.3.2 Implementation Details

Data object creation solely on connection or disconnection of Nodes disregards some major problems. These problems are identical to those for detecting circular Pipelines described in detail in section 7.10. Their solution is comparable, too: The approach described above has to be extended to determine usage of the data allocator dynamically during data processing.

For upstream announcement, a new way to transfer information has to be introduced because data flow is currently limited to downstream direction. DirectShow offers comparable possibilities for example via the interface *IQualityControl* with the method *Notify()*. The following solution introduces *notifications* as a synonym for announcements.

- An new interface *IAllocator* is introduced. The DirectShow interface *IMemAllocator* can be used as a template.
- A new generic allocator class *AbstractAllocator* implementing *IAllocator* is introduced.
- New data allocator classes derived from *AbstractAllocator* for all data types are introduced.
- The Piping class is extended by an attribute *allocators* representing a collection of data allocators implementing *IAllocator*. If a Piping object publishes multiple data types, a dedicated allocator for each type is operated.
- The Piping class is extended by a method *duplicateData()* which requires a Pipe, a Squidy data type, and a boolean flag as parameters. The method enables or disables duplication of data objects and thereby usage of the internal allocator for the specified data type during the data publishing process.
- The Piping class is extended by an abstract boolean method *isModifyingData()* which indicates if it modifies incoming data objects. Inherited objects, especially Nodes, have to implement this method to force the developer of the Node to provide this information and also allow to provide it dynamically. Per default, Piping objects

also iterate over their sub-Processables to accumulate the attribute from all Piping objects which they contain.

- The *IData* interface and the *AbstractData* class are extended by an attribute to hold a boolean flag *multiplied* which indicates that an upstream Piping object has multiplied the data object without duplicating it. Object serialization of the *AbstractData* class is extended to support the *multiplied* attribute. It is set to *false* during instantiation.
- A new interface *INotification* is introduced. It follows the *visitor* design pattern (see section 2.3.1) and contains the method *visit()* which requires a Piping object and a Pipe as parameters. The interface can be extended to multiple *visit()* methods for later extension of the notification functionality.
- A new class *DuplicateDataNotification* which implements *INotification* is introduced. On construction, it requires a Squidy data type and a boolean flag indicating the desired state of data object duplication as parameters, both are stored in attributes. Its *visit()* method advises a Piping object to duplicate or not to duplicate data objects for the specified Pipe and data type by invoking the Piping's *duplicateData()* method.
- An new interface *INotifyable* is introduced. It contains the method *notify()* which requires *INotification* and a Pipe as parameter.
- The Piping class is extended to implement *INotifyable*. The method *notify()* and the internal processing structure is implemented to provide the same functionality upstream as the method *process()* of *AbstractData* provides downstream. A data allocator for notifications shall be omitted because of their relatively rare occurrence. The *notify()* method is invoked by the downstream Piping object which publishes notification objects upstream with the notification object and the employed Pipe as parameters.
- The *AbstractNode* class is extended by a processing queue which holds objects implementing the *INotification* interface. The processing queue can be processed within the thread which is also used for data processing.
- When a Piping object multiplies a data object without duplicating it, it sets the *multiplied* flag.
- When a Piping object which modifies incoming data objects receives a data object with the *multiplied* flag set, it creates a new *DuplicateDataNotification* object and publishes it upstream.

7.3.3 Transfer of unique Attributes

Data objects can carry unique properties in their attributes. This can be for example a device identifier or the time when the original data object has been captured from a device. When a data object passes a Node during data processing, especially a Transformer Node which converts input data and creates new output data objects of a different type, the unique attributes should be retained and transferred to the corresponding new data objects.

Without an allocator, this task has to be carried out by the developer of a Node after he has created a new data object. He might forget it, do it in a wrong way or not adapt it every time a new unique attribute is added. An allocator provides a solution to encapsulate this task, force the developer not to forget it, and release him from implementing attribute transfer.

An allocator can demand for a source data object when a new data object is requested by the application. This can be realized by a method requiring a source data object as parameter. Thus it can copy all unique attributes before it provides the new data object to the caller. When using individual allocator classes for each Squidy data type, specialized copy procedures can be implemented, too.

Due to existence of Node categories, a Node can create different types of allocators or initialize an allocator depending on the Node category. A Source Node can allow to request new data objects without providing a source data object, whereas a Transformer Node can force the developer to provide a source data object.

Squidy Remote also instantiates new data objects but it uses serialization to reconstruct the data objects. Thus their unique attributes can be reconstructed, too.

7.3.4 Dynamic Data Allocator Capacity

One might argue that it is useful to enable the capacity of a data allocator to grow and shrink dynamically, depending on the number of object instances required. This allows a Node which cannot cope with the rate of incoming data objects to buffer unprocessed data objects in memory and to process it when it is ready.

Thorough reflection on this approach reveals the following arguments:

- A Node which cannot cope with the rate of incoming data objects is most likely not able to catch up with its buffered data objects, except its processing behavior is very unsteady. Therefore, it only makes sense to provide a small number of additional data samples for temporary fluctuations of a Node's processing performance. It should be possible to estimate a reasonable static number of available data objects in the allocator.
- Buffering and not immediate processing data objects introduces additional latency which is the opposite of the requirements for this framework. Depending on the type of processed input data, it can make more sense to drop unprocessed data objects to keep latency low at the expense of high data rate and completeness of data transfer.
- Dynamic allocator capacity might be useful for very long processing chains with extensive parallel processing where static allocator capacity must be chosen very large by default to be able to provide enough data objects for parallel processing in all stages (compare section 6.8). This is likely to be a rare case.

- Validating and resizing a data allocator can be a costly operation which can affect overall processing performance. Moreover, if allocator capacity is reduced and objects are freed, the Java garbage collector is invoked and might interfere input processing on a platform with heavy CPU load.

To sum up, dynamic data allocator capacity can be useful in some cases but its usage should be avoided if possible. An implementation of a data allocator should provide two optimized classes for static and dynamic capacity. This allows to leave the choice to the developer or user of a particular Node which implementation to use.

7.4 Direct Invocation and Blocked Execution

The concept of data input queues of the Node class is related to the concept of dynamic allocator capacity. A data input queue acts as a buffer for data objects and cooperates very well with dynamic allocator capacity. Thus all arguments listed above (see section 7.3.4) apply to data input queues, too. It depends on the individual processing chain and demands of the user if such a buffer is useful or counterproductive.

In order to allow this choice, a new publishing mode for Nodes called *direct invocation* is introduced which also eliminates possible additional latency (see section 3.3). With direct invocation, the *process()* command of the target Node directly invokes data processing without placing incoming data objects in its data input queue. Thereby the internal processing thread is bypassed and data processing is executed by the thread of the preceding Node. The explicit purpose of this publishing mode becomes clear in section 7.5.3.

Within the attributes of a Pipe it is possible to enable direct invocation optionally and separately for each data type. Nodes can simultaneously operate with traditional processing and direct invocation depending on the data type.

7.4.1 Parallel Processing

By using *direct invocation* parallel processing disappears. All processing is done within the thread of the first Node of a processing chain segment which uses direct invocation.

If parallel processing shall remain enabled but buffering of data should be prevented, direct invocation cannot be used. The maximum capacity of a Node's data input queue has to be limited to one item instead. Additionally, the *AbstractNode.process()* method gains the potential to *block execution* optionally until the data input queue has free capacity again. Pins in DirectShow can behave similarly. If all Nodes throughout the entire processing chain stick to *Blocked execution*, all Nodes are forced to assimilate the speed of the slowest Node in the processing chain. Thus data congestion cannot occur because Nodes which introduce data to the processing chain are limited, too.

If maximum input data queue capacity is set and blocked execution is not effective, all incoming data samples to a Node which exceed the maximum queue capacity are dropped automatically by the Node. This data loss should be visually reflected in Squidy Designer to warn the user.

Direct invocation and blocked execution reduce functionality to match a special purpose instead of adding new functionality. This can be useful depending on the application.

7.4.2 Implementation Details

Direct invocation, maximum queue capacity and blocked execution can be implemented in *AbstractNode.process()* with a few lines of code. All three limitations are disabled by default. Attributes to store the state of maximum queue capacity and blocked execution have to be added to the *AbstractNode* class.

The state of direct invocation needs to be stored in an attribute of the Pipe class for each data type transferred, and in an attribute of the *AbstractData* class. During data publishing the state of direct invocation has to be copied from the Pipe to the published data object and has to be evaluated by the *AbstractNode.process()* method.

7.5 Multi-Touch Input Node

A basic idea of Squidy is to reuse and combine the best state-of-the-art solutions for a particular problem. This reduces the workload required for research projects and builds upon proven knowledge. Thus the current multi-touch input Node of Squidy was combined from several programming languages and techniques from several sources.

Multi-touch image processing runs in a separate process launched by Squidy from an executable file on demand. This executable file is implemented in C++ and borrows blob detection from *Touchlib* (see section 4.3.15). Moreover, it features image pre-processing for background subtraction and image filtering by CUDA, which runs on NVIDIA graphics adapters. The detected blobs are analyzed by a rudimentary finger processing algorithm and transferred via OSC protocol to the corresponding Squidy multi-touch Node which is implemented in Java and makes use of the OSC Bridge.

7.5.1 Advantages of the current Implementation

The multi-touch tracker module uses GPGPU for image pre-processing. Thereby it transfers a large amount of workload to a specialized ASIC (application-specific integrated circuit) and operates very fast compared to the same algorithms executed on a conventional general-purpose CPU. It introduces only a small amount of latency into the input processing chain.

7.5.2 Disadvantages and Proposals for Improvements

- The current implementation of image pre-processing is implemented using *CUDA*[77] *(Compute Unified Device Architecture)*, a proprietary architecture from NVIDIA[78]. It

77 http://www.nvidia.com/object/cuda_home.html
78 http://www.nvidia.com/

requires a graphics adapter manufactured by NVIDIA to operate. To mitigate vendor dependency, the source code specific to CUDA should be ported to use the functionally equivalent *OpenCL*[79] *(Open Computing Language).* OpenCL is an open standard and is supported by multiple vendors. Moreover, an additional CPU-only implementation should be provided to extend platform portability to platforms without a GPU or an OpenCL API.

- Image processing for blob recognition is currently executed by the CPU while all other image processing is executed by GPGPU. Image processing on the GPU promises better performance, thus blob processing should also be transferred to the GPU.
- Efficient GPGPU programming requires detailed technical knowledge of the GPU and all involved components such as graphics memory management. Therefore, taking care of efficiency can be an expensive task and is not necessarily a focus of software development for research purposes. It should be verified if the current GPGPU algorithms exploit the full potential of the hardware.
- The module uses fixed pre-processing steps and image filters only. It also applies blob tracking and post-processing of blob data within an immutable workflow. This is not a convenient situation for evaluation and research of multi-touch applications which requires separation and arbitrary recombination of these tasks. The fixed processing steps should be replaced by a flexible processing chain.

  The Pipes and Filters architecture of Squidy would be ideal for this task. However, it very likely causes a vast loss of performance if some filters in a processing chain operate on the GPU and not all involved filters are aware of this fact. These filters could cause expensive data transfer in memory between host and graphics device.

  To gain maximum possible processing performance, the host CPU has to operate using data located in the host memory and the GPU has to operate using data located in its graphics memory. Graphics memory is located on the graphics device which is connected to the host system via a bus subsystem (often PCI Express). Depending on the task to execute, it can become necessary to transfer data between host memory and graphics memory which is an expensive process and should therefore be reduced to a minimum.
- The module currently supports only a vendor-specific camera interface (by IDS Imaging[80]) because it makes use of features which are special to the corresponding cameras. Therefore, only a limited number of cameras is supported. Moreover, the multi-touch tracker operates satisfactorily on Windows only. IDS Imaging also offers drivers for Linux, but according to the authors of Squidy they do not work as stable as their Windows counterpart. To overcome these limitations, the camera interface implementation should be replaced by a more general and platform independent one. The module already contains a replaceable class representing the camera interface, but it makes more sense to adapt the entire, well-proven video subsystem of another

79 http://www.khronos.org/opencl/
80 http://www.ids-imaging.de/

existing multi-touch framework to save time for implementation and reduce the required effort.

7.5.3 Proposal for an overall Architectural Redesign

Based on the enumeration of disadvantages described above, the following solution for an overall architectural redesign of the multi-touch input Node is proposed:

- Two new Squidy data types are introduced: *DataImage* and *DataImageGPU*. *DataImage* provides a reference counted image in host RAM, *DataImageGPU* provides a similar image in graphics RAM. The memory of *DataImageGPU* is allocated as a frame buffer object (FBO) which enables it to be accessed as OpenGL texture, OpenGL frame buffer, or GPGPU memory object for versatile GPU image processing. With these data types, a concept comparable to DirectShow transport interfaces [31] is introduced.
- A data allocator is introduced to gain performance.
- A new *Camera* Source Node is introduced which makes use of the data allocator class and fills *DataImage* objects from its data allocator with acquired images from an attached physical camera. The Camera Node connects to a native code implementation via JNI.
- A new Squidy Transformer Node for image data transfer between host and graphics RAM is introduced. It acts as converter between *DataImage* and *DataImageGPU* in both directions depending on the input data type and connects to a native code implementation via JNI.
- Parallel processing controlled by the CPU is not very useful for GPGPU as a GPU operates its own thread management and does not require any CPU resources. Therefore, all data objects of type *DataImageGPU* are published using direct invocation by default (see section 7.4).

  This assumes that there is only one GPU available which is fully utilized by processing *DataImageGPU* objects. If multiple GPUs are available or processing does not fully utilize the GPU, this situation has to be detected and direct invocation may not be used. It this situation it would be counterproductive because it would prevent parallel processing of data objects.
- A number of new Nodes for GPGPU image processing are introduced, such as Nodes for background subtraction, high-pass filtering, color conversion, and blob detection. These Nodes operate on the *DataImage* and *DataImageGPU* types and connect to a native code implementation via JNI. They replace the internal image filters of the current multi-touch Node implementation and provide equivalent implementations of their individual filter task for CPU and GPU processing which are executed depending on the input data type.

- A new Squidy data type *DataBlob2D* is introduced. It derives from the data type class *DataPosition2D* and contains additional information for blob size. This data type is used by the blob detection Node for data output.
- A new Squidy data type *DataTouch* is introduced. It derives from *AbstractData* and contains properties for touch event, identifier, origin, bounds, time, device, path, and ambiguity (compare section 8.1).
- A new Squidy data type *DataTouchCommand* is introduced. It derives from data type class *DataString* and contains a command property as a string interpreted from one or multiple *DataTouch* objects as well as all identifiers of *DataTouch* objects used to create this command (compare section 8.1).
- A new multi-touch abstraction Transformer Node is introduced which processes DataBlob2D objects and publishes interpreted multi-touch events as *DataTouch* objects (compare section 8.1).
- A new multi-touch interpretation Transformer Node is introduced which processes DataTouch objects and publishes interpreted multi-touch commands as *DataTouchCommand* objects (compare section 8.1).

Multi-touch processing can be modeled by a Pipeline shown in illustration 6.

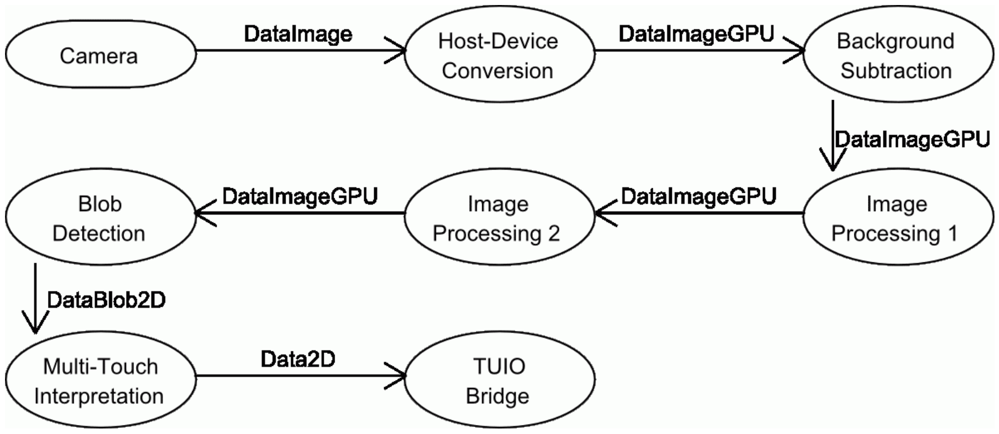


Illustration 6: A new multi-touch data processing chain

7.5.4 Benefits of the proposed Redesigned Architecture

The new architecture complies with the Pipes and Filters architectural pattern and separates different functionality into different small modules which can be refined and maintained with less effort than one large and comprehensive module. This also complies with the architectural concepts of Squidy and with the concept of separation of concerns. Further benefits are the following:

- It introduces new types of Nodes which can be reused in different contexts.
- It allows flexible image processing which is required to cope with different multi-touch environments.

- It allows to model, store, and reuse multiple different Pipelines for multi-touch applications.
- It accounts for the requirements of GPGPU and enhances utilization of the GPU's processing potential.

7.6 Multi-Touch Calibration

This section starts with a discussion about calibration of multi-touch input hardware to explain the need for calibration and its backgrounds. After that, a possible implementation for calibration in Squidy is proposed.

A multi-touch application requires knowledge about the position of touches from users on the multi-touch surface which are detected by a camera. It needs this position to interpret and to assign touches appropriately to elements visible on the surface at the particular position. Thus a *surface calibration* is needed to correlate image pixels seen by a camera with positions on the surface. For this purpose, a *two-dimensional (2D)* coordinate space is assigned to the multi-touch surface which is commonly chosen to be identical to the coordinate space of the graphical workspace presented to the user on the surface.

