# ViSTA Widgets: a Framework for Designing 3D User Interfaces from Reusable Interaction Building Blocks

Sascha Gebhardt*, Till Petersen-Krauß*, Sebastian Pick*, Dominik Rausch*, Christian Nowke*, Thomas Knott*,
Patric Schmitz*, Daniel Zielasko*, Bernd Hentschel*, and Torsten W. Kuhlen*
Visual Computing Institute, RWTH Aachen University
JARA – High-Performance Computing

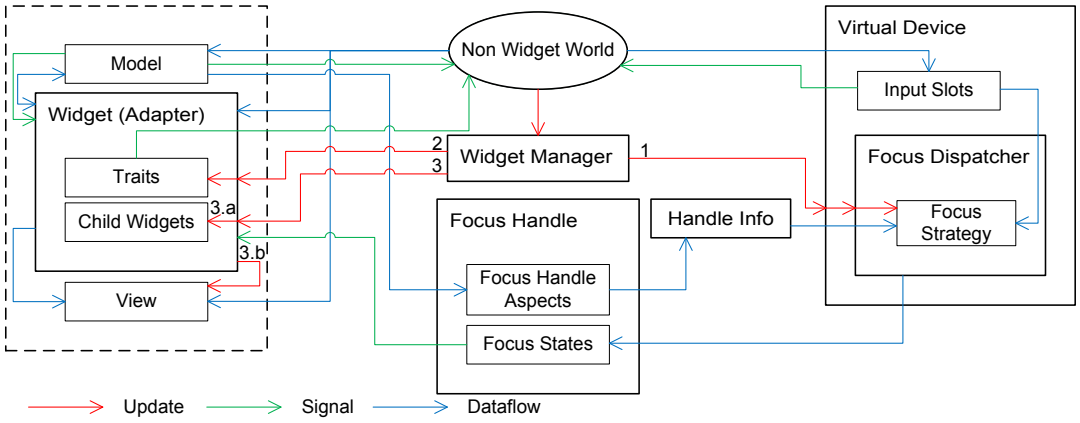


**Figure 1:** *Overview of the components of ViSTA Widgets and their interplay. Red arrows show the individual steps of the update cycle with the numbers indicating their order. Green arrows illustrate communication via signals. Blue arrows illustrate data flow.*

## Abstract

Virtual Reality (VR) has been an active field of research for several decades, with 3D interaction and 3D User Interfaces (UIs) as important sub-disciplines. However, the development of 3D interaction techniques and in particular combining several of them to construct complex and usable 3D UIs remains challenging, especially in a VR context. In addition, there is currently only limited reusable software for implementing such techniques in comparison to traditional 2D UIs. To overcome this issue, we present ViSTA Widgets, a software framework for creating 3D UIs for immersive virtual environments. It extends the ViSTA VR framework by providing functionality to create multi-device, multi-focus-strategy interaction building blocks and means to easily combine them into complex 3D UIs. This is realized by introducing a device abstraction layer along sophisticated focus management and functionality to create novel 3D interaction techniques and 3D widgets. We present the framework and illustrate its effectiveness with code and application examples accompanied by performance evaluations.

**Keywords:** 3D user interfaces, 3D interaction, virtual reality, multi-device, framework

**Concepts:** •**Human-centered computing** → **User interface toolkits;** •**Computing methodologies** → **Virtual reality;**

*e-mail:{gebhardt|petersen-krauss|pick|rausch|nowke|knott|schmitz|zielasko|hentschel|kuhlen}@vr.rwth-aachen.de



VRST '16, November 02 - 04, 2016, Garching bei München, Germany
ISBN: 978-1-4503-4491-3/16/11
DOI: http://dx.doi.org/10.1145/2993369.2993382

## 1 Introduction

While VR has recently seen a major boost in popularity due to the release of various VR consumer devices, it has already been an active field of research for several decades. This is particularly true for 3D interaction and 3D UIs. The high degree of freedom in 3D UI design, compared to a 2D desktop setting, in conjunction with the wide variety of in- and output devices for 3D interaction, gives rise to a multitude of novel 3D interaction techniques every year. However, developing such techniques and especially combining them into complex and reusable 3D UIs in the context of VR is still a challenging task. An obvious reason is that the design is challenging from a Human Computer Interaction (HCI) point of view. On top, there is only limited software toolkit support for creating new techniques and integrating them with existing ones.

For an illustration of the challenges in combining techniques, consider the following VR application example. Its scenery contains objects that can be grabbed using a touch metaphor to move them around. Additionally, it is possible to select other, distant objects—e.g., light switches—via a pointing metaphor. Finally, these interactions can be performed independently with either hand.

While this is a rather basic setting, it is representative of many VR applications and already imposes several design challenges. One major challenge is object indication or focusing, which is the process of determining the object that the user wants to interact with. This process can be realized using a variety of approaches, the so-called Focus Strategies. In the above example, users are able to focus objects by using either a touch- or a pointing-based Focus

Strategy. For this to properly work, the system must ensure that the results of both strategies are fused in a consistent and comprehensible manner, e.g., when the user touches an object but also points at another behind it at the same time. Failing to do so can easily result in confusing interaction outcomes, e.g., pressing a button which turns on a light when the intention was to grab an object. While this might seem an easily-resolvable issue at first, it is compounded by the multitude of Focus Strategies that are available for VR applications. For example, pointing can, amongst many others, be realized via ray-casting or IntenSelect [de Haan and Post 2009]. When choosing the former, the aforementioned conflict can be resolved by performing a simple distance comparison between touch and intersection point. If the latter is chosen, however, the required intersection point is not defined and another solution must be found. Given the many different Focus Strategies that offer special strengths and weaknesses, the described scenario is very likely. Unfortunately, it is rarely addressed by existing toolkits. The focus issue becomes even more severe since it is common to combine similar strategies to cover more scenarios, such as ray-casting plus IntenSelect for pointing. Another challenge in VR is bi-manual interaction, which requires all the aforementioned situations to be well-defined for multiple devices. Here, hand-overs of object manipulations from one hand to the other are an often-occuring theme.

So even in this simple example, combining different interaction elements can be challenging. This problem gets even more pronounced when additional and more specialized techniques—like a world-in-miniature or combinations of 2D and 3D interfaces—are added. With most existing frameworks, it is possible to somehow address these challenges when creating new VR applications. However, this usually involves a lot of state management and often comes at the cost of limited flexibility and reusability. To help with such tasks, we developed ViSTA Widgets, a framework for creating 3D UIs with special focus on VR. It is built on top of the ViSTA (Virtual Reality in Scientific and Technical Applications) VR framework [Assenmacher and Kuhlen 2008] and provides functionality to create multi-device, multi-focus-strategy interaction building blocks and thus allows to easily develop complex 3D UIs. This is realized by adding a further device abstraction layer that facilitates, among others, bi-manual interaction, advanced focus management, and a mechanism to combine these elements into novel 3D interaction techniques and 3D widgets.

To introduce our framework, we first present related work, followed by a discussion of technical requirements that form the foundation of its implementation. We then describe elementary components and their interplay. Next, we demonstrate its use by means of code and application examples that illustrate the realization of complex 3D UIs. This is followed by a performance evaluation. We conclude the paper with a discussion of limitations and future work.

