Advance Access publication on 3 July 2018 doi:10.1093/iwc/iwy014

I-Typed DMML: A Novel DSL for Direct Manipulation Interaction with Virtual Objects

JOONG-JAE LEE[1] AND JUNG-MIN PARK[2,*]

[1]*Center of Human-centered Interaction for Coexistence, 5, Hwarang-ro 14-gil, Seongbuk-gu, CHIC, Seoul, Republic of Korea*
[2]*Center for Intelligent Robotics Research, Robotics and Media Institute, Korea Institute of Science and Technology, 5, Hwarang-ro 14-gil, Seongbuk-gu, Seoul, Republic of Korea*
[*]*Corresponding author: pjm@kist.re.kr*

The demand for interactive applications in 3D virtual environments is increasing, but they are not easy to develop. In this article, we propose an I-Typed direct manipulation modeling language (DMML) for designing and developing interaction applications to directly manipulate virtual objects in 3D virtual environments. The I-Typed DMML consists of a domain-specific language to describe direct manipulation interaction easily and intuitively with a DMI engine that supports I-Type-based direct manipulation interaction. The feasibility of the proposed language for a variety of applications is confirmed by user study results, which demonstrate usability in terms of efficiency, effectiveness and satisfaction.

RESEARCH HIGHLIGHTS

- Presented an I-Typed direct manipulation modeling language (I-Typed DMML).
- Presented a domain-specific language (DSL) to describe direct manipulation interaction intuitively.
- Highlighted the need for directly manipulating virtual objects in 3D virtual environment.
- Presented a direct manipulation interaction (DMI) engine supporting direct manipulation.
- Showed the results that our method overcomes the limitation of existing DSL and development methods and enables both non-specialists and experts to easily develop interaction application.

Keywords: interaction type (I-Type); domain-specific language (DSL); direct manipulation modeling language (DMML); direct manipulation interaction (DMI) engine; virtual object; virtual reality

Editorial Board Member: Dr Rod McCall

Received 11 April 2017; Revised 20 April 2018; Editorial Decision 30 May 2018; Accepted 4 June 2018

1. INTRODUCTION

Interactive 3D applications are being used in various fields ranging from entertainment, scientific simulation and military training to education. Recently, many researches have applied model-driven engineering (MDE) in software developments and also in HCI application developments. For building task-modeling tools, notations such as ConcurTaskTrees (CTT) (Paternò, 2004) and languages such as HAMSTERS (Martinie *et al.*, 2011) have been developed. CTT, proposed by Paternò, easily translates user activities into everyday language and is suitable for designing interactive applications. Studies applying MDE process are not only underway to generate conceptual models for interactions but have also started on transformation, which automatically generates codes for interactions (Ammar and Mahfoudhi, 2013; Duval *et al.*, 2014; Jung *et al.*, 2015; Lenk *et al.*, 2012). However, unlike other software, applying MDE concepts in interactive 3D application development is not common (Duval *et al.*, 2014; Jung *et al.*,

2015). The reasons lie in the complexity of the development process of interactive 3D applications. Developing interactive 3D applications needs the use of diverse technologies. The development process involves experts from a variety of disciplines, for example, interaction designers, 3D content developers and programmers; besides, a wide variety of tools and terminology are used, resulting in implementation outcomes different from design intent (Lenk *et al.*, 2012).

Generally, an MDE development method transforms a platform-independent model into one or more platform-specific models and code models. In one transformation process, it converts a source model into a target model. One source model can be transformed into various target models using various methods. The final model remains the same in terms of functionalities but can be different in terms of usability. It is very crucial to determine the desired usability attributes during the transformation process, but most of the MDE methods show limitations in applying them (Ammar and Mahfoudhi, 2013; Schmidt, 2006). In other words, MDE methods are good at translating everyday tasks in abstraction, however, many possible models can be generated since the description of the interaction covers wide ranges of task activities. Thus, it is very difficult for programmers to develop exactly what designers intend. There have been two approaches to solve this kind of problem. The first reflects functions that users demand in the design process by the adaption of implemented rules or through customization (Ammar and Mahfoudhi, 2013; Anzalone *et al.*, 2015; Khaddam *et al.*, 2015; Popp *et al.*, 2012; Raneburger *et al.*, 2015). The other obtains code models of what users want through an iterative process (Blouin and Beaudoux, 2010; Raneburger, 2010). The modified transformation method through the latter solution minimizes errors in development resulting from miscommunications between developers and designers or users. However, earlier studies on transformation have focused on interactions using the WIMP interface such as menu GUI (Pleuss *et al.*, 2013; Raneburger *et al.*, 2011, 2012, 2015) and there is a need for transformation based on 3D interaction metaphors.

With the increase in demand for 3D interactions, many studies have been carried out on 3D collaborative virtual environments (CVEs) (Csisinko and Kaufmann, 2010; Duval *et al.*, 2014; Figueroa *et al.*, 2002; Jung *et al.*, 2015; Lenk *et al.*, 2012; Valkov *et al.*, 2012). These studies developed NUI-based interactions instead of GUI-based menus in immersive VR and AR environments. With the advancements in relevant input hardware such as perception neuron (Perception Neuron, 2017), and Leap Motion sensor (Leap motion, 2017), and advancements in relevant output hardware such as head-mounted displays (HMD) (Oculus Rift, 2017) and HoloLens (Hololens, 2018), there has been increased demand for bare-hand manipulations in 3D environments. Previously developed interactive 3D applications used ray casting, wand-based interaction, and physics-based contact or touch for manipulations. Mauney (Mauney *et al.*, 2010) stated that users prefer to use direct manipulation methods, which are more intuitive and easier than abstraction interactions such as gestures or symbolic interactions in 3D environments. Therefore, this study uses an interactive 3D metaphor that enables users to manipulate virtual objects using their bare hands.

Many interactive 3D applications are developed in code-centric or ad-hoc manner. As a result, reusing the application is difficult due to compatibility and customization issues. Several studies have been conducted that provides flexible abstraction for defining interaction metaphors or interfaces to address this issue. For example, Viargo is a comprehensive virtual reality library that provides an independent software layer from the application and associated libraries (Valkov *et al.*, 2012). In the Viargo library, input devices are abstracted to provide events to interaction metaphor components. This allows the interaction metaphor to seamlessly interoperate with other devices, even if certain devices are changed at runtime. Gismo, which is a gesture-based domain-specific language (DSL), separates interactions from gestures through visual programming language (VPL) and provides modeling language and editing tools for simulating and running interactive 3D applications (Meyers *et al.*, 2014; Romuald and Mens, 2015). However, Viargo supports such various interaction metaphors that relevant knowledge is required before using it. The gesture-based interaction model developed by using Gismo has a low degree of usability and compatibility when compared to direct manipulation metaphors because users need to have knowledge of the gestures beforehand. Furthermore, it produces tightly coupled source code since it maps interaction metaphors to states or events.

On the other hand, VITAL (Csisinko and Kaufmann, 2010) and InTml (Figueroa *et al.*, 2002) automatically convert documents written by using scripts with specific metaphors and abstractions into Java or C++ applicable formats. One problem with these two methods is that they do not support an iterative development process and desired functions can only be reflected during the design process and not during the development process. To address this problem, the scene structure and integration modeling language (SSIML) (Jung *et al.*, 2015; Lenk *et al.*, 2012; Vitzthum and Pleuß, 2005) has been developed, and it enables the iterative development process. This study follows the MDE process used by the SSIML model and this modeling language considers three different developers (software designer, 3D content developer and programmer). The software designer creates the SSIML model and uses two code generators. First one is used for generating code for 3D scene developed by a 3D content developer. Second one is used for generating program code and behavior code developed by a programmer. Through this approach, different developers can be participated in one development process, thereby facilitating iterative development. For describing 3D graphics scenes and objects in SSIML, X3D is adopted which is a standard for declaratively representing 3D computer graphics using XML, and focuses more on spatial and object information. It also

supports low-level interaction such as event handling used in script languages, but it is not easy to understand other developers except programmer.

In this article, we introduce a new method for developing Interaction Type (I-Type)-based direct manipulation applications for manipulating virtual objects in 3D environments. Following are the two contributions of this article.

First, we propose a novel I-Typed direct manipulation modeling language (DMML) focused on direct manipulation. The I-Typed DMML minimizes miscommunication among heterogeneous developer groups by DSL that transforms an interaction model designed by designers into a code model that programmers use to develop. Since traditional task models are human-centered abstract models and the models are implemented in low-levels such as event handlings, there is a possibility of having communication errors in understanding the intention of the interactions.

Second, we propose a direct manipulation interaction (DMI) engine that supports I-Type-based direct manipulation interactions. The main purpose of the DMI engine is supporting direct manipulation and it consists of a three-tiered architecture comprising an I-Type layer, an inference unit layer, and a perception unit layer. When transforming task-based interaction models, instead of converting rules or event handlings, descriptions from I-Types can be used to develop applications. This approach helps prevent the tight coupling of perception modules and inference algorithms as in traditional interaction models. In addition, it is possible to map units from each layer flexibly so that the exact intentions of the interaction designer, programmer and user can be described.

This article is organized as follows. Section 2 describes the I-Typed DMML, and Section 3 describes the DMI engine. Section 4 illustrates the feasibility of the proposed method through a case study and user study. Section 5 concludes the article.

2. I-TYPED DIRECT MANIPULATION MODELING LANGUAGE

We propose a new interaction modeling language called I-Typed DMML in this article. The proposed I-Typed DMML is a DSL that is used for direct manipulation of virtual objects in 3D environments. When describing various user interactions formally, mostly either the model-based development (MBD) method or the task-based development (TBD) method is used. An example of MBD is the CTT model that can easily describe everyday tasks; but, the description is so comprehensive that it is hard to manage many generated models. When implementing an application from a design model that results from CTT, there is a need for converting a model to code to fit the intention of the design. To satisfy this need, we propose the I-Typed DMML that can intuitively describe DMIs with virtual objects in 3D environments.

Figure 1. Stacking block in real world (CreativeZone/shutterstock.com, 2016).

Figure 1 represents direct manipulation in real space and can be described as follows in terms of interactions:

> A girl is playing with toy blocks of various shapes, sizes and colors. She is building a castle with three triangular prism roofs as well as a tall structure that looks like a tower. She is also making another tower by placing a red block carefully above the orange block.

When designing and implementing interactions such as those shown in Fig. 1, at the design level, interactions are described as human-friendly task-based interactions. On the other hand, at the programming level, a low-level description such as event handling is required. In the absence of such a description, during application development, there is a possibility of miscommunication between heterogeneous developers due to their level of expertise because of the use of algorithm-centered descriptions instead of interaction-centered descriptions. When implementing the interaction that is shown in Fig. 2 (the user's hand getting closer to the orange block), the program needs to calculate the distance between the hand and the orange block periodically. If the distance decreases, the program needs to detect that there is an approach interaction. This is the typical manner in which an interaction is described from an algorithm-centric perspective. However, in everyday life, people do not describe interactions in this manner as it is complicated for average people to understand. To reduce the gap between designing and implementing an interaction, it is necessary to intuitively describe the interaction in a human-centered manner. Therefore, we design a new concept of describing interactions by I-Type. The implementation of this concept can reduce the miscommunication between heterogeneous developers since it provides high-level descriptions even in the transformation step. I-Type is the base unit of describing direct manipulation and describes various interactions that humans can do with their bare hands. As an example, we can find the following I-Types in Fig. 2.

