ISSN 2464-4617 (print) WSCG 2016 - 24th Conference on Computer Graphics, Visualization and Computer Vision 2016
ISSN 2464-4625 (CD-ROM)

MorphableUI: A Hypergraph-Based Approach to Distributed Multimodal Interaction for Rapid Prototyping and Changing Environments

Andrey Krekhov
High Performance Computing
University of Duisburg-Essen
47057, Duisburg, Germany
andrey.krekhov@uni-due.de

Jürgen Grüninger
Intel VCI
Saarland University
66123, Saarbrücken, Germany
juergen.grueninger@dfki.de

Kevin Baum
Intel VCI
Saarland University
66123, Saarbrücken, Germany
baum@intel-vci.uni-saarland.de

David McCann
Intel VCI
Saarland University
66123, Saarbrücken, Germany
mccann@intel-vci.uni-saarland.de

Jens Krüger
High Performance Computing
University of Duisburg-Essen
47057, Duisburg, Germany
jens.krueger@uni-due.de

ABSTRACT

Nowadays, users interact with applications in constantly changing environments. The plethora of I/O modalities is beneficial for a wide range of application areas such as virtual reality, cloud-based software, or scientific visualization. These areas require interfaces based not only on the traditional mouse and keyboard but also on gestures, speech, or highly-specialized and environment-dependent equipment.
We introduce a hypergraph-based interaction model and its implementation as a distributed system, called MorphableUI. Its primary focus is to deliver a user- and developer-friendly way to establish dynamic connections between applications and interaction devices. We present an easy-to-use API for developers and a mobile frontend for users to set up their preferred interfaces.
During runtime, MorphableUI transports interaction data between devices and applications. As one of the novelties, the system supports I/O transfer functions by automatically splitting, merging, and casting inputs from different modalities. MorphableUI emphasizes rapid prototyping and, e.g., facilitates the execution of user studies due to easy UI reconfiguration and device exchangeability.

Keywords

Dynamic interfaces; scenario-dependent interaction; rapid prototyping.

1 INTRODUCTION

Present-day technologies allow applications to run in heterogeneous and changing environments. Different environments provide users with different input and output devices. Even in the same environment, users typically have different needs and preferences with respect to such interaction devices. This wanted flexibility creates a demand for user interfaces that are adaptable to changing environments and user preferences by spanning the plethora of contemporary I/O modalities and devices. However, the engineering workload involved in making applications fully adaptable in this sense is very high, and, as a result, applications nowadays often support only a limited number of devices.

Consider the following use-case: A group of experts wants to perform a deep brain stimulation on a patient. This kind of brain surgery requires various medical datasets to be explored in advance as well as being monitored during the process. Further assume that a 3D visualization application that is able to handle those datasets is available. In the preparation stage, the experts review the dataset at the office. The interaction setup involves well-known devices such as mouse and keyboard, and the dataset is displayed on a monitor. Later, the experts meet in the conference room and review the surgery roadmap on a large display wall while standing in front of it. The interaction is done via gestures, speech, and personal mobile devices. During the surgery, the doctor relies on a big touchscreen to monitor the process and change parameters on the fly via touch-based or Leap-Motion-captured gestures. The latter is a benefit in aseptic environments where touching should be avoided or is not possible.

The scenario above outlines three different environments and workflows based on the same application but with different interaction requirements. One way to tackle this issue is to add support for various devices to the

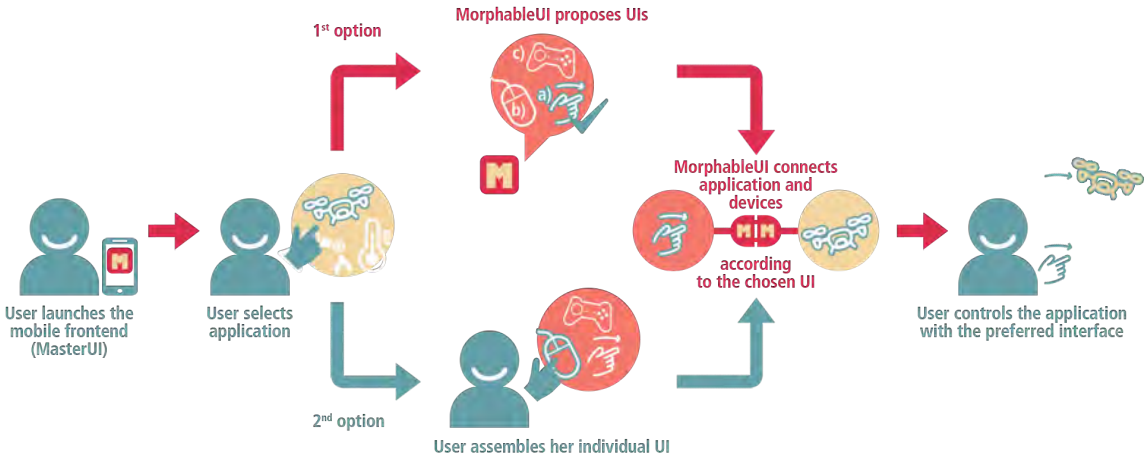

Figure 1: MorphableUI guides users through the UI configuration process by proposing UIs learned from previous decisions and assists with manual UI configuration. During runtime, the system dynamically connects the associated application, devices and interaction data streams over a network.

application itself and extend that range when needed. When new types of devices are introduced within the field of an application, the latter must be modified in order to accommodate these new interaction possibilities. In contrast to this approach, we propose a model that allows for dynamically connecting applications and devices in a way that makes user interfaces adaptable to changing environments and user preferences. This method helps developers avoid having to adapt their application to various types of devices and enhances rapid prototyping possibilities. Users are given the freedom to control applications in the way that best suits their needs by making use of any device that is available in their environment.