## 2 Related Work

VR technology has constantly evolved and is used for various applications. Thus, a lot of different—and often very specialized—hardware has been developed, which has led to a high number of 3D input devices with different capabilities. The different input and output devices are usually well abstracted by VR toolkits (e.g., [Bierbaum et al. 2001], [Kelso et al. 2002], [Assenmacher and Kuhlen 2008]) or tracking frameworks like Virtual Reality Peripheral Network (VRPN) [Taylor II et al. 2001] or OpenTracker [Reitmayr and Schmalstieg 2005]. However, the creation of appropriate 3D UIs is still challenging.

For 2D UIs, standard techniques like the Windows, Icons, Menus and Pointers (WIMP) metaphor have been established and are well supported by powerful UI toolkits like Qt [The Qt Company 2016], Swing [Oracle 2016], or Windows Presentation Foundation (WPF) [Microsoft 2016]. However, while it has been tried to bring classical 2D UIs to Immersive Virtual Environments (IVEs) [Andujar et al. 2006], they are not capable of using the full potential of the third dimension. Recently, low-cost commercial consumer VR Head-mounted Displays (HMDs) have hit the markets which led to a certain degree of support for the creation of 3D UIs through commercially available game engines like the Unreal Engine [Epic Games, Inc. 2016] or Unity [Unity Technologies 2016]. However, they lack support for projection-based VR systems, while also being limited in their capabilities regarding hardware abstraction.

One challenge of designing new 3D interaction techniques that is frequently mentioned in the literature is their high implementation effort. To reduce the transition effort from conceptual ideas to a working implementation, several design models have been proposed. One such model is Concept-Oriented Design [Wingrave and Bowman 2008], which was deduced from examining the design process and aims at increasing reusability and documentation. It uses different tiers—from target behavior formulated in natural language, to implementation code—and is integrated in Chasm, a toolkit to support the design process by appropriate language tools for each tier. [Frees 2010] proposes another design model, which focuses on grouping interaction methods by contexts that are switched at run-time in order to adopt to the current requirements (e.g., precise vs. approximate manipulation).

In an early attempt to provide generic 3D interfaces, [Stevens et al. 1994] propose a system to combine multiple geometric primitives using constraints and relationships. By providing a visual language, this allows for an easy construction of simple 3D widgets.

A common approach to modeling interaction for VR is the use of Data Flow Networks (DFNs). One of the first models for designing 3D UIs [Jacob et al. 1999] combines event-based and continuous data-flow-based interaction, as well as a definition language and visual editor to define the interaction. InTml [Figueroa et al. 2002] uses DFNs to connect semantic components, in order to better separate the interaction from device input and graphical output. [Csisinko and Kaufmann 2010] propose a framework for interaction design that combines DFNs with state machines to model context switches and use adapters between the interaction and device input as well as graphical output. Further research has been conducted on the usage of distributed DFNs [Assenmacher and Raffin 2009] to facilitate interaction in distributed Virtual Environments (VEs). Data-flow-based interaction modeling has continuously been refined and often provides good abstraction and reusability. Also, frameworks often provide specialized languages and editors to aid the interaction design. While data-flow-based interaction description is well suited for high-level interaction, it requires appropriate components (often called filters) to perform specific interaction tasks, which have to be developed on a case-by-case basis. Especially when including graphical interface elements, this can be challenging.

Another approach to make interaction techniques reusable is the Virtual Interactive Namespace (VINS) [Valkov et al. 2012b], which uses named variables in a shared memory space. Since all communication of drivers and graphical output with the interaction is performed over these named variables, a very compact Application Programming Interface (API) suffices. While this facilitates an exchange of the application framework, its limitation to variables (without callbacks or explicit function calls) severely constrains the versatility of the actual design process.

The Viargo library [Valkov et al. 2012a] is an event-based interaction toolkit and incorporates device abstraction and a custom scene-graph for object handling. However, while individual interaction metaphors can be created and reused, it does not allow for the cre-

ation of fully integrated 3D UIs, since compound techniques and inter-dependencies are not modeled.

In summary, we observe that a lot of work has been done on enhancing the design process, providing hardware or toolkit abstraction, and providing reusability. However, these often focus on the interface between the interaction component and the application logic. The actual interaction techniques for which they are used are usually regarded in isolation, while the combination of different simultaneous interaction techniques adds additional challenges. Furthermore, current interface libraries often concentrate on higher-level interaction design, so that low-level aspects like the assembly of interface widgets from basic components (similar to classical 2D UIs) is rarely supported. In this work, we address these aspects.

## 3 Requirements

ViSTA Widgets aim at facilitating the creation of advanced interaction concepts for VR applications. As such, their main purpose is not to provide ready-to-use interaction designs that *end users* can immediately benefit from. Rather, they should support developers in overcoming technical and design-related challenges that are unique to 3D UI design in a VR context. Therefore, the main target audience are *interaction developers*, who create new interaction building blocks, and *application developers*, who use these building blocks to realize application-level interaction concepts.

Compared to the creation of WIMP-style 2D UIs, developers of VR-based 3D UIs face various additional challenges. These mostly stem from the large interaction design space that is available for VR applications. In contrast to 2D UIs, no general design guidelines exist, which leaves a vast number of design alternatives open for consideration. This situation is compounded by the large amount of non-standard input devices that are available for solving interaction problems. The possibility to use multi-modal or bi-manual designs further adds to the complexity of creating interaction concepts.

It is the goal of ViSTA Widgets to provide developers with an approach to handle this kind of complexity. To reach this goal in a reusable and generic fashion, we derived a set of requirements that need to be fulfilled in addition to general requirements like a modular design and platform-independence.

**Requirement R1:** *Provide a proper abstraction mechanism for input devices.* To make interaction building blocks ready for arbitrary and changing device configurations, an appropriate device abstraction has to be found. It is often unclear which devices are to be used with a given interaction metaphor. Here, it is of key importance to anticipate the presence of multiple devices at the same time. Such aspects can already be partly covered by the device abstraction that is provided by DFNs, which is, e.g., the case in ViSTA. They usually provide an abstraction for individual devices and directly enable the realization of basic interaction metaphors, like steering. However, interaction techniques that are created via DFNs are mostly isolated, which often hampers their combination.

**Requirement R2:** *Focus management must be able to handle multiple devices.* It might be possible that certain interaction tasks can be fulfilled by one of many present interaction devices, such as grabbing and dragging an object with either hand. To prepare interaction building blocks for these scenarios, they have to be able to track which devices are currently being used for a given operation.

**Requirement R3:** *Focus management must be able to handle multiple different focus strategies.* Several different techniques exist to focus objects in 3D space to perform tasks on them, all with their very own advantages and disadvantages. Consider examples like ray casting and IntenSelect [de Haan and Post 2009], which both realize the pointing metaphor. While ray casting is well-suited to focus large geometries, IntenSelect is superior in focusing and tracking small objects in crowded scenes. However, the latter comes with the drawback of only being able to focus point-like objects. Consequently, in several settings, the need to combine different techniques emerges, e.g., in a scenery where a swarm of birds moves in front of a mountainous landscape and individual birds as well as single mountains are to be focused. This gets even more complex if, e.g., proximity selection for pushing virtual buttons or selection via speech are added. To account for this, the underlying focus management system needs to be able to evaluate and combine the results of all involved focus strategies.