Figure 2. The scene shows that a right hand of the girl is approaching the orange block to grasp it. The dotted circle denotes the current position of the hand. Image at time t (Left) and image at time $t + 1$ (Right).

- A (right) hand is approaching the red block. ⇒ *Approach*
- Holding the red block with the hand. ⇒ *Grasp*
- Lifting up the red block. ⇒ *Lift up*
- Moving the red block above the orange block. ⇒ *Move*

An I-Type is defined by the quintuple in the I-Typed DMML as follows:

$$I\text{-}Type = (N, C, H, TO), TO \rightarrow SO|MO, \quad (1)$$

where N is the name of the I-Type, C is the confidence level representing the likelihood of the interaction, H represents information of the hand involved in the interaction (e.g. hand state: grasped or free), TO stands for the target object that represents an object involved in an interaction and can indicate single (SO) or multiple (MO) objects. A confidence level usually represents the likelihood of specific interactions between a user and objects in the overall interaction process. In previous interaction methods, the confidence level is discrete and users only know if an event has occurred or not. However, in the proposed method, the confidence level is a continuous variable from 0 to 1. For example, among the I-Types from Fig. 2, as the hand of the user gets closer to the orange block, I-Type (*Approach*) is activated. At this moment, the confidence level is inversely proportional to the distance between the block and the hand.

As shown in Fig. 3, the I-Types used in an application are represented in a three-layered graph, which we call the I-Graph. Each node within the I-Graph is connected to specific nodes of the adjacent layer. For example, I-Type IT_2 is a graph that consists of IU_2 from the inference unit and PU_2 and PU_3 from the perception unit.

Here, the layered graph G_L is a generalization of bipartite graphs, and can be formalized as follows.

A graph $G_L = (V_L, E_L)$ is a layered graph if the following conditions hold true:

$$\begin{cases} V_L = \cup_{i=1\ldots k}^{n} V_{L_k} \\ \forall_e = (v, w) \in E_L\text{: } v \in V_{L_i} \text{ and } w \in V_{L_{i+1}} \end{cases} \quad (2)$$

Using this definition, the I-Graph can mathematically be classified as a three-layered tripartite graph whose nodes are partitioned into three sets V_1, V_2 and V_3 such that $V_1 \cup V_2 \cup V_3$ and there is no $e = (v, w) \in E$ such that $v, w \in V_1$ or $v, w \in V_2$ or $v, w \in V_3$, i.e. a tripartite graph is a graph on three sets of nodes, where there are edges connecting only between these three sets of nodes. In short, edges in the I-Graph link vertices belonging to adjacent layers.

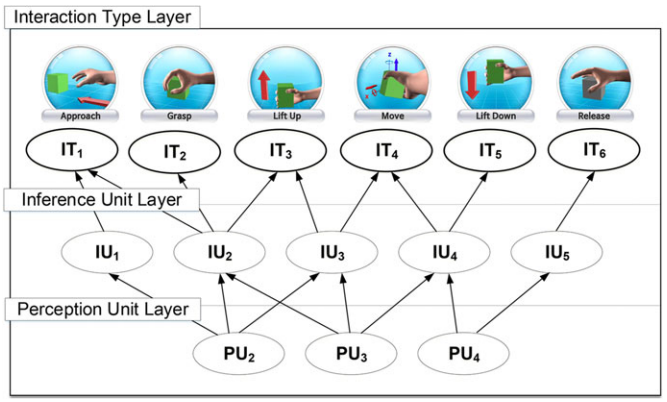

Figure 3. Three-layered graph structure constructed by I-Types. IT_i denotes the ith interaction type, IU_j denotes the jth inference unit, and PU_k denotes the kth perception unit.

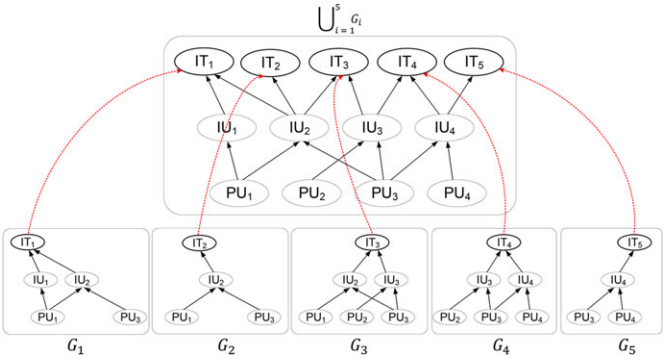

Figure 4. Example of an I-Graph that contains subgraphs representing each I-Type. Note that a certain application can be implemented as combinations of sub-I-Graphs.

The advantages of using the layered structure mentioned above are that units can be developed individually and reused in different applications as it is loosely coupled when creating I-Types. For example, an expert in hand detection algorithms can implement a PU specialized in hand detection in the perception unit layer. Another expert can use the results from hand detection to implement a grasp algorithm in the IU layer to determine grasp interaction. An application developer can combine the units for the two layers appropriately to develop any applications that use grasp interaction. Thus, the architecture allows developers to develop parts of the applications related to their own fields of expertise.

If an application requires more than one interaction as shown in Fig. 4, the I-Graph can be a union of sub-I-Graphs

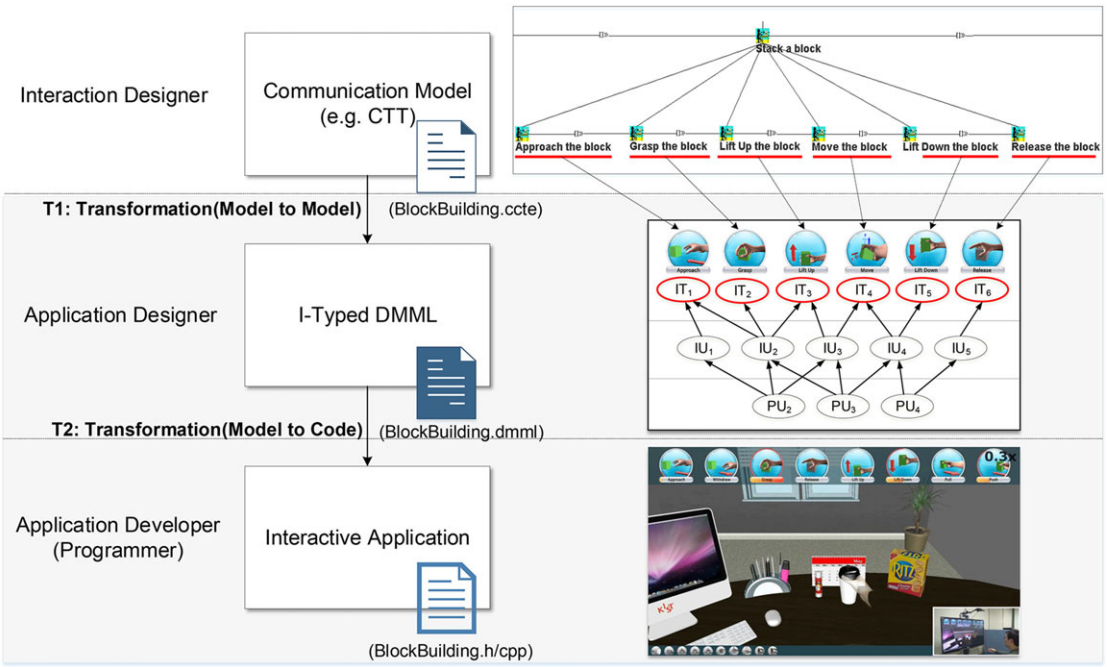

Figure 5. Transformations using I-Typed DMML.

that represent each I-Type. An application comprises selected I-Types, and represents an I-Graph. Various applications consist of combinations of different I-Graphs. From the combinations of I-Graphs, we can also extract interaction patterns. This feature can be useful in analyzing interaction patterns.

Figure 5 shows the process of application development from an interaction design model using task-centered description and the code-level implementation using the I-Typed DMML. Let us suppose that the interaction designer used the TBD tool called the CTT Environment (Paternò, 2004) and generated the communication model. In Fig. 5, the CTT model describes the interaction of virtual object manipulation as 'stacking blocks'. This task is further broken down into subtasks such as 'approach/grasp/lift up/lift down/release the block'. Understanding the interaction is very intuitive and easy since it is described as a set of human-centered tasks. The interaction design model is implemented through transformation, and the I-Typed DMML is in charge of the transformation. In the proposed method, transformation takes place in two steps. In the first step, the interaction design model is described by the I-Typed DMML and selects I-Types that are able to map to the interaction design model. For example, the 'lift up the block' task can be mapped to ITypeLiftUp. In the same way, one can select I-Types for the task to create the I-Typed DMML. We assume that **T1** transformation in Fig. 5 should be manually described instead of being automatically generated. In the second step, C++ code is generated automatically by inputting the I-Typed DMML. ITypeHandler functions in the generated code template are used to handle visual, sound, or haptic feedback when each

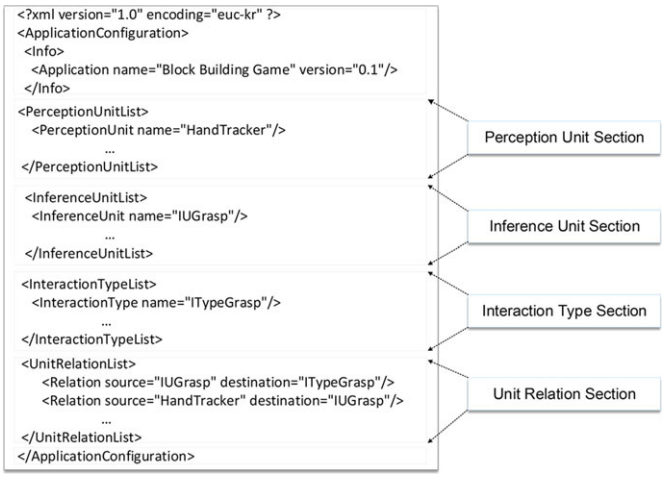

Figure 6. I-Typed DMML document structure.

I-Typed interaction occurs. For more details, please refer to Section 3.

In summary, to reduce the disadvantages of directly implementing task-based manipulation interactions with hard coding, we provide a method to accurately implement manipulation interactions according to the intention of the interaction designer by applying a one-to-one transformation based on the I-Type. 3D direct manipulation applications are developed by selecting and combining different I-Types and configuring them.

Figure 6 represents the document structure described using the I-Typed DMML. The I-Typed DMML, which is a declarative description of 3D direct manipulation, consists of I-Type

Table 1. DNOTGS Criteria for selecting I-Types.

	DNOTGS criteria	Example
D	**D**etailed fingertip interactions	pinch grasp
N	**N**on-tabletop activities (e.g. navigation or lower-limb movements)	bow, walk, run
O	**O**bject property (e.g. soft body, thin, complex virtual objects and so on)	tear, wringing
T	**T**ool-based interactions	hammer, screw
G	**G**eneral or ambiguous interactions	open, close
S	**S**pecific task-oriented interactions	assemble

Table 2. Feature comparison between conventional and I-Type-based methods.