In addition, we present an implementation of the proposed model. The implementation provides a uniform, easy-to-use API for arbitrary devices and applications and exposes a service that allows users and developers to configure and dynamically change their interfaces. We demonstrate the service's capabilities by a mobile application suitable for rapid and easy reconfiguration of user environments.

2 RELATED WORK

To solve the outlined issues, two major tasks need to be addressed. First, a way to abstract from actual devices, manufacturers, and even modalities to cover all available interaction possibilities is required. Future devices should also be captured by the developed abstraction. Second, one needs a way to dynamically set up and modify interfaces by taking into account the environment and user preferences. A number of different approaches, especially to the first task, have been presented in the past. Our work builds on these achievements and establishes an interaction environment that includes device and application classification, UI generation, and I/O data streaming.

2.1 I/O Abstraction

I/O hardware abstraction layers hide the details of the underlying hardware. They are often used in VR/AR/MR environments where one has to deal with various kinds of often highly specialized I/O equipment such as motion tracking or 6 degree-of-freedom (6 DOF) devices. One example of the latter is the Control Action Table (*CAT*) [13]. It combines both 3D and 2D interaction techniques and extends the UI design space. Another option is to combine the CAT with other devices such as HMDs or the sensors of a smartphone. In the case of 6 DOF controls, one should also pay attention to the human ability to coordinate movements [27]. One well-established approach to wiring input devices and applications that is used in a number of VR environments is the *VRPN* [25] system. Apart from introducing abstract classes such as joysticks, VRPN streams the device input data over the network, allowing, for example, distributed applications and scenarios.

As opposed to the broad hierarchy of VRPN, the abstraction layer of *DEVAL* [22] establishes a deep hierarchy that puts more emphasis on the exchangeability of devices. Both approaches are based on abstracting from concrete devices and introducing hardware or device classes. These approaches have limitations when it comes to multimodal exchangeability of interaction techniques. In contrast, *DEMIS* [15] relies on events. It also accounts for multi-level composite events and is placed between the operating system and an application.

Frameworks such as emphMidas [24] focus on multi-touch and further enhance the I/O abstraction. In terms of distributed output and cross-device interaction, *PolyChrome* [1] can be used to seamlessly connect multiple devices for collaborative, web-based visualizations. Systems that want to support multi-device interaction can benefit from the Device Indepentent Architecture [5]. Similar to our system, the authors propose to decouple devices from applications in order to adapt to the given environment. The work around the *Virtual Interactive*

Namespace (VINS) [26] provides a distributed memory space that permits the reuse and exchange of various interactive techniques, which also enhances the development of reusable interaction components. Another library that supports designers and researchers with regard to the development of novel interaction techniques is *Squidy* [18]. It unifies various device drivers, frameworks, and tracking toolkits and exposes a visual design environment to increase the overall ease of use.

In order to establish our interaction event types, we have chosen the contributions of Card et al. [4] and Mackinlay et al. [19] as our starting point. Their work in this area focuses on the design space for input devices. One key idea is to split a device into a set of atomic capabilities, e.g., a mouse wheel and mouse buttons and the movement sensor in the case of a mouse. These capabilities are captured by a taxonomy consisting of classes such as 1D-3D motion or rotation. Hence, mouse movement would be classified as 2D motion on the x- and y-axes. In contrast to that rather mechanical point of view, the work of Javob et al. [14] focuses more on the perceptual structures of interaction tasks.

There is also work on other taxonomies dealing with less traditional I/O techniques. For example, one might consider gesture recognition. Here, *Proton* [17] proposes a regular expression-based classification of touch input. The work of Nebelig et al. [21] evaluates user-defined Kinect gestures and speech commands for the interaction with a wall-projected web browser. For mobile devices, user-defined gestures are composed into a motion gesture taxonomy of Ruiz et al. [23]. Widgets are another important approach to generating user input. One example of widget classification with a focus on 3D tasks can be found in the work of Dachselt and Hübner [6].

We do not merely aim to classify devices but also to establish exchangeable connections to applications. For that reason, one has to deal with the application side of the interaction pipeline as well. Applications can have a variety of interaction tasks to be performed. The following six tasks, mainly suited for 2D, were proposed by Foley [9]: select, position, orient, path, quantify, and text. For 3D interaction, five basic interaction tasks were introduced and refined by Bowman [2, 3]: navigation, selection, manipulation, system control, and symbolic input.

2.2 UI Adaptation

Our scenario involves varying environments, tasks, and user preferences. The task of adapting a user interface to such constraints can be tackled in multiple ways. For instance, users can assign the output of directly connected devices to application functions with the visual editor *ICON* [7]. To a limited degree, input transformation is possible as well, but requires a skilled user to perform the configuration. *SUPPLE* [10] formalizes the UI configuration problem and focuses on the graphical aspect of automated UI generation. Its successor, *SUPPLE++* [11], adds support for physically disabled users by including user models. Kim et. al [16] introduce interaction layering and abstraction based on device capabilities to overcome the issue with different interaction environments. UI adaptation also plays an important role in the automotive industry, driven especially by the amount of external infotainment possibilities as discussed in [20].

3 MORPHABLEUI

We start off with outlining an abstract model for UIs and user interaction in general. We enhance the construct by enabling dynamic transformation of interaction events via the split, merge, and cast operators. Based on that model, we describe our novel approach of generating admissible UIs using a hypergraph-based algorithm. We conclude by presenting an implementation of these concepts and offering a user- and developer-friendly way to establish dynamic connections between arbitrary applications and interaction devices.