## 4 ViSTA Widgets Architecture

The ViSTA Widgets library is written in C++ and built on top of the ViSTA VR framework [Assenmacher and Kuhlen 2008]. It has been iteratively developed, where requirements were incrementally fulfilled. Eventually, they were employed in several VR applications. This approach ensured continuous feedback from Widget and application developers, thus leading to ongoing improvements of the system. In this section we describe the framework that resulted from this process, by first giving a short overview, before describing its individual components in more detail.

### 4.1 Overview

In ViSTA Widgets, any component a user can interact with is considered a Widget. Widgets are realized with a Model-View-Adapter (MVA) concept with exchangeable Views for easy customization. So-called Interaction Traits can be used by Widgets to use already implemented interaction techniques. A device abstraction layer comprises Input Slots in Virtual Devices, which encapsulate Focus Strategies to interact with Widgets. However, Focus Strategies are only one part of a Focus Management system that consists of several components and facilitates the simultaneous usage of multiple Focus Strategies and Virtual Devices to interact with Widgets. The communication between the single components is mostly realized via a custom Signals & Slots implementation. For a general overview, the interplay of these components is illustrated in Figure 1. In the following, we first describe the single components before we elucidate their interplay in more detail.

### 4.2 Device Abstraction

VR frameworks like ViSTA often already provide device abstraction—in ViSTA's case by using a DFN—that can be configured for different platforms and input devices. However, there is usually no standard device input routing for interaction techniques. This makes it hard to develop a consistent API for interaction techniques and limits reusability. To overcome these limitations, we introduce the concept of Virtual Devices within ViSTA Widgets.

In contrast to the technical abstraction that is provided by DFNs, Virtual Devices are associated with a semantic. Virtual Devices provide the possibility to structure logical devices that often represent real physical input devices, but can also consist of a multitude or only parts of such. For this, they aggregate so-called Input Slots, which provide the input history for one datum of a device, e.g., the 6-Degrees-of-Freedom (6-DoF) transformation of a 3D input device or the button of a mouse. For the realization of interaction techniques, Virtual Devices and Input Slots are identified via naming conventions. For instance, a "Primary Pointer" Virtual Device for selection can be constructed by adding a 6-DoF "Pose" Input Slot for the device transformation and a Boolean "Select" input slot to map a button of the physical device.

In addition, Input Slots can be shared by Virtual Devices. This enables applications to make use of different interaction techniques, e.g., a primary 6-DoF input device with a button for selection and the same 6-DoF input alongside with another one for two-handed object manipulation. In such cases, an application developer can construct two Virtual Devices: first, one with Input Slots for the 6-DoF and Boolean values of the first device' transformation. Second, a Virtual Device with the Input Slots for the 6-DoF values of both devices' transformations. Hence, Widgets that need 6-DoF and Boolean data, e.g., for object selection can make use of the first Virtual Device, while Widgets that need two 6-DoF inputs, e.g., for two-handed object manipulation, can use the second Virtual Device.

## 4.3 Focus Management

3D UIs have more complex needs for focus management than classical 2D UIs. While in the 2D case, it is usually sufficient to test whether the 2D coordinates of an input device are inside of the front-most surface to determine the focused object, a wide variety of Focus Strategies exists for 3D UIs. In this context, a Focus Strategy is regarded as an algorithm that identifies objects in a scene to be in focus, based on the current state of an input device. As illustrated in the requirements, it is often necessary to combine multiple Focus Strategies in certain applications. To account for this, ViSTA Widgets are capable of managing objects in focus with multiple Focus Strategies and multiple Virtual Devices, thereby always determining unambiguous focus. The remainder of this section describes the components needed to achieve this and explains their interplay.

### 4.3.1 Focus Handles

Every focusable entity in ViSTA Widgets needs to provide at least one so-called Focus Handle. It provides all necessary information so that Focus Strategies can evaluate whether a Focus Handle is a potential candidate of focus. Such information is provided by so-called Focus Handle Aspects which attach semantics to it. For instance, a spherical volume is defined via a center position and a radius. A Focus Handle encapsulates this information by aggregating a Center Focus Handle Aspect and a Radius Focus Handle Aspect. By not only storing a point and a scalar value with the Focus Handle, but also including the semantic information, Focus Strategies can interpret these values according to their functionality. For instance, a Ray Casting Focus Strategy constructs a sphere for the ray intersection test using the provided center point and the radius. In contrast, an IntenSelect Focus Strategy only uses the center point information to evaluate objects in focus, thereby ignoring the radius. This enables different Focus Strategies to evaluate focus using the same Focus Handle as long as it provides the necessary Focus Handle Aspects. Ambiguities are avoided by restricting Focus Handles to aggregate each type of Focus Handle Aspect only once. A Focus Handle Aspect can also contain non-geometric information, e.g., a string identifier for speech input or an object ID to realize focus synchronization in distributed applications.

To account for multiple Virtual Devices, Focus Handles store focus per Virtual Device. This type of information can be used to realize multi-device interaction metaphors.

While it is mostly desired to only focus one dedicated object, e.g., an entry in a menu, situations exist where multiple objects are to be focused simultaneously. This is, e.g., the case when a whole group of particles is to be selected in a flow simulation. To this end, we distinguish two Focus Types for Focus Handles: Passive Focus and Primary Focus. A Focus Handle with Primary Focus can only be focused individually by a Virtual Device. Hence, no other Focus Handle can be focused by the same device at the same time. In contrast, an arbitrary number of Focus Handles with Passive Focus can be focused concurrently.

In addition to Focus Types, any currently focused Focus Handle can request Exclusive Focus. In this case, all other Focus Handles lose focus so that only the requesting Focus Handle maintains focus until it explicitly releases it. This concept allows for interaction techniques where focus and object manipulation need to be decoupled, i.e., when an object could get out of focus while being manipulated. An example for such a behavior is a virtual camera, which is illustrated by a factory layout planning application in Section 5.4.

### 4.3.2 Focus Strategies

Focus Strategies implement techniques to focus objects. More precisely, they evaluate focus on Focus Handles. Depending on their particular nature, Focus Strategies require additional information via Input Slots. The focus evaluation returns a Focus Result List that contains all Focus Handles that are candidates for focus, ordered by a best match scoring. For instance, a Ray Casting Focus Strategy will return a list of all Focus Handles whose geometric descriptions intersect a device' pick ray in ascending order by distance to the input device.

Focus Strategies use so-called Handle Infos to interpret Focus Handle Aspects. This indirection enables application and Widget developers to use their own Focus Handle Aspects with existing Focus Strategies. Handle Infos transform the generic description provided by Focus Handle Aspects to data that can be directly processed by Focus Strategies. They are provided via a factory, where they can be registered and queried at run-time. For instance, a Ray Casting Focus Strategy can query the factory to provide a Handle Info that transforms the Center and Radius Focus Handle Aspects of a particular Focus Handle to an intersection point of the respective ray and sphere.

Particular interaction techniques require information from Focus Strategies that go beyond a ranking of Focus Handles. This is, e.g., the case when the intersection point of a pick ray and the respective focused object is needed to realize a certain behavior. Focus Strategies can provide this type of information by attaching a so-called Focus Result Aspect to a processed Focus Handle. Later, such information can be used by querying respective Result Aspects from Focus Handles.