Conventional method	I-Type-based method
Bottom-up approach (Algorithm to interaction)	Top-down approach (Interaction to algorithm)
Procedural or event-based programming	I-Type-based programming
Algorithm-oriented	Interaction-oriented
Narrow user spectrum	Wide user spectrum

sets and relations of each I-Type needed for describing the 3D DMIs. It consists of four sections describing I-Type, inference unit and perception unit for manipulation interactions to be processed by the application, and a unit relation section, which is necessary to describe the relation of each unit. The unit of each layer is not fixed but extensible. Experts can add new units or update the existing units and add new I-Types.

In this article, the DNOTGS (Do NOT over Generalize and Specialize) is proposed as criteria for selecting I-Types as shown in Table 1 to maintain consistency of system. The DNOTGS criteria are as follows. (i) Exclude interactions that require detailed finger manipulation. (ii) Interactions are limited to tabletop environments. Exclude interactions that require navigation or lower-limb movements. (iii) Exclude interactions that manipulate very thin or complex virtual objects. (iv) Exclude interactions that use tools instead of bare hands. (v) Exclude interactions that are too general or ambiguous. (vi) Exclude interactions that target specific tasks. The criteria for selecting I-Types is derived by considering the following two aspects. First, commercially available sensors have limitations with respect to stable fingertip detection and we only consider realizable hand manipulation interactions (of course, there are no limitations on describing interactions as well). Second, I-Types should only describe DMIs and not be too general or specific.

Table 2 compares the features of conventional and I-Type-based methods when developing direct manipulation applications. While the conventional method interprets the interaction as a low-level event and implements it as an algorithm-oriented method, the I-Type-based method can translate interactions into more descriptive and intuitive descriptions using everyday language. In other words, existing interactive application developments are algorithm-oriented, whereas the proposed method is based on I-Types, which represent interactions. This method provides a way for anyone without professional knowledge to develop applications easily. As a result, a wide range of users can develop interactive applications using this method.

3. DMI ENGINE

This article introduces a DMI system that helps overcome the following limitations encountered while developing traditional interactive applications: (i) the dependency of the perception module on input devices, (ii) low reusability of perception modules and (iii) low usability owing to a method that is not human-centric.

The reason why the perception module depends on input devices is that the code is device-dependent. In addition, different input devices have different data structures for information. Without having a unified data structure for different input devices, it is impossible to have device-independent codes. Typically, perception modules and inference modules are coupled in interactive applications. If a new inference method is added to a perception module, relevant codes should be also modified, and perception and inference modules have low reusability when applied to a new application. Most interactive applications define and process interactions in a system-centric manner, for instance, by event handling. It is different from the way humans define and implement interactions and is difficult to implement.

3.1. Overall structure of the DMI system

In this section, an overview of the DMI system structure is described. Figure 7 shows the overall structure of the DMI system that consists of a device abstraction layer and three kinds of engines, namely, a graphics rendering engine, a physics engine and a DMI engine. The system also includes a memory manager and an AppRunner.

The device abstraction layer eliminates the dependency of the perception module and input devices and enables to switch to a new device without modifying the module. The most commonly used devices for detecting hand pose for 3D virtual object manipulation are 3Gear Camera (3Gear, 2017), Leap Motion (Leap motion, 2017) and other wearable-type motion capture devices. The device abstraction layer allows the addition of various devices and expansion without limitations. The graphics rendering engine and the physics engine use scene graphs to manage virtual objects for natural interactions, which are widely used in physics-based interaction frameworks. It efficiently manages scenes from graphics to physics or vice versa in the graphics rendering engine and the physics engine. Consequently, the scene graph manages the objects that result from the physics interaction, which can be

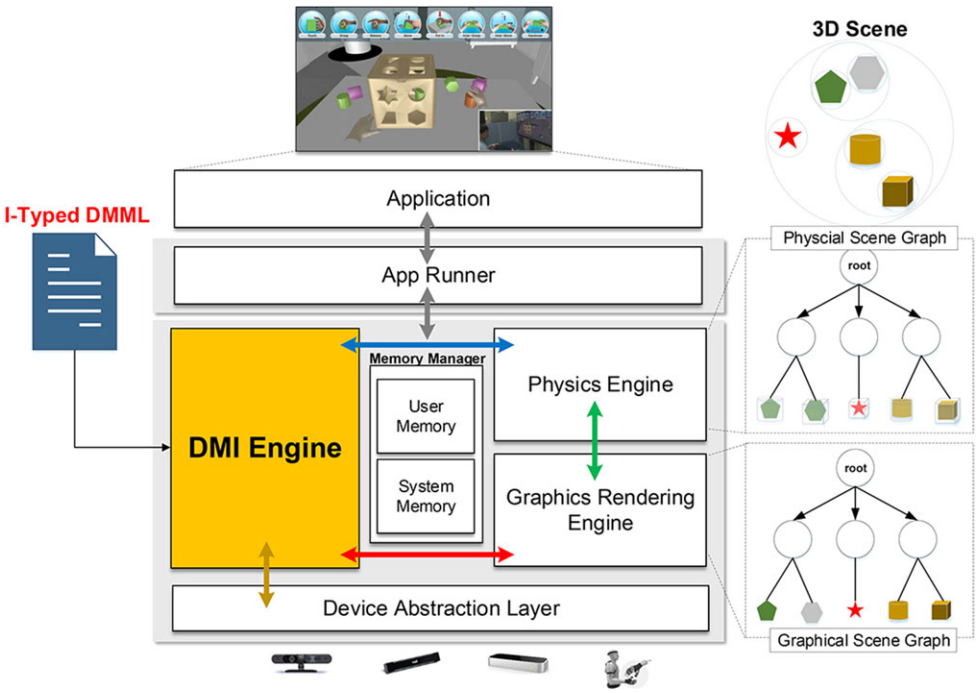

Figure 7. DMI system outline. The DMI engine, the core of the DMI system, is highlighted in orange.

synced with the graphics instantly. The proposed system uses a graphics rendering based on OpenGL and the Bullet physics engine (Bullet physics, 2017). The physics engine offers a common interface, for instance PAL (Physics Abstraction Layer, 2017), to support various physics engines to provide extensibility on demand.

The memory manager manages shared data in each engine by dividing the memory into user memory and system memory. The system memory stores and manages information obtained from input devices, graphic objects for 3D scenes or physics objects for physics simulations. In DMIs, managing hand data is important since it is based on bare hand direct manipulation. Hand and finger pose information, hand states such as grasped or released, and grasped objects information are managed by the system memory. The user memory manages data generated from the DMI engine such as inputs, parameters and outputs. The AppRunner runs applications separately from the DMI system. Consequently, it decouples the DMI system and the application, guaranteeing independency and restricting illegal access to tighten security.

3.2. Features of the proposed DMI engine

The DMI engine is the core of the system that differs from the existing interaction system and supports direct manipulation. It is structurally independent of the graphics rendering engine or the physics engine, which can be easily ported on other platforms with slight modifications. The proposed DMI engine has two main features. First, the DMI engine comprises three layers, namely, the I-Type layer, inference unit layer and perception unit layer. Especially, the I-Type layer that abstracts the interactions improves the usability because the interaction application can be developed intuitively and easily. Second, the DMI engine provides a common interface between each layer. The common interface allows developers with diverse backgrounds to create and modify their units independently of other units of the hierarchy using their own expertise. It also allows developers to add new units and combine with other units without any restrictions. Furthermore, it provides unified input and output methods that enhance usability by providing easier connections to other units without modifying the engine and support the addition of new units for expansions.

As shown in Fig. 8, the proposed DMI engine creates the I-Graph, which consists of the I-Type layer, inference unit layer, and perception unit layer, during run time and executes it. The top layer is the I-Type layer that consists of I-Types related to manipulations. This means that users do not need to create modules using C++ or Java but instead just select the appropriate interactions described by verbs used in everyday life that they need. Specifically, if a user grasps, pushes, or pulls an object, such interactions can be mapped one to one with I-Types. The second layer is the inference unit layer that restricts coupling to the perception module. It enables the application of different perception modules to the same inference module without modifying the code, and provides a method of selecting an appropriate perception module through performance comparison. The last layer is the perception unit

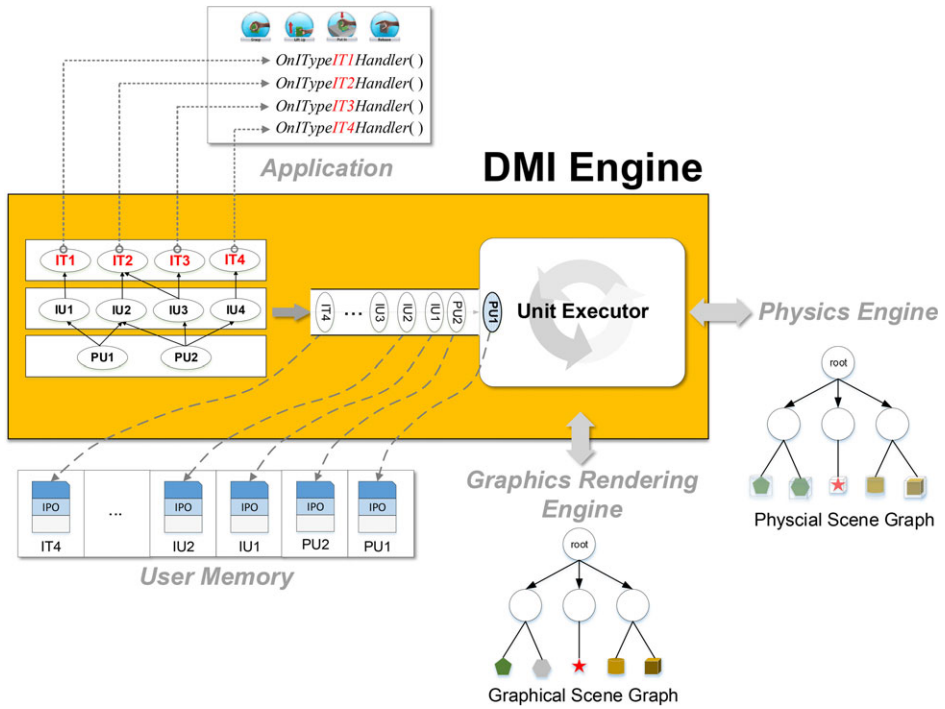

Figure 8. Structure of the DMI engine.

layer that defines the natural user interface (NUI) based perception module. For example, the perception unit layer performs head pose estimation, hand and finger pose estimation, face tracking, gaze tracking and so on. Even if these perception modules have the same input and output, there can be different versions depending on the core algorithms. Because there can be many combinations among devices and the perception module as well as the perception module and the inference module, all modules are extensible and reusable in new applications. The important point here is that the top layer (i.e. I-Type layer) does not change even though the lower layers are modified. Unlike the other interactive applications, the application developed using the proposed method is based on I-Types and the application codes do not change even when the perception module changes.