3.1 Model

3.1.1 Events, Capabilities, and Requirements

Different interaction devices can be used to perform the same user task. In our medical example, the visualization application allows users to move, i.e., pan, the dataset, which can be achieved by moving the mouse and also by swiping over a smartphone touchscreen. From a more abstract point of view, what the mouse and the smartphone provide is the ability to generate *interaction events* of a specific type that are sent to and interpreted by the application. Both devices generate the same *type* of interaction event, precisely, a two-dimensional motion event. Because the mouse and the smartphone provide the means of generating interaction events of the same type, they can be exchanged with respect to the task to be performed.

The device characteristics or *capabilities* describe the type of generated or processed interaction events. Input capabilities generate interaction events triggered by the user, whereas output capabilities process interaction events received from the application such as video output. Note that some devices, e.g., smartphones, have input as well as output capabilities.

The capability classification includes low-level types of interaction events, e.g., `Firing Event` or `2D Position`, as well as higher-level types such as `3D Manipulation`. An example classification illustrating both input and output capabilities of a smartphone is

Capability	Interaction event type
Pinch gesture	`Zoom Event`
Gyroscope	`3D Rotation`
Slider widget	`1D Motion`
Touchscreen position	`2D Position`
Touchscreen display	`Video`
Voice recognition	`Text`

Table 1: Excerpt of the capabilities of a smartphone.

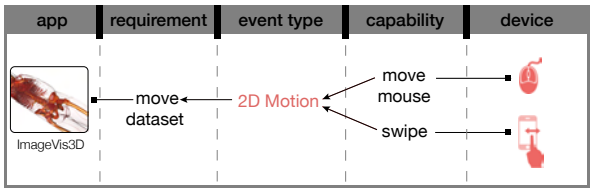

Figure 2: By introducing requirements and capabilities, the basic model decouples applications from devices. Both sides are associated with the corresponding interaction event types. In this example, moving the mouse can be used to pan the dataset. Since the swipe gesture is associated with the same interaction event type, these two interaction techniques can be exchanged.

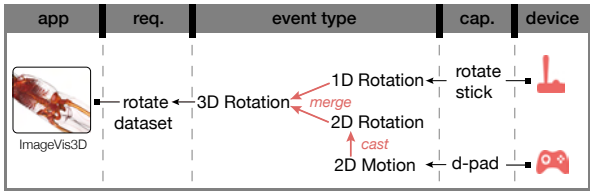

Figure 3: Adding the novel split, merge, and cast operators allows the transformation of generated interaction events and combination of different devices to perform a task. Hence, rotating the dataset can be achieved by a combination of the directional pad (d-pad) of a gamepad and the stick rotation of a joystick.

given in Table 1. A `Zoom Event` can also be regarded as an event of type `1D Motion`. However, the pinch gesture capability of a smartphone is tailored to accomplish the very specific task of zooming in or out, which is why we associate it with that higher-level event type. As explained in the next section, such a fuzzy specification is not an issue since event types can be transformed into other types if certain criteria are met.

Analogous to capabilities of devices, applications have *requirements* for specific user tasks. Each one is tied to an event type. Viewing devices and applications in this way allows the exchange of devices if their capabilities cover the requirements of the application as shown in Figure 2. We call an admissible connection between a capability and a requirement a *wiring*. Since an application usually consists of multiple requirements, the complete user interface can be formally defined as a set of wirings.

Another aspect to be mentioned regarding the user experience is the I/O data sensitivity and range. These additional properties can be provided on both the application and device sides to enable automated unification inside the framework that internally uses a unit hypercube, which often results in improved interaction compared to raw input that might differ significantly across devices.

3.1.2 Dynamic Event Modification

In addition to panning, we now want to rotate our dataset, which requires a `3D Rotation` event. One might use a gyroscope in a smartphone to generate the necessary input. However, one also could combine, i.e., merge, different lower-dimensional input capabilities. Hence, the definition of a wiring must be extended to also include connections between one requirement and multiple capabilities. The latter have to generate interaction events that can be transformed to yield a single event matching the application requirement.

We suggest three types of operations on interaction events that allow such transformations: cast, split, and merge. The *cast operator* transforms the semantics of an interaction event if possible. In the case of a `Zoom Event`, one is able to cast it to `1D Motion`. The *split operator* splits one event into multiple, in most cases lower-dimensional, events. Hence, a 3D gyro sensor can be used for panning a picture in 2D by splitting the underlying `3D Motion` capability into `1D Motion` and `2D Motion`. The *merge operator* is its inverse and merges multiple interaction events into one. An example transformation pipeline for the `3D Rotation` requirement is depicted in Figure 3.

3.2 Graph

Being able to transform device I/O according to the three introduced operators clearly enhances the UI design space. This section tackles the issue of computing such wirings. First, a number of different representations for the interaction event types and their interconnection are discussed. Second, we present an iterative algorithm that proposes admissible wirings for a given requirement.

Taking interaction events as input, the operators execute a certain transfer function and return the corresponding interaction event (or events, in the case of a split operation) as a result. From the point of view of an interaction event, operators are perceived as incoming, if that event is the result, or outgoing, if that event is the input.

One way to project this model onto a data structure is to use trees with the event types as vertices and operators as edges. Another approach is to use context-free grammars with event types as symbols and operators as production trees. Intuitively, both approaches share the same computational logic: one starts at the type of the application requirement and examines all possible decompositions. At this point, two major drawbacks can already be observed. First, both representations contain duplicates of event types since each one can have multiple outgoing and incoming operators. As a result, the representation is difficult to maintain since one has to care about all production rules or trees if a type or operator is added or removed. Second, the need to account for all possible decompositions leads to an exponential runtime of the algorithm, which is a problem in cases with a mentionable number of devices and operators.