For the combined use of multiple Focus Strategies, we introduce so-called Merger Focus Strategies. In contrast to regular Focus Strategies, they operate on other Focus Strategies as input. They merge the respective Focus Result Lists of the input Focus Strategies and return a single, combined Focus Result List as output.

To cover most application scenarios out of the box, ViSTA Widgets come with a set of predefined Focus Strategies. First, a Ray Casting Focus Strategy is provided to focus geometric objects. Second, an IntenSelect Focus Strategy is available, which performs better in focusing small objects in crowded scenes. Third, a Proximity Focus Strategy enables to focus objects at close range by touching. Additionally, an Always-in-focus Strategy grants focus to all Focus Handles that have a Focus Handle Aspect of a certain, application-developer-defined type. It is used to realize, e.g., modal dialogs like menus or to capture focus if no other object is currently focused, e.g., to trigger actions when a user clicks into an empty area of the scene. However, such a behavior is only useful when using multiple Focus Strategies. Therefore, the Priority Merger Focus Strategy combines the results of multiple Focus Strategies based on an application-developer-defined priority order.

While a wide variety of use cases are covered with these Focus Strategies, application developers are encouraged to add own ones,

whenever special behavior is to be realized. This accounts for new Focus Strategies, but also for special-purpose Merger Strategies.

#### 4.3.3 Focus Dispatcher

After Focus Strategies have evaluated focus, it has to be assigned to the respective Focus Handles. The responsibility for this lies at the Focus Dispatcher. One such Focus Dispatcher is present in each Virtual Device and it operates on one Focus Strategy. If a combination of Focus Strategies is needed, Merger Focus Strategies are to be used.

Focus dispatching is triggered via an update from the associated Virtual Device. As a result, the Focus Dispatcher first triggers the evaluation of the assigned Focus Strategy. Next, it iterates over the returned Focus Result List and assigns focus to the first Focus Handle with Primary Focus. If the list does not contain a Focus Handle with Primary Focus, it assigns focus to all Focus Handles with Passive Focus. Finally, it resigns focus from all Focus Handles that had focus in the previous evaluation, but lost it in the current one. Widgets and other components can react to focus changes by observing the respective Signals of Focus Handles.

### 4.4 Signal System

Communication between components in ViSTA Widgets is primarily realized by a custom Signals and Slots system. It provides functionality similar to boost Signals2 [Gregor and Hess 2016] or Qt Signals & Slots [The Qt Company 2016]. Slots can be connected to Signals, which notify events to all connected Slots. Signals include a Signal Argument when emitted, containing the Signal sender and optional further information. In most cases, this additional information communicates a value change. Slots are methods that must accept the argument type of the Signal they shall be connected to.

Signal arguments can either be read-only or read-write. The first case is used when a Slot is meant to trigger a reaction to a particular event. The second case is used to provide a feedback-channel for slots to manipulate values in the arguments and thus give the signal sender the opportunity to react to such a manipulation. This behavior is, e.g., used to constrain values, which is described later.

### 4.5 Interaction Traits

While the focus management system and the input data from Virtual Devices already allow for implementing nearly arbitrary interaction techniques, it is undesired to re-implement basic interaction concepts multiple times. To account for this, ViSTA Widgets provide reusable building blocks for interaction tasks that are regularly needed, so-called Interaction Traits.

In situations where more than one Focus Handle is used within a Widget, focus switches from one Focus Handle to another have to be managed correctly. This also accounts for managing the focus of several Virtual Devices. To this end, the so-called Focusable consolidates the focus of several Focus Handles and notifies focus changes via Signals.

Usually, users should not only be able to focus Widgets, but also to interact with them. Building upon the Focusable Trait, ViSTA Widgets provide a Clickable Trait, which notifies button presses and clicks via Signals when an aggregated Focusable is in focus. It accounts for multiple Virtual Devices, thus allowing, e.g., the handover of button presses between them.

Extending these Traits, a Draggable Trait allows for dragging Widgets around by focusing them and holding a button pressed. The Draggable notifies a new position when dragging occurs. Similar to the Clickable Trait, it accounts for multiple Virtual Devices and allows handover from one device to another.

### 4.6 Fields

The properties of single components in ViSTA Widgets are realized as so-called Fields. Fields encapsulate a data item with a setter and a getter, each, and can be constrained and observed via Signals. To this end, every field has two Signals, a Constraint Signal and a Value Changed Signal. Both Signals emit Value Changed Arguments. However, the argument of the Value Changed Signal is read-only while the argument of the Constraint Signal is read-write and has an additional flag to determine if Slots have actually constrained the value.

Slots can be attached to the Constraint Signal to constrain the value of a Field. If a new value does not match the requirements of the constraining Slot, it can change it in the Value Changed Argument. All following Constraint Slots see the previously constrained value as the new value and can further constrain it, if necessary. However, this could lead to conflicts. To account for this, the Constraint Signal is raised once more after all Slots have been called in case its value got constrained. If the value gets constrained again, this reveals a constraint conflict and a warning is emitted.

ViSTA Widgets come with three fundamental Constraints that can be attached to any suitable Field. First, a Range Constraint limits a Field's value to a certain range. Further, a Minimum Constraint and a Maximum Constraint prevent the value of the Field from exceeding a certain lower or upper bound, respectively.

### 4.7 Widgets

As already mentioned, any component a user can interact with is considered a Widget. Widgets are realized with the MVA pattern, a variant of the Model-View-Controller (MVC) pattern. The framework offers base classes for each component, where the adapter base class is called Widget. It forms the primary contact point to work with Widgets. While the Model and the Adapter are tightly coupled, the View is exchangeable for every Widget. This way, the look of single Widgets can be changed, while adaptions to new render engines or VR toolkits can also be performed without logic changes. A View interface must be provided, enabling the adapter, i.e., the Widget, to set all necessary View properties. Model and View are independent components that do not rely on each other or the Adapter. In contrast, the Adapter aggregates both Model and View and is responsible for keeping the data in both up-to-date while also being responsible for the interaction logic.

As common with MVA implementations, the basic description of the Widget is stored in the Model. The Model is realized as a class containing several publicly accessible Fields. They are used to configure the Widget, but also to react to user input by observing them.

The View is responsible for generating a representation of the Widget that is displayed to the user. Here the term "View" is not to be taken literally, since a Widget does not necessarily have a visual representation. It could as well be, e.g., acoustic, haptic, olfactory, or multi-modal. For Widgets that are designed for input without direct user feedback, e.g., a Widget intercepting clicks in empty regions of a scene, it is also valid to not provide a View at all.

By default, every Widget comes with the three Interaction Traits Focusable, Clickable, and Draggable. These can be enabled or disabled according to functionality needs. To allow for an aggregated behavior in case of Compound Widgets, which are described below, only the Signals of the default Traits are exposed by the Widget's API. However, custom Traits can be added by Widget developers.

To create a focusable Widget, Focus Handles have to be registered, which are automatically forwarded to the contained Focusable. In addition, Focus Handle Aspects need to be provided to describe how the Widget can obtain focus.