As shown in Fig. 8, each unit described in the I-Typed DMML is created and registered within the user memory for the application. The I-Graph is created using the relationship of each unit. In the DMI engine, the execution of the I-Graph is carried out first in the perception unit layer (bottom layer), then in the inference unit layer (middle layer), and lastly in the I-Type layer (top layer). Each unit is input to the unit executor in the form of first in first out (FIFO), and the unit executor automatically executes the unit using the same common interface defined for all units. The input, output and parameter values to be used in each unit are obtained by referring to the corresponding memory block in the user memory described above. The unit that needs to refer to the graphic object

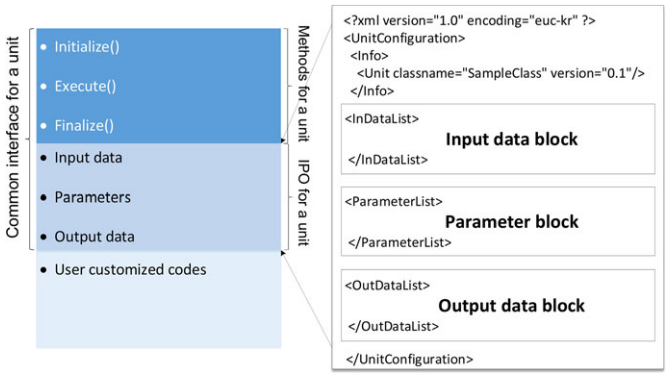

Figure 9. Common interface structure of a unit.

information gets the information of that object from the scene graph managed by the graphics rendering engine. Similarly, physics simulation results such as contact, collision, etc. can be exchanged with the physics engine. When an I-Type is activated, the I-Type handler defined in the application is called, and the object information and the hand state related to the corresponding I-Type are provided as arguments. Therefore, application developers can simply make an interactive application by implementing user feedback codes in the I-Type handler.

A unit consists of a common interface and user customized codes as shown in Fig. 9. The common interface is a feature provided to the DMI engine that executes and manages units consistently. In order to do so, the common interface has the

same execution routine (Initialize/Execute/Finalize) and the data structures Input/Parameter/Output (IPO) required by each unit when processing them. As shown on the right side of Fig. 9, the IPO is described in a XML file and the memory for the IPO is automatically allocated and registered to the user memory when a unit is created. Users can easily access the unit's input and output values by using a key value from the user memory. In addition, parameter values can be accessed to tune the performance of the algorithm by unit developers.

3.3. Development process using the I-Typed DMML

The development process using the I-Typed DMML comprises defining the task model, creating the units and creating the interactive application as shown in Fig. 10. The involved developers are an interaction designer (software engineer), a unit developer, and an interactive application developer. The interaction designer creates and designs interactions from tasks using CTT. The unit developer can develop I-Type, inference or perception units of his/her expertise. The interactive application developer can create a direct manipulation application by combining the I-Types provided by the DMI system, and change the combination of the inference unit and the perception unit that make up the I-Type as needed.

The unit authoring process can be divided into three steps shown in Fig. 11. First, in the unit configuration step, a unit developer sets the name for the unit, selects the type and describes the IPO used in the unit, and stores them in a XML file. Second, in the code generation step, the configuration file is given as an input into the unit template generator and a code template for the unit is created. Lastly, in the implementation step, the unit developer implements the algorithms in the code template generated in the second step. Configuration files and code templates created in the above steps are stored in the DMI system unit repository, to which application developers have access.

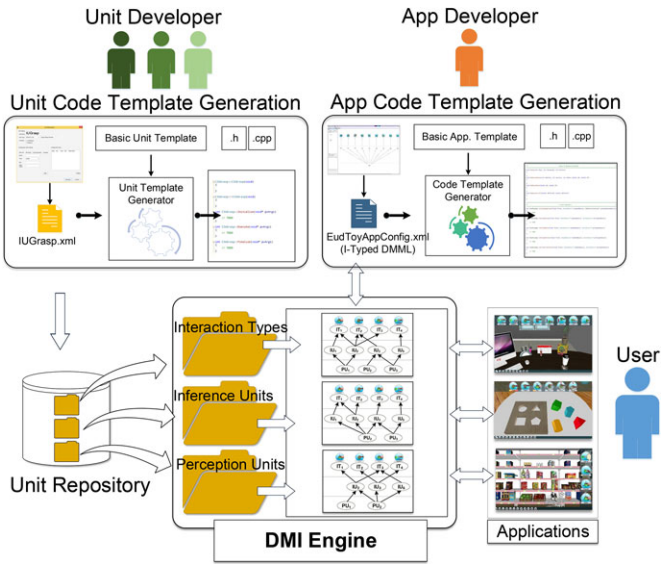

Figure 10. I-Type-based interactive application development process.

Similar to Figs. 11, 12 shows the development process of interactive applications. First, an app configuration file is created with the app name and I-Types to be used are selected. Then an application code template is generated in the second step, in which feedback can be implemented. Application development is broadly divided into basic and expert mode. The basic mode provides I-Types with default relations for inference and perception units. Hence, application developers do not need to have any professional knowledge in inference or perception algorithms to develop applications. Conversely, in the expert mode, application developers are able to select inference and perception units and customize relations for I-Types. They can also combine and test algorithms or add and modify units to enhance performance. During the application development phase, one can leave the I-Types as they are but can freely modify or combine lower layer units without changing the application codes.

Listing 1 is an example of an auto-generated application code template of the I-Type handler, which processes feedback in each I-Type when an interaction has occurred. It is necessary to implement user feedback (visual, sound, haptics, etc.) in the I-Type handler. For example, when a user grasps a cube block, a specific I-Type handler (OnITypeGrasp) is invoked. Then the grasp's confidence level, hand information

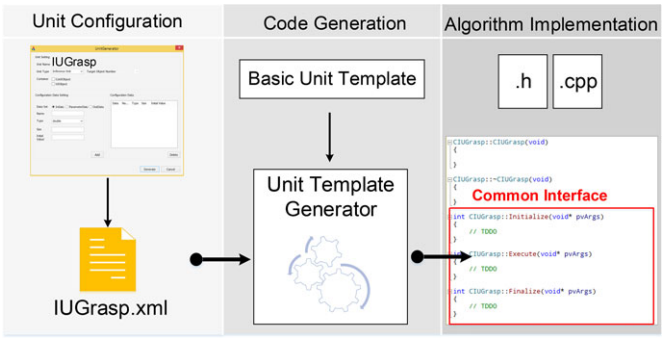

Figure 11. Unit authoring process.

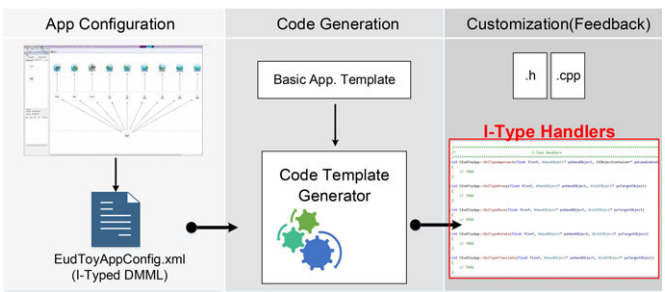

Figure 12. Application authoring process.

```
1  int CApplication::OnITypeApproach(float fConf, AHandObject*
       poHandObject, CGObjectContainer* poCandidateObjects)
2  {
3  // TODO
4  return 0;
5  }
6
7  int CApplication::OnITypeGrasp(float fConf, AHandObject*
       poHandObject, AGraphicObject* poTargetObject)
8  {
9  // TODO
10 return 0;
11 }
12
13 int CApplication::OnITypeMove(float fConf, AHandObject*
       poHandObject, AGraphicObject* poTargetObject)
14 {
15 // TODO
16 return 0;
17 }
18
19 int CApplication::OnITypeRotate(float fConf, AHandObject*
       poHandObject, AGraphicObject* poTargetObject)
20 {
21 // TODO
22 return 0;
23 }
24
25 int CApplication::OnITypeTranslate(float fConf, AHandObject*
       poHandObject, AGraphicObject* poTargetObject)
26 {
27 // TODO
28 return 0;
29 }
```

Listing 1. A part of the generated application code for I-Type handlers.

and object information are passed as parameters. These information can be used by the application developers to add feedback to the grasped object, such as changing colors or sound effects.

4. CASE STUDY AND USABILITY EVALUATION

In this section, we define 3D manipulation interaction using the proposed I-Typed DMML and explain the method of developing interactive applications using three examples. We also assess and analyze usability in terms of efficiency, effectiveness and satisfaction.

4.1. Case study

4.1.1. Experimental setup and I-Type dictionary

The experimental environment is shown in Fig. 13 for manipulating virtual objects directly with a bare hand in the virtual environment. The virtual environment is shown on a 55-inch flat 3D display with full HD (1920 × 1080). An RGBD camera (Primesense camera) is mounted on the top, facing downward to track the user's hand pose at 30 fps. The PC used has an i7-4790 Intel CPU with 32 GB memory and NVidia GTX 780 running on Windows 8.1. Visual Studio C++ was used for developing the IDE used to develop the DMI system and the interactive application in C++.

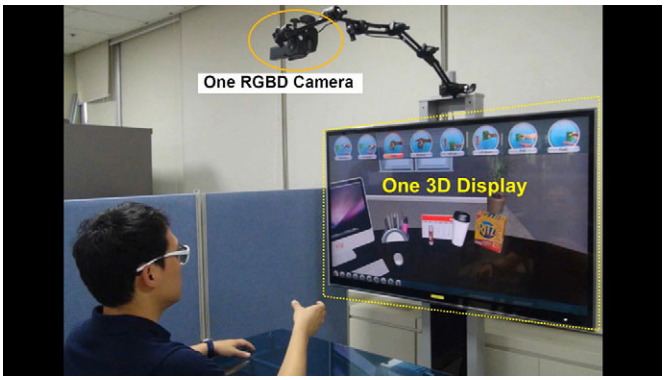

Figure 13. Experimental setup for case study.

Before looking at the examples of interactive applications, I-Types for various manipulation interactions in daily life were selected as shown in Table 3 according to the DNOTGS criteria described in Section 3. Table 3 gives the name, graphical symbol, explanation, and category of the chosen interactions. I-Types are divided into two groups: grasp-based and non-grasp based. Similar to units, I-Types are not limited to Table 3 and can be added or removed.

4.1.2. Application: EduToy

EduToy is an example of an interactive application that is similar to matching blocks for children. The virtual environment is a 3D scene, which has a box comprising holes of different shapes and little blocks around the box. The blocks are polygons, cubes, and cones and so on. Users are supposed to put the blocks into the appropriate holes of matching shapes. Interactions occurring in this application can be implemented using I-Types. Figure 14 shows the I-Types used for *EduToy* and the relevant IUs and PUs as an I-Graph. The user grasps an object (ITGrasp), lifts it up (ITLiftUP), moves it to the appropriate hole (ITMove), and puts the block in (ITPutIn). This process can be mapped to I-Types as shown in Fig. 14.

As explained in Section 3, one selects the appropriate I-Types for *EduToy* and generates the app configuration as shown in Listing 2.

Listing 3 shows a part of auto-generated code using the app configuration, which comprises I-Type handlers invoked by I-Types when the interaction occurs. The application developer implements the interactive application by providing proper feedback based on the confidence level, which is the likelihood of the interaction to occur, hand information, and object information. ITypePutIn represents putting a block into a hole by changing the object's color to blue when the confidence level is above 0.5 and gives a sound effect when the

Table 3. I-Type dictionary.