If an application operating with a curved multi-touch surface wants to handle versatile variants of surfaces, it can become vital to know the *three-dimensional (3D)* spatial properties of the surface. Especially in combination with tangible and multi-modal interaction, spatial position information of objects used for interaction is likely to be valuable. This might exceed the practical requirements of the first prototype of the Curve desk but can become vital for further research. A prerequisite for capture of spatial positions is *camera calibration* which will be described later in this section.

7.6.1 Surface Calibration

Planar surfaces which are located in the field of view of one camera can be calibrated with little effort. Due to prior knowledge of the rectangular and planar characteristics of a surface, a simple linear coordinate transformation can be established between the 2D image coordinate space and the 2D surface coordinate space.

This allows to implement a straightforward *surface calibration* process which is used by many multi-touch frameworks (see section 4). The user touches the four corners of the rectangular surface, the touches are captured by the camera and the surface calibration application determines the image coordinates of the touches.

An issue for multi-touch input is image distortion of the image provided by a camera which causes error-prone coordinate transformations. Image distortion originates from (often radial) distortion caused by the lens mounted on the camera. The smaller the focal distance of the used lens the larger becomes the distortion. Distortion needs to be determined to be able to define an accurate transformation between image coordinates and surface coordinates.

Based on the assumption that image display on the surface is already calibrated and provides a final undistorted view, the surface calibration procedure can be extended to reduce the impact of distortion of the camera's lens. The multi-touch application displays a grid of points spread over the entire surface to the user who has to touch all these points. The application can integrate the image coordinates of these grid points into its coordinate transformation. However, this does not really correct distortion completely but reduces its impact by a more accurate calibration of smaller areas of the surface.

In most cases when using a curved surface, it is still possible to calibrate this surface with the procedure just described. It is necessary to use a dense array of surface calibration points in curved regions of the surface to be able to construct a sufficiently accurate coordinate transformation. If bending or deformation of the surface is very intense or the angle between the camera viewing direction and the surface is small, calibration and detection of points on the surface are not very accurate.

If the image display on the surface is not calibrated and requires a camera-assisted surface calibration process first, more complex calibration techniques have to be applied. This mainly concerns presentation to the user and is not part of this work.

7.6.2 Camera Calibration

If one wants to use complex or deformed surfaces viewed by multiple cameras it can become necessary to retrieve more information about the cameras itself. Moreover, tracking the position in 3D space of objects located above the surface is an interesting task for Curve research (see section 1.1).

The following properties of a multi-touch surface table have to be determined to handle these requirements properly:

- lens distortion of each camera to allow its accurate correction
- spatial position and field of view of of each camera to allow 3D calculations
- curvature or deformation of the surface to enable a correlation between 2D image and 2D surface coordinates

It is impossible to determine all these parameters by the simple approach of surface calibration (see section 7.6.1) because one cannot determine whether displacement of coordinates results from lens distortion, camera position or orientation, or an also unknown deformation of the surface. Therefore, other methods have to be applied to determine all these properties.

Camera Parameters

Tsai [64] describes *camera calibration* in the following way: “A camera calibration in the context of 3D machine vision is the process of determining the internal camera geometric and optical characteristics (*intrinsic parameters*) and/or the 3D position and orientation of the camera frame relative to a certain world coordinate system (*extrinsic parameters*).”

Methods for camera calibration allow to determine these camera parameters. Camera calibration is a thoroughly explored field of research, only few examples of related work can be named here. Another method besides the technique which Tsai proposes is *DLT* [65]. The *extended DLT* method [66] also includes basic radial distortion correction. Further methods for distortion correction can for example handle fish-eye lenses with very short focal distance [67; 68].

Depending on the technique used for camera calibration it is often required to provide a defined number of points with known spatial 3D coordinates together with a correlating set of 2D image coordinates captured from the image of the camera to calibrate. These points should be chosen to be evenly distributed throughout the field of view of the camera which is commonly a part of the multi-touch surface. It is difficult to obtain a sufficient number of points with known specific spatial 3D coordinates from an unspecifically curved surface as input data for a camera calibration algorithm because these 3D coordinates are part of the unknown variables to be determined.

A solution for this dilemma is provided by *wand-based camera calibration* techniques [69]. While being tracked by all cameras, a wand which carries markers is moved throughout the space to be calibrated. The same technique can be applied to a multi-touch surface if the user moves his fingers over the entire surface while the position of the fingers is being tracked by the cameras. Additionally, some fixed reference points from the edges of the surface can be involved to determine orientation and scaling of the camera views.

3D and 2D Reconstruction

If a particular spatial point can be seen from at least two cameras which are calibrated in the same world coordinate system, it is possible to *reconstruct* the 3D spatial coordinates of this point from its 2D coordinates on the camera images. The DLT method can achieve this by solving a system of linear equations [70].

Camera calibration allows to calculate an accurate coordinate in 3D space but this position has no direct relation to the 2D coordinate space of a multi-touch surface. Thus a correlation between the two coordinate spaces must be established to determine the 2D position of a touch on the surface.

The obvious way to achieve this is to construct a virtual 3D representation of the multi-touch surface onto which a 2D coordinate space representing the real surface is mapped. When a touch occurs on the surface, the shortest possible line between the detected spatial 3D coordinates and the virtual 3D representation of the surface can be calculated. The intersection point of this line with the virtual surface provides the required 2D coordinates of the touch.

Construction of a virtual 3D representation of the multi-touch surface can be done based on the coordinates of fingers tracked during wand-based camera calibration because these coordinates are already spread over the entire surface. A more accurate but also more expensive method is camera-based *3D scanning* of the surface. The typical setup of a multi-touch table including back projection and cameras enables use of *structured light* to reconstruct the surface [71; 72]. Various approaches for both structured light and camera-based 3D scanning exist which include 3D scanning without prior camera calibration and use of structured light for camera calibration itself.

Further Aspects

Since the camera calibration steps of tracking, calculation, and reconstruction are error-prone, there are also techniques which aim to minimize this error [73]. Furthermore, multiple frameworks for camera calibration exist. One is the *Camera Calibration Toolbox* for Matlab[81], another one is *BazAR*.[82] Stephan Rupp presents a modular software framework for camera calibration [74] which handles the camera calibration steps described above including an abstraction of the required GUI.

7.6.3 Implementation Details

Except the necessity of a suitable calibration procedure for the user, surface and camera calibration information has to be published to Nodes interested in camera calibration. Surface calibration is called 2D camera calibration in the implementation because it realizes a versatile technique which does not necessarily have to be applied to multi-touch surfaces only. A convenient solution consists of the following points:

- A new Squidy data type *DataPoint2DArray* is introduced. It contains an array of 2D coordinates.
- A new Squidy data type *DataPoint3DArray* is introduced. It contains an array of 3D coordinates.
- A new Squidy data type *DataCameraCalibration2D* is introduced. It derives from *DataPoint2DArray* and fills it with 2D coordinates in the coordinate space of a camera image. Additionally, it complements *DataPoint2DArray* by correlating 2D coordinates in the coordinate space of the surface.
- A new Squidy data type *DataCameraCalibration3D* is introduced. It derives from *DataPoint2DArray* and fills it with 2D coordinates in the coordinate space of a camera image. Additionally, it complements *DataPoint2DArray* by correlating 3D coordinates in the world coordinate space and includes a set of DLT camera calibration parameters.
- A new *CameraTransformation2D* Transformer Node is introduced which accepts *DataCameraCalibration2D*, *DataPoint2DArray*, and *DataPoint2D* as input objects

81 http://www.vision.caltech.edu/bouguetj/calib_doc/
82 http://cvlab.epfl.ch/software/bazar/

and applies 2D transformation to *DataPoint2D+* objects based on *DataCameraCalibration2D* and outputs corresponding *DataPoint2D+* objects.

- A new *CameraTransformation3D* Transformer Node is introduced which accepts *DataCameraCalibration3D*, *DataPoint2DArray*, *DataPoint3DArray*, *DataPoint2D*, and *DataPoint3D* as input objects. It applies 3D reconstruction into world coordinate space to *DataPoint2D+* objects based on *DataCameraCalibration3D* and outputs corresponding *DataPoint3D+* objects. It applies 2D projection into image coordinate space to *DataPoint3D+* objects based on *DataCameraCalibration3D* and outputs corresponding *DataPoint2D+* objects.
- A Camera Node stores its camera calibration information in *DataCameraCalibration+* objects.
- A new class *DataPublishRequest* which implements *INotification* is introduced (see section 7.3.2). On construction, it requires a Squidy data type (*DataCameraCalibration+*) as parameter which is stored in an attribute. Its *visit()* method advises a Node to publish its calibration information if it produces information of the requested data type. Visitor objects of this class can be published upstream by Nodes to request calibration information on demand.
- *DataCameraCalibration+* objects are published by a Camera Node once when data processing starts, when camera calibration is modified, or when a *DataPublishRequest* initiates the publishing process. A *CameraTransformation+* Node always stores the last *DataCameraCalibration+* object it receives.

7.7 Image Stitching

Multiple cameras are required for the Curve desk (see section 3.5) to capture its entire surface for multi-touch input processing. The field of view of each camera can overlap with the field of view of all adjacent cameras to get a gapless representation of the surface. When all cameras have sufficient overlap and are accurately calibrated, it is possible to detect multi-touch input seamlessly and correctly even if it spreads the field of view of multiple cameras.

However, this does not provide a visual representation of the entire surface of the desk but only partial views. Moreover, these views probably look severely distorted because of lens distortion and especially because of the curved surface if no image correction is applied. This stands in contrast to the requirement for Curve to have an undistorted seamlessly combined view of the surface which allows use cases such as document scanning and tracking of fiducial markers on the entire surface.

7.7.1 Stitching Process

The solution for these requirements is *image stitching.* Image stitching is the process of blending multiple adjacent images into a single and ideally seamless image. It is commonly used for panorama photography to construct a panorama picture from multiple photos. This kind of photos is normally shot one after another by a single camera. The same technique can be used to blend the images which were shot at a single point in time by multiple cameras.

Particular approaches for image stitching vary slightly in the processing steps applied to the images. However, the process of image stitching is complex and requires much processing power. It includes distortion correction, key point or image feature finding and matching, color and intensity adaption (normalization), and blending of all input images into one output image [75-77]. A tutorial which describes all steps of the process in detail is available [78].

For blending video images captured by a fixed installation of multiple cameras such as a multi-touch desk like Curve, most of these steps have to be performed only once because the sole variable parameter are the images itself. The cameras are normally not moved or adjusted, and relative color and intensity of their images remain constant. Thus all steps except image blending can be omitted during normal operation of the multi-touch desk [79; 80]. These steps can be performed once and can also be combined with the calibration procedure of a multi-touch surface (see section 7.6.1).

Camera calibration (see section 7.6.2) data can be an additional input for the image stitching algorithm [81] or can add precision if required camera calibration information is only estimated [77].

7.7.2 Available Implementations

An implementation of image stitching comes from the *PTStitcherNG* tool from the *Panorama Tools*[83]. It has been optimized for speed on various hardware platforms. Another implementation is *Nona* which is part of *Hugin*[84], a GUI for the Panorama Tools and Nona. Nona supports multi-threading and more image correction features compared to PTStitcherNG. Additionally, Nona has experimental support for image processing on the GPU.

An employee of NVIDIA implemented an image stitching pipeline in CUDA which runs on GPU and operates very fast[85]. Fast operation is a very useful quality to achieve low processing latency of an input processing chain. Parts of the image stitching pipeline are planned to be published within the *OpenVIDIA*[86] project, another part (*SIFT*) is already available to the public[87].

During a student workshop[88] at the University of Erlangen-Nürnberg some parts of the Panorama Tools were implemented using GPGPU with CUDA. The new implementations partially show a speedup of a factor of multiple hundred times compared to the implementations running on a common CPU.

83 http://panotools.sourceforge.net/
84 http://hugin.sourceforge.net/
85 http://developer.download.nvidia.com/presentations/2009/SIGGRAPH/Advances_in_GPU_based_Image_Processing.pdf
86 http://openvidia.sourceforge.net/
87 http://www.csc.kth.se/~celle/
88 http://www12.informatik.uni-erlangen.de/edu/map/

Any of these implementations have to be adapted for the use with Squidy and multi-touch input applications as they are targeted at still image stitching. This means they execute the entire stitching process for each image by default instead of the image blending step only. Sticking to this behavior wastes a lot of processing power and introduces additional latency.

7.7.3 Implementation Details

The foundation for image stitching - or more appropriate *video stitching* - has been laid with the new multi-touch processing chain (see section 7.5.3) which introduces new data types and a new Camera Node.

- A new *ImageStitcher* Multiplexer Node is introduced which accepts *DataCameraCalibration2D*, *DataCameraCalibration3D*, *DataImage*, and *DataImageGPU* as input objects. It synchronizes all incoming *DataImage+* objects of the same data type from multiple Pipes and applies an image stitching process to these objects based on *DataCameraCalibration+*. The Node provides equivalent implementations for image stitching for CPU and GPU processing which are executed depending on the input data type. The output data of the Node is a blended image in a *DataImage+* object matching the input data type as well as a composed *DataCameraCalibration+* object matching the blended image.
- An ImageStitcher Node initiates calculation of all image stitching parameters each time it receives new or modified camera calibration information or when a new camera is added to its input port. It stores these parameters and applies only a minimal set of operations required for image blending based on the stored parameters to incoming *DataImage+* objects.
- An ImageStitcher Node stores its composed camera calibration information in *DataCameraCalibration+* objects.
- *DataCameraCalibration+* objects are published by an ImageStitcher Node once when data processing starts, when camera calibration is modified, or when a *DataPublishRequest* initiates the publishing process. An ImageStitcher Node always stores the last *DataCameraCalibration+* object it receives.

7.8 Management Interface

The preceding sections presented possible improvements regarding existing features of Squidy. Based on these proposals the need for image stitching has been discussed. This section introduces entirely new functionality.

One of the requirements defined for the input framework (see section 3) is support of scripting of input tasks. Another requirement is support for a control channel to allow an application which uses Squidy to send commands upstream to any Node. Both requirements rely on a *management interface* which allows to control and to monitor Squidy Manager and the Processables it holds. In this context the term *management* *interface* refers to the concept of an interface with the purpose of management, not to a specific interface class or technology.

Currently, Squidy Designer and Squidy Manager are strongly coupled: Squidy Designer directly invokes methods of Squidy Manager to control it. A management interface in between both components is able to decouple them and allow external control of Squidy Manager. When a management interface has been realized in Squidy, Squidy Designer acts as a *client* and Squidy Manager acts as a *server*. This allows Squidy Designer to remotely connect to Squidy Manager, too (see section 6.6).

Additionally, many control tasks are currently directly implemented in Shape classes within Squidy Designer. Thus Squidy Designer and Squidy Manager have to be refactored to an MVC architecture first (see section 2.3.2). Controllers are not only required to separate model and view but also as junctions for the management interface.

In DirectShow, the Filter Graph Manager provides COM interfaces for all management tasks and allows access to all filters inside the graph. Each filter and each pin exposes a standard interface (*IBaseFilter* and *IPin*) for common tasks such as requesting its name or state. Additionally, each filter and pin can expose any number of specialized interfaces. DirectShow defines overall 190 COM interfaces[89] which already include a number of specialized interfaces for tasks such as camera control. A software developer is free to add and implement further custom interfaces if required.

7.8.1 Requirements

Some use cases of the management interface can be differentiated:

A) Direct access from within one instance of a JVM (Java Virtual Machine) by
- Squidy Designer.
- classes within Squidy Manager itself.
- a Java client application which integrates Squidy classes.

B) Remote access using the Java language (from another instance of a JVM) by
- Squidy Designer.
- another instance of Squidy.
- an external Java client application.

C) Remote access not using the Java language by
- a non-Java client application.
- a scripting language.

Since Squidy Designer enables the user to access all relevant management tasks of Squidy Manager, it can be used as a template to identify groups of functionality which the management interface has to expose:

- enumeration, creation, modification, interconnection, and deletion of Processables
- readout and modification of properties of Processables

89 http://msdn.microsoft.com/en-us/library/dd390343(VS.85).aspx

- invocation of methods of Processables
- enumeration of specific properties and methods of Processables
- serialization and deserialization of Processables
- monitoring state of Squidy Manager, Processables, and data objects

This outline of use cases and functionality reveals that access to nearly any aspect of Squidy Manager is required. These aspects can change and evolve over time. Thus a suitable management interface has to allow access to a mutable set of methods and properties. Of course, the basic management interface itself has to be invariant.

Another challenge to solve is to find ideally one single solution which is able to handle local and remote access as well as language independent access to the management interface.

7.8.2 Remote Access using the Java Language

Use case A reflects the current situation in Squidy and works with and without a management interface. The following list picks a small selection out of the pool of existing solutions suitable to cover use case B which is a common use case in many applications. All of these solutions support TCP and/or UDP as transport interface and can be used over LAN, WLAN or the Internet:

Java RMI (Remote Method Invocation)

The *Java RMI*[90] API is a package supplied with the Java Runtime Environment. It allows to treat remote Java objects in the same way as local Java objects. Moreover, it requires classes and its methods to be modified for remote invocation. Beside this disadvantage, it offers a low degree of abstraction. Thus it is used as a base technology for other remote management solutions.

Cajo

The *Cajo*[91] framework has been built to provide an easy-to-use and dynamic way of remote method invocation. It is based on reflection and does not require modification of classes which should be remotely accessible. Cajo does not contain more features than required for remote method invocation and complies with the reflection based processing of Squidy.