To realize complex Widgets, it is often useful to combine several basic Widgets. To this end, Widget developers create Compound Widgets, which are composed of multiple sub-Widgets. For this, a parent Widget creates Widgets and registers them as child Widgets of its own. This way, the hierarchy of a Widget is hidden from application developers, exposing its functionality as a single entity. This also accounts for the focus management in Compound Widgets. A Widget is considered as being in focus whenever any of its Focus Handles or its child Widgets is in focus.

### 4.8 Widget Manager

The Widget Manager forms the central component of ViSTA Widgets, since it serves as connecting point to the rest of an application. In order to use ViSTA Widgets in an application, an instance of the Widget Manager has to be created. All Virtual Devices and Widgets must be registered with the Widget Manager. It is then responsible to trigger updates on the respective components.

### 4.9 Putting it All Together—The Widget Update Cycle

While the previous sections described the individual components of the ViSTA Widgets system, this section focuses on the interplay of the components, i.e., the Widget update cycle. This process is illustrated in Figure 1.

The update cycle is performed periodically—e.g., once per frame—by calling the update routine of the Widget Manager. Before performing the update cycle, all Input Slots must be filled with current data, so that all components are up-to-date. The update cycle consists of three steps. First, focus is evaluated, since all other components depend on a consistent focus state of the whole system. Next, the Interaction Traits are updated to ensure that they are all in a consistent state, too, when the Widgets get updated in the last step.

To illustrate this in more detail, the Widget Manager first updates all Virtual Devices. This triggers an update of the Focus Dispatchers, which trigger an evaluation of the associated Focus Strategies. If Merger Strategies are used, they recursively trigger the evaluation of their respective child Focus Strategies. Finally, the Focus Dispatchers update all Focus Handles whose states have changed.

In the next step of the update cycle, the Widget Manager triggers an update of Interaction Traits on all registered Widgets. The Widgets then forward the call to all associated Interaction Traits and recursively to their child Widgets' Traits, if present.

In the last step, the Widget Manager triggers update calls on all Widgets. These are recursively forwarded to child Widgets, if present. Finally they trigger an update of their Views.

## 5 Usage Examples

To illustrate how ViSTA Widgets are used in practice, we provide some examples. First, a code example is given to demonstrate how to implement an application. Next, we present selected applications to illustrate the usefulness of different features of ViSTA Widgets. Afterwards, we illustrate the impact on latency introduced by the framework with a performance evaluation.

### 5.1 Code Example

In the following example, we use two 6-DoF input devices with a ray casting and an IntenSelect Focus Strategy, each. Furthermore, a Sphere Widget is added to be manipulated by them. To create this setup, first, a Widget Manager is instantiated and registered at the system.

```
manager = new WidgetManager;
handler = new WidgetManagerUpdateHandler(
  manager,ViSTA->GetEventManager());
```

In this example, an update handler for ViSTA is used. However, with a different toolkit, the application developer simply has to take care that the Widget Manager's update method is called periodically. Next, the Input Slots and Virtual Devices are created:

```
pose = new InputSlot<TransformMatrix>;
button = new InputSlot<bool>;
device = new VirtualDevice("Pointer");
device->AddInputSlot("Pose",pose);
device->AddInputSlot("Select",button);
manager->AddVirtualDevice(device);
```

When using ViSTA, the Input Slots are usually filled each frame via ViSTA's DFN, where the Input Slots are identified via their respective names. With a different toolkit, the application developer is responsible for filling the Input Slots with current values before the Widget Manager's update method is called. In the next step, the Focus Strategies are created and connected to the Input Slots and Virtual Devices:

```
ray = new RayCastingFocusStrategy;
ray->SetPoseInputSlot(pose);
inten = new IntenSelectFocusStrategy;
inten->SetPoseInputSlot(pose);
merger = new PriorityMergerFocusStrategy;
merger->AddStrategy(ray);
merger->AddStrategy(inten);
device->GetFocusDispatcher()->SetRootStrategy(merger);
```

After two Virtual Devices, `left_device` and `right_device`, have been created as illustrated above, the application is prepared to use Widgets, e.g., a Sphere Widget:

```
sphere = new SphereWidget;
sphere->Init();
sphere->RegisterVirtualDevice(left_device);
sphere->RegisterVirtualDevice(right_device);
manager->AddWidget(sphere);
```

Now a Sphere Widget exists that can be clicked and dragged by both devices without further configuration. More complex 3D UIs can be created by adding further Widgets and reacting to interactions performed on them. Examples are illustrated in the remainder of this section.

### 5.2 Coordinating Interactions in Multi-view Systems

ViSTA Widgets are used for coordinating interactions in VisNEST, a Coordinated Multiple Views (CMV) system that visualizes neuroscientific simulation results [Nowke et al. 2013]. CMV is a category of visualization systems that use two or more distinct views to support the investigation of a single conceptual entity [North and Shneiderman 1997; Wang Baldonado et al. 2000]. Note that a view of a CMV systems is not to be confused with a view of the MVA concept, as it was discussed before.

In VisNEST, views are single applications running on multiple computers interlinked through a network. To synchronize selection

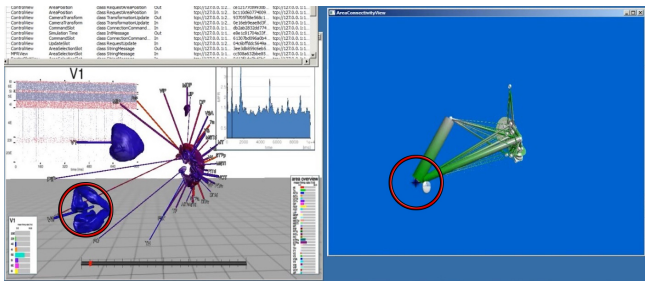

**Figure 2:** *Two applications are coordinated via a Network Focus Strategy: a manipulation in the right application updates the positions of the brain areas in the left application (red circles).*

states across all applications, they must be shared via the respective network interfaces. However, one major challenge arising from such a design are conflicting input states if users interact with multiple applications simultaneously. Consider the case when a user selects a visual entity in an application *A* and then selects another one in an application *B*. A simple yet ineffective solution would be to use the last selection state received, hence discarding any previously performed interaction. However, the Widget systems allows for a more elegant solution: a Network Focus Strategy which receives selection states from a remotely connected Widget (see Figure 2). To this end, a Network Focus Handle Aspect has been realized in order to synchronize Widget interaction. A Network Focus Strategy is fed selection states from remotely connected Widgets via an asynchronous poll mechanism. To allow for non-conflicting input state handling, the Network Focus Strategy is attached to a Virtual Device, which coexists with the standard device for the particular platform to allow for, e.g., the selection with a flying joystick in a CAVE and a coupled desktop application. One benefit of this approach is the non-distracting interaction behavior for users simultaneously using the system: a user in a CAVE can see the selection performed on the desktop application but is not affected in her own interaction and vice versa. In addition, this approach enables to prioritize local interactions, e.g., when the remotely coupled Widget currently operates on the same entity but is overruled by local changes.