I-Type	Symbol	Usage and description	Category	I-Type	Symbol	Usage and description	Category
Approach		come near or nearer to an object that is at a distance	NON-GRASP	Pull		exert force on object by taking hold of it to move it toward oneself	GRASP
Withdraw		move away from an object	NON-GRASP	Push		exert force on object with one's hand to move it away from oneself	GRASP
Touch		put your hand onto and off an object	NON-GRASP	Put In		place an object in a specified location	GRASP
Grasp		seize and hold an object	GRASP	Take Out		remove an object from a hole	GRASP
Release		free from holding an object	GRASP	Adjust		alter or move an object slightly to achieve the desired fit or appearance	GRASP
Move		move an object from one place to another	GRASP	Attach		affix or stick an object	GRASP
Lift Up		take an object and lift it upward	GRASP	Detach		separate or unfasten an object	GRASP
Lift Down		take an object and lift it downward	GRASP	Translate		make an object move in one direction	GRASP
				Rotate		make an object turn in a circular direction	GRASP

confidence level is above 0.9 as an indicator of the block being successfully put in the hole.

As mentioned in the DMI system, the IPO described in the unit of each layer is automatically allocated and registered in the user memory at runtime, making it easy to read and write memory from the unit. For example, Fig. 15 shows a memory block of units generated at runtime in the DMI system for the *EduToy* application. Each unit has access to input, parameter and output from the memory, and each memory block applies data synchronization to ensure consistency of data shared among the units.

Figure 16 shows interactions through the interactive application *EduToy*. A user in real space interacts with virtual objects in the virtual space with a bare hand in Fig. 16a and b. The user is putting the block into the box as shown in Fig. 16b. The virtual space the user sees is shown in Fig. 16c and d. The I-Type gauges on the top of the screen are simultaneously displayed with activated interactions by the user's hand movement and show the confidence levels to aid understanding, and are not a requirement for applications.

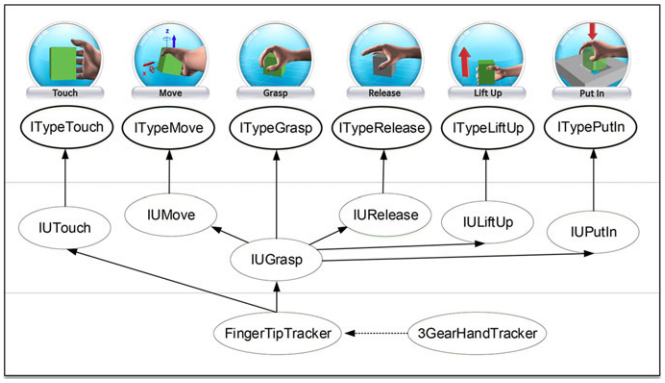

Figure 14. I-Graph for *EduToy* application.

```
<?xml version="1.0" encoding="euc-kr" ?>
<ApplicationConfiguration>
  <Info>
    <Application name="EduToy application" version="0.1"/>
  </Info>
  <PerceptionUnitList>
     <PerceptionUnit name="3GearHandTracker"/>
     <PerceptionUnit name="FingerTipTracker"/>
  </PerceptionUnitList>
  <InferenceUnitList>
    <InferenceUnit name="IUTouch"/>
    <InferenceUnit name="IUMove"/>
    <InferenceUnit name="IUGrasp"/>
    <InferenceUnit name="IURelease"/>
    <InferenceUnit name="IULiftUp"/>
    <InferenceUnit name="IUPutIn"/>
  </InferenceUnitList>
  <InteractionTypeList>
    <InteractionType name="ITTouch"/>
    <InteractionType name="ITMove"/>
    <InteractionType name="ITGrasp"/>
    <InteractionType name="ITRelease"/>
    <InteractionType name="ITLiftUp"/>
    <InteractionType name="ITPutIn"/>
  </InteractionTypeList>
  <UnitRelationList>
    <Relation source="3GearHandTracker" destination="
      FingerTipTracker"/>
    <Relation source="FingerTipTracker" destination="IUTouch"/>
    <Relation source="FingerTipTracker" destination="IUGrasp"/>
    <Relation source="IUTouch" destination="ITypeTouch"/>
    <Relation source="IUGrasp" destination="IUMove"/>
    <Relation source="IUMove" destination="ITypeMove"/>
    <Relation source="IUGrasp" destination="ITypeGrasp"/>
    <Relation source="IUGrasp" destination="IURelease"/>
    <Relation source="IURelease" destination="ITypeRelease"/>
    <Relation source="IUGrasp" destination="IULiftUp"/>
    <Relation source="IULiftUp" destination="ITypeLiftUp"/>
    <Relation source="IUGrasp" destination="IUPutIn"/>
    <Relation source="IUPutIn" destination="ITypePutIn"/>
  </UnitRelationList>
</ApplicationConfiguration>
```

Listing 2. Modeling *EduToy* application with I-Typed DMML.

4.1.3. *Application: VirtualOffice*

It is possible to develop various interactive applications using the same I-Types. *VirtualOffice* is another example developed using I-Types. In this virtual office, there are many objects such as a glue, a highlighter, a calendar, a tumbler and so on for manipulations such as grasp, lift up, move, put down and so on. Figure 17 shows the I-Graph used for the application.

```
int CEduToy::OnITypeGrasp(float fConf, AHandObject*
      poHandObject, AGraphicObject* poTargetObject)
{
  // TODO
  return 0;
}

int CEduToy::OnITypeRelease(float fConf, AHandObject*
      poHandObject, AGraphicObject* poTargetObject)
{
  // TODO
  return 0;
}

int CEduToy::OnITypeMove(float fConf, AHandObject*
      poHandObject, AGraphicObject* poTargetObject)
{
  // TODO
  return 0;
}

int CEduToy::OnITypePutIn(float fConf, AHandObject*
      poHandObject, AGraphicObject* poTargetObject)
{
  if( fConf >= 0.9f ) {
    // sound feedback
    PlaySound(''alarm.wav'');
  }
  else if( fConf > 0.5f) {
    // visual feedback
    ChangeColor(poTargetObject, eCOLOR_BLUE);
  }
  return 0;
}
```

Listing 3. A part of auto-generated C++ code skeleton from the I-Typed DMML (Listing 2).

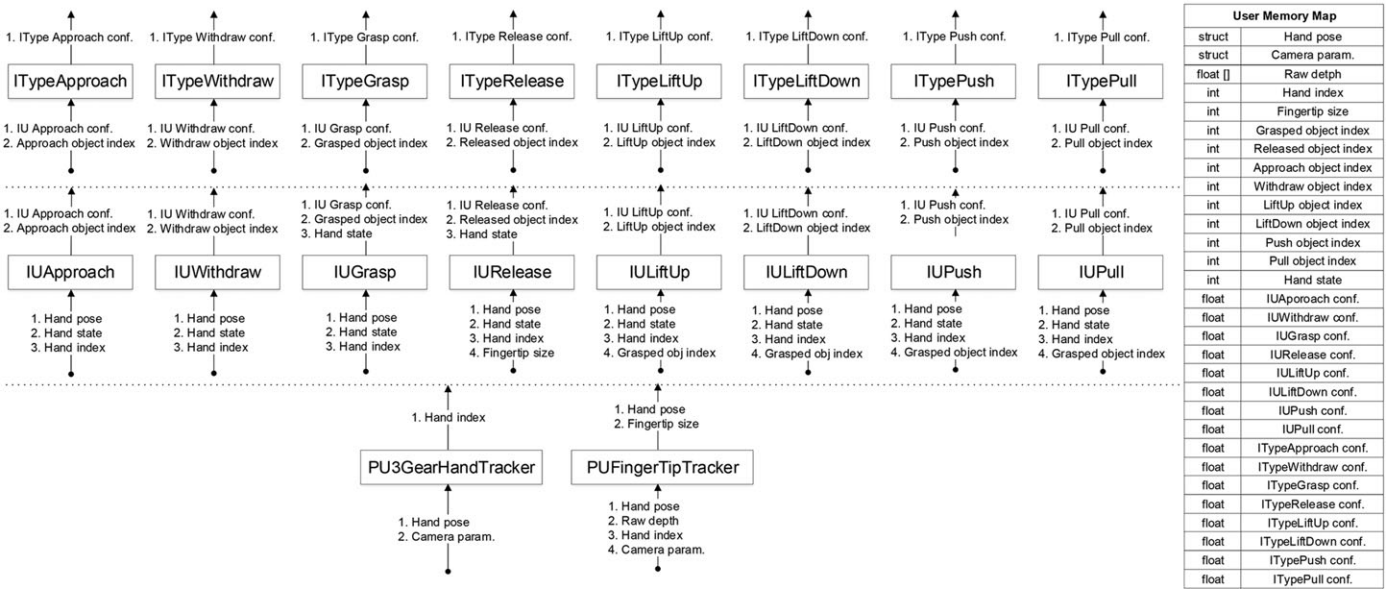

Figure 15. Memory configuration for *EduToy* application.

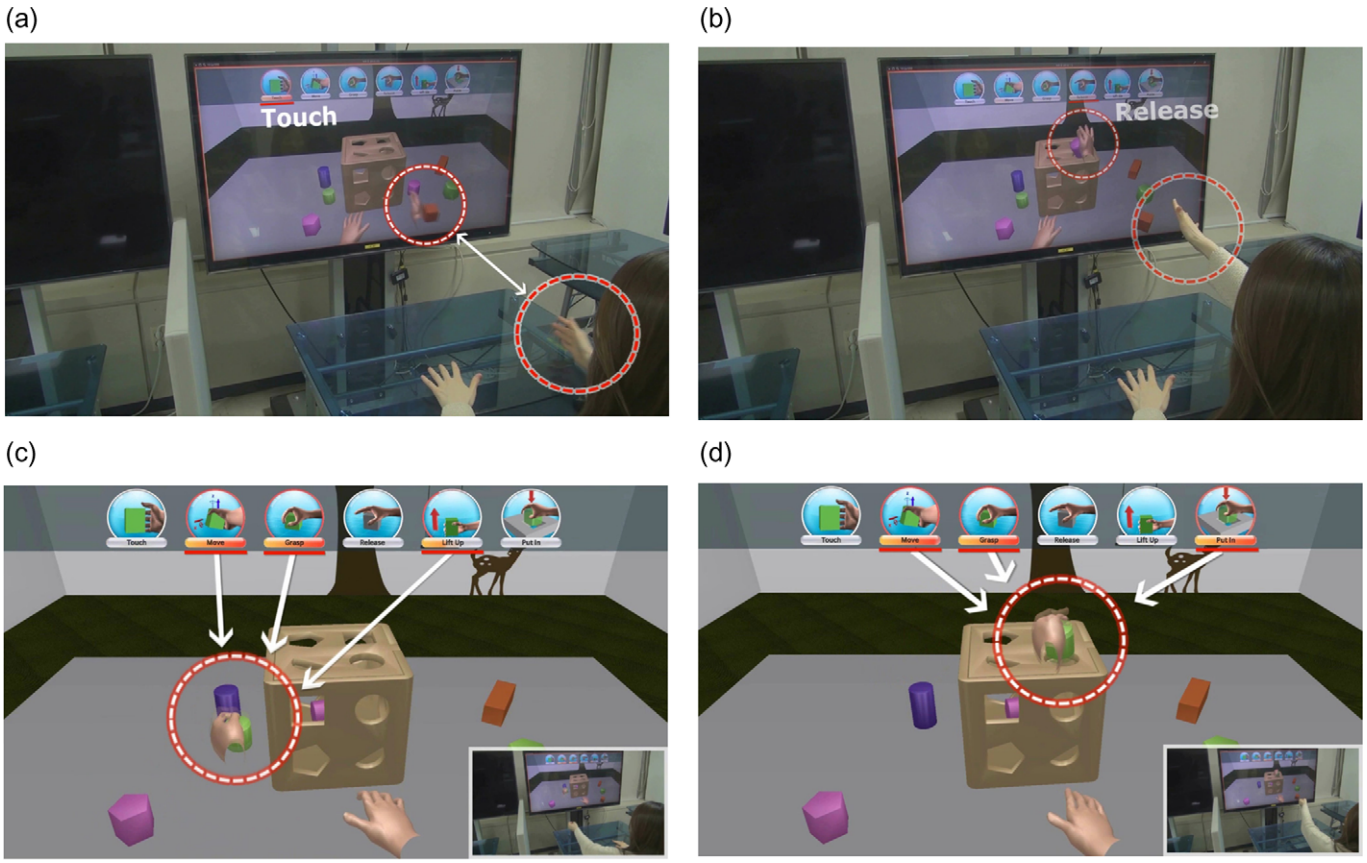

Figure 16. Screenshot of *EduToy* application: (a) Touch in real space; (b) Release in real space; (c) Move, Grasp and LiftUp in virtual space; and (d) Move, Grasp and PutIn in virtual space.