JMX (Java Management Extensions)

JMX[92] has been designed for management and monitoring of applications. It requires management objects called *MBeans* to be implemented which are responsible for management tasks. Furthermore, JMX includes a lookup service, security management and allows the use of different connectors for transport between client and server. The standard connector uses RMI.

90 http://java.sun.com/javase/technologies/core/basic/rmi/
91 http://cajo.dev.java.net/
92 http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

These features might have to be supplemented for a management interface based on RMI or Cajo when it is used in large environments with many users. On the other hand, the additional management objects of JMX require additional effort for implementation. This effort can be reduced when the MBeans concept is considered during refactoring of Squidy to an MVC architecture. Controllers can be designed with this purpose in mind and can implement an MBean interface.

JMS (Java Message Service)

JMS[93] is a message oriented middleware API which defines a communication standard to enable asynchronous communication between components. Asynchronous communication reduces dependencies but requires more effort for handling of outstanding requests between client and server. A JMS implementation available from Sun is the *Java System Message Queue* service which also allows to use JMX over JMS.

JMS is not a management protocol itself but a transport protocol. In case of using it in conjunction with JMX it adds one more level of complexity to the communication process. It is currently hard to assess if Squidy can make use or even requires this feature. To keep overall complexity of Squidy low, JMS should only be used if practical use of the management interface shows that JMS is needed.

Spring Remoting

Spring[94] is a large application framework written in Java which provides functionality needed in many applications. One part of the framework is *Spring Remoting*. It is a wrapper for multiple underlying remote method invocation technologies including RMI, JMS, and JAX-RPC for web services.

Similar to Cajo, Spring Remoting allows to build remote interfaces without modifying the affected classes. It uses a configuration file where the developer has to define remote objects. Spring Remoting does not explicitly provide management functionality.

Conclusion

To sum up, Cajo is a good choice for a quick implementation of the management interface for a small environment with few users. JMX is harder to implement but it offers more possibilities and because of its convenient architecture it will pay off if Squidy becomes larger. It can even be used over JMS if required later. Thus JMX is the recommended solution to integrate into Squidy to implement the management interface .

Spring Remoting would be a good alternative to Cajo. Unfortunately it is not very reasonable to use Spring Remoting within an application not based on the Spring framework because Remoting makes use of features provided by the framework. However, it makes sense to think about migrating Squidy to the Spring framework.

93 http://java.sun.com/products/jms/
94 http://www.springsource.org/

Spring offers sophisticated solutions for the MVC concept, AOP, serialization, and remote connection including support for JMX.

7.8.3 Remote Access by a non-Java Client Application

Use case C for the management interface demands for a way to connect the Java programming language natively to other programming languages. This is not expected to be a focus of Squidy but is discussed briefly. The remote access solutions listed above mainly connect to a Java counterpart, only RMI can cooperate with different programming languages. The following solutions are platform independent and except for Jace also language independent.

OSC (Open Sound Control)

It is obvious to consider the *OSC*[95] protocol as it is widely used to transport multi-touch input data (see section 3.6.1). OSC is a message-based protocol for data transfer. Its messages are encoded using ASCII text. The protocol itself does not define any specific commands, it defines the syntax and contextual structure of data and commands which are transported by OSC.

Therefore, a management interface based on OSC has to specify a management protocol consisting of a set of management commands. Besides an OSC bridge, it also requires an encoder and an interpreter to be designed to translate between textual OSC commands and the object structure of Squidy. A management protocol based on OSC either has to define a general set of commands which rely on Java reflection or a specific set of commands for specific functionality. The former approach means a lot of manual programming effort for a user of the protocol, except he builds a framework like Cajo around it. The latter approach is easy to use but the protocol and its translation must be maintained and extended manually for each modification and each new feature which should be supported.

OSC is a good choice if one aims at implementing a minimal approach with a very reduced set of management functions. Squidy Remote could be extended to support some management commands with little effort. However, if more complex management tasks should be performed, such as remotely working with Squidy Designer, it is very inefficient to rely on OSC due to the characteristics of the protocol described above. In terms of abstraction, OSC resides far below RMI because it defines nothing but rules for a text-based protocol.

SOAP

SOAP[96] is an XML-based protocol targeted for information exchange with web services. It also defines RPC and supports object-oriented development. It can be compared with RMI or Cajo as it allows to invoke methods remotely but does not include specific

95 http://opensoundcontrol.org/
96 http://www.w3.org/TR/soap/

management functionality. Compared to RMI or Cajo, it requires additional processing power and bandwidth due to XML parsing and XML transport overhead.

SOAP is commonly implemented in frameworks for web services. Classes or objects which should be published for remote access must be specified in a description file written in WSDL (Web Services Description Language). Frameworks supporting SOAP for Java are for example Apache Axis2/Java and Apache CXF. Both frameworks have a lot of additional functionality mainly for web services. Apache CXF includes support for JAXB which can be useful as Squidy currently relies on this technology for serialization.

However, JAXB support cannot compensate the missing management architecture compared to JMX. Moreover, JAXB serialization in Squidy shall be replaced by Hibernate anyway (see section 6.6). Thus integration of a web services framework should only be considered if language-independent access to the management interface becomes essential for some reason.

CORBA (Common Object Request Broker Architecture)

Another solution to access the management interface from an application not written in Java is to use a middleware architecture such as *CORBA*.[97] CORBA is the name of a standard for interoperability of objects and data between software components written in multiple computer languages and running on multiple computers.

CORBA implementations are available for many platforms and programming languages. An implementation for Java is the EJB (Enterprise JavaBeans) technology. It can replace Cajo or JMX. However, the concepts of CORBA and EJB are complex compared to JMX and even more Cajo. Moreover, EJB do not include a remote management concept as JMX does. Thus CORBA and EJB appear to be oversized and provide too little abstraction for Squidy.

With RMI-IIOP[98] (Remote Method Invocation over Internet Inter-ORB Protocol) it is possible to use RMI in Java and connect to a CORBA counterpart. RMI-IIOP is a transport protocol for RMI which replaces the standard JRMP (Java Remote Method Protocol) and is integrated into the Sun JRE.

Jace

If Java should be accessed from an application implemented with C++, the *Jace*[99] library can be used. Jace provides a set of C++ classes to access JNI from the C++ programming language. This requires to create a *remote proxy client application* based on the *proxy* design pattern [24] which wraps the management interface and acts as a standard Java client. The proxy client application has to provide access to the management interface via JNI to the Jace library.

97 http://www.corba.org/
98 http://java.sun.com/products/rmi-iiop/
99 http://sourceforge.net/projects/jace/

7.8.4 Use of Scripting Languages

A number of scripting languages exist which can be executed in a JVM. They come with a compiler which compiles the scripting language to Java byte code or an interpreter which runs within the JVM. The big advantage of these approaches is that the scripting language can make use of all features of the JVM and has full access to Java classes. This allows a seamless integration with applications written in Java such as Squidy.

An example for implementations of scripting languages is *BeanShell*[100], a Java-based interpreter with a language syntax comparable to Java. Other examples are *Jython*[101] (an implementation of Python in Java), *Rhino*[102] (JavaScript in Java), *Tcl/Java*[103] (Tcl in Java), or *Kahlua*[104] (Lua in Java).

Beside scripting languages, there are also implementations of mature programming languages for the Java platform which follow the same technical concepts as the scripting languages. Examples are *Groovy*[105] and *Scala*[106]. Groovy extends the features of the Java language but also simplifies it, while Scala intends to be an alternative to the Java language to enable programmers to be more productive.

Languages based on the JVM could also be one of the next steps of the evolution of Squidy. Interactive design and development could be improved for the user by integration of multiple interpreting languages into Squidy. The user could then choose the language of his choice to implement processing tasks of a Node in place of being forced to use the Java language and dynamic compilation (see section 6.2.10). Together with the management interface and a source code injection functionality into Nodes, this would enable complete external scripting support for presumably all relevant use cases of Squidy.

Another way to use a scripting language in conjunction with Java is to use JPype[107]. “JPype is an effort to allow Python programs full access to Java class libraries. This is achieved not through re-implementing Python [...] but rather through interfacing at the native level in both Virtual Machines.”

7.8.5 Remote Access by Scripting Languages

If Cajo is used to implement the management interface, BeanShell would be a good choice for scripting. “Since BeanShell and cajo both use Java reflection as the basis for object method invocation; cajo makes it possible to have remote method invocations appear syntactically identical to local ones, using a very small wrapper.”

100 http://www.beanshell.org/
101 http://www.jython.org/
102 http://www.mozilla.org/rhino/
103 http://tcljava.sourceforge.net/
104 http://code.google.com/p/kahlua/
105 http://groovy.codehaus.org/
106 http://www.scala-lang.org/
107 http://jpype.sourceforge.net/

Scripting languages running in the JVM can make direct use of all features and techniques available for Java, including remote access and management technologies. Thus, the management interface can directly be accessed, too.

The technical approach of JPype requires a remote proxy client application as described in section 7.8.4 to access the management interface because the Python virtual machine it needs to interact with a JVM.

Last but not least, JavaScript can be used from a web browser which supports Java and JavaScript. If a proxy client application is implemented as a Java applet, it can be loaded into the web browser. Due to the integrated interface of the web browser between JavaScript and Java, JavaScript can gain full access to the management interface of Squidy.

7.9 Monitoring

The term *monitoring* combines tasks such as logging and inspection of data objects or system status. The following proposals rely on the management interface (see section 7.8). The management interface can offer functions to monitor the system status as it can directly access the classes containing this information.

Squidy already contains monitoring mechanisms for a user of Squidy Designer. The user gets visual feedback from a colored glow effect which represents the status of Nodes, Pipelines, and Pipes. He is able to visualize the data flow within each Pipe, he can use the *Data Recorder* Node for logging, and the GUI of each Node allows access to the textual error logging output of the Node.

However, there is no comprehensive possibility to inspect and trace data objects from the perspective of a software developer. Furthermore, textual logging of errors, warnings, and information is only performed if a developer cares for this task and does not forget to insert appropriate logging statements in his source code. More extensive monitoring facilities would enable a developer or an application using Squidy to detect problems faster. Thus it helps during research and evaluation of new configurations and to resolve malfunctions during practical use.

7.9.1 Inspection of Data Objects

Squidy already contains a monitoring feature which allows to inspect data objects passing through a Pipe. For this purpose, a *feedback object* which implements the interface *ProcessingFeedback* can register itself with a Pipe. However, this concept is currently limited to one feedback object per Pipe and it does allow only a momentary view on the data objects. Therefore, the following improvements shall be integrated into Squidy.

- Allow more than one feedback object per pipe by managing a collection of feedback objects. Thus, passing data objects can be inspected by multiple inspectors simultaneously, for example by Squidy Designer and via the management interface.

- An attribute consisting of a collection of visitor objects shall be added to the class *AbstractData*. A visitor object has to implement a new interface *IDataVistor* which follows the *visitor* design pattern (see section 2.3.1). The interface includes a *visit()* method which accepts *IProcessable* and *IData* as parameters.
- A visitor object has to implement serialization and deserialization to allow persistence and transfer via Squidy Remote. *IDataVistor* includes the methods *serialize()* and *deserialize()* which are called by *AbstractData*.
- An *acceptVisitor()* and a *dismissVisitor()* method which accept a visitor object implementing the interface *IDataVistor* shall be added to the interface *IData* and implemented in the class *AbstractData*. The methods store or remove the visitor object in or from its collection of visitor objects. This allows multiple visitor objects to be attached to or detached from a data object.
- Transformer Nodes which make use of their data allocator and create new outgoing data objects from their incoming data objects have to transfer all visitor objects to the corresponding new data objects (see section 7.3.3). Consequently, if a Node has multiple outgoing Pipes, the *publish()* method in *AbstractNode* has to copy all visitor objects to all instances of a published data object. If a Node has multiple incoming Pipes, possible duplicate visitor objects have to be eliminated. Therefore, *AbstractData* shall reject duplicate visitor objects which are added to its collection of visitors by the *acceptVisitor()* method.
- Piping objects shall provide the methods *addDataVisitorFactory()* and *removeDataVisitorFactory()* which operate on arguments of type *IDataVisitorFactory* and store them in an attribute consisting of a collection of visitor factory objects. *IDataVisitorFactory* is an interface following the *abstract factory* design pattern (see section 2.3.1) and provides a function *createDataVisitor()* which returns a new object which implements *IDataVisitor*.
- Every time a data object is published by a Piping object, it invokes *createDataVisitor()* of all visitor factory objects attached to it to create new data visitor objects. These new objects are attached to the collection of visitor objects of the data object by invoking its *acceptVisitor()* method.
- Every time a data object is published by a Piping object, the *visit()* method of all objects stored in the collection of visitor objects of the data object has to be invoked with the Piping object itself and the data object as parameters. *IData* and *AbstractData* shall provide a method *notifyVisitors()* for this task which accepts with *IProcessable* a parameter.
- All methods related to visitors shall be synchronized to allow safe concurrent processing.

The modifications described above allow to connect multiple inspectors to a Pipe and to trace the path of a data object throughout the processing chain. An external module or application can inject visitor objects to trace each data object starting from any

Piping object by providing an implementation of *IDataVisitorFactory*. The visitor factory is free to return no, always the same, or multiple custom implementations of visitor objects to inspect or even modify data objects. Additionally, visitor objects can remove themselves from a data object at any time.

7.9.2 Logging

Squidy includes basic logging by Apache log4j. As already mentioned in section 6.6, log4j should be replaced by SLF4J. Both frameworks do not significantly differ in their usage; logging methods have to be invoked manually by the developer.

Detailed logging, which allows to trace each function call automatically, can be done by using AOP (see section 2.2.2). Logging is a classical example when to apply AOP to implement a new concern throughout an entire application without affecting or modifying the source code of the rest of the application. AOP allows to define an aspect *logging* for any selection of join points defined by a pointcut. By adding the advices *before* and *after* and letting them invoke logging methods, entry and exit of each method matching the pointcut can be logged.[108]

Aspect-oriented logging can provide very detailed feedback with little effort. Logging control and output can be provided though the management interface, too. It is also supported by Java through the *AspectJ*[109] extension [22]. The Eclipse platform supports AOP by *AJDT*[110] *(AspectJ Development Tools).*

Moreover, the Spring framework integrates complete support for AOP. This feature can be used by applications which use the Spring framework as basis (see conclusion of section 7.8.2).

7.10 Circular Pipelines

It is possible to construct circular Pipelines with Squidy. This means output data objects of a Node returns via the processing chain to the input port of the Node. When such a Pipeline is started this can result in multiple presumably undesirable situations with negative consequences, depending on the behavior of the participating Nodes:

- A fixed number of data objects circulates in the Pipeline, uses processing power and does not perform any useful task.
- A continuously growing number of data objects circulates in the Pipeline until the program terminates due to a low memory situation.
- A deadlock occurs because resources occupied by the data objects or the data objects itself are not released due to their continuous circulation while the same resources shall be allocated elsewhere.
- A deadlock or stack overflow occurs when direct invocation (see section 7.4) is used because this situation results in an infinite recursive function call sequence.

108 http://en.wikipedia.org/wiki/Aspect-oriented_programming
109 http://eclipse.org/aspectj/
110 http://eclipse.org/ajdt/

Since Squidy does not solely operate locally but can also operate as a distributed system, it is necessary to find a solution which can cope with this situation. The solution either has to prevent construction of circular Pipelines or dissolve the negative consequences of circular Pipelines.

In rare cases circular Pipelines can be constructed intentionally. This can be useful to allow a simple way to provide feedback output from a Node as input to an upstream Node. The data filtering mechanism of Pipes allows to control the data flow accordingly. However, the user may forget to set these filters correctly and has to use different Squidy data types for the actual payload and the feedback data. A solution which handles the issue of circular Pipelines should still allow this Pipeline configuration but should eliminate its potential problems.

7.10.1 Prevent Construction

Chandy et al. [82] described an algorithm for *distributed deadlock detection*. Its basic idea is that a process waiting for a resource sends queries to all dependent processes containing the originator of the query and the last sender of the query. The query is updated and forwarded by all traversed processes to their dependent processes. If the query returns to the originator a deadlock situation is found.

A modified version of this algorithm could be used to prevent construction of circular processing Pipelines. This is basically the same situation except that the problematic resource is a circulating data object itself:

- The Processable class is extended by an immutable attribute for a *UUID (universally unique identifier)* which is created and assigned during instantiation of a Processable object. Object serialization of the Processable class is extended to support the UUID attribute. A UUID is required to identify a specific object instance to allow safe comparison with instances of the same class in other instances of Squidy.
- The new Squidy data type *DataPipe* is introduced. It contains a UUID for a Node and a UUID for a Pipe.
- When a connection between two Nodes shall be established, the employed Pipe is connected to the source Node first. The target Node publishes a new *DataPipe* object and initializes it with the UUID of the source Node of the Pipe and with the UUID of the Pipe. Additionally, it blocks execution for a definable timeout and waits for a notification containing both UUIDs to continue execution. In case of a timeout, execution continues normally. Otherwise if a matching notification arrives, connection between the two Nodes is rejected.
- When a Node receives a *DataPipe* object, it compares the contained UUID with its own UUID. If the UUIDs differ, the Node publishes the *DataPipe* object unmodified to all outgoing Pipes. Otherwise the Node looks for the outgoing Pipe matching the UUID contained in the *DataPipe* object and notifies the target Node of this Pipe. The notification includes both UUIDs as a distinctive feature for this notification.