We design a hypergraph with event types as vertices and operators as hyperedges. Informally, this generalized graph form is needed because the split and merge operators represent a 1-to-N connection and involve more than two vertices. Hyperedges allow N-to-M connections and are a feasible data structure for our task. One additional

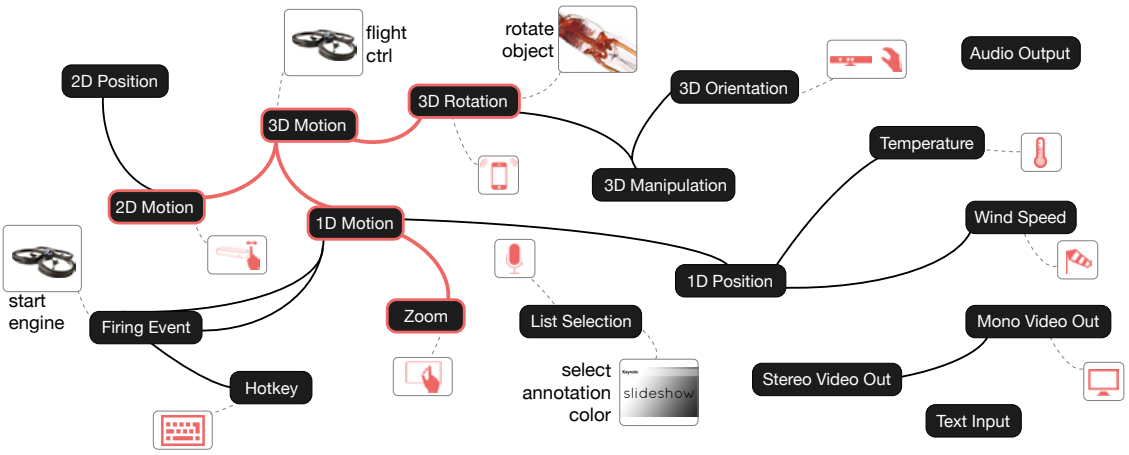

Figure 4: A subset of the established hypergraph. Event types are captured as vertices whereas hyperedges represent the operators. To maintain clarity, a number of edges and vertices are omitted. The proposed iterative algorithm uses device tokens that traverse the hypergraph until the requirement vertex is reached. The result is a subgraph representing the wiring between device capabilities and a requirement. One example of a wiring is highlighted.

concept based on the work in [12] is utilized, the so-called *backward* and *forward arcs*. Both are special types of directed hyperedges, either 1-to-N (forward) or N-to-1 (backward). Hence, a forward arc precisely expresses the layout of the split operator, and a backward arc represents a merge operation. Thus, the task of computing admissible UIs can be completed by computing a sub-graph connecting the vertex associated with the application requirement to one or more vertices representing device capabilities as depicted in Figure 4.

Note that the length of a path between two vertices directly corresponds to the resulting transfer function applied on the I/O data. Thus, a large distance, i.e., a large number of required operators, corresponds to a less direct mapping. The distance aligns with one's intuition since using three `1D Motion` events to accomplish a `3D Motion` task is less direct than using a single `3D Motion` event. Based on that property, an iterative algorithm that presents possible wirings ordered by ascending distance between requirement and device capabilities is beneficial. Hence, a user would first receive a number of adjacent solutions and demand further solutions if needed.

Our key idea is to use tokens commonly known from Petri Nets. Each token represents a device and is initially placed at the corresponding vertex. For example, a token for the swipe gesture will start in the `2D Motion` vertex. Tokens can be moved over edges to adjacent vertices if the traversal requirements outlined in Table 2 are met.

Possible traversals are executed sequentially, ordered by their cost. Similar to Dijkstra's shortest path algorithm, the cheapest traversal is estimated by computing the distance we already traveled as formalized in Table 2. The approach is summarized in Algorithm 1. Tokens arriving at the vertex corresponding to the requirement carry a valid wiring since the token history stores the sequence of executed traversals. In this way, solutions are presented to the user step by step. Again, later proposals indicate a less direct transfer function is needed to transform the I/O data required by the application. To sum up, the main advantages of the presented approach include the iterative solution generation, the in-place search with a data structure without duplicated event types, and the amortized polynomial time and space of the algorithm.

Finally, we establish a way to validate external, e.g., handcrafted, assignments of device capabilities for a requirement. For this purpose, the same algorithm can be employed. The corresponding device capability tokens are inserted, and the algorithm executed until a solution is found, no further traversals can be executed, or a step limit is reached. If the algorithm finds a solution, the demanded mapping is admissible, and the wiring including the required operator chain is returned. Note that this procedure allows for black box proposals consisting of the endpoints—requirement and capabilities—without the need to provide the complete operator sequence.

3.3 Implementation of MorphableUI

We addressed the distributed multi-device design issue by developing an interaction model and a corresponding algorithm that computes admissible wirings for given application requirements. In the following, we demonstrate our implementation of MorphableUI to prove the established concepts. We introduce three main components: *Gates* that serve as entry points for application and device developers. A *server* that maintains the interaction topology and provides external services, and the *MasterUI*, a mobile frontend that builds on such a service and allows users to select and configure UIs.

3.3.1 Gates

A MorphableUI gate is a C-library that allows users to plug applications and devices into the interaction topology spanned by our framework. The gate component

Algorithm 1 Iterative computation of admissible wirings. The algorithm returns tokens that arrive at the requirement vertex. The corresponding sequence of operators can be extracted from $hist(t)$. Traversal rules and definitions can be found in Table 2. The algorithm sequentially executes the next cheapest traversal. After an execution and the resulting token movement, traversals of the affected vertices have to be updated.