Note that it uses the same I-Types used in *EduToy* application. This means that it is possible to reuse the I-Types in different applications without any modification.

Figure 18c and d show interactions of a user lifting up a tumbler, pulling it toward the user, and putting it down on the desk. Several I-Types are activated at one point until the tumbler is lifted and placed in the desired position. When the user lifts up the tumbler, not only is LiftUp activated but also is Pull. This means that two actions are combined in real life. This shows that the confidence of an I-Type is not limited to interaction application development but can also be used in the analysis of interaction patterns of users. Figure 19 shows the changes in I-Type Pull and Push when the user brings the tumbler towards him/her and also in the opposite direction in the *VirtualOffice* application. By analyzing the interactions, their associated target object, and their confidence level with

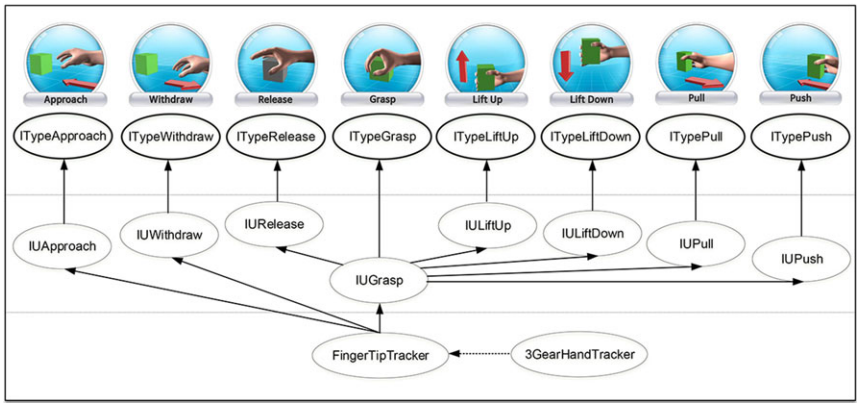

Figure 17. I-Graph for *VirtualOffice* application.

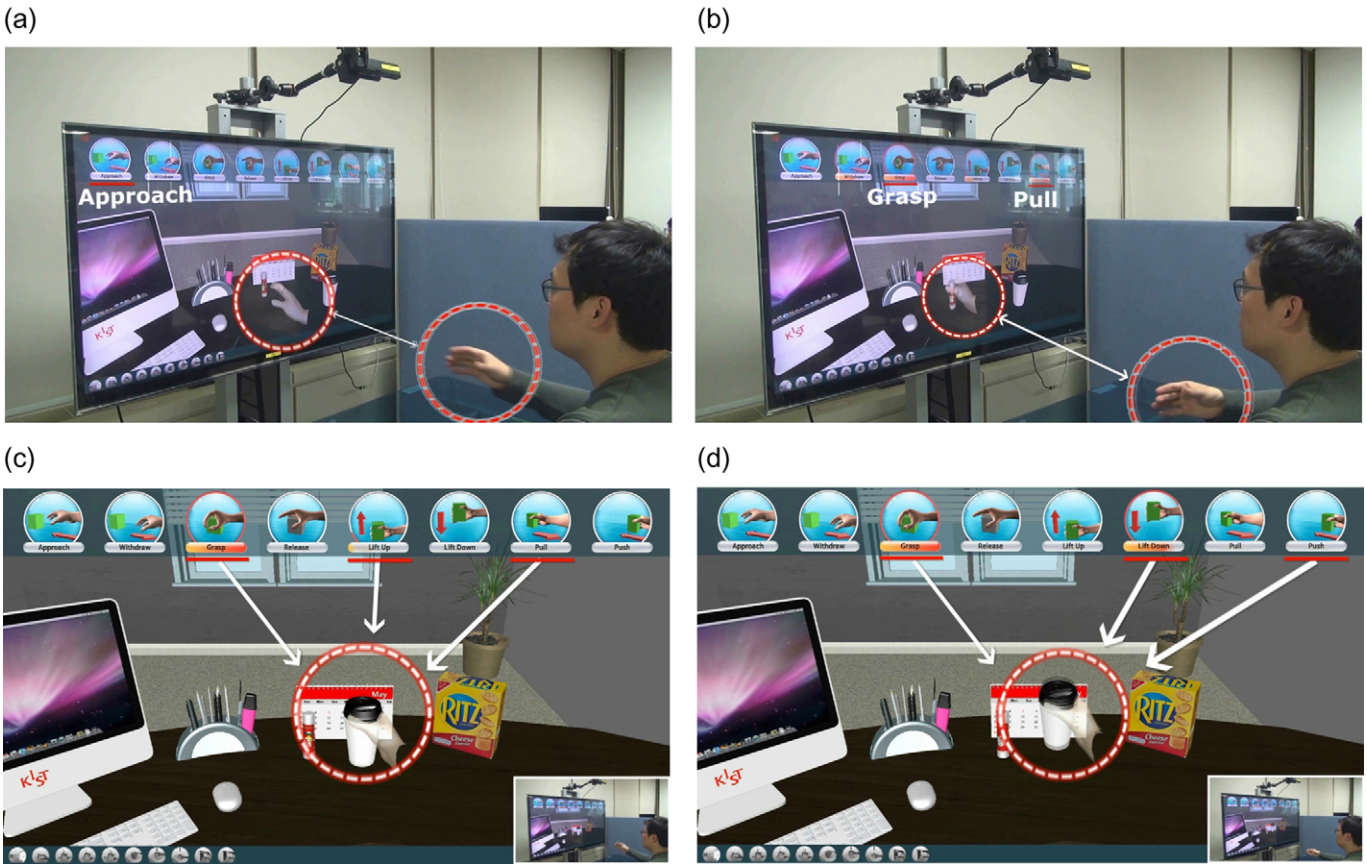

Figure 18. Screenshot of *VirtualOffice* application: (a) Approach in real space; (b) Grasp and Pull in real space; (c) Grasp, LiftUp and Pull in virtual space; and (d) Grasp, LiftDown and Push in virtual space.

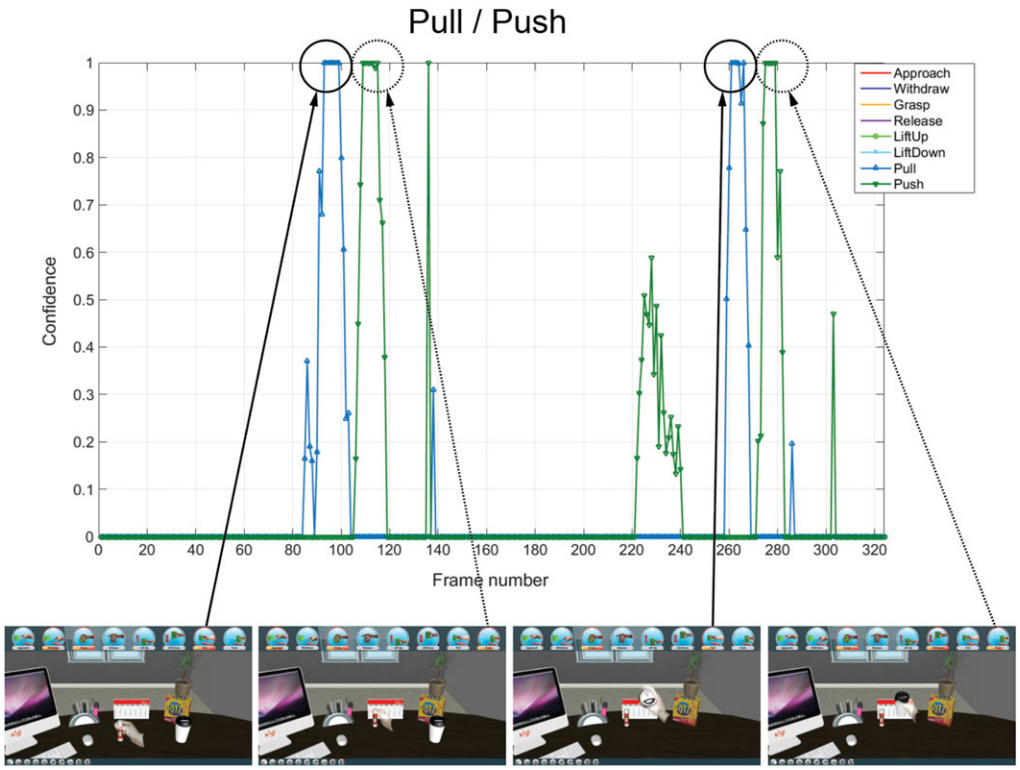

Figure 19. The confidence changes relevant to the I-Types of Pull and Push. The key frames corresponding to each I-Type are shown in the bottom of the figure. The solid circles denote the peak confidence of Pull interaction and the dashed circles denote the peak confidence of Push interaction.

respect to time, we can deduce the interaction frequency, interest objects and manipulation patterns for interaction analysis. In addition, we can analyze the performance of algorithms in units with the same interaction.

4.1.4. *Application: VirtualChess*

This section shows that it is possible to develop an application without having to go through the hassle of reimplementation when the used sensors are changed. Interaction applications are mostly dependent on sensors and adding a new sensor or modifying the perception algorithm can be time consuming. However, when developing the *VirtualChess* application using the proposed method and switching the sensor from 3Gear camera to Leap Motion, the developer only needs to change 3GearHandTracker PU to LeapMotionHandTracker PU. The developer does not need to change the I-Graph, I-Types or any inference units. As shown in Fig. 20a and b, it is possible to develop a new interactive application like *VirtualChess* by only modifying the PU. Figure 20c shows the *VirtualChess* using the Leap Motion sensor.

4.2. Usability evaluation

4.2.1. *Usability test outline*

The I-Typed DMML intends to provide an environment to create interactive applications for non-professional users as well as experts, such as algorithm developers, interaction designers and interactive application developers. In this section, we evaluate the effectiveness of the interactive application development process when using the I-Typed DMML according to ISO-9244-11 (Bevan, 2000), which is a usability testing technique. The evaluation criteria for usability are as follows.

- *Efficiency*: This indicates whether the entire development process is efficient, and the amount of time taken for a user to finish a given task is measured.
- *Effectiveness*: This indicates if resources or functions are effectively provided during the entire development process, and the number of errors that occur during the development process or after completion of the development process is measured.
- *Satisfaction*: The ease of use is evaluated through a survey (use of development resources and ease of coding and error correction).