### 5.3 Bottle Blowing

**Figure 3:** *In Bottle Blowing, bottles are realized as Widgets while each finger of the user's hands represents a Virtual Device.*

[Zielasko et al. 2015] created Bottle Blowing as a virtual music instrument. Using an IVE, classical blowing of bottles is simulated to create according sounds. Additionally, more versatile interaction is possible than in reality, for example by allowing to play several bottles simultaneously. To play the instrument, the user triggers a virtual air stream by blowing into a microphone. This stream is then redirected along her arms and through her fingers, which allows to intone virtual bottles by pointing at them (see Figure 3).

Zielasko et al. created this multi-modal application by combining interaction building blocks of ViSTA Widgets. To this end, every bottle is instantiated as Widget using predefined standard components. While the View represents a geometrical model of a bottle, a Focus Handle with a Center and a Radius Focus Handle Aspect is used for the focus management. The Focus Handle marks the opening of a bottle, as this is the point that should be aimed at with the airstream. The behavior of the bottle Widget is defined using the available Interaction Traits, i.e., Clickable and Focusable. The bottle is highlighted by glowing when focused (see Figure 3). When getting clicked, the Widget raises a Signal, which triggers the application to play the corresponding tone. Additionally, bottle Widgets can be put into a draggable mode, which allows to reposition them.

With this setup the bottles are already playable with every point-and-click device, e.g., a mouse in a desktop setting or a flying joystick in an IVE. However, in [Zielasko et al. 2015] bottles are played by hand gestures and blowing into a microphone. For this, the output of hand tracking and a blow detection algorithm are fed into the Widget system using the concept of Virtual Devices: first, eight 6-DoF and one Boolean Input Slots are created and marked as "Pose" and "Select" slots, respectively. The former each receive the pose of the respective fingers, which are tracked by a Leap Motion. The latter is set based on the blow detection running in the background. A Virtual Device is created for each finger, which combines the respective pose Input Slot with the shared select Input Slot. Finally, an IntenSelect Focus Strategy is instantiated for each Virtual Device. As result, the user can use all fingers at the same time to play various sets of bottles simultaneously.

In summary, a novel multi-modal, multi-hand interaction was created by combining predefined interaction building blocks of the Widget system. This use case shows that ViSTA Widgets can be used to quickly develop IVEs that go beyond the classic notion of Widget-based interaction. Although, the application itself is rather playful, the created Virtual Devices could be directly reused in more serious applications, e.g., for system control [Zielasko et al. 2015], showing furthermore the benefits of the modular design of the proposed Widget system.

### 5.4 VR-based Factory Layout Planning

[Pick et al. 2014] presented flapAssist, a VR-based factory layout planning application that allows planners to review and modify machine layouts during virtual walkthroughs. Planners are supported by interactive visualizations that provide comprehensible access to relevant planning-related data, such as material flow costs. An annotation system further allows to record comments and decisions using various interaction techniques [Pick et al. 2016]. Users can input texts, take virtual photos, create sketches, or produce voice recordings to form different kinds of annotations. This interaction has been realized using ViSTA Widgets. The application mainly targets immersive VR systems, but also fully supports desktop PCs.

The most vital part for the integration of the various interaction techniques is the focus management of ViSTA Widgets. Interaction techniques utilize different kinds of Focus Strategies and rely on the guarantees the Widget system makes in terms of when focus is granted and to whom. One characteristic situation occurs when elements of different interaction techniques occlude each other, e.g., configuration dialogs opened in front of visualization elements. In

**Figure 4:** *Interaction in flapAssist, like the SceneCamera shown here, is realized using ViSTA Widgets. Focus Strategies are extensively used to consistently combine all Widgets.*

these situations it must be ensured that the overall interaction behavior remains consistent and predictable to the user to avoid confusion and erroneous handling. For this, Merger Focus Strategies are used extensively. Since most occlusions were identified to be caused by Widgets that primarily rely on ray casting, the related Focus Strategy is given priority over others, e.g., IntenSelect, using a Priority Merger Focus Strategy. Modal interaction techniques, such as the *SceneCamera* for taking virtual photos (see Figure 4), or Extended Pie Menus [Gebhardt et al. 2013] for configuration, make heavy use of the Exclusive Focus mechanism and the Always-in-focus Strategy. To avoid accidental interaction with other Widgets, these techniques request Exclusive Focus as soon as they become active.

Another part of ViSTA Widgets, the Input Slots, is used for the integration of speech recognition to perform multi-modal interaction tasks. For object selection, a Virtual Device is defined, which holds a 6-DoF Input Slot that is fed by a 6-DoF input device but uses a speech-recognition-based Boolean Input Slot for selection. Depending on the command a user speaks, the state of the Input Slot is modified. To emulate a click, a command like 'select' toggles the selection slot's state to *true* and immediately back to *false*. In contrast, to realize grabbing, a command like 'grab' first sets the slot's state to *true*, but then awaits another command such as 'release' to set it back to *false*.

### 5.5 Analysis of Multi-dimensional Functions

In the configuration of manufacturing processes, e.g., the configuration of laser cutting machines, parameters like the power or the focal position of the laser affect quality criteria like the roughness of the resulting cut or the cutting speed. This mapping can be modeled via multi-dimensional functions, so-called metamodels [Khawli et al. 2016]. To aid engineers in optimizing manufacturing processes, memoSlice, a CMV application for the analysis of such metamodels was created.

It is designed to be used as stand-alone application for desktop systems, but also as Widget that can be brought up for the optimization of machine settings within flapAssist as illustrated in Figure 5 [Gebhardt et al. 2014]. To achieve this, it has been built upon ViSTA Widgets. The whole visualization system is designed as a multi-layered Widget architecture. A single Compound Widget comprises several child Widgets that are again composed of child Widgets and so on. This design allows for easily realizing direct manipulation on different types of visualizations for the shown metamodel.

For stand-alone desktop use, this Widget can be instantiated in an application that sets up a basic Widget system. Here, a Virtual Device, which uses mouse input that has been transformed to 6-DoF input via a DFN is used to interact with the Widget. To be used within flapAssist, the Widget can be instantiated and registered with the existing Virtual Devices like any other Widget. Due to the device and focus abstractions provided by ViSTA Widgets no further setup is required to be able to interact with memoSlice in flapAssist.

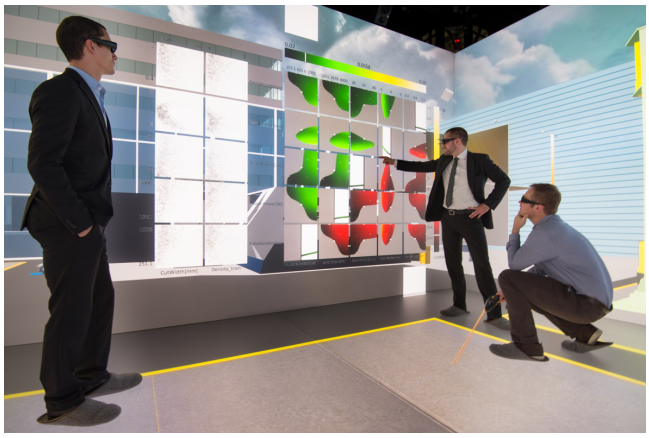

**Figure 5:** *Collaborative analysis session with memoSlice as Widget inside of flapAssist.*

### 5.6 Performance Evaluation

In VR applications, performance is crucial, since latency and low frame rates can reduce immersion and introduce simulator sickness. Consequently, we took care to minimize the impact of ViSTA Widgets on the overall performance of applications that make use of it. The latency added by our framework is determined by the overall computation time required by the Widget system's update call, as well as the rendering of the Views. It cannot be generally named, since it depends on the actual Widget setup. The rendering delay of the views heavily depends on the respective quality and implementation and will not be further analyzed here.