Input:
application requirement r
device capabilities $c_1, ..., c_n$

Initialization:
for all c_i **do**
  insert new t into corresponding v
end for
create empty $TraversalList$
mark requirement vertex as v_r
for all e **do**
  compute cheapest $trav_{V \to W}$ on e (see R)
  add $trav_{V \to W}$ to $TraversalList$
end for

Iteration:
repeat
  exec. cheapest $trav_{V \to W}$ in $TraversalList$ (see R)
  for all $v \in V, W$ **do**
    for all e, e incident to v **do**
      remove $trav_{V \to W}$ associated with e from $TraversalList$
      compute cheapest $trav'_{V \to W}$ on e
      add $trav'_{V \to W}$ to $TraversalList$
    end for
  end for
until new t arrives in v_r
return t

comes with a simple API that allows users to send and receive interaction events as shown in the Listing 1. An application that demands `3D Rotation` only has to call such a receive function or register a callback to obtain incoming events. On the other side of the pipeline, e.g., the gyro sensor of a smartphone constantly pushes its captured rotation events via a send function.

Gates gather the information about application requirements and device capabilities from a developer-provided json file. Our implementation in plain C allows the use of the same gate implementation on desktops, mobile (iOS, Android), and other platforms such as Raspberry Pi. To faciliate the integration into modern software, a set of wrappers in other languages is available. The wrappers expose the same API and are available in languages such as Python, JavaScript, Java, C++, Objective-C, and Go. In terms of interaction event streaming performance, we point out that the gate-to-gate streaming is executed directly, i.e., without routing data over the server component presented in the next section.

Traversal rules R

Definitions and notation
- v, w : vertices, e : edge, V, W : sets of vertices
- $trav_{V \to W}$ traversal on edge connecting V and W
- traversal types: $split_{v \to W}$, $cast_{v \to w}$, $merge_{V \to w}$
- t a token with history $hist(t)$ of executed traversals
- t associated with one or more (after merging) device capabilities c
- $cost(t) = |hist(t)|$

Candidate tokens for a traversal $trav_{V \to W}$
- all t in $v \in V$ not yet visited any $w \in W$
- merge constraint: one t from each $v \in V$ required and selected tokens must not be associated with the same c (prevents merging a capability with itself)
- cheapest $trav_{V \to W}$ (not necessarily unique) defined as: $min(\sum cost(t) \mid t$ participating in $trav_{V \to W})$

Executing a traversal $trav_{V \to W}$
- $split_{v \to W}$: $\forall w \in W$: insert duplicate t_d of t_{src} in w
- $cast_{v \to w}$: insert duplicate t_d of t_{src} in w
- $merge_{V \to w}$: insert new t_n in w, $\forall t_{src}$: add $hist(t_{src})$ to $hist(t_n)$
- $\forall t (t_d$ or $t_n)$ add $trav_{V \to W}$ to $hist(t)$
- $\forall t_{src}$: if visited all adjacent v and $\nexists$ outgoing merge edge: $delete(t_{src})$
- *note:* the second condition is needed since a potential merge candidate might arrive later

Table 2: Our Algorithm 1 operates on tokens. They initially represent device capabilities and are moved in a hypergraph on edges standing for operators between vertices representing the interaction event types.

3.3.2 Server

While gates are spread over the network, their operability depends on a central server that is responsible for the environment coordination. Precisely, the server contains a memory-efficient C++ implementation of Algorithm 1, maintains user sessions, and keeps track of available gates. The server behaves as a broker between the gates and the users. It exposes necessary information about available applications and devices gathered from the gates to the users and configures gate streaming pipelines according to the user-definded interfaces. Apart from this UI configuration functionality, further explained in the next section, the server exposes a set of external services available for developers. For instance,

```
// initialization
Gate gate("ImageVis3D.json");
gate.start();
// runtime
Event evt = gate.receiveEvent("Pan␣dataset");
// something inside the target software
translate(glm::vec3(evt.x, evt.y, 0), data);
```

Listing 1: Example integration in C++. The gate is initialized with a json file containing a list of requirements or capabilities. During runtime, the target software polls or sends interaction events.

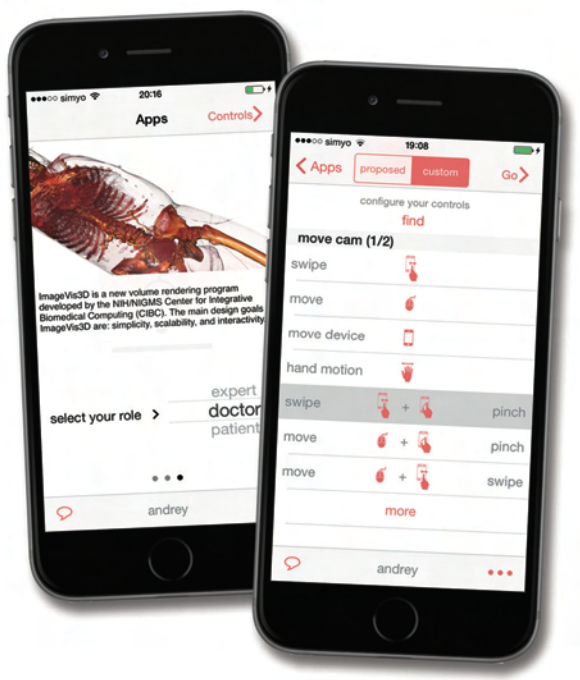

Figure 5: The MasterUI is a mobile frontend for our system that allows to select and customize UIs. The application selection screen already prototypes the role feature addressed in future work. The right image displays possible assignments for a given requirement.

REST interfaces are provided for both UI generation and interaction environment monitoring.

Assume that the server received a UI configured by a user via our frontend. According to our model, the UI consists of one wiring for each application requirement, while each wiring represents an I/O processing pipeline with a sequence of operators. This information is sent to all participating gates that then dynamically set up the necessary gate-to-gate streaming connections. The operator chain is always placed on the receiving side since data might be incomplete prior to that point. Hence, in the case of an input requirement, the operator chain resides in the application gate and vice versa.