This approach could be implemented in *AbstractNode*, transparently for inherited Node classes. However, it is not recommended to implement it because it has three major disadvantages:

First, either the procedure of connecting two Nodes has to be executed asynchronously in a separate thread, or it has to block execution of the calling thread to wait for the return of the published *DataPipe* object. In both cases a timeout has to be defined because no confirmation for absence of a circular Pipeline is available. It cannot be safely determined if a circular Pipeline exists because a circulating *DataPipe* object can still arrive after the timeout. If the connection between the Nodes is established regardless of this problem after the timeout has occured, data objects can be published over the Pipe and can cause one of the undesirable situations described above when a circular Pipeline exists.

Second, the data objects might not be able to return safely to the source Node. Either due to a partially stopped Pipeline or due to lost or dropped data objects for example because of lacking processing power or a temporarily unreachable remote target Node. It is possible to ignore the current processing state of a Pipeline or Pipeline segment and to always publish *DataPipe* objects, but it is not possible to avoid the problem of lost or dropped data objects. Thus it cannot be safely determined if a circular Pipeline exists.

Third, this approach requires that *DataPipe* objects are always routed through all Pipes independently of active data filters. This can cause detection of a circular data flow which is actually not existent. Moreover, it contradicts the demand that it should still be possible to construct a partially circular Pipeline if desired. It would be possible to solve these problems by adding and evaluating a list of data types to *DataPipe* objects. However, this would generate a complex process.

7.10.2 Eliminate Negative Consequences

Instead of detecting the circular Pipeline during connection of Nodes, it can be detected and treated during data processing. The basic idea of the approach described above remains but its major disadvantages are eliminated:

- The Node class is extended by an immutable attribute for a UUID as described above for Processables.
- The *IData* interface and the *AbstractData* class are extended by an immutable attribute to hold the UUID of the source Node. Object serialization of the *AbstractData* class is extended to support the UUID attribute. A UUID is required to identify a specific Node instance because during data processing, data objects can be converted to different data types and can be serialized and deserialized for transfer to other instances of Squidy.
- The Node class is extended by an integer attribute *dropped* which counts the number of dropped data objects.

- When a data object is instantiated, its UUID attribute is initialized with the UUID of the source Processable.
- When a Node receives a data object, it compares the contained UUID with its own UUID. If the UUIDs are equal the Node drops the data object and increases the count of *dropped*.
- When a Transformer Node which creates new data objects processes a data object, the UUID has to be transferred to the corresponding new data object (see section 7.3.3).

This functionality can be implemented in *AbstractNode*, transparently for inherited Node classes. The *dropped* attributes can be used to provide feedback to the user or to an application employing Squidy.

The drawback of this approach compared to the prevention of construction of a connection is the need of additional processing power for comparison of the UUIDs of each data object in each Node in the Pipeline during data processing. This drawback should have negligible impact because a single comparison of a 128-bit value (size of UUID) is a cheap operation.

7.11 Dynamic Reconnection

The process of dynamic reconnection of Nodes is currently not synchronized in Squidy. Read and write access on the involved objects is unsafe because they can be executed concurrently and interleaved. Thus it can lead to undefined behavior or abnormal program termination on connection or disconnection of a Pipe during execution of the method *AbstractNode.publish()*. The affected attributes are *Piping.outgoingPipes*, *Pipe.source*, *Pipe.target*, additionally all related getter and setter methods, and related operations.

The method *AbstractNode.publish()* contains a loop iterating over all connected outgoing Pipes of the Node. Access on the involved objects needs to be synchronized which can be done in several steps:

- Synchronize access to *Piping.outgoingPipes* by adding the synchronized attribute to the methods *AbstractNode.publish()*, *Piping.setOutgoingPipes()*, *Piping.addOutgoingPipe()*, and *Piping.removeOutgoingPipe()*.
- In *AbstractNode.publish()*, encapsulate the body of the loop over the outgoing Pipes by a synchronized block using the intrinsic lock provided by the Pipe object processed in the current loop iteration.
- Synchronize access to *Pipe.target* by adding the synchronized attribute to the methods *Pipe.getTarget()*, *Pipe.setTarget()*, and *Pipe.delete()*. Synchronization of *getTarget()* and *setTarget()* is not mandatory because the target attribute is initialized before a Pipe is added to a Node but it adds safety to the class. These methods might be used in different contexts or might be called in different order which can cause a failure.

For the latter reason, the methods *Pipe.getSource()* and *Pipe.getSource()* should be synchronized to protect access to the source attribute, too. Moreover, the incoming collection of Pipes in Piping should be synchronized equally, even if the current program flow does not trigger potential concurrency failures.

7.12 Further Improvements

This section describes some further approaches for improvements in Squidy in a very short form. Before they can be realized, they have to be reconsidered and specified. This is not done within this work because of constraints regarding the author's time available for this task and the permissible extent of this work. However, these ideas should not get lost.

Data Type Hints

Besides the visualization of Node types (see section 7.1), it would be convenient for a user of Squidy Designer to know which data types a particular Node handles on it input and output ports. Therefore, this information should be presented to the user. A concrete concept for a realization within the user interface has to be developed first. A possible approach can be a temporary information display when the mouse pointer touches the visual representation of a port. Moreover, Squidy Designer can present Nodes ordered by support for a specific data type.

Abstract Caching Class

Squidy currently contains some manually implemented object caches and multiple manually predefined static objects of the same type but with different properties because these objects are repeatedly required and should not be constructed and destructed at every use. A more consistent and convenient way to handle these objects can be an abstract caching class which can be used for any kind of object. This class can basically work like an allocator (see section 6.7.5) but can also suppress duplicate objects and allow simultaneous access to one particular object.

Index for Data Type Lookup

The various enumerated custom *process()* methods in *AbstactNode* for different data types are stored using a *HashMap* class for caching (see section 6.2.7). This is not an ideal solution because many lookups (for many data objects) in a hash map can become relatively expensive in terms of processing power.

Thus the hash map should be replaced by a simple array and *AbstractData* should carry an integer index for this array depending on its data type. The index value can be generated by data object allocators consistently for a specific data type within an instance of Squidy. Using an enumeration or static assignment of an integer index per data type (as currently contained in *IData*) is inappropriate because it must be maintained and synchronized with *IData* implementations manually.

Injection of Processing and Properties

A radical change of one of the basic models of Squidy helps to gain flexibility, reusability and separation of concerns. Concrete Nodes are derived from the *AbstractNode* class to implement their specific operations on data objects. Thereby data transport functionality of the Node and data processing functionality are mixed.

By using the *dependency injection* design pattern (see section 2.3.1) data processing can be separated from a Node. It can be realized with an interface which contains the abstract methods of *AbstractNode* and a *process()* method. Processing classes implementing this interface can then be injected into a Node. Moreover, *AbstractNode* does not need to be abstract anymore but can just be a finalized internal implementation of a Node. A developer of new Nodes does not need to care about inheritance from *AbstractNode* anymore.

Properties of Nodes are currently located directly in the Node implementation as attributes of the Node class and are enumerated using JAXB annotations for persistence and presentation to the user. They can be stored in a property class instead. This class can provide an universal interface[111] for all types of properties. The processing interface can refer to this property interface and reusable sets of properties could be implemented.

This circumvents the potential problem that changes in *AbstractNode* affect all derived Nodes. It also allows implementation of Squidy Bridges and all other data processing tasks as reusable modules. Processing functionality can be combined using the *proxy* design pattern and sets of properties can be combined in collections or using the *composite* design pattern (see section 2.3.1). The dependency injection also simplifies the realization of internal and external scripting (see section 7.8.4).

AOP for Persistence

Currently all attributes of classes have to be annotated manually for persistence functionality with JAXB. Even with the planned migration to the Hibernate persistence framework (see section 6.6) either annotations or XML configuration files have to be used to manually define for each class which attributes have be stored at which location.

With aspect-oriented programming (see section 2.2.2) it is possible to keep the concern of persistence away from the implementation of classes which should be stored. Persistence can be defined, maintained, and changed in an aspect at one centralized point. The aspect can be designed to match and to weave itself for example into new and existing Nodes and a developer of a Node does not have to know what he has to consider for persistence support because it is added automatically.

111 http://msdn.microsoft.com/en-us/library/bb761474(VS.85).aspx

Partial Image Filtering

Image filtering as it is required for multi-touch input or processing of other large data objects is commonly applied to an entire data object before the next processing step is applied. Therefore, a latency equal to the duration of data processing for one data object is introduced each time a processing step is applied to a real-time data source. It would be helpful to reduce this latency as far as possible to meet the requirements for an input framework (see section 3). A solution to achieve this can be partial data processing which are illustrated in the following using the example of image filtering.

Since most image filtering algorithms do not process the entire image simultaneously, images can be partitioned into smaller parts for transport and processing. It can also be necessary to leave a small overlap between image parts depending on the image filtering algorithm or to create image filters tailored to partitioned images. Additionally, a container data type for partitioned images can be introduced which automatically cares for decomposition and recomposition of the images.

Smaller parts of an image can be for example half of the image or only one line of pixels. In a processing chain using parallel processing, partitioning into continuously smaller parts can cause overall latency of the chain to converge to the processing time of one image of the slowest filter in the chain.[112] However, more parts cause more data objects and more overhead for handling of these data objects. Thus a suitable balance must be found.

Improved Configuration Management

Squidy currently uses one configuration file which stores the entire configuration of the workspace. There is no explicit functionality for the user to load and save specific configurations or files. The idea behind this limitation is that all data should be stored in a single workspace which can be hierarchically organized by grouping of objects and semantic zooming.

The current implementation has some drawbacks such as no folders similar to a file system, no moving of objects between hierarchical levels, no user rights management, no possibility for partial copies or backups, etc. Constant development of the project can become a problem because of incompatible changes in a single Node implementation. This happened multiple times to the author during exploration and testing of Squidy for this work and can lead to an unusable configuration file containing the entire workspace.

These findings should be considered when the persistence functionality of Squidy is modified to work with Hibernate (see section 6.6).

112 An example: In a processing chain consisting of 5 filters each filter requires 10 ms (*tf*) to process an entire image data object. This sums up to 50 ms latency (*tl*) for an image when it has passed the processing chain. Now, the data object is partitioned into 10 parts (*p*), each filter requires 1/10 of 10 ms to process one part which is 1 ms. This results in 5 ms latency (*tlp*) until processing of the first part is finished and seems to sum up to 50 ms again for all 10 parts. But the first filter in the chain already starts to process the second part after 1 ms which also takes 5 ms to pass the filter chain and passes the last filter 1 ms after the first part. After only 5 + 9 = 14 ms the entire image has passed the processing chain ($tl = tlp + (p - 1) * tf / p$ with *tf* being the processing time of the slowest filter and if partitioning into parts scales linearly and all filters process data in parallel).

Bridge to libtisch

One of the requirements defined for this work is a connection to *libtisch* (see sections 3 and 4.3.5). To fulfill this requirement, a Bridge Node for the libtisch protocols can be developed.

Tools for Bridge Implementations

This is just a note for people interested in developing Squidy Bridges to native programming languages: Tools such as *SWIG*[113] *(Simplified Wrapper and Interface Generator)*, *cxxwrap*[114], or *xFunction*[115] can automate and simplify this task. They relieve the developer from implementing and maintaining JNI source code manually.

113 http://www.swig.org/
114 http://cxxwrap.sourceforge.net/
115 http://www.excelsior-usa.com/xfunction.html

8 Remarkable Features of Competitors to Squidy

Some of the projects which were compared to Squidy in sections 4 and 5 offer remarkable features which are not or not equally included in Squidy. The following selection of features emphasizes and describes their singularity and should be another inspiration for the future development of Squidy.

8.1 Input Interpretation

Squidy only contains basic functionality in terms of multi-touch input abstraction (see section 7.5). Blob tracking and rudimentary identification of fingers is integrated in the multi-touch Node. Compared to specialized systems for multi-touch input, it is a simple and inflexible solution. This limitation results from the need for a tailored solution for previous research projects. More powerful solutions can be easily integrated in Squidy but have not been realized, yet.

The *mu3* framework (see section 4.3.6) concentrates on this functional gap. It abstracts multi-touch input data, in particular detected blob coordinates, into information about pressure, distance, type of touch and the tangible which interacts with the multi-touch surface. This information is combined to segments. These segments are tracked and analyzed and as a result, multi-touch events in terms of *new*, *released*, *split*, *joined*, *joined and split*, and *tracked* touch are generated. These events refer to an identifiable touch object with properties for origin, bounds, time, device, path, and ambiguity.

mu3 is currently in an early state of development and written in Java, as is Squidy. Therefore, it can be integrated directly by implementing a suitable Node.

8.2 Gesture Recognition

Gesture recognition in Squidy is based on the *wiigee*[116] library. Wiigee is specialized on interpretation of data generated by acceleration sensors, especially by the sensors of the Nintendo Wiimote. Usage of wiigee is comfortable because it can be trained for new gestures by the user who simply performs the desired gesture in a training mode. However, wiigee is not tailored to recognition of complex multi-touch gestures.

Expertise in multi-touch and tangible gesture recognition and input interpretation can be contributed by *TWING* (see section 4.5.3). It has a sophisticated but simple architecture which hands over incoming *Requests* (multi-touch events) to *Behaviors* which match a Request. Behaviors such as a *OneFingerScrollingBehavior* issue *Commands* such as a *ScrollCommand* which are handled and published by a *Command Processor*. The resulting behaviors are lightweight, reusable, exchangeable at runtime, and separate concerns [57]. TWING currently includes 22 Commands and 23 Behaviors.

116 http://wiigee.org/

TWING is written in C# using Microsoft Visual Studio and requires Windows and the .NET Framework to run. Since it concentrates on data processing and does only few interaction with the underlying platform, TWING should also work on the Mono platform to be independent from the operating system. Its architecture allows to connect input and output interfaces to use for example TUIO or another OSC-based protocol. However, it is also possible to port the C# class structure of TWING to Java without large modifications for integration into Squidy because both programming languages have comparable language features.

8.3 Input Abstraction

As described in the preceding section, the *mu3* framework abstracts multi-touch input data. The *Unit* framework (see section 4.1.1) goes one step further and defines an *interaction technique* abstraction layer for all kinds of input devices. The concept for the abstraction layer of Unit involves that interaction techniques can be constructed from input data of devices or from other interaction techniques. Moreover, interaction techniques are modular and can be exchanged with others during runtime.

Unit has a Pipes and Filters architecture which is roughly comparable to Squidy's. Its interaction technique abstraction layer is implemented as a set of units which are comparable to Nodes. This concept complies well with the concepts of Squidy and can be integrated by creating a set of Nodes and a unified interaction technique command data type for the same purpose. Gestures are a concrete form of interaction techniques. Thus Squidy already contains limited support for interaction techniques by means of its gesture recognition Nodes. However, comprehensive support of interaction techniques would relieve applications using Squidy of the task of data interpretation and data mapping to application semantics and would make applications more independent from the used input devices.

The *Papier-Mâché* toolkit (see section 4.5.1) also integrates complex mechanisms for input abstraction and is a potential source for ideas for a comparable solution for Squidy.

8.4 Image Filtering

The current implementation of the multi-touch Node and the proposal for a new multi-touch input concept (see section 7.5.3) already include a solution for fast image filtering. The *CCV* and *Touché* frameworks (see section 4.3.2, 4.3.17) also execute image filtering on the GPU but they use OpenGL pixel shaders for this task. This can be an alternative to the use of CUDA and OpenCL.

Although OpenGL pixel shaders have only limited functionality compared to GPGPU, they have some advantages. Their limited functionality makes them easier to understand for developers. Furthermore, an application might want to display the images on the screen after they have been processed by the GPU. It would save processing power and time to display the image which is already located as a 2D texture in the memory of the graphics adapter by simply using an OpenGL command instead of transferring the image to the host memory and back.

This feature might require that the displaying application runs in the same JVM or process as Squidy. In any event it requires that the displaying application runs on the same platform as Squidy to allow access to the graphics adapter. Finally, OpenGL and OpenCL can also cooperate and access the same memory locations on the graphics hardware.

8.5 Fiducial Markers

The *reacTIVision* framework (see section 4.5.2) is one of the few frameworks which is able to detect and track fiducial markers. Another fiducial maker tracker is integrated in *ARToolKit*[117] which was not included in the comparison of all frameworks because it is mainly a solution for prototyping and realization of small augmented reality projects.

Squidy currently supports input from reacTIVison via TUIO. With the image processing chain proposed in section 7.5.3, a feature similar to reactTIVision's fiducial marker tracking could easily be added directly to Squidy. The *fidtrack* library is part of reactTIVision and handles the marker processing. It is licensed under the *LGPLv2*[118] *(Lesser General Public License 2.1)*. Its integration would allow to omit the additional separate framework for fiducial markers.

8.6 Data Processing and Synchronization

In contrast to the preceding ideas which are about functionality, this proposal refers to the Pipes and Filters architecture of Squidy.