For the update cycle, we found that the required time strongly depends on the current setup. However, it is split among the three steps of the update cycle. First, in the update of Virtual Devices the evaluation of Focus Strategies consumes most of the calculation time. Second, the update of Traits is usually the smallest consumer of calculation time. Third, in the update of Widgets we could observe the strongest variance among different applications. Examples of the performance impact are illustrated in Table 1, where we present measurements for different applications and test setups. The measurements were performed on a desktop PC with an Intel® Xeon® E5-1650 v2 CPU (6 cores, 3.5 GHz) with 12 GB of RAM.

These measurements include some of the sample applications presented earlier. In Bottle Blowing, the described setup of eight Virtual Devices (one for each finger), each with an IntenSelect Focus Strategy is used and ten bottle Widgets are present in the example setup. Here, the total performance impact for the use of ViSTA Widgets is $< 0.04$ ms and thus negligible.

The applications flapAssist and memoSlice are among the most complex ones that were realized with ViSTA Widgets so far. They were measured stand-alone and as combined version. The times

| | # Widgets | Device | Trait | Widget | Total |
|---|---|---|---|---|---|
| BB | 10 | 0.031 | 0.006 | 0.000 | 0.039 |
| fA | 1,019 | 1.101 | 0.132 | 1.094 | 2.331 |
| mS | 1,717 | 1.625 | 0.301 | 0.686 | 2.615 |
| fA+mS | 2,041 | 1.432 | 0.306 | 2.174 | 3.916 |
| SWT | 10,000 | 8.482 | 0.724 | 0.069 | 9.278 |
| EPMT | 8,849 | 0.001 | 0.919 | 0.427 | 1.351 |

**Table 1:** *Number of Widgets and update times (ms) for the applications Bottle Blowing (BB), flapAssist (fA), memoSlice (mS), memoSlice integrated in flapAssist (fA+mS), Sphere Widget Test (SWT), and Extended Pie Menu Test (EPMT). Times are depicted for the single steps of the update cycle and its total time.*

depict the average performance impact per frame over an analysis session with different data sets from productive environments. Both applications make use of a variety of pre-defined and custom Widgets with different complexities. In both applications, the number of present Widgets depends on the currently analyzed data set. Each setup makes use of two Virtual Devices. A viewer device is used for aligning certain Widgets to the viewer's eye position, but does not contain any Focus Strategies. The second one is used for interacting with the application. It uses a combination of ray casting and IntenSelect together with multiple Always-in-focus Strategies that realize various modal dialogs. Most of the calculation time for the update cycle is consumed by the evaluation of Focus Strategies and the Widget updates, while Trait updates consume the smallest proportion. The largest overall calculation time is present in the combined solution, with a total update time of $< 4$ ms. Consequently, a target frame rate of, e.g., 60 Hz can be maintained if the rest of the application does not require a calculation time of $> 12$ ms. If the Widget system is run concurrently to the render thread, the added latency of $< 4$ ms is negligible.

In addition to benchmarks of the presented sample applications, we included two test settings to illustrate performance boundaries. First, we show measurements for a settings where 10,000 Sphere Widgets are randomly distributed in empty space, all of which can be grabbed and dragged around. We chose this setting to illustrate an upper boundary for the performance impact of the evaluation of Focus Strategies, because 10,000 active Widgets by far exceed any real scenario we encountered until now. Two Virtual Devices are present, each with a ray casting and an IntenSelect Focus Strategy combined via a Priority Merger Focus Strategy. Consequently, focus evaluation on all Widgets is performed for four Focus Strategies. In this setting it is the biggest impact factor, contributing with over 90 % to the overall update time of $< 10$ ms. While 10 ms are definitely a considerable amount of update time, a frame rate of 60 Hz can still be maintained if the rest of the application does not a require calculation time of $> 6$ ms.

As second test setting, we included measurements for a scene that only contains a complex Extended Pie Menu, to illustrate the impact of modal menus. This menu consists of four layers with 4 entries each. Every sub-menu and menu entry is implemented as a Widget, which results in 8,849 Widgets in total, which, again, exceeds any real-world scenario we encountered so far. The provided measurements apply for the menu being opened to the lowest layer, which means that one sub-menu per hierarchy is visible, while all others are invisible. One Virtual Device is present, only facilitating an Always-in-focus Strategy to grant Exclusive Focus. Since the menu system was developed independently and later adapted to our Widget system, it handles entry selection internally within its Widget and Trait updates. Consequently, these two update steps are the strongest contributors to the overall update time, which is slightly above 1 ms. This example illustrates that even complex modal menus only have a marginal performance impact when utilizing ViSTA Widgets.

In summary, the presented measurements show that the performance impact of our framework strongly depends on the current setup. Contributing factors are the number and type of used Widgets, the amount of currently active Widgets, their complexity, and especially the number and type of used Focus Strategies. For the most common, relatively simple scenarios, the added latency is negligible. Even in complex setups with several thousand active Widgets, only a tolerable amount of end-to-end latency is added to the overall system by using ViSTA Widgets. Furthermore, it has to be kept in mind that tasks like focus evaluation and interaction handling also need to be performed if a custom solution is realized. Consequently, the presented times cannot be seen as pure overhead of ViSTA Widgets.

## 6 Limitations and Future Work

As already mentioned and illustrated by the example applications, ViSTA Widgets have been developed in an iterative development process where application developers participated as early as possible. While the framework is in a stable state and being used in production, there are still limitations that will be addressed in future development cycles.

One of these limitations is due to the implementation of the current constraint mechanism. While constraining Fields via the aforementioned Constraint Signal mechanism enables flexibility in development, this feature is prone to breaking Widgets if application developers add Constraints that conflict with Constraints included by Widget developers. This can happen, e.g., if a slider Widget's Range Constraint enforces a range that conflicts with that of another user-set Range Constraint. While this would currently only produce a run-time warning, we plan to include measures in future development cycles that prevent such misuse.

Another limitation arises from the flexibility of the framework. While it provides developers with many possibilities in terms of the design of interaction techniques and concepts, it also requires them to craft noticeable amounts of glue code to set up Input Slots, Virtual Devices, Focus Strategies, and so on. Based on developer feedback and code reviews, we plan to address this overhead by providing building blocks for recurring application scenarios. However, it is not trivial to identify a reasonable granularity of such building blocks that, on the one hand, significantly reduces setup code while, on the other hand, maintains flexibility. Consequently, further analysis will be required until we successively add such features.

Until now, ViSTA Widgets are not publicly available, but we have scheduled an open source release in the near future.

## 7 Conclusion

In this paper, we presented ViSTA Widgets, a new framework for the creation of 3D UIs for IVEs. We illustrated how the modular design with its device abstraction and focus management, along with other features, enables the creation of reusable interaction building blocks. We demonstrated how these building blocks can easily be combined and extended to form complex 3D UIs by presenting example applications. Since we are continuously improving the framework, it still has limitations, which were also discussed to outline the direction of future development cycles. Nonetheless, ViSTA Widgets is already a valuable framework that makes the creation of 3D UIs easier and faster for application developers and is a great help in the daily work of developing VR applications.