3.3.3 UI Configuration App

One of the services provided by the server is to allow configuration and launch of user-defined interfaces. To deliver a user- and developer-friendly contribution, MorphableUI comes with a default mobile frontend, the *MasterUI*, depicted in Figure 5. This app guides users through the UI generation pipeline and allows on-the-fly customization during runtime, which is beneficial for, e.g., rapid prototyping tasks. This component is designed as a personal assistant that behaves according to the bring-your-own-device paradigm. Hence, everyone is able to use their private smartphone to create and apply desired UIs.

An example configuration process is depicted in Figure 5. First, the user has to choose the application he or she wants to control. In a second step, the UI is assembled. The straightforward way is to configure each application requirement manually. Hereby, wirings are requested from the hypergraph algorithm in an iterative way until the user sees a satisfying device assignment.

Remember that the proposal ordering corresponds to the length of the involved operator chain and thus reflects the number of needed event transformations between the capabilites and the application requirement. Instead of manually configuring each wiring, users are also able to request automated proposals for complete UIs and choose between or reconfigure them.

The ability to obtain automated UI proposals is based on stored information about previous usage, i.e., on what the user already designed for this or similar applications. Similarity, then, is defined by the percentage of equal requirement types. Intuitively, there is a chance that requirements associated with the same type of interaction event behave analogously. Thus, we implicitly port UIs across applications by generating such proposals. This approach also accounts for interfaces designed by others since it turned out to be a good starting point compared to blank initialization. One of our future goals is to enhance this automated proposal and learning ability to anticipate users' needs and minimize the UI configuration efforts.

4 INTEGRATING MORPHABLEUI

To demonstrate how the theoretical model and its realization behave in the real world, we have added support for a set of devices and applications and evaluated the integration efforts of our framework in external projects. A few lines of code suffice to enable full access to the MorphableUI interaction features. The Listing 1 provides an overview of the necessary steps including setup and runtime. Note that the appications does not need to know the available devices at all nor to restart or recompile if a new device becomes available.

4.1 Sample Devices and Applications

4.1.1 Device Support

Our prototype covers conventional desktop environments, joysticks, gamepads, Kinects, Leap Motions, monitors and mobile displays for mono video output, and head-mounted displays for stereo video output. The underlying video streaming relies on a JPEG-encoded frame transmission, i.e., each frame is packed into an interaction event and transported to the output device. Furthermore, MorphableUI supports iOS and Android smartphones and tablets. These devices expose capabilities such as swipe and pinch gestures, gyroscope and accelerometer sensors, speech input, and widgets such as sliders and virtual joysticks. To demonstrate the range of possible use-cases, smart home sensors for temperature, wind speed, and air pressure were also integrated.

4.1.2 Application Example: ImageVis3D

The volume rendering software *ImageVis3D* [8] scales to very large biomedical datasets. It already accounts

Requirement	Interaction event type
Rotate dataset	`3D Rotation`
Pan dataset	`2D Motion`
Resize dataset	`Zoom`
Toggle between 1D-TF and Iso	`Toggle`
Rotate clipping plane	`3D Rotation`
Set smoothstep function for TF	`2D Position`

Table 3: A set of ImageVis3D requirements. The one-dimensional transfer function is denoted by 1D-TF and isosurface rendering by Iso.

for heterogeneous environments by being able to run on everything from mobile devices to high-end graphics workstations. To allow such flexibility on the I/O side, we have connected the software to our framework by capturing a set of basic functionalities as shown in Table 3.

From the captured interaction tasks, manipulating the transfer function was of increased interest for the developers of ImageVis3D. One surprisingly intuitive interface was moving the hand over the leap motion and changing the inflection point of the smoothstep function by lifting or lowering the hand. Alternatively, rotating one's hand was also rated as intuitive for changing the slope of the smoothstep function.

4.2 Developer Survey

To gain feedback on our approach, we asked nine application developers (all male) interested in MorphableUI to fill out a questionnaire after the first successful integration of our system into their target software, including both academic and industrial collaborations to cover a wide sample range. The questions included both subjective topics (difficulty of integrating the software, required support) and objective topics (needed development time for integrating the libraries into their software including glue code, time for defining the requirements/capabilities). The subjective questions were answered via a 7-point Likert scale, with 1 meaning very easy/none and 7 indicating very hard/always. The objective questions used minutes as a scale, since they targeted development efforts. Complementary questions about the programming experience and age of the participants were designed to provide hints about possible side effects for inexperienced users.

Our results show that the integration of MorphableUI could be done in less than one hour in all cases, but most participants required less than 30 minutes. The design time for capabilities/requirements fluctuated more, linearly depending on the amount and complexity of the targeted interaction. For the question regarding the difficulty of integrating MorphableUI into existing software, we see a mean value of 2.0 with a standard deviation (SD) of $\sigma = 0.71$. Hence, the developers found the process easy and encountered no major difficulties. This result is further strengthened with the outcome for the question concerning the required support for integrating the software: it shows a mean value of 2.0 with a SD of $\sigma = 1.0$, meaning the developers only needed little support. We cannot conclude that the developer experience had a direct impact on the integration or requirement/-capabilities design time. Hence, the complexity of the target software seems to be a more prominent factor.

	Difficulty of Integration	Needed Help	Time for Integration	Time for Requirements	Exp.
P1	2	2	20 min	30 min	3 yr.
P2	3	3	40 min	30 min	1 yr.
P3	1	1	5 min	5 min	7 yr.
P4	2	1	15 min	10 min	2 yr.
P5	3	2	30 min	5 min	4 yr.
P6	2	4	40 min	20 min	6 yr.
P7	2	2	20 min	5 min	4 yr.
P8	2	2	30 min	5 min	4 yr.
P9	1	1	10 min	10 min	3 yr.
mean	2.0	2.0	$23.\overline{3}$ min	$13.\overline{3}$ min	$3.\overline{7}$ yr.
sd	1	1	12.5 min	10.61 min	1.86 yr.