DataLaViSTA (see section 4.1.2) uses a Pipes and Filters architecture, multi-threading, actively processing pipes, handles synchronization issues, introduces push and pull models for data transfer, can cope with upstream and downstream data, supports distributed processing, and encapsulates system specific implementations to handle inter-process communication. It also integrates a concept similar to allocators (see section 6.7.5) which is called *packet recycling.*

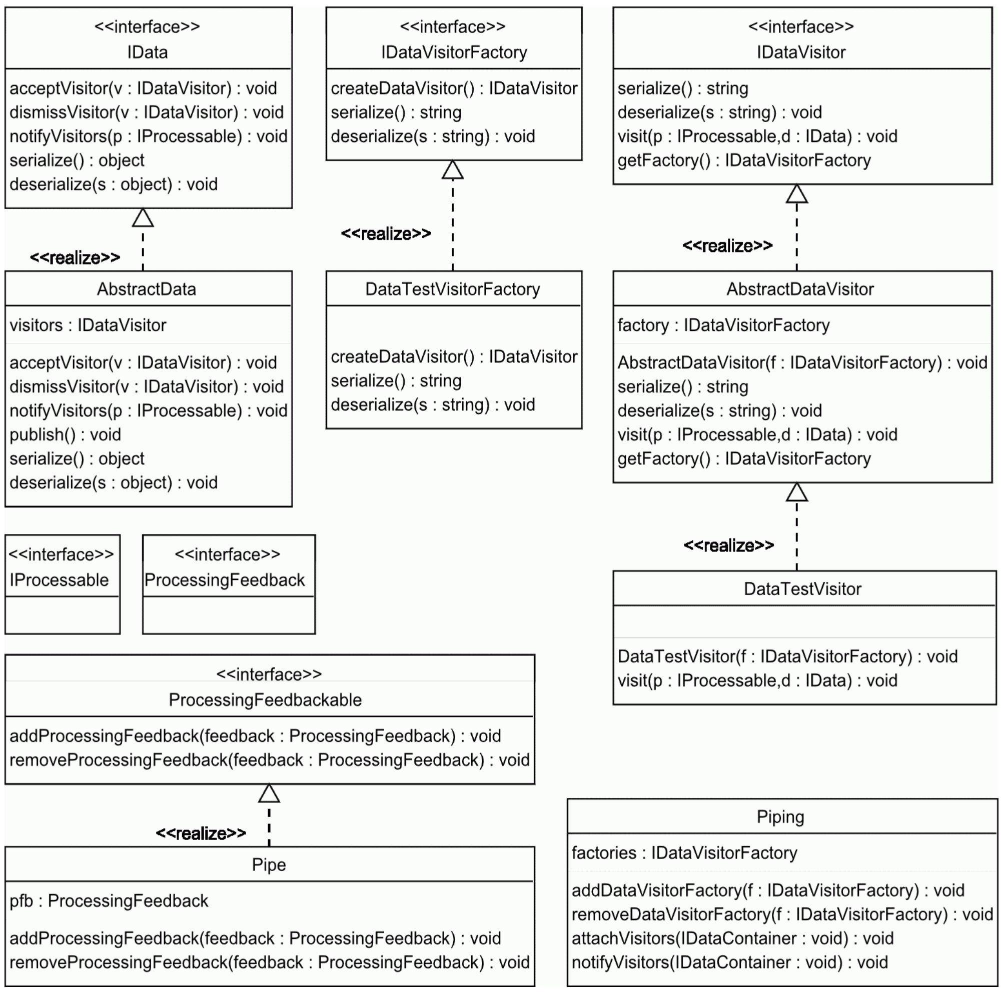
This makes DataLaViSTA an interesting reference implementation of the Pipes and Filters architectural pattern. Inspection of the source code is likely to reveal new technological ideas and solutions for Squidy. According to the project website the source code of DataLaViSTA is freely available for research institutions.

117 http://artoolkit.sourceforge.net/
118 http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

9 Implementation of Data Object Monitoring using Visitors

In the course of this diploma thesis a data object monitoring mechanism based on visitor objects (see section 7.9.1) has been implemented for Squidy. This mechanism has been chosen for implementation because it has no dependencies to most of the other proposals described in section 7 which show a sequential dependence. Moreover, the extent of required modifications for the realization was manageable within the time constraints for this work.

The proposal for data object monitoring includes an abstract factory interface for data visitor objects which allows to provide a custom factory producing any kind of visitor objects. During implementation of serialization of data visitor objects it turned out that the corresponding factory has to be serialized, too. Otherwise deserialization is not able to reconstruct data visitor objects because these are only produced by their specific factory. Therefore, serialization was extended to *IDataVisitorFactory*.



<<interface>>
IData
acceptVisitor(v : IDataVisitor) : void
dismissVisitor(v : IDataVisitor) : void
notifyVisitors(p : IProcessable) : void
serialize() : object
deserialize(s : object) : void
<<interface>>
IDataVisitorFactory
createDataVisitor() : IDataVisitor
serialize() : string
deserialize(s : string) : void
<<interface>>
IDataVisitor
serialize() : string
deserialize(s : string) : void
visit(p : IProcessable,d : IData) : void
getFactory() : IDataVisitorFactory
<<realize>>
AbstractData
visitors : IDataVisitor
acceptVisitor(v : IDataVisitor) : void
dismissVisitor(v : IDataVisitor) : void
notifyVisitors(p : IProcessable) : void
publish() : void
serialize() : object
deserialize(s : object) : void
DataTestVisitorFactory
createDataVisitor() : IDataVisitor
serialize() : string
deserialize(s : string) : void
AbstractDataVisitor
factory : IDataVisitorFactory
AbstractDataVisitor(f : IDataVisitorFactory) : void
serialize() : string
deserialize(s : string) : void
visit(p : IProcessable,d : IData) : void
getFactory() : IDataVisitorFactory
<<interface>>
IProcessable
<<interface>>
ProcessingFeedback
DataTestVisitor
DataTestVisitor(f : IDataVisitorFactory) : void
visit(p : IProcessable,d : IData) : void
<<interface>>
ProcessingFeedbackable
addProcessingFeedback(feedback : ProcessingFeedback) : void
removeProcessingFeedback(feedback : ProcessingFeedback) : void
Pipe
pfb : ProcessingFeedback
addProcessingFeedback(feedback : ProcessingFeedback) : void
removeProcessingFeedback(feedback : ProcessingFeedback) : void
Piping
factories : IDataVisitorFactory
addDataVisitorFactory(f : IDataVisitorFactory) : void
removeDataVisitorFactory(f : IDataVisitorFactory) : void
attachVisitors(IDataContainer : void) : void
notifyVisitors(IDataContainer : void) : void

Illustration 7: UML diagram of the implementation of data object visitors

Additionally, a data visitor object has to know the factory by which it was produced which is also the one required for serialization. Thus *IDataVisitor* has been extended to provide this information. Moreover, the class *AbstractDataVisitor* has been introduced as a base class for data visitor objects. It stores the reference to its factory and forces a developer to initialize the factory attribute by providing solely a parameterized constructor method.

The source code included in the printed version of this work contains entirely new files created for the implementation and differences to the existing source code of Squidy. Required changes in the *import* section for modified files have been omitted in the printed version. The electronic version includes the entire source code of Squidy and a set of all files modified for this implementation.

For testing purposes, the Squidy Node *MouseIO* which processes input from the computer's mouse was modified. A test implementation of a data visitor object factory producing data visitor objects which output the mouse coordinate to the debugging console was added. Furthermore, a *MouseIO* Node, a *FlipXY2D* Node, and an empty Node were composed to a Pipeline. The *FlipXY2D* Node swaps x and y coordinates of 2D coordinates. When being executed, the logging output of this Pipeline shows the messages of the visitor objects attached to *DataPosition2D* objects traveling through the processing chain. These messages are generated each time a data object is being published through a Pipe between two Nodes:

```
DEBUG ... de.ukn.hci.squidy.extension.basic.MouseIO: x = 0,342 y = 0,485
DEBUG ... de.ukn.hci.squidy.extension.basic.FlipXY2D: x = 0,485 y = 0,342
DEBUG ... de.ukn.hci.squidy.extension.basic.MouseIO: x = 0,321 y = 0,493
DEBUG ... de.ukn.hci.squidy.extension.basic.FlipXY2D: x = 0,493 y = 0,321
DEBUG ... de.ukn.hci.squidy.extension.basic.MouseIO: x = 0,283 y = 0,519
DEBUG ... de.ukn.hci.squidy.extension.basic.FlipXY2D: x = 0,519 y = 0,283
DEBUG ... de.ukn.hci.squidy.extension.basic.MouseIO: x = 0,250 y = 0,530
DEBUG ... de.ukn.hci.squidy.extension.basic.FlipXY2D: x = 0,530 y = 0,250
DEBUG ... de.ukn.hci.squidy.extension.basic.MouseIO: x = 0,213 y = 0,541
```

The source code for data object monitoring using visitors can be found in the Appendix of this work.

10 Conclusion

Overall 28 existing input libraries, frameworks and toolkits have been presented, reviewed, and rated. The rating result is a definite justification for the recommendation to chose the Squidy Interaction Library as the software platform for processing of input data for the Curve project.

The amount of related work that serves almost identical purposes, while coming from several different areas of research, shows that it is profitable to stay informed about developments from "foreign" domains. If the domain of multi-modal interaction had not been considered, the Squidy Interaction Library, a versatile input framework fit for use, might have remained undiscovered. Furthermore, the initial approach to start a new software development project for an input software platform for the Curve project would have consumed much more resources than the way of adapting Squidy will consume.

The description of Squidy from the perspective of software development complements the recent publications from its inventors. Future software development on Squidy, on the Curve project, and in fact any project employing Squidy will benefit from these insights and the manifold recommendations introduced in this work.

Besides realization of the suggested improvements of multi-touch support, the next required steps to advance the Curve project will be further research on image stitching techniques and their integration into Squidy. Results from these efforts and practical evaluation of the Curve desk will show how the project could benefit from adding support for camera calibration and reconstruction of spatial object positions. Implementation of data synchronization, allocators for data objects, and monitoring functionality will improve reliability and performance and supports further development and evaluation processes of Squidy.

Last but not least, the included tabular survey is also of independent interest: By adapting its rating parameters it can be used to make well founded decisions for other projects which require a software platform for input processing as well. Thus it can hopefully serve as a valuable guide for anyone looking for a basis for his own work.

Bibliography

[1] Raphael Wimmer, Florian Schulz, Fabian Hennecke, Sebastian Boring, Heinrich Hußmann. Curve: Blending Horizontal and Vertical Interactive Surfaces. In *Adjunct Proceedings of the 4th IEEE Workshop on Tabletops and Interactive Surfaces (IEEE Tabletop 2009).* 2009.

[2] Johannes Schöning, Peter Brandl, Florian Daiber, Florian Echtler, Otmar Hilliges, Jonathan Hook, Markus Löchtefeld, Nima Motamedi, Laurence Muller, Patrick Olivier, Tim Roth, Ulrich von Zadow. Multi-Touch Surfaces: A Technical Guide, 2008, [p. 19].

[3] Jefferson Y. Han. Low-Cost Multi-Touch Sensing through Frustrated Total Internal Reflection. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology.* 2005.

[4] F. Echtler, T. Sielhorst, M. Huber, G. Klinker. A Short Guide to Modulated Light. In *TEI '09: Proceedings of the conference on tangible and embedded interaction.* 2009.

[5] Werner A. König, Roman Rädle, Harald Reiterer. Interactive Design of Multimodal User Interfaces - Reducing technical and visual complexity. In: Journal on Multimodal User Interfaces (JMUI), 2009

[6] Werner A. König, Roman Rädle, Harald Reiterer. Squidy: A Zoomable Design Environment for Natural User Interfaces. In *CHI EA '09: Proceedings of the 27th International Conference Extended Abstracts on Human Factors in Computing Systems.* 2009.

[7] H.C. Jetter, W.A. König, H. Reiterer. Understanding and Designing Surface Computing with ZOIL and Squidy, 2009.

[8] Christoph Endres, Andreas Butz, Asa MacWilliams. A Survey of Software Infrastructures and Frameworks for Ubiquitous Computing. In: Mobile Information Systems 1, January 2005, [pp. 41-80].

[9] Pablo Figueroa, Walter F. Bischof, Pierre Boulanger, H. James Hoover. Efficient comparison of platform alternatives in interactive virtual reality applications. In: Int. J. Hum.-Comput. Stud. 1. Duluth, MN, USA, 2005, [pp. 73-103].

[10] Bruno Dumas, Denis Lalanne, Dominique Guinard, Reto Koenig, Rolf Ingold. Strengths and weaknesses of software architectures for the rapid creation of tangible and multimodal interfaces. In *TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction.* 2008.

[11] NUI Group Authors. Multi-Touch Technologies. NUI Group, 2009.

[12] Scott R. Klemmer, James A. Landay. Toolkit Support for Integrating Physical and Digital Interactions. In: Human-Computer Interaction 3, July 2009, [pp. 315-366].

[13] R.S. Pressman, D. Ince. Software Engineering: A Practitioner's Approach. McGraw-Hill New York, 2005.

[14] Daniel Mölle. Design by Demut - Die Auswirkungen von Entwurfsentscheidungen. In: iX Magazin für professionelle Informationstechnik 9. Hannover, Germany, September 2009, [pp. 92-95].

[15] R.C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR Upper Saddle River, NJ, USA, 2008.

[16] Tom Mens, Michel Wermelinger. Separation of Concerns for Software Evolution. In: Journal of Software Maintenance 5. New York, NY, USA, 2002, [pp. 311-315].

[17] E.W. Dijkstra. On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective New York, NY, USA, 1982, [pp. 60-66].

[18] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[19] K. Beck, C. Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2004.

[20] William F. Opdyke. Refactoring Object-oriented Frameworks. 1992.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, J. Irwin. Aspect-Oriented Programming, 1997.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold. An Overview of AspectJ. In: Lecture Notes in Computer Science, 2001, [pp. 327-353].

[23] C. Alexander, S. Ishikawa, M. Silverstein. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, USA, 1977.

[24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., 1995.

[25] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004

[26] Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Vol. 1. John Wiley & Sons, Inc., 2002.

[27] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-oriented Software Architecture: A System of Patterns. John Wiley & Sons, Inc., 1996.

[28] D. Garlan, M. Shaw. An Introduction to Software Architecture. In: Advances in Software Engineering and Knowledge Engineering, 1993, [pp. 1-40].

[29] H. Zimmermann. OSI reference model - The ISO model of architecture for open systems interconnection. In: IEEE Transactions on communications 4, 1980, [pp. 425-432].

[30] Ingo Assenmacher, Bernd Hentschel, Marc Wolter, Torsten Kuhlen. DataLaViSTA: A Packet-based Pipes and Filters Architecture for Data Handling in Virtual Environments. September, 2006.

[31] Microsoft Windows SDK Documentation (MSDN) - DirectShow. http://msdn.microsoft.com/en-us/library/dd375454(VS.85).aspx.

[32] Glenn E. Krasner, Stephen T. Pope. A Cookbook for using the Model-View Controller User Interface Paradigm in Smalltalk-80. In: J. Object Oriented Program. 3. Denville, NJ, USA, 1988, [pp. 26-49].

[33] W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc. New York, NY, USA, 1998.

[34] B. Foote, J. Yoder. Big Ball of Mud. In: Pattern Languages of Program Design 654-692, 2000, [p. 99].

[35] Rance Cleaveland, Scott A. Smolka. Strategic Directions in Concurrency Research. In: ACM Comput. Surv. 4. New York, NY, USA, 1996, [pp. 607-625].

[36] P. Gepner, MF Kowalik. Multi-Core Processors: New way to Achieve High System Performance, 2006.

[37] Jason I. Hong, James A. Landay. An Infrastructure Approach to Context-Aware Computing. In: Hum.-Comput. Interact. 2. Hillsdale, NJ, USA, 2001, [pp. 287-303].

[38] Ralph E. Johnson. Frameworks = (Components + Patterns). In: Commun. ACM 10. New York, NY, USA, 1997, [pp. 39-42].

[39] Mohamed Fayad, Douglas C. Schmidt. Object-Oriented Application Frameworks. In: Commun. ACM 10. New York, NY, USA, 1997, [pp. 32-38].

[40] Eric von Hippel, Ralph Katz. Shifting Innovation to Users via Toolkits. In: Manage. Sci. 7. Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, 2002, [pp. 821-833].

[41] Alex Olwal, Steven Feiner. Unit: Modular Development of Distributed Interaction Techniques for highly Interactive User Interfaces. In *GRAPHITE '04: Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. 2004.

[42] Alex Olwal, Steven Feiner. Interaction Techniques using prosodic Features of Speech and Audio Localization. In *IUI '05: Proceedings of the 10th International Conference on Intelligent User Interfaces*. 2005.

[43] Christian Sandor, Alex Olwal, Blaine Bell, Steven Feiner. Immersive Mixed-Reality Configuration of Hybrid User Interfaces. In *ISMAR '05: Proceedings of the 4th IEEE/ACM International Symposium on Mixed and Augmented Reality*. 2005.

[44] T. Van Reimersdahl, T. Kuhlen, A. Gerndt, J. Henrichs, C. Bischof. ViSTA: a multimodal, platform-independent VR-Toolkit based on WTK, VTK, and MPI. June, 2000.

[45] Jean-Yves Lionel Lawson, Ahmad-Amr Al-Akkad, Jean Vanderdonckt, Benoit Macq. An Open Source Workbench for prototyping Multimodal Interactions based on Off-the-Shelf Heterogeneous Components. In *EICS '09 Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2009.

[46] M. Kaltenbrunner, T. Bovermann, R. Bencina, E. Costanza. TUIO: A Protocol for Table-Top Tangible User Interfaces. In *Proc. of the The 6th Int⊠Workshop on Gesture in Human-Computer Interaction and Simulation*. 2005.

[47] Christopher Wolfe, J. David Smith, T. C. Nicholas Graham. A Low-cost Infrastructure for Tabletop Games. In *Future Play '08: Proceedings of the 2008 Conference on Future Play.* 2008.

[48] Florian Echtler, Gudrun Klinker. A Multitouch Software Architecture. In *NordiCHI '08: Proceedings of the 5th Nordic Conference on Human-Computer Interaction.* 2008.

[49] Florian Echtler, Manuel Huber, Gudrun Klinker. Shadow Tracking on Multi-Touch Tables. In *AVI '08: Proceedings of the Working Conference on Advanced Visual Interfaces.* 2008.