We used a within-subjects design for the user study. This method has a smaller sample size than a between-subjects design and can detect differences between design metrics. However, it has the disadvantage in that a learning effect (also called a sequence effect or carryover effect) can occur, in which participants become better as they participate in the experiments. Fortunately, there is an effective way to reduce many (but not all) of the

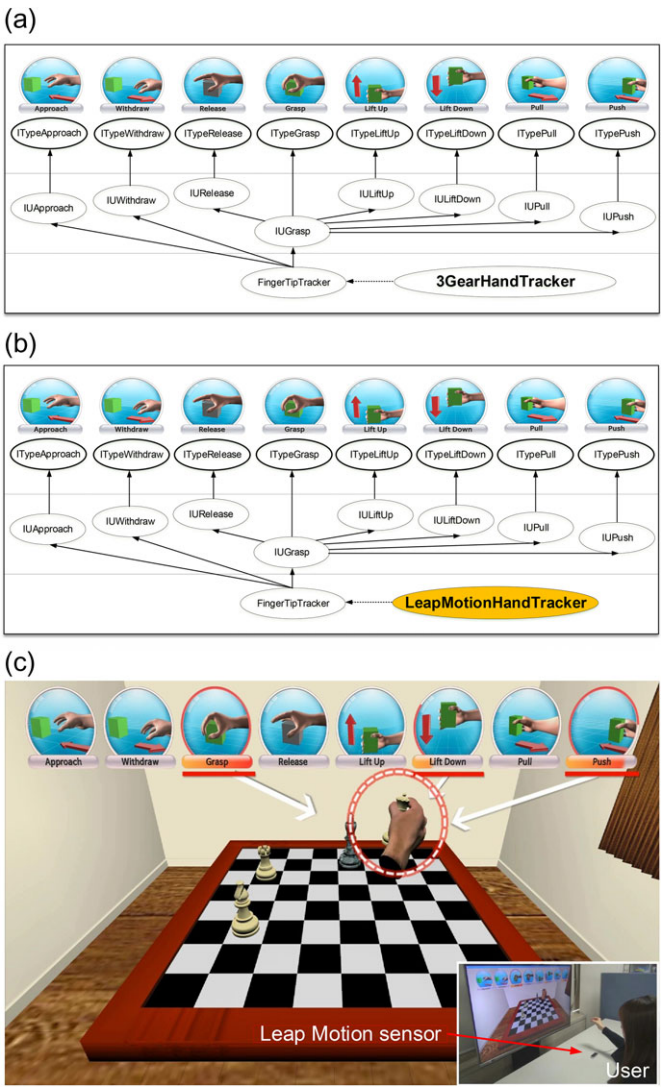

Figure 20. *VirtualChess* Application: (a) I-Graph for *VirtualChess* with 3Gear camera; (b) I-Graph for *VirtualChess* with Leap Motion sensor; and (c) Screenshot of *VirtualChess* application with Leap Motion sensor.

negative consequences that carryover effects bring through counterbalancing (MacKenzie, 2012). There is not much existing research in which user studies on the development methodology have been performed, so we employed the user study methodology that was introduced in Gismo, a gesture-based DSL (GISMO's Tech. report, 2017; GISMO's Wiki, 2017). The study participants comprised volunteers belonging to three organizations (Korea Institute of Science and Technology, Center of Human-centered Interaction for Coexistence and Kwangwoon University), to minimize the bias in the experiment.

4.2.2. *Experimental procedure*

The experiment consists of the following four stages.

(i) *Learning stage*: We describe the concept of direct manipulation application of 3D virtual objects, introduce the I-Typed DMML, and explain the use of the API and DMITool used in the experiment through video prepared in advance. These concepts were explained using a toy scenario similar to the one that the participants would be asked to create later in the experiment.

(ii) *Pre-survey stage*: A preliminary survey is performed (see Table A1 in Appendix).

(iii) *Experimental stage*: The proposed interaction application is created (see *Usability Testing* in Appendix). All participants create the interaction application using two methods, that is, the existing interaction application development method and the proposed method. To reduce the learning effect between the two methods, the participants were divided into two groups. One group was asked to develop the interaction application first using the existing method and then the proposed method, and the other group was asked to develop the interaction application first with the proposed method and then the existing method.

(iv) *Post-survey stage*: After the experiment, a survey is performed on the ease of use and satisfaction of the interaction application development method, an so on (see Table A2 in Appendix).

4.2.3. *Evaluation results*

- *Participant background knowledge*

A user evaluation was carried out for 24 participants. To make it fair, those who had been exposed to the I-Typed DMML were excluded from the evaluation. In the preliminary survey, familiarity with the application or the development method was assessed by asking the participants their majors and their experience using the DMI application. Figure 21 shows the distribution of majors of the experiment participants. A total of 24 students and researchers (14 males, 10 females), aged 20–35 years old, took part in this study. Most were majoring in engineering-related fields and a few were majoring in mathematics. This indicates that the participants are familiar with logical thinking, which is fundamental to the implementation of interactive application programs.

As shown in Figs 22 and 23, the participants were not familiar with interactive applications and had no previous experience in developing them. The work description prior to the experiment was appropriate, as shown in Figs 24 and 25.

- *Assessment of efficiency*

Efficiency was assessed by measuring the time taken to write code for the direct manipulation application presented in the experimental stage with the two development methods. In order to make a fair comparison of the two development methods and for the convenience of the experiment participants, we provided an API that supports DMI for the existing

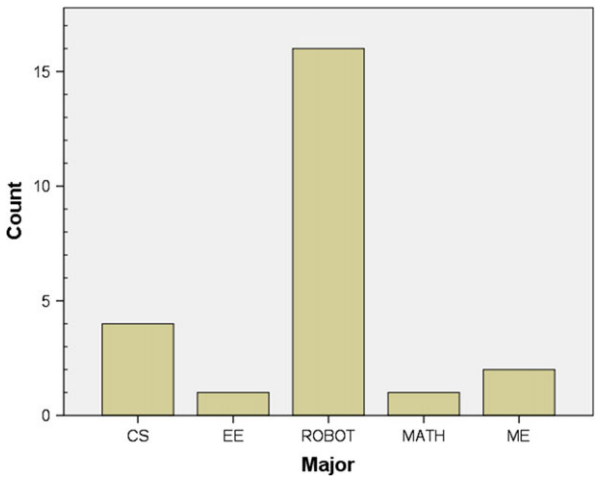

Figure 21. Participant majors.

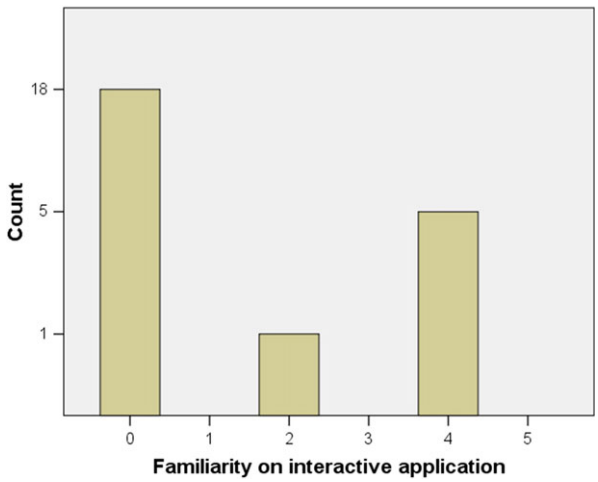

Figure 22. Familiarity with interactive applications.

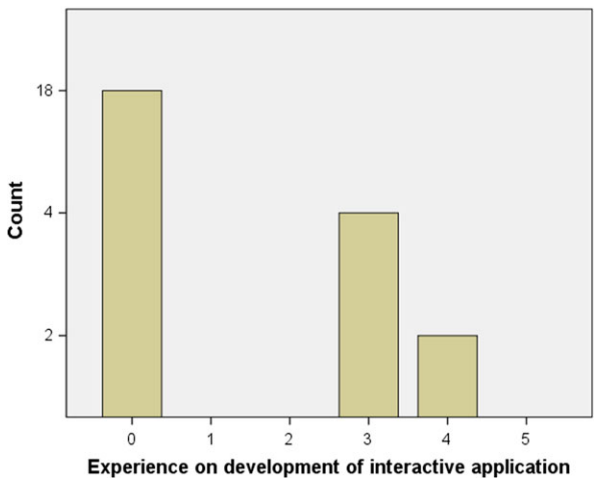

Figure 23. Past experience with interactive application development and participation in related works.

development method and provided a DMITool interface for the I-Typed DMML. For user feedback, the same API was provided for both development methods.

For comparison of the above two methods, Table 4 provides the time measured and the errors found in the development results. As evident in the results shown in Table 4, with the proposed method users were overall able to write the code in a shorter time in terms of time taken for the development efficiency comparison. Figure 26 shows a statistical analysis of the development time distribution for the two methods based on the results shown in Table 4. An independent-samples *t*-test was conducted to compare development of time for each method. There was a significant difference in the times for the API ($M = 28.4167$, SD $= 13.00474$) and the DMITool ($M = 10.3750$, SD $= 4.88843$); $t(29.372) = 6.362$,

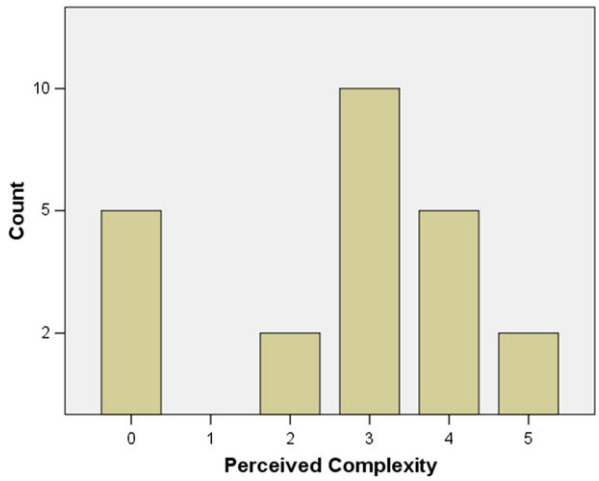

Figure 24. Survey result of perceived complexity.

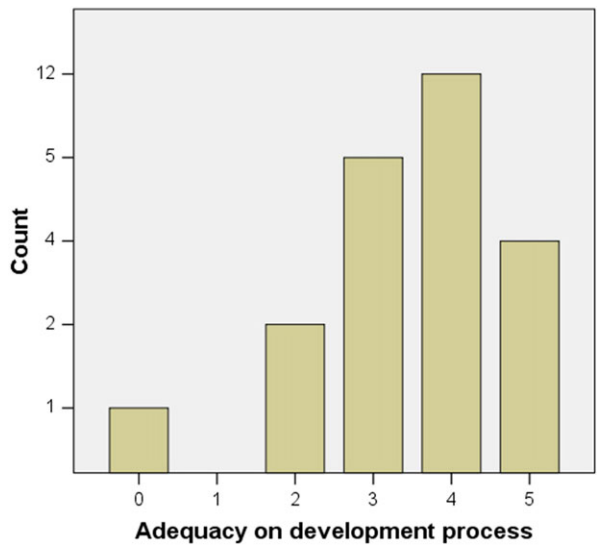

Figure 25. Adequacy of the development process for the interactive application using the proposed method.