Table 4: P1 to P7 were integrations of MorphableUI into existing applications, P8 and P9 added new interaction devices to our system. In detail, P1-P4 were interactive 3D visualization tools, P5 an interactive physical simulation, P6 a connection to the FMI standard, P7 the integration into OgreVR, P8 connected the Community Core Vision, and P9 added support for the Leap Motion.

5 BENEFITS AND LIMITATIONS

A common question is whether MorphableUI is beneficial for a particular application. Despite that results from preliminary user studies indicate high acceptance, this topic requires further discussion. On the one hand, mapping a complex software such as Photoshop with hundreds of different tasks does not seem feasible with the proposed technique. First, the UI generation process will generally consume more time, and reassigning certain controls will often result in scrolling through large lists compared to, for example, browsing well-structured settings menus. Second, applications that are tightly coupled to a specific environment or device setup often cannot take advantage of the offered I/O exchangeability. On the other hand, applications often have a set of basic functionalities that are accessible in different environments and can be controlled in multiple ways depending on the use-case. In the Photoshop example, users still might want to control panning and zooming via a tablet with the non-dominant hand.

Hence, we recommend combining MorphableUI with traditional hard-wired interfaces. That is, we suggest using our system to cover only a small subset of requirements where device exchangeability is expected to be important. In the case of our ImageVis3D scenario, we recommend users stick to a traditional UI for tasks such as opening a file and rely on MorphableUI for object manipulation or the streaming of the video output. During development, prototyping tasks benefit massively from the effortless integration, as the system allows to try out a plethora of I/O devices out of the box.

6 SUMMARY AND FUTURE WORK

The paper established a requirement- and capability-based model for distributed, multimodal interaction. We

introduced a classification building upon interaction event types and expressed their relations by three operators: split, merge, and cast. This formalization allows higher-order I/O data transformation and enhances the exchangeability compared to other approaches. The problem of computing admissible UIs by generating wirings for each requirement is tackled by a token-based, iterative algorithm working on a hypergraph. The vertices correspond to the interaction event types, and edges represent the operators. The generated wiring proposals are ordered by the length of the resulting operator chain. Hence, the order expresses the amount of performed I/O data transformation.

The implementation consists of three main components. The gate is a library that serves as an entry point for applications and devices. During runtime, I/O data is streamed directly between the gates over the network. The server monitors the devices and applications and exposes services such as the UI configuration. MasterUI is a mobile frontend built upon such a service. It allows users to dynamically configure their interfaces and receive notifications about changes in the interaction topology. Finally, support was added for a set of sample devices and applications to showcase the effortless integration of our system which particularly enhances the area of rapid prototyping of multimodal interfaces.

There are a number of different issues to be targeted in the future. One goal is to increase the number of supported devices such as the Microsoft HoloLens, which is mainly an engineering task. At this point, it might be of interest to extend the existing model by hardware characteristics of the input devices. Considering widgets as I/O capabilities, a sophisticated arrangement would be beneficial. For now, the system does not have any hierarchical concepts and places the widgets, such as virtual joysticks, at predefined positions. To fully support that kind of interaction, the UI configuration pipeline has to be extended to deal with layout settings.

Another idea is to arrange requirements into roles on the application side as already prototyped in Figure 5. In our brain stimulation example, one would differ between the doctor and patient. The latter role has limited interaction requirements allowing, for example, only to panning and rotating the dataset. Another user-related feature that will be included in future work is security and authentication. One use-case is to prevent unauthorized I/O device access to a set of private devices limited to one particular user.

In this paper, we mainly focused on I/O exchangeability and enabled a novel approach for multimodal, distributed interaction and rapid prototyping. One of our next goals is to measure and enhance user experience in MorphableUI by conducting usability studies. Clearly, our framework can also be used to provide interfaces that are not very user-friendly. For this reason, we plan to combine the UI generation with a sophisticated online learning algorithm to further improve the UIs being proposed automatically based on prior knowledge. In addition, multi-user setups will be focused more since the framework does not impose any limitations on the number of users for one application.

The set of interaction events that we used for the capability and requirement classification does not pretend to be complete. Further refinement is needed depending on the application area and the use-case. To tackle this issue, we are developing a MorphableUI tool chain. The chain will include a GUI-based hypergraph modification tool that allows the addition of new types of events without the need to touch or (re-)compile code. Also, a web application that facilitates monitoring the interaction environment and assists developers and administrators would further enhance the framework. The main issue, therefore, is to find a compact and meaningful graphical representation of the environment including device locations, active user interfaces and the corresponding wirings between devices and applications.

REFERENCES

[1] S. K. Badam and N. Elmqvist. “PolyChrome: A Cross-Device Framework for Collaborative Web Visualization”. In: *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*. ITS ’14. Dresden, Germany: ACM, 2014, pp. 109–118. ISBN: 978-1-4503-2587-5.

[2] D. A. Bowman et al. *3D User Interfaces: Theory and Practice*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 0201758679.

[3] D. A. Bowman et al. *Interaction Techniques For Common Tasks In Immersive Virtual Environments - Design, Evaluation, And Application*. 1999.

[4] S. K. Card, J. D. Mackinlay, and G. G. Robertson. “The design space of input devices”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’90. Seattle, Washington, United States: ACM, 1990, pp. 117–124. ISBN: 0-201-50932-6.