[50] Michiel Hakvoort. A Unifying Input Framework for Multi-Touch Tables. In *10thTwente Student Conference on IT.* 2009.

[51] S. Hafeneger, M. Weiss, G. Herkenrath, J. Borchers. PocketTable: Mobile Devices as Multi-Touch Controllers for Tabletop Application Development. In *Proceedings of Extended Abstracts of Tabletop '08.* 2008.

[52] Malte Weiss, Julie Wagner, Yvonne Jansen, Roger Jennings, Ramsin Khoshabeh, James D Hollan, Jan Borchers. SLAP Widgets: Bridging the Gap Between Virtual and Physical Controls on Tabletops. In *CHI '09: Proceeding of the twenty-seventh annual SIGCHI conference on Human factors in computing systems.* 2009.

[53] Prasad Ramanahally, Stephen Gilbert, Thomas Niedzielski, Desirée Velázquez, Cole Anagnost. Sparsh UI: A Multi-Touch Framework for Collaboration and Modular Gesture Recognition. In: ASME Conference Proceedings 43376, 2009, [pp. 137-142].

[54] Scott R. Klemmer. Papier-Mâché: Toolkit Support for Tangible Interaction. In *Adjunct Proceedings of the UIST.* 2003.

[55] Scott R. Klemmer, Jack Li, James Lin, James A. Landay. Papier-Mache: Toolkit Support for Tangible Input. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems.* 2004.

[56] Martin Kaltenbrunner, Ross Bencina. reacTIVision: A Computer-Vision Framework for Table-Based Tangible Interaction. In *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction.* 2007.

[57] Andreas Angerer, Markus Bischof, Bettina Conradi, Peter Lachenmaier, Kai Linde, Max Meier, Philipp Pötzl, Elisabeth André. TWING Framework: An architecture targeting the challenges of multitouch and tangible input on a table surface.

[58] Markus Bischof, Bettina Conradi, Peter Lachenmaier, Kai Linde, Max Meier, Philipp Pötzl, Elisabeth André. Xenakis: Combining tangible interaction with probability-based musical composition. In *TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction.* 2008.

[59] H.-C. Jetter, W. A. König, J. Gerken, H. Reiterer. ZOIL - A Cross-Platform User Interface Paradigm for Personal Information Management. In *Personal Information Management: PIM 2008, CHI 2008 Workshop.* 2008.

[60] Werner A. König. Referenzmodell und Machbarkeitsstudie für ein neues Zoomable User Interface Paradigma. University of Konstanz. 2006.

[61] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. In: Computer 8. Los Alamitos, CA, USA, 1983, [pp. 57-69].

[62] I. Scott MacKenzie, Colin Ware. 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*. 1993.

[63] Wesley M. Johnston, J. R. Paul Hanna, Richard J. Millar. Advances in Dataflow Programming Languages. In: ACM Comput. Surv. 1. New York, NY, USA, 2004, [pp. 1-34].

[64] R. Tsai. A versatile Camera Calibration Technique for High-accuracy 3D Machine Vision Metrology using Off-the-Shelf TV Cameras and Lenses. In: IEEE Journal of robotics and Automation 4, 1987, [pp. 323-344].

[65] YI Abdel-Aziz, HM Karara. Direct Linear Transformation from Comparator Coordinates into Object Space Coordinates in Close-Range Photogrammetry, 1971.

[66] GT Marzan, HM Karara. A Computer Program for Direct Linear Transformation Solution of the Colinearity Condition, and some Applications of it, 1975.

[67] Richard Hartley, Sing Bing Kang. Parameter-Free Radial Distortion Correction with Center of Distortion Estimation. In: IEEE Transactions on Pattern Analysis and Machine Intelligence 8. Los Alamitos, CA, USA, 2007, [pp. 1309-1321].

[68] Thomas Stehle, Daniel Truhn, Til Aach, Christian Trautwein, Jens Tischendorf. Camera Calibration for Fish-Eye Lenses in Endoscopy with an Application to 3D Reconstruction. In *Proceedings IEEE International Symposium on Biomedical Imaging (ISBI)*. 2007.

[69] J. Mitchelson, A. Hilton. Wand-based multiple Camera Studio Calibration, 2003

[70] DLT Method. http://www.kwon3d.com/theory/dlt/dlt.html.

[71] C. Rocchini, P. Cignoni, C. Montani, P. Pingi, R. Scopigno. A low cost 3D Scanner based on Structured Light, 2001.

[72] Hiroshi Kawasaki, Ryo Furukawa, Ryusuke Sagawa, Yasushi Yagi. Dynamic scene shape reconstruction using a single structured light pattern. In: Computer Vision and Pattern Recognition, IEEE Computer Society Conference, Los Alamitos, CA, USA, 2008, [pp. 1-8].

[73] Stephan Rupp, Matthias Elter, Michael Breitung, Walter Zink, Christian Kueblbeck. Robust Camera Calibration using Discrete Optimization. In: International Journal of Applied Science 13, 2006, [pp. 250-254].

[74] Stephan Rupp. A modular software framework for camera calibration. In *SEPADS'06: Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*. 2006.

[75] Thomas Haenselmann, Marcel Busse, Stephan Kopf, Thomas King, Wolfgang Effelsberg. Multicamera Video-Stitching, June 2006

[76] Matthew Brown, David G. Lowe. Automatic Panoramic Image Stitching using Invariant Features. In: Int. J. Comput. Vision 1. Hingham, MA, USA, 2007, [pp. 59-73].

[77] Martin Byröd, Matthew Brown, Kalle Åström. Minimal Solutions for Panoramic Stitching with Radial Distortion. In *Proceedings of the British Machine Vision Conference (BMVC).* 2009.

[78] Richard Szeliski. Image Alignment and Stitching: A Tutorial. In: Found. Trends. Comput. Graph. Vis. 1. Hanover, MA, USA, 2006, [pp. 1-104].

[79] H. Dhand, LP Daggubati. Towards Obtaining an Ideal Real Time Panoramic Video. In: Lecture Notes in Computer Science, 2006, [p. 698].

[80] Mai Zheng, Xiaolin Chen, Li Guo. Stitching Video from Webcams. In *ISVC '08 Proceedings of the 4th International Symposium on Advances in Visual Computing, Part II.* 2008.

[81] Wai-Kwan Tang, Tien-Tsin Wong, Pheng-Ann Heng. A System for Real-time Panorama Generation and Display in Tele-immersive Applications. In: IEEE Transactions on Multimedia 2, 2005, [pp. 280-292].

[82] K. Mani Chandy, Jayadev Misra, Laura M. Haas. Distributed Deadlock Detection. In: ACM Trans. Comput. Syst. 2. New York, NY, USA, 1983, [pp. 144-156].

Appendix

Source Code of Data Object Monitoring using Visitors

```
File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/ProcessingFeedbackable.java
Changes: extended interface for multiple simultaneous feedbacks

public interface ProcessingFeedbackable {
   public void addProcessingFeedback(ProcessingFeedback feedback);
   public void removeProcessingFeedback(ProcessingFeedback feedback);
}
```

```
File: core/squidy-designer/src/main/java/de/ukn/hci/squidy/designer/model/PipeShape.java
Changes: changed call according to interface

   public void initialize() {
      [...]
      pipe.addProcessingFeedback(visualization);
      [...]
   }
```

```
File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/model/Pipe.java
Changes: implemented support for multiple simultaneous feedbacks

   public IDataContainer process(IDataContainer dataContainer) {
      [...]
      // Processing feedback.
      if (processingFeedback != null) {
         for (ProcessingFeedback feedback : processingFeedback) {
            feedback.feedback(incomingData.toArray(new IData[incomingData.size()]));
         }
      }
      [...]
   }

   private Collection<ProcessingFeedback> processingFeedback;

   public void addProcessingFeedback(ProcessingFeedback feedback) {
      if (feedback == null) {
         return;
      }
      if (processingFeedback == null) {
         processingFeedback = new ArrayList<ProcessingFeedback>();
      } else if (processingFeedback.contains(feedback)) {
         // prevent duplicates
         return;
      }
      processingFeedback.add(feedback);
   }

   public void removeProcessingFeedback(ProcessingFeedback feedback) {
      if (processingFeedback != null && feedback != null) {
         processingFeedback.remove(feedback);
      }
   }
```

```
File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/IData.java
Changes: extended interface

public interface IData {
   [...]
   public boolean acceptVisitor(IDataVisitor visitor);
   public boolean dismissVisitor(IDataVisitor visitor);
   public void notifyVisitors(IProcessable<?> processable);
   [...]
}
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/AbstractData.java
Changes: added collection and handling of visitors, extended serialization

```java
   // all visitors assigned to this object
   private Collection<IDataVisitor> visitors;

   /**
    * @param visitor to add
    * @return true if visitor has been added
    */
   public boolean acceptVisitor(IDataVisitor visitor) {
      if (visitor == null) {
         return false;
      }
      if (visitors == null) {
         visitors = new ArrayList<IDataVisitor>();
      } else if (visitors.contains(visitor)) {
         // prevent duplicates
         return false;
      }
      return visitors.add(visitor);
   }

   /**
    * @param visitor to remove
    * @return true if visitor has been removed
    */
   public boolean dismissVisitor(IDataVisitor visitor) {
      return (visitors != null) ? visitors.remove(visitor) : false;
   }

   /**
    * notify all visitors that we are ready to receive their visit
    */
   public void notifyVisitors(IProcessable<?> processable) {
      if (visitors != null && processable != null) {
         for (IDataVisitor visitor : visitors) {
            visitor.visit(processable, this);
         }
      }
   }

   public void deserialize(Object[] serial) {
      source = ReflectionUtil.loadClass((String) serial[0]);
      timestamp = TimeUtility.getTimestamp((String) serial[1]);
      attributes = CoreUtility.getAttributesOfSerial((String) serial[2]);
      visitors = CoreUtility.getVisitorsOfSerial((String) serial[3]);
   }

   public Object[] serialize() {
      String attributesSerial = CoreUtility.getSerialOfAttributes(attributes);
      String visitorsSerial = CoreUtility.getSerialOfVisitors(visitors);
      return new Object[] { source.getName(), String.valueOf(timestamp), attributesSerial,
                            visitorsSerial };
   }
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/IDataVisitor.java
Changes: new file, definition of IDataVisitor

```java
package de.ukn.hci.squidy.manager.data;

import de.ukn.hci.squidy.manager.IProcessable;

public interface IDataVisitor {
   public IDataVisitorFactory  getFactory();
   public void                 visit(IProcessable<?> processable, IData data);
   public String               serialize();
   public void                 deserialize(String serial);
}
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/IDataVisitorFactory.java
Changes: new file, definition of IDataVisitorFactory

```java
package de.ukn.hci.squidy.manager.data;

public interface IDataVisitorFactory {
   public IDataVisitor createDataVisitor();
   public String       serialize();
   public void         deserialize(String serial);
}
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/AbstractDataVisitor.java
Changes: new file, implementation of AbstractDataVisitor

```
package de.ukn.hci.squidy.manager.data;

public abstract class AbstractDataVisitor implements IDataVisitor {

   private IDataVisitorFactory factory;

   public AbstractDataVisitor(IDataVisitorFactory factory) {
      super();
      this.factory = factory;
   }

   /*
    * @return factory which produced this visitor
    */
   public IDataVisitorFactory getFactory() {
      return factory;
   }

   /*
    * custom deserialization for remote transport if required
    */
   public void deserialize(String serial) {
   }

   /*
    * custom serialization for remote transport if required
    */
   public String serialize() {
      return null;
   }
}
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/model/Piping.java
Changes: implemented support for data visitor factories

```
   private Collection<IDataVisitorFactory> visitorFactories;

   /**
    * @param visitor factory to add
    * @return true if visitor factory has been added
    */
   public boolean addDataVisitorFactory(IDataVisitorFactory factory) {
      if (factory == null) {
         return false;
      }
      if (visitorFactories == null) {
         visitorFactories = new ArrayList<IDataVisitorFactory>();
      } else if (visitorFactories.contains(factory)) {
         // prevent duplicates
         return false;
      }
      return visitorFactories.add(factory);
   }

   /**
    * @param visitor factory to remove
    * @return true if visitor factory has been removed
    */
   public boolean removeDataVisitorFactory(IDataVisitorFactory factory) {
      return (visitorFactories != null) ? visitorFactories.remove(factory) : false;
   }


   /**
    * create new visitors and attach them to all data objects
    * @param container of data objects
    */
   protected void attachVisitors(IDataContainer container) {
      if (visitorFactories != null && container != null) {
         for (IDataVisitorFactory factory : visitorFactories) {
            for (IData data : container.getData()) {
               data.acceptVisitor(factory.createDataVisitor());
            }
         }
      }
   }
```

```
    /**
     * notify all visitors that we are ready to receive their visit
     * @param container of data objects
     */
    protected void notifyVisitors(IDataContainer container) {
       if (container != null) {
          for (IData data : container.getData()) {
             data.notifyVisitors(this);
          }
       }
    }
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/model/AbstractNode.java
Changes: added data visitor processing during publishing

```
    public final void publish(final IDataContainer dataContainer) {
       [...]
          attachVisitors(container);
          notifyVisitors(container);
       [...]
    }
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/util/CoreUtility.java
Changes: implementation of serialization procedure for data visitors and data visitor factories

```
    public static String getSerialOfAttributes(Map<DataConstant, Object> attributes) {
       if (attributes == null) {
          return "";
       }
       [...]
    }

    /**
     * @param visitors
     * @return serialized string
     */
    public static String getSerialOfVisitors(Collection<IDataVisitor> visitors) {

       if (visitors == null) {
          return "";
       }

       StringBuilder serial = new StringBuilder("");
       for (IDataVisitor visitor : visitors) {
          serial.append(visitor.getFactory().getClass().getName()).append(":");
          serial.append(visitor.getFactory().serialize()).append(":");
          serial.append(visitor.serialize()).append(";");
       }

       return serial.substring(0, serial.length() - 1);
    }

    /**
     * @param serial
     * @return list from serialized string
     */
    public static Collection<IDataVisitor> getVisitorsOfSerial(String serial) {

       if ("".equals(serial)) {
          return null;
       }

       Collection<IDataVisitor> visitors = new ArrayList<IDataVisitor>();

       StringTokenizer attributeTokens = new StringTokenizer(serial, ";");
       while (attributeTokens.hasMoreTokens()) {
          String[] parts = attributeTokens.nextToken().split(":");

          IDataVisitorFactory factory = ReflectionUtil.createInstance(parts[0]);
          if (factory != null) {
             factory.deserialize(parts[1]);
             IDataVisitor visitor = factory.createDataVisitor();
             if (visitor != null) {
                visitor.deserialize(parts[2]);
                visitors.add(visitor);
             }
          }
       }

       return visitors;
    }
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/impl/DataTestVisitor.java
Changes: new file for testing purposes, implementation of DataTestVisitor

```java
package de.ukn.hci.squidy.manager.data.impl;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import de.ukn.hci.squidy.manager.IProcessable;
import de.ukn.hci.squidy.manager.data.IData;
import de.ukn.hci.squidy.manager.data.IDataVisitorFactory;

public class DataTestVisitor extends AbstractDataVisitor {

   private static final Log LOG = LogFactory.getLog(DataTestVisitor.class);

   public DataTestVisitor(IDataVisitorFactory factory) {
      super(factory);
   }

   public void visit(IProcessable<?> processable, IData data) {
      if (data instanceof DataPosition2D) {
         DataPosition2D d2d = (DataPosition2D) data;
         LOG.debug(processable.getClass().getName() + ": x = " + String.format("%.3f", d2d.x)
                  + " y = " + String.format("%.3f", d2d.y));
      }
   }
}
```

File: core/squidy-manager/src/main/java/de/ukn/hci/squidy/manager/data/impl/DataTestVisitorFactory.java
Changes: new file for testing purposes, implementation of DataTestVisitorFactory

```java
package de.ukn.hci.squidy.manager.data.impl;

import de.ukn.hci.squidy.manager.data.IDataVisitor;
import de.ukn.hci.squidy.manager.data.IDataVisitorFactory;

public class DataTestVisitorFactory implements IDataVisitorFactory {

   /*
    * create a new visitor
    */
   public IDataVisitor createDataVisitor() {
      return new DataTestVisitor(this);
   }

   /*
    * custom deserialization for remote transport if required
    */
   public void deserialize(String serial) {
   }

   /*
    * custom serialization for remote transport if required
    */
   public String serialize() {
      return null;
   }
}
```

File: basic/src/main/java/de/ukn/hci/squidy/extension/basic/MouseIO.java
Changes: temporarily added constructor and data visitor factory for testing of new features

```java
   // TEST: DataPosition2D visitor
   public MouseIO() {
      super();
      addDataVisitorFactory(new DataTestVisitorFactory());
   }
```

Source Code of Synchronization Latency Test

```
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#include <math.h>
#include <windows.h>

#define WORKLOAD   20
#define ITERATIONS 100000

HANDLE  hMasterReady, hSlaveDone;
__int64 iSum;

void Workload()
{
  // overflows quickly but is for testing only
  for (int i = 0; i < WORKLOAD; i++)
    iSum += iSum - (__int64)sqrt((double)iSum);
}

DWORD WINAPI Slave(LPVOID pbQuit)
{
  SetEvent(hSlaveDone);
  while (!*(LPBOOL)pbQuit) {
    WaitForSingleObject(hMasterReady, INFINITE);
    Workload();
    SetEvent(hSlaveDone);
  }
  CloseHandle(hMasterReady);
  CloseHandle(hSlaveDone);
  return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
  LARGE_INTEGER liStart, liEnd, liFreq;
  HANDLE        hSlave;
  BOOL          bQuitSlave;
  DWORD         dwMasterTime, dwSlaveTime, i;
  __int64       iMasterSum, iSlaveSum;

  bQuitSlave = FALSE;
  QueryPerformanceFrequency(&liFreq);

  hMasterReady = CreateEvent(NULL, FALSE, FALSE, NULL);
  hSlaveDone   = CreateEvent(NULL, FALSE, FALSE, NULL);
  hSlave       = CreateThread(NULL, 0, Slave, &bQuitSlave, 0, 0);

  // Master
  QueryPerformanceCounter(&liStart);
  iSum = 2;
  for (i = 0; i < ITERATIONS; i++)
    Workload();
  QueryPerformanceCounter(&liEnd);
  dwMasterTime = (DWORD)((liEnd.QuadPart - liStart.QuadPart) * 1000 / liFreq.QuadPart);
  iMasterSum   = iSum;

  // Slave
  WaitForSingleObject(hSlaveDone, INFINITE);
  QueryPerformanceCounter(&liStart);
  iSum = 2;
  for (i = 0; i < ITERATIONS; i++) {
    SetEvent(hMasterReady);
    WaitForSingleObject(hSlaveDone, INFINITE);
  }
  QueryPerformanceCounter(&liEnd);
  dwSlaveTime = (DWORD)((liEnd.QuadPart - liStart.QuadPart) * 1000 / liFreq.QuadPart);
  iSlaveSum   = iSum;

  bQuitSlave = TRUE;
  SetEvent(hMasterReady);

  printf("Master: sum %I64d in %d ms\nSlave:  sum %I64d in %d ms\n\nDifference: %d ms\n
          Overhead: %.6f ms\n",
         iMasterSum, dwMasterTime, iSlaveSum, dwSlaveTime, dwSlaveTime - dwMasterTime,
         (double)(dwSlaveTime - dwMasterTime) / ITERATIONS);

  return 0;
}
```

Comparison of Libraries, Frameworks, and Toolkits

This section consists of two parts: A tabular comparison of Libraries, Frameworks and Toolkits and which is introduced by the legend of scoring criteria.