## Acknowledgements

The depicted research was partly funded by the German Research Foundation DFG as part of the Cluster of Excellence "Integrative Production Technology for High-Wage Countries". The authors would like to acknowledge the support by the Excellence Initiative of the German federal and state governments and in particular the Jülich Aachen Research Alliance – High-Performance Computing.

## References

ANDUJAR, C., FAIREN, M., AND ARGELAGUET, F. 2006. A Cost-effective Approach for Developing Application-control GUIs for Virtual Environments. In *3DUI 2006*, IEEE, 45–52.

ASSENMACHER, I., AND KUHLEN, T. 2008. The ViSTA Virtual Reality Toolkit. In *Proc. of IEEE VR SEARIS Workshop*, IEEE, 23–28.

ASSENMACHER, I., AND RAFFIN, B. 2009. A Modular Framework for Distributed VR Interaction Processing. In *JVRC'09*, Eurographics.

BIERBAUM, A., JUST, C., HARTLING, P., MEINERT, K., BAKER, A., AND CRUZ-NEIRA, C. 2001. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proc. of IEEE VR, 2001*, IEEE, 89–96.

CSISINKO, M., AND KAUFMANN, H. 2010. VITAL - The Virtual Environment Interaction Technique Abstraction Layer. In *Proc. of IEEE VR SEARIS Workshop*, IEEE, 77–86.

DE HAAN, G., AND POST, F. H. 2009. StateStream. In *EICS '09*, ACM, 13–22.

EPIC GAMES, INC., 2016. Unreal Engine Technology. https://www.unrealengine.com/. Accessed: 2016-10-04.

FIGUEROA, P., GREEN, M., AND HOOVER, H. J. 2002. InTml: A Description Language for VR Applications. In *Proc. of the Seventh International Conference on 3D Web Technology*, ACM, 53–58.

FREES, S. 2010. Context-driven Interaction in Immersive Virtual Environments. *Virtual Reality 14*, 4, 277–290.

GEBHARDT, S., PICK, S., LEITHOLD, F., HENTSCHEL, B., AND KUHLEN, T. 2013. Extended Pie Menus for Immersive Virtual Environments. *IEEE TVCG 19*, 4, 644–651.

GEBHARDT, S., PICK, S., AL KHAWLI, T., VOET, H., REINHARD, R., HENTSCHLE, B., AND KUHLEN, T. 2014. Integration of VR- and Visualization Tools to Foster the Factory Planning Process. In *11. Fachtagung "Digital Engineering zum Planen, Testen und Betreiben technischer Systeme"*, Fraunhofer-Institut für Fabrikbetrieb und -automatisierung IFF.

GREGOR, D., AND HESS, F. M., 2016. Chapter 30. Boost.Signals2. http://www.boost.org/doc/libs/1_61_0/doc/html/signals2.html. Accessed: 2016-30-06.

JACOB, R. J., DELIGIANNIDIS, L., AND MORRISON, S. 1999. A Software Model and Specification Language for non-WIMP User Interfaces. *ACM TOCHI 6*, 1, 1–46.

KELSO, J., ARSENAULT, L. E., SATTERFIELD, S. G., AND KRIZ, R. D. 2002. DIVERSE: A framework for building extensible and reconfigurable device independent virtual environments. In *Proc. of IEEE VR, 2002*, IEEE, 183–190.

KHAWLI, T. A., GEBHARDT, S., EPPELT, U., HERMANNS, T., KUHLEN, T., AND SCHULZ, W. 2016. An Integrated Approach for the Knowledge Discovery in Computer Simulation Models with a Multi-dimensional Parameter Space. *AIP Conference Proc. 1738*, 1.

MICROSOFT, 2016. Windows Presentation Foundation. https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx. Accessed: 2016-10-04.

NORTH, C., AND SHNEIDERMAN, B., 1997. A Taxonomy of Multiple Window Coordination.

NOWKE, C., SCHMIDT, M., ALBADA, S. J. V., EPPLER, J. M., BAKKER, R., DIESMANN, M., HENTSCHEL, B., AND KUHLEN, T. 2013. VisNEST – Interactive Analysis of Neural Activity Data. *IEEE BioVis*, 65–72.

ORACLE, 2016. java.com: Java + You. https://www.java.com/en/. Accessed: 2016-10-04.

PICK, S., GEBHARDT, S., KREISKÖTHER, K., REINHARD, R., VOET, H., BÜSCHER, C., AND KUHLEN, T. 2014. Advanced Virtual Reality and Visualization Support for Factory Layout Planning. In *Proc. of the Conference Entwerfen Entwickeln Erleben*, TUDpress.

PICK, S., WEYERS, B., HENTSCHEL, B., AND KUHLEN, T. W. 2016. Design and Evaluation of Data Annotation Workflows for CAVE-like Virtual Environments. *IEEE TVCG 22*, 4, 1452–1461.

REITMAYR, G., AND SCHMALSTIEG, D. 2005. OpenTracker: A Flexible Software Design for Three-dimensional Interaction. *Virtual Reality 9*, 1, 79–92.

STEVENS, M. P., ZELEZNIK, R. C., AND HUGHES, J. F. 1994. An Architecture for an Extensible 3D Interface Toolkit. In *ACM UIST*, ACM, 59–67.

TAYLOR II, R. M., HUDSON, T. C., SEEGER, A., WEBER, H., JULIANO, J., AND HELSER, A. T. 2001. VRPN: A Device-independent, Network-transparent VR Peripheral System. In *Proc. of ACM VRST*, ACM, 55–61.

THE QT COMPANY, 2016. Qt. http://www.qt.io/. Accessed: 2016-10-04.

UNITY TECHNOLOGIES, 2016. Unity - Game Engine. https://unity3d.com/. Accessed: 2016-10-04.

VALKOV, D., BOLTE, B., BRUDER, G., AND STEINICKE, F. 2012. Viargo - A Generic Virtual Reality Interaction Library. In *Proc. of IEEE VR SEARIS Workshop*, IEEE, 23–28.

VALKOV, D., GIESLER, A., AND HINRICHS, K. 2012. VINS: Shared Memory Space for Definition of Interactive Techniques. In *Proc. of ACM VRST*, ACM, 145–152.

WANG BALDONADO, M. Q., WOODRUFF, A., AND KUCHINSKY, A. 2000. Guidelines for Using Multiple Views in Information Visualization. In *Proc. of the Working Conference on Advanced Visual Interfaces*, ACM, 110–119.

WINGRAVE, C. A., AND BOWMAN, D. A. 2008. Tiered Developer-centric Representations for 3D Interfaces: Concept-oriented design in Chasm. In *IEEE VR'08*, IEEE, 193–200.

ZIELASKO, D., RAUSCH, D., LAW, Y. C., KNOTT, T. C., PICK, S., PORSCHE, S., HERBER, J., HUMMEL, J., AND KUHLEN, T. W. 2015. Cirque des Bouteilles: The Art of Blowing on Bottles. In *In Proc. of IEEE 3DUI*, IEEE, 209–210.