[5] J. Chmielewski and K. Walczak. “Application Architectures for Smart Multi-device Applications”. In: *Proceedings of the Workshop on Multi-device App Middleware*. Multi-Device ’12. Montreal, Quebec, Canada: ACM, 2012, 5:1–5:5. ISBN: 978-1-4503-1617-0.

[6] R. Dachselt and A. Hübner. “A Survey and Taxonomy of 3D Menu Techniques”. In: *Proceedings of the 12th Eurographics Conference on Virtual Environments*. EGVE’06. Lisbon, Portugal: Eurographics Association, 2006, pp. 89–99. ISBN: 3-905673-33-9.

[7] P. Dragicevic and J.-D. Fekete. “Input Device Selection and Interaction Configuration with ICON”. In: *Proceedings of the HCI01 Conference on People and Computers XV*. Springer, 2001, pp. 543–558.

[8] T. Fogal and J. Krüger. "Tuvok, an Architecture for Large Scale Volume Rendering". In: *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*. 2010.

[9] J. D. Foley, V. L. Wallace, and P. Chan. "The human factors of computer graphics interaction techniques". In: *IEEE Computer Graphics and Applications* 4.11 (1984), pp. 13–48. ISSN: 0272-1716.

[10] K. Gajos and D. S. Weld. "SUPPLE: Automatically Generating User Interfaces". In: *Proceedings of the 9th International Conference on Intelligent User Interfaces*. IUI '04. Funchal, Madeira, Portugal: ACM, 2004, pp. 93–100. ISBN: 1-58113-815-6.

[11] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld. "Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities". In: *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. UIST '07. Newport, Rhode Island, USA: ACM, 2007, pp. 231–240. ISBN: 978-1-59593-679-0.

[12] G. Gallo et al. "Directed hypergraphs and applications". In: *Discrete Appl. Math.* 42.2-3 (Apr. 1993), pp. 177–201. ISSN: 0166-218X.

[13] M. Hachet, P. Guitton, and P. Reuter. "The CAT for Efficient 2D and 3D Interaction As an Alternative to Mouse Adaptations". In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '03. Osaka, Japan: ACM, 2003, pp. 225–112. ISBN: 1-58113-569-6.

[14] R. J. K. Jacob et al. "Integrality and Separability of Input Devices". In: *ACM Trans. Comput.-Hum. Interact.* 1.1 (Mar. 1994), pp. 3–26. ISSN: 1073-0516.

[15] H. Jiang, G. D. Kessler, and J. Nonnemaker. "DEMIS: A Dynamic Event Model for Interactive Systems". In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '02. Hong Kong, China: ACM, 2002, pp. 97–104. ISBN: 1-58113-530-0.

[16] S. J. Kim et al. "Adaptive interactions in shared virtual environments for heterogeneous devices". In: *Computer Animation and Virtual Worlds* 21.5 (2010), pp. 531–543. ISSN: 1546-427X.

[17] K. Kin et al. "Proton: Multitouch Gestures As Regular Expressions". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: ACM, 2012, pp. 2885–2894. ISBN: 978-1-4503-1015-4.

[18] W. A. König, R. Rädle, and H. Reiterer. "Interactive Design of Multimodal User Interfaces - Reducing technical and visual complexity". In: *Journal on Multimodal User Interfaces* 3.3 (2010), pp. 197–213.

[19] J. Mackinlay, S. K. Card, and G. G. Robertson. "A semantic analysis of the design space of input devices". In: *Hum.-Comput. Interact.* 5.2 (June 1990), pp. 145–190. ISSN: 0737-0024.

[20] G. de Melo et al. "Towards a Flexible UI Model for Automotive Human-machine Interaction". In: *Proceedings of the 1st International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. AutomotiveUI '09. Essen, Germany: ACM, 2009, pp. 47–50. ISBN: 978-1-60558-571-0.

[21] M. Nebeling et al. "Web on the Wall Reloaded: Implementation, Replication and Refinement of User-Defined Interaction Sets". In: *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*. ITS '14. Dresden, Germany: ACM, 2014, pp. 15–24. ISBN: 978-1-4503-2587-5.

[22] J. Ohlenburg, W. Broll, and I. Lindt. "DEVAL: a device abstraction layer for VR/AR". In: *Proceedings of the 4th international conference on Universal access in human computer interaction: coping with diversity*. UAHCI'07. Beijing, China: Springer-Verlag, 2007, pp. 497–506. ISBN: 978-3-540-73278-5.

[23] J. Ruiz, Y. Li, and E. Lank. "User-defined Motion Gestures for Mobile Interaction". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: ACM, 2011, pp. 197–206. ISBN: 978-1-4503-0228-9.

[24] C. Scholliers et al. "Midas: A Declarative Multitouch Interaction Framework". In: *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction*. TEI '11. Funchal, Portugal: ACM, 2011, pp. 49–56. ISBN: 978-1-4503-0478-8.

[25] R. M. Taylor II et al. "VRPN: a device-independent, network-transparent VR peripheral system". In: *Proceedings of the ACM symposium on Virtual reality software and technology*. VRST '01. Baniff, Alberta, Canada: ACM, 2001, pp. 55–61. ISBN: 1-58113-427-4.

[26] D. Valkov, A. Giesler, and K. H. Hinrichs. "VINS - Shared Memory Space for Definition of Interactive Techniques". In: *ACM Symposium on Virtual Reality Software and Technology (VRST 2012)*. ACM, 2012, pp. 145–153.

[27] S. Zhai and P. Milgram. "Quantifying Coordination in Multiple DOF Movement and Its Application to Evaluating 6 DOF Input Devices". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '98. Los Angeles, California, USA: ACM Press/Addison-Wesley Publishing Co., 1998, pp. 320–327. ISBN: 0-201-30987-4.