Table 4. Time taken and errors found in the developed code (Majors: Robotics(16), Electrical Engineering(1), Mechatronics Engineering(2), Mathematics(1), and Computer Science(4)).

Participant no.	1	2	3	4	5	6	7	8	9	10	11	12
Time taken (API in minutes)	35	34	27	30	28	22	53	24	19	28	18	20
Time taken (DMITool in minutes)	9	13	14	12	2	4	7	12	10	13	16	2
Errors in API	0	0	2	0	0	0	0	4	0	0	0	0
Errors in DMITool	0	0	0	0	1	0	0	0	0	0	0	0
Participant no.	13	14	15	16	17	18	19	20	21	22	23	24
Time taken (API in minutes)	33	43	17	56	38	50	11	12	5	29	20	30
Time taken (DMITool in minutes)	12	15	3	11	3	13	8	8	17	15	11	19
Errors in API	2	2	2	2	0	0	2	3	3	0	3	2
Errors in DMITool	0	0	0	1	0	0	0	0	0	0	0	0

$P = 0.00000056$. These results suggest that DMITool really does have an effect on efficiency to develop interaction application and this is almost three times faster with the DMITool than the average time it took with the API to achieve the same goal.

- *Assessment of effectiveness*

To assess effectiveness, errors found in the developed code were counted as displayed in Table 4. Even though the program was developed quickly, the program's completeness must also be considered, so the development time and the error count are separated and measured. That is, the program must be able to produce the correct output for an input, and we used error to measure the degree to which this is achieved. We consider both syntax errors and semantic errors. Possible syntactic errors in the API can be syntax errors in the C language (i.e. we also call them compile-time errors). Semantic errors occur when a DMI API function or feedback API is not selected appropriately for a given specification. For example, when the *IsRotated* function, which checks rotation, is used instead of *IsTranslated*, which checks whether or not an object has moved, this is considered a semantic error. A typical syntax error in the I-Typed DMML is the creation of an incomplete model, which includes errors like missing the link that shows the relation between the units selected when editing an interaction model. Similar to the DMI API, if the used I-Type is incompatible with the given specification, this is considered to be a semantic error even if it does not cause a syntax error.

Figure 27 compares the error occurrences when participants made programs with each of the two methods. Figure 27 shows that hardly any errors were found in code written with the proposed method as compared to that written with the existing method. This confirms that the proposed method provides better results in terms of completeness. In some cases, however, the task was completed within a short period of time using the proposed method. Thus, the errors shown here could be regarded as mistakes resulting from the short development time. Figure 28 shows a statistical analysis of the

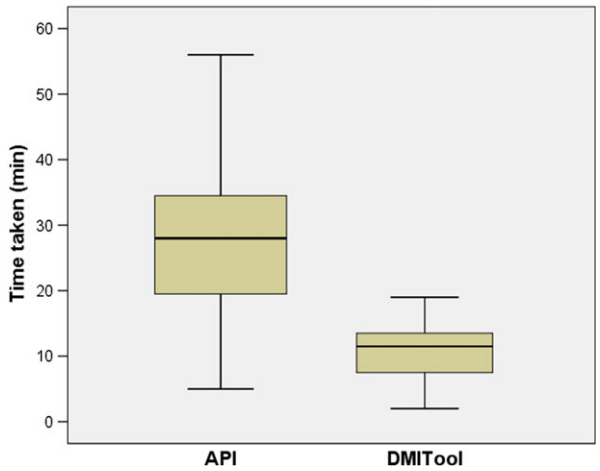

Figure 26. Boxplot of the time taken (in minutes) to carry out the task with the two methods.

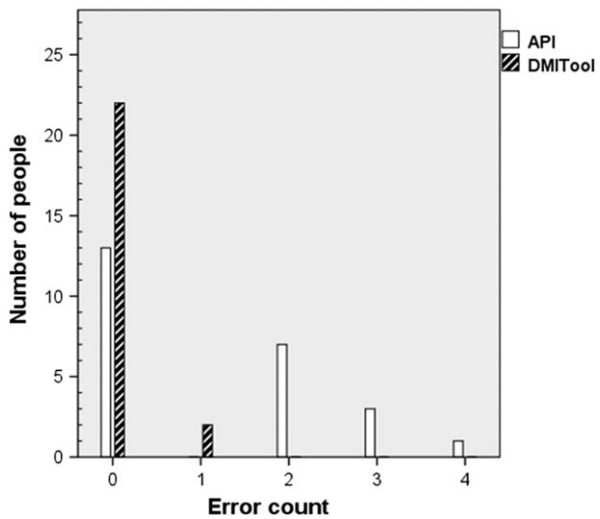

Figure 27. Comparison of the error occurrences when participants created programs with each of the two methods.

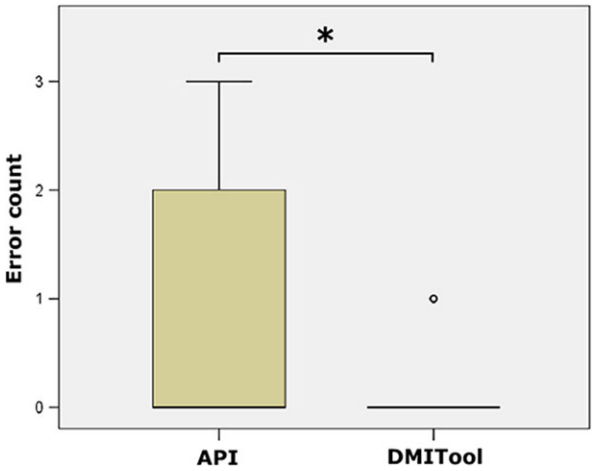

Figure 28. Boxplot of the error occurrences when participants created programs with each of the two methods.

number of error occurrences with each of the two methods. An independent-samples *t*-test was conducted to compare error count on each method. There was a significant difference in the errors for the API ($M = 1.0$, SD $= 1.17954$) and the DMITool ($M = 0.0833$, SD $= 0.28233$); $t(25.627) = 3.703$, $*P = 0.001028 < 0.05$.

- *Assessment of satisfaction*

Satisfaction was assessed through a survey after the experiment (see Table A2 in Appendix). The results of the survey demonstrate that the I-Typed DMML has a higher satisfaction rate in terms of convenience and suitability than the existing development method as indicated in Figure 29. For satisfaction with writing the application code, such as the ease of writing code, the ease of writing user feedback, and the ease of fixing problems, the results show that satisfaction with the proposed method is high, as indicated in Figs 30–32.

For satisfaction with the application code output, evaluation of the reflection of user intention, evaluation of the portability of the code and recommendation of the development method to others all show satisfactory results as displayed in Figs 33–35. The proposed method is rated as excellent since the code is portable without knowing the API, the interaction unit, which is a basic unit, is modularized, and the application developers can focus on user feedback even if they do not know the detailed algorithms. In addition, it is evaluated as satisfactory from the perspective of the developers of the perception algorithms because it is easy to replace and extend the existing perception modules.

Based on the results of the user evaluation, it is clear that the proposed method overcomes the limitations of existing DSL's shortcomings in sufficiently defining DMIs, and enables both non-specialists and experts to easily develop interactive applications. In order to discern and evaluate the limitations and possible improvements for the current iteration

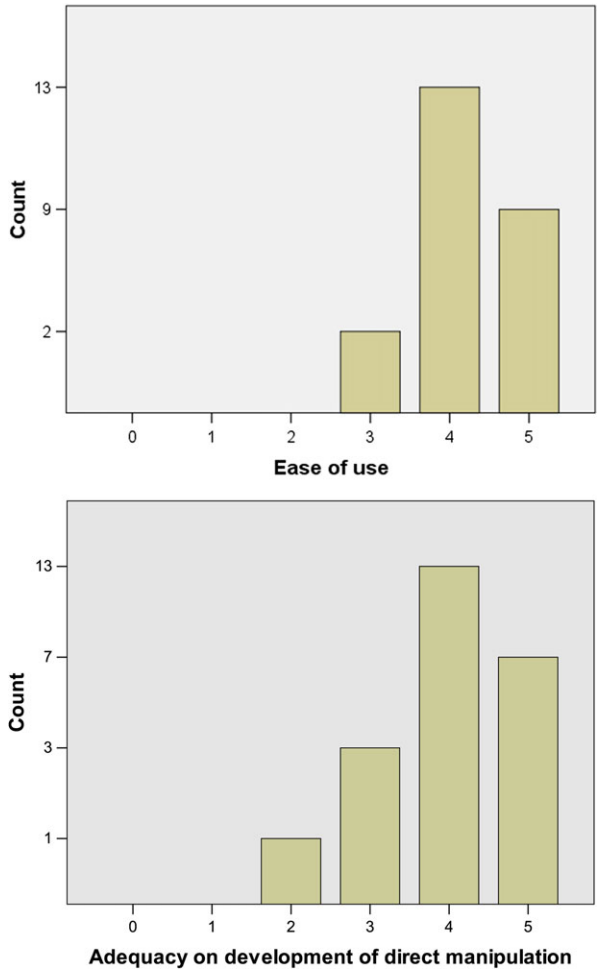

Figure 29. Overall satisfaction rating.

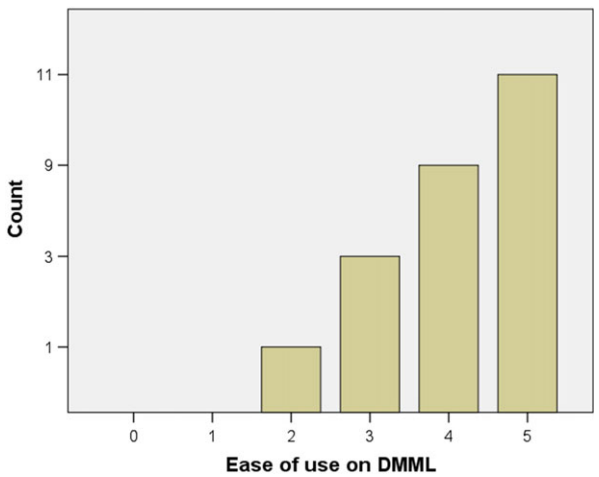

Figure 30. Ease of writing code using the I-Typed DMML.

of the I-Typed DMML and the DMI engine, we list several open questions in Table A2 within the Appendix. The following observations were made on these questions.

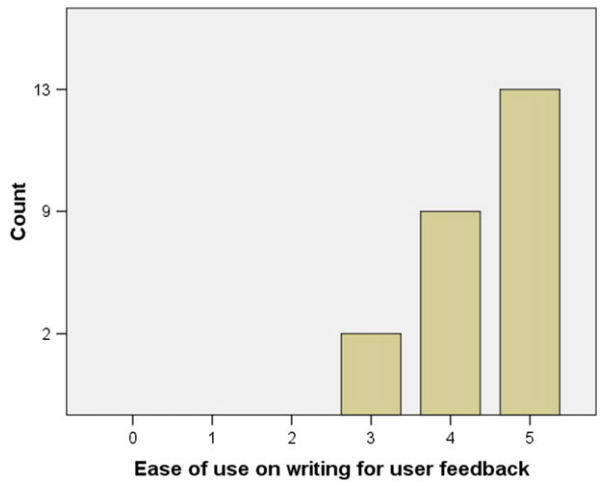

Figure 31. Ease of writing user feedback.