Criterion	Value	Score	Description
Name			name of the project
Former Names			former names of the project
URL			primary URL
URL (secondary)			secondary URL
Year of Invention			year of invention
Latest Version			version number of latest released version or latest revision from repository of version control system
Authors			authors of the project, only names of people
Organizations			organizations participating in the project
Scope			scope of the project, such as multi-touch input, input abstraction, graphical visualization
Special Features			features which are really special to the project, such as XNA support, Python interface, artificial intelligence
Input Devices			list of supported devices such as camera, mouse, laser pointer, or any if just a driver or interface has to be added
Programming Languages			programming languages used for the implementation
Number of Contributors	>10	30	number of people who have contributed source code according to authors or log of version control system
	6..10	20	
	3..5	10	
	1..2	0	
	unknown	0	
Origin	research	0	has been invented during research at university or comparable institution
	community	0	has been invented by the Internet community or private groups or persons
	commercial	0	has been invented by a company or another commercial institution
License	royalty-free	40	free for everything and everyone
	BSD	35	BSD License[119]
	LGPLv2	30	GNU Lesser General Public License 2.1[120]
	LGPLv3	30	GNU Lesser General Public License 3[121]
	GPLv2	20	GNU General Public License 2[122]
	GPLv3	20	GNU General Public License 3[123]
	commercial/free	15	you need to pay for it but may it use for free for non-commercial purposes
	commercial	0	you need to pay for it or at least you need to pay for the platform to use it
	unknown	0	
Project Status	stable	20	suitable for production use
	beta	15	might still contain bugs, suitable for testing, research, and early development stages of own projects
	alpha	0	early access, incomplete, suitable for testing only
	unknown	0	
Maintenance Status	active	35	somebody maintains the project
	inactive	0	nobody maintains the project
Implemented	yes	50	the project has been implemented, not only described somewhere
	no	0	
Binaries available	yes	20	executable files of the project are available to the public
	no	0	
Source Code available	yes	50	source code of the project is available to the public
	limited	25	source code of the project is available under certain conditions
	no	0	
Windows	yes	25	supports the Microsoft Windows platform
	no	0	

119 http://www.opensource.org/licenses/bsd-license.php
120 http://www.opensource.org/licenses/lgpl-2.1.php
121 http://www.opensource.org/licenses/lgpl-3.0.html
122 http://www.opensource.org/licenses/gpl-2.0.php
123 http://www.opensource.org/licenses/gpl-3.0.html

	unknown	0	
Linux	yes	20	supports the Linux platform
	no	0	
	unknown	0	
Mac OS X	yes	20	supports the Apple Mac OS X platform
	no	0	
	unknown	0	
API	yes	40	provides an API for developers who want to use the project
	no	0	
	unknown	0	
Declarative Programming	yes	10	data processing can be configured by one or more editable configuration files or comparable descriptive files
	no	0	
	unknown	0	
Visual Programming	yes	40	provides a user interface for visual programming
	no	0	
	unknown	0	
Scripting	yes	20	provides an interface to a scripting language for configuration and control, this is always true for implementations on the Java or .NET platform because compatible scripting languages are available
	no	0	
	unknown	0	
Architectural Type	library	0	it is a framework
	framework	20	it is a library
	toolkit	25	it is a toolkit
	unknown	0	
Architectural Model	layers	0	it has a layer architecture
	pipes & filters	0	it has a pipes and filters architecture
	model-view-controller	0	it has a model-view-controller architecture
	unknown	0	
Extensible	yes	40	provides interfaces and infrastructure which can be used to extend major aspects of the project without modifying its existing implementation
	no	0	
	unknown	0	
Modular	yes	30	has a recognizable modular structure
	no	0	
	unknown	0	
Reusable Components	yes	30	contains mainly components or modules with universal interfaces which can be used in different contexts or projects without modifying the components
	no	0	
	unknown	0	
Overall Complexity	simple	20	[subjective impression] small project which is easy to understand entirely
	complex	10	[subjective impression] larger project which requires some time to understand entirely
	very complex	0	[subjective impression] large or very large project which is hard to oversee and understand entirely
	unknown	0	
Consistent Conventions	yes	15	[subjective impression] source code follows consistent rules for formatting, structure, naming, comments etc.
	no	0	
	n/a	0	
	unknown	0	
Overall Code Quality	good	30	[subjective impression] well structured project, self-explaining code, use of proven design patterns
	acceptable	15	[subjective impression] partial inconsistent design or conventions, sometimes hard to read
	bad	0	[subjective impression] little structured, few or no comments and conventions, no use of design patterns
	n/a	0	
	unknown	0	
Number of Files	unknown	0	number of lines of source code files, counted using CLOC[124]
Lines of Code (SLOC)	unknown	0	number of lines of source code excluding blank and comment lines, and third-party contributions, counted using CLOC
Lines of Comment	unknown	0	number of comment lines of source code, counted using CLOC
Documentation Level	high	40	detailed documentation of architecture and functionality, comparable with commercial products such as documentation of Java API or Windows API
	medium	20	some documentation of architecture and functionality

124 http://cloc.sourceforge.net/

	low	0	few or no documentation
	unknown	0	
Parallel Processing	yes	30	processing steps (at least some) can be processed in parallel on platforms capable of multiprocessing
	no	0	
	n/a	0	
	unknown	0	
Distributed Processing	yes	15	processing steps (at least some) can be divided to multiple platforms in a distributed environment
	no	0	
	n/a	0	
	unknown	0	
Rearrangeable Steps	yes	25	processing steps (at least some) can be rearranged and processed in different order or combinations
	no	0	
	n/a	0	
	unknown	0	
Access Raw Data of each Step	yes	10	raw input and output data of each processing step can be accessed through the standard application interface of the project
	no	0	
	n/a	0	
	unknown	0	
Interface to Devices	yes	100	includes a direct interface to at least one input device
	no	0	
	unknown	0	
Multiple Types of Devices	yes	50	supports multiple types of input devices such as camera, Wiimote, mouse, etc.
	no	0	
	n/a	0	
	unknown	0	
Multiple Devices	yes	50	supports simultaneous use of more than one input device, independently from useful correlation between these devices
	no	0	
	n/a	0	
	unknown	0	
Data Synchronization	yes	25	provides functionality to synchronize input data from multiple input devices which provide input data simultaneously
	no	0	
	n/a	0	
	unknown	0	
Multi-Touch Workflow	yes	30	integrates processing workflow by means of multi-touch specific processing of input data, tracking and/or interpretation
	no	0	
	unknown	0	
Heterogeneous Input Data	yes	25	supports input data of devices from different multi-touch environments or setups, such as camera-based or DiamondTouch, includes at least flexible data type support for this purpose
	no	0	
	unknown	0	
Multiple Camera Models	yes	15	includes device driver implementations for different cameras such as webcams and industrial cameras, or cameras from multiple manufacturers
	no	0	
	n/a	0	
	unknown	0	
Image Filters	>10	20	number of included or used image filters as an indicator for capabilities of image processing
	6..10	15	
	1..5	10	
	no	0	
	unknown	0	
Input Interpretation	comprehensive	30	comprehensive interpretation mechanisms for detected input data such as reconstruction of hand and complex gesture recognition
	basic	15	basic interpretation mechanisms for detected input data such as tracking or identification of fingers
	no	0	
	unknown	0	
OSC included	yes	10	includes classes or a library which make it easy to implement a custom OSC protocol
	no	0	
	unknown	0	
TUIO implemented	yes	20	provides an implementation to transfer input data using the TUIO protocol
	no	0	
	unknown	0	

	Augmented and Vitual Reality		
Name	**Unit**	**ViSTA (Core Libs only)**	**ViSTA (DataLaViSTA)**
Former Names			
URL	www.csc.kth.se/~alx	www.rz.rwth-aachen.de/ca/c/piz	www.rz.rwth-aachen.de/ca/c/piz
URL (secondary)		www.sourceforge.org/projects/vistavrtoolkit	www.sourceforge.org/projects/vistavrtoolkit
Year of Invention	2002	1997	1997
Latest Version	unknown	r4818	r4818
Authors	Alex Olwal	VR Group	VR Group
Organisations	The Royal Institute of Technology (KTH)	RWTH Aachen University	RWTH Aachen University
Scope	distributed multi-device input	data transport and processing	data transport and processing
Special Features	interaction technique abstraction, Java3D UI	multi-threading	multi-threading
Input Devices	any	various (mouse/keyboard, spacemouse, tracking devices, MIDI, RAW USB (HID), Wiimote)	any (data transport only)
Programming Languages	Java	C++	C++
Number of Contributors	unknown	>10	>10
Origin	research	research	research
License	unknown	LGPLv3	LGPLv3
Status			
Project Status	unknown	beta	beta
Maintenance Status	active	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	no	no	no
Source Code available	limited	yes	yes
Platform			
Windows	yes	yes	yes
Linux	unknown	yes	yes
Mac OS X	unknown	yes	yes
Development			
API	yes	yes	yes
Declarative Programming	unknown	yes	no
Visual Programming	yes	no	no
Scripting	unknown	no	no
Architecture			
Architectural Type	framework	toolkit	library
Architectural Model	pipes & filters	unknown	pipes & filters
Extensible	yes	yes	yes
Modular	yes	yes	yes
Reusable Components	yes	yes	yes
Overall Complexity	unknown	very complex	complex
Source Code			
Consistent Conventions	unknown	yes	yes
Overall Code Quality	unknown	good	good
Number of Files	unknown	1006	76
Lines of Code (SLOC)	unknown	144255	4626
Lines of Comment	unknown	74675	5073
Documentation			
Documentation Level	unknown	low	low
Processing			
Parallel Processing	unknown	yes	n/a
Distributed Processing	yes	yes	n/a
Rearrangeable Steps	yes	n/a	n/a
Access Raw Data of each Step	yes	n/a	n/a
Input Devices			
Interface to Devices	yes	yes	no
Multiple Types of Devices	yes	yes	n/a
Multiple Devices	yes	yes	n/a
Data Synchronization	unknown	yes	n/a
Multi-Touch			
Multi-Touch Workflow	unknown	no	no
Heterogeneous Input Data	yes	yes	yes
Multiple Camera Models	unknown	n/a	n/a
Image Filters	unknown	no	no
Input Interpretation	comprehensive	no	no
Protocols			
OSC included	unknown	no	no
TUIO implemented	unknown	no	no
Overall Score	**640**	**790**	**495**

	Multi-Modal		
Name	**OpenInterface**	**Squidy Interaction Library**	**vvvv**
Former Names			
URL	www.openinterface.org/platform	www.squidy-lib.de	vvvv.org
URL (secondary)	www.oi-project.org		
Year of Invention	2007	2007	1998
Latest Version	0.3.6	1.0.0	4.0 beta 21
Authors	Jean-Yves Lionel Lawson	Werner König, Roman Rädle, Toni Schmidt	
Organisations	Université catholique de Louvain (UCL)	University of Konstanz	vvvv group
Scope	mutli-modal input, multi-modal design space	multi-modal input	input, graphics, audio, visual effects, device control
Special Features	multi-language support (C/C++, Java, Matlab, C#), Eclipse integration	multi-threading, GPU image processing, interactive configuration, dataflow visualization, reusability	many effects, requires runtime environment
Input Devices	any	any	any
Programming Languages	C/C++	Java, C++	unknown
Number of Contributors	1..2	6..10	unknown
Origin	research	research	commercial
License	BSD	LGPLv3	commercial/free
Status			
Project Status	beta	stable	beta
Maintenance Status	active	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	yes	yes
Source Code available	limited	yes	no
Platform			
Windows	yes	yes	yes
Linux	yes	yes	no
Mac OS X	no	yes	no
Development			
API	yes	yes	no
Declarative Programming	yes	yes	yes
Visual Programming	yes	yes	yes
Scripting	no	yes	no
Architecture			
Architectural Type	framework	framework	toolkit
Architectural Model	pipes & filters	pipes & filters	pipes & filters
Extensible	yes	yes	yes
Modular	yes	yes	yes
Reusable Components	yes	yes	yes
Overall Complexity	very complex	complex	complex
Source Code			
Consistent Conventions	unknown	yes	n/a
Overall Code Quality	unknown	good	n/a
Number of Files	287	409	unknown
Lines of Code (SLOC)	25092	35861	unknown
Lines of Comment	11717	27633	unknown
Documentation			
Documentation Level	unknown	low	medium
Processing			
Parallel Processing	yes	yes	yes
Distributed Processing	yes	yes	yes
Rearrangeable Steps	yes	yes	yes
Access Raw Data of each Step	yes	yes	yes
Input Devices			
Interface to Devices	yes	yes	yes
Multiple Types of Devices	yes	yes	yes
Multiple Devices	yes	yes	yes
Data Synchronization	yes	no	no
Multi-Touch			
Multi-Touch Workflow	no	yes	yes
Heterogeneous Input Data	yes	yes	yes
Multiple Camera Models	yes	no	yes
Image Filters	>10	6..10	1..5
Input Interpretation	unknown	basic	no
Protocols			
OSC included	yes	yes	no
TUIO implemented	yes	yes	no
Overall Score	830	970	725

	Multi-Touch		
Name	**Bespoke Multi-Touch**	**Community Core Vision**	**EquisFTIR**
Former Names		**tbeta**	**OpenFTIR**
URL	www.bespokesoftware.org/multi-touch	ccv.nuigroup.com	research.cs.queensu.ca/~wolfe/equisftir
URL (secondary)	bespokemultitouch.codeplex.com	nuicode.com/projects/tbeta	
Year of Invention	2008	2008	2007
Latest Version	4.2.0.0	1.2	1.0
Authors	Paul Varcholik		Christopher Wolfe
Organisations	Bespoke Software	NUI Group	Queen's University Kingston
Scope	multi-touch input	multi-touch input	multi-touch input
Special Features	XNA integration	uses OpenFrameworks, GUI, GPU image filters	optimized image filters
Input Devices	camera	camera	camera
Programming Languages	C#	C++	C++
Number of Contributors	1..2	3..5	1..2
Origin	community	community	research
License	BSD	GPLv3	LGPLv3
Status			
Project Status	stable	stable	alpha
Maintenance Status	active	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	yes	yes
Source Code available	yes	yes	yes
Platform			
Windows	yes	yes	yes
Linux	no	yes	no
Mac OS X	no	yes	no
Development			
API	yes	yes	yes
Declarative Programming	yes	yes	no
Visual Programming	no	no	no
Scripting	yes	no	no
Architecture			
Architectural Type	framework	framework	library
Architectural Model	layers	layers	layers
Extensible	yes	yes	no
Modular	yes	yes	yes
Reusable Components	yes	yes	no
Overall Complexity	simple	simple	simple
Source Code			
Consistent Conventions	yes	no	yes
Overall Code Quality	good	acceptable	acceptable
Number of Files	163	146	74
Lines of Code (SLOC)	20998	15265	4802
Lines of Comment	8802	5974	600
Documentation			
Documentation Level	medium	low	low
Processing			
Parallel Processing	no	no	no
Distributed Processing	no	no	no
Rearrangeable Steps	no	yes	no
Access Raw Data of each Step	no	no	no
Input Devices			
Interface to Devices	yes	yes	yes
Multiple Types of Devices	no	no	no
Multiple Devices	yes	yes	yes
Data Synchronization	no	no	no
Multi-Touch			
Multi-Touch Workflow	yes	yes	yes
Heterogeneous Input Data	no	no	no
Multiple Camera Models	yes	yes	yes
Image Filters	>10	6..10	1..5
Input Interpretation	no	no	no
Protocols			
OSC included	yes	yes	no
TUIO implemented	no	yes	no
Overall Score	735	740	535

	Multi-Touch		
Name	libavg	libtisch	mu3
Former Names			
URL	www.libavg.de	tisch.sourceforge.net	code.google.com/p/mu3
URL (secondary)			
Year of Invention	2003	2008	2009
Latest Version	0.90	r1169	1.0
Authors	Ulrich von Zadow	Florian Echtler, Gudrun Klinker	Michiel Hakvoort
Organisations	Archimedes Solutions GmbH	Technische Universität München	University of Twente
Scope	multi-media processing	multi-touch and tangible input	multi-touch input abstraction
Special Features	audio and graphics support, Python scripting	distributed architecture, lightweight	mutli-user and object identification and tracking
Input Devices	camera	any	any
Programming Languages	C++, Python	C++	Java
Number of Contributors	>10	1..2	1..2
Origin	community	research	research
License	LGPLv2	LGPLv3	BSD
Status			
Project Status	stable	beta	alpha
Maintenance Status	active	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	yes	no
Source Code available	yes	yes	yes
Platform			
Windows	yes	yes	yes
Linux	yes	yes	yes
Mac OS X	yes	yes	yes
Development			
API	yes	yes	yes
Declarative Programming	yes	no	no
Visual Programming	no	no	no
Scripting	yes	no	no
Architecture			
Architectural Type	library	framework	framework
Architectural Model	layers	layers	layers
Extensible	yes	no	yes
Modular	yes	yes	yes
Reusable Components	yes	yes	no
Overall Complexity	complex	simple	simple
Source Code			
Consistent Conventions	yes	yes	yes
Overall Code Quality	good	acceptable	good
Number of Files	428	123	71
Lines of Code (SLOC)	52114	8135	4290
Lines of Comment	11094	1775	994
Documentation			
Documentation Level	medium	medium	low
Processing			
Parallel Processing	n/a	no	yes
Distributed Processing	n/a	yes	no
Rearrangeable Steps	n/a	no	no
Access Raw Data of each Step	n/a	no	no
Input Devices			
Interface to Devices	yes	yes	yes
Multiple Types of Devices	no	yes	yes
Multiple Devices	yes	yes	yes
Data Synchronization	no	no	no
Multi-Touch			
Multi-Touch Workflow	no	yes	yes
Heterogeneous Input Data	no	yes	yes
Multiple Camera Models	yes	yes	no
Image Filters	>10	1..5	no
Input Interpretation	no	basic	comprehensive
Protocols			
OSC included	no	no	no
TUIO implemented	no	no	yes
Overall Score	730	765	765

	Multi-Touch		
Name	**Multi-Touch Vista**	**multitouch**	**MultiTouch.framework SDK**
Former Names			**PocketTable**
URL	www.codeplex.com/MultiTouchVista	code.google.com/p/multitouch	hci.rwth-aachen.de/multitouch
URL (secondary)			
Year of Invention	2008	2007	2008
Latest Version	second release refresh 2	r224	unknown
Authors	Daniels Danilins		Stefan Hafeneger
Organisations		TU Berlin, Deutsche Telekom AG Laboratories	RWTH Aachen
Scope	multi-touch input translation, GUI	multi-touch input	multi-touch input
Special Features	WPF controls, Windows 7 driver		support for mobile devices
Input Devices	any	camera	camera, iPhone, iPod
Programming Languages	C#, VB.NET	Java	ObjC
Number of Contributors	1..2	3..5	1..2
Origin	community	research	research
License	GPLv2	GPLv2	unknown
Status			
Project Status	beta	alpha	beta
Maintenance Status	active	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	no	no
Source Code available	yes	yes	no
Platform			
Windows	yes	yes	no
Linux	no	yes	no
Mac OS X	no	yes	yes
Development			
API	yes	yes	yes
Declarative Programming	yes	no	unknown
Visual Programming	no	no	unknown
Scripting	yes	no	unknown
Architecture			
Architectural Type	framework	framework	framework
Architectural Model	layers	layers	unknown
Extensible	yes	no	yes
Modular	yes	no	yes
Reusable Components	no	no	unknown
Overall Complexity	complex	simple	unknown
Source Code			
Consistent Conventions	yes	yes	unknown
Overall Code Quality	good	bad	unknown
Number of Files	215	50	unknown
Lines of Code (SLOC)	13347	5834	unknown
Lines of Comment	2339	2249	unknown
Documentation			
Documentation Level	low	low	unknown
Processing			
Parallel Processing	no	no	unknown
Distributed Processing	yes	no	unknown
Rearrangeable Steps	no	no	unknown
Access Raw Data of each Step	no	no	unknown
Input Devices			
Interface to Devices	yes	yes	yes
Multiple Types of Devices	yes	no	yes
Multiple Devices	yes	no	unknown
Data Synchronization	no	n/a	unknown
Multi-Touch			
Multi-Touch Workflow	no	yes	yes
Heterogeneous Input Data	yes	no	yes
Multiple Camera Models	yes	yes	unknown
Image Filters	no	no	unknown
Input Interpretation	no	no	unknown
Protocols			
OSC included	no	yes	no
TUIO implemented	yes	yes	no
Overall Score	705	500	455

	Multi-Touch		
Name	multitouchframework	OpenTouch	PyMT
Former Names			
URL	code.google.com/p/multitouchframework	code.google.com/p/opentouch	pymt.txzone.net
URL (secondary)			
Year of Invention	2008	2007	2009
Latest Version	1.0	r158	0.3
Authors	Arnoud de Jong	Pawel Solyga	
Organisations		Wroclaw University of Technology	community
Scope	multi-touch GUI	multi-touch input	multi-touch GUI
Special Features	WPF controls		many widgets, OpenGL output
Input Devices	none	camera	none
Programming Languages	C#	C++, Java	Python
Number of Contributors	1..2	1..2	>10
Origin	community	community	community
License	GPLv3	LGPLv3	GPLv2
Status			
Project Status	beta	alpha	stable
Maintenance Status	inactive	inactive	active
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	no	no
Source Code available	no	yes	yes
Platform			
Windows	yes	no	yes
Linux	no	no	yes
Mac OS X	no	yes	yes
Development			
API	yes	yes	yes
Declarative Programming	yes	no	no
Visual Programming	no	no	no
Scripting	yes	no	yes
Architecture			
Architectural Type	framework	framework	framework
Architectural Model	unknown	layers	layers
Extensible	unknown	unknown	yes
Modular	unknown	unknown	yes
Reusable Components	unknown	unknown	yes
Overall Complexity	unknown	simple	simple
Source Code			
Consistent Conventions	unknown	yes	yes
Overall Code Quality	unknown	acceptable	good
Number of Files	unknown	48	93
Lines of Code (SLOC)	unknown	3417	10737
Lines of Comment	unknown	989	651
Documentation			
Documentation Level	low	low	high
Processing			
Parallel Processing	unknown	no	no
Distributed Processing	unknown	no	no
Rearrangeable Steps	n/a	unknown	n/a
Access Raw Data of each Step	n/a	unknown	n/a
Input Devices			
Interface to Devices	no	yes	no
Multiple Types of Devices	n/a	yes	n/a
Multiple Devices	n/a	no	n/a
Data Synchronization	n/a	n/a	n/a
Multi-Touch			
Multi-Touch Workflow	yes	yes	yes
Heterogeneous Input Data	yes	no	yes
Multiple Camera Models	n/a	yes	n/a
Image Filters	no	>10	no
Input Interpretation	comprehensive	no	comprehensive
Protocols			
OSC included	no	no	yes
TUIO implemented	yes	yes	yes
Overall Score	325	495	670

	Multi-Touch		
Name	**Sparsh-UI**	**TouchKit**	**touchlib**
Former Names			
URL	code.google.com/p/sparsh-ui	labs.nortd.com/touchkit	www.touchlib.com
URL (secondary)	www.vrac.iastate.edu/uav/touchtable		www.whitenoiseaudio.com/touchlib
Year of Invention	2008	2008	2006
Latest Version	r715	v005	2.0
Authors	Stephen Gilbert		David Wallin, NUI Group
Organisations	Iowa State University	NOR_/D	NUI Group, White Noise Audio
Scope	mutli-touch input and gestures	mutli-touch input	multi-touch input
Special Features		lightweight	
Input Devices	multi-touch devices	camera	camera
Programming Languages	C++, Java	C++	C++
Number of Contributors	>10	unknown	6..10
Origin	research	commercial	community
License	LGPLv3	LGPLv3	BSD
Status			
Project Status	stable	stable	beta
Maintenance Status	active	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	no	yes
Source Code available	yes	yes	yes
Platform			
Windows	yes	yes	yes
Linux	yes	no	no
Mac OS X	yes	yes	no
Development			
API	yes	yes	yes
Declarative Programming	no	no	yes
Visual Programming	no	no	no
Scripting	no	no	no
Architecture			
Architectural Type	framework	framework	framework
Architectural Model	layers	layers	pipes & filters
Extensible	yes	no	yes
Modular	yes	yes	yes
Reusable Components	no	no	yes
Overall Complexity	simple	simple	simple
Source Code			
Consistent Conventions	no	yes	no
Overall Code Quality	acceptable	acceptable	acceptable
Number of Files	220	17	90
Lines of Code (SLOC)	11643	1614	7965
Lines of Comment	3492	454	1315
Documentation			
Documentation Level	low	low	low
Processing			
Parallel Processing	no	no	no
Distributed Processing	yes	no	no
Rearrangeable Steps	no	no	yes
Access Raw Data of each Step	no	no	no
Input Devices			
Interface to Devices	yes	yes	yes
Multiple Types of Devices	no	no	no
Multiple Devices	no	no	yes
Data Synchronization	n/a	n/a	no
Multi-Touch			
Multi-Touch Workflow	yes	yes	yes
Heterogeneous Input Data	yes	no	no
Multiple Camera Models	no	yes	yes
Image Filters	no	>10	>10
Input Interpretation	comprehensive	no	no
Protocols			
OSC included	no	yes	yes
TUIO implemented	no	no	yes
Overall Score	**665**	**545**	**725**

	Multi-Touch		
Name	**touchpy**	**Touché**	**xTouch**
Former Names			**BBTouch**
URL	code.google.com/p/touchpy	gkaindl.com/software/touche	code.google.com/p/opentouch
URL (secondary)		code.google.com/p/touche	benbritten.com/category/multitouch
Year of Invention	2008	2008	2007
Latest Version	r78	1.0b3	not released
Authors	Goran Medakovic	Georg Kaindl	Ben Britten Smith
Organisations		Vienna University of Technology	
Scope	multi-touch GUI	multi-touch input	multi-touch input
Special Features	Compiz plugin	comfortable GUI, image filtering with OpenGL shaders	
Input Devices	none	camera, Wiimote	camera
Programming Languages	Python, C	ObjC	ObjC
Number of Contributors	1..2	1..2	1..2
Origin	community	research	community
License	GPLv3	LGPLv3	LGPLv3
Status			
Project Status	alpha	beta	beta
Maintenance Status	inactive	active	active
Implementation			
Implemented	yes	yes	yes
Binaries available	no	yes	no
Source Code available	yes	yes	yes
Platform			
Windows	no	no	no
Linux	yes	no	no
Mac OS X	no	yes	yes
Development			
API	yes	yes	yes
Declarative Programming	no	no	no
Visual Programming	no	no	no
Scripting	yes	no	no
Architecture			
Architectural Type	framework	framework	framework
Architectural Model	layers	pipes & filters	model-view-controller
Extensible	no	yes	yes
Modular	no	yes	yes
Reusable Components	no	yes	no
Overall Complexity	simple	complex	simple
Source Code			
Consistent Conventions	yes	yes	yes
Overall Code Quality	acceptable	good	acceptable
Number of Files	8	285	105
Lines of Code (SLOC)	2116	23108	5710
Lines of Comment	180	7028	3560
Documentation			
Documentation Level	low	low	low
Processing			
Parallel Processing	no	yes	no
Distributed Processing	no	no	no
Rearrangeable Steps	no	no	no
Access Raw Data of each Step	no	no	no
Input Devices			
Interface to Devices	no	yes	yes
Multiple Types of Devices	n/a	yes	no
Multiple Devices	n/a	yes	no
Data Synchronization	n/a	no	n/a
Multi-Touch			
Multi-Touch Workflow	yes	yes	yes
Heterogeneous Input Data	yes	yes	no
Multiple Camera Models	n/a	yes	yes
Image Filters	no	>10	>10
Input Interpretation	basic	basic	no
Protocols			
OSC included	yes	yes	yes
TUIO implemented	yes	yes	yes
Overall Score	**370**	**800**	**575**

	Multi-Touch Commercial	
Name	**Microsoft Surface SDK**	**Microsoft Windows Touch SDK**
Former Names		
URL	www.surface.com	msdn.microsoft.com/en-us/library/dd562197%28VS.85%29.aspx
URL (secondary)		msdn.microsoft.com/en-us/library/ee332414.aspx
Year of Invention	2008	2009
Latest Version	1.0 SP1	6.1 build 7600
Authors		
Organisations	Microsoft Corporation	Microsoft Corporation
Scope	multi-touch input, GUI	multi-touch input
Special Features	supports Surface table and WPF	included in Windows 7
Input Devices	Microsoft Surface Table	any
Programming Languages	unknown	unknown
Number of Contributors	unknown	unknown
Origin	commercial	commercial
License	commercial	commercial
Status		
Project Status	stable	stable
Maintenance Status	active	active
Implementation		
Implemented	yes	yes
Binaries available	yes	yes
Source Code available	no	no
Platform		
Windows	yes	yes
Linux	no	no
Mac OS X	no	no
Development		
API	yes	yes
Declarative Programming	yes	no
Visual Programming	no	no
Scripting	yes	no
Architecture		
Architectural Type	framework	framework
Architectural Model	unknown	unknown
Extensible	yes	yes
Modular	unknown	unknown
Reusable Components	unknown	unknown
Overall Complexity	unknown	unknown
Source Code		
Consistent Conventions	unknown	unknown
Overall Code Quality	unknown	unknown
Number of Files	unknown	unknown
Lines of Code (SLOC)	unknown	unknown
Lines of Comment	unknown	unknown
Documentation		
Documentation Level	high	high
Processing		
Parallel Processing	yes	yes
Distributed Processing	no	no
Rearrangeable Steps	no	no
Access Raw Data of each Step	no	no
Input Devices		
Interface to Devices	yes	yes
Multiple Types of Devices	no	yes
Multiple Devices	no	yes
Data Synchronization	no	no
Multi-Touch		
Multi-Touch Workflow	yes	yes
Heterogeneous Input Data	no	yes
Multiple Camera Models	no	n/a
Image Filters	unknown	no
Input Interpretation	comprehensive	comprehensive
Protocols		
OSC included	no	no
TUIO implemented	no	no
Overall Score	**510**	**605**

	Tangible Input		
Name	**Papier-Mâché**	**reacTIVision**	**TWING (Xenakis)**
Former Names			
URL	hci.stanford.edu/research/papier-mache	reactivision.sourceforge.net	xenakis.origo.ethz.ch
URL (secondary)		www.reactable.com	xenakis.3-n.de
Year of Invention	2003	2005	2008
Latest Version	r1068	1.4	r109
Authors	Scott R. Klemmer	Ross Bencina, Martin Kaltenbrunner	Markus Bischof, Bettina Conradi, Peter Lachenmaier, Kai Linde, Max Meier, Philipp Pötzl
Organisations	UC Berkeley, Stanford University	Universitat Pompeu Fabra, Barcelona, Spain	University of Augsburg
Scope	pyhsical input	fiducial input	tangible and multi-touch input
Special Features	input abstraction and association	fiducial tracking	input interpretation, command generation
Input Devices	any	camera	none
Programming Languages	Java	C++	C#
Number of Contributors	6..10	1..2	3..5
Origin	research	research	research
License	BSD	GPLv2	LGPLv3
Status			
Project Status	stable	stable	stable
Maintenance Status	inactive	active	inactive
Implementation			
Implemented	yes	yes	yes
Binaries available	yes	yes	no
Source Code available	yes	yes	yes
Platform			
Windows	yes	yes	yes
Linux	yes	yes	no
Mac OS X	yes	yes	no
Development			
API	yes	yes	yes
Declarative Programming	no	no	no
Visual Programming	no	no	no
Scripting	yes	no	yes
Architecture			
Architectural Type	toolkit	framework	framework
Architectural Model	layers	layers	model-view-controller
Extensible	yes	no	yes
Modular	yes	yes	yes
Reusable Components	yes	yes	yes
Overall Complexity	complex	complex	simple
Source Code			
Consistent Conventions	no	yes	yes
Overall Code Quality	acceptable	acceptable	good
Number of Files	162	202	353
Lines of Code (SLOC)	20914	25767	28973
Lines of Comment	6846	9941	19467
Documentation			
Documentation Level	low	low	high
Processing			
Parallel Processing	no	no	yes
Distributed Processing	no	no	no
Rearrangeable Steps	no	no	no
Access Raw Data of each Step	no	no	no
Input Devices			
Interface to Devices	yes	yes	no
Multiple Types of Devices	yes	no	n/a
Multiple Devices	yes	no	n/a
Data Synchronization	no	n/a	n/a
Multi-Touch			
Multi-Touch Workflow	no	yes	yes
Heterogeneous Input Data	yes	yes	yes
Multiple Camera Models	yes	yes	n/a
Image Filters	>10	1..5	no
Input Interpretation	no	comprehensive	comprehensive
Protocols			
OSC included	no	yes	no
TUIO implemented	no	yes	no
Overall Score	**730**	**660**	**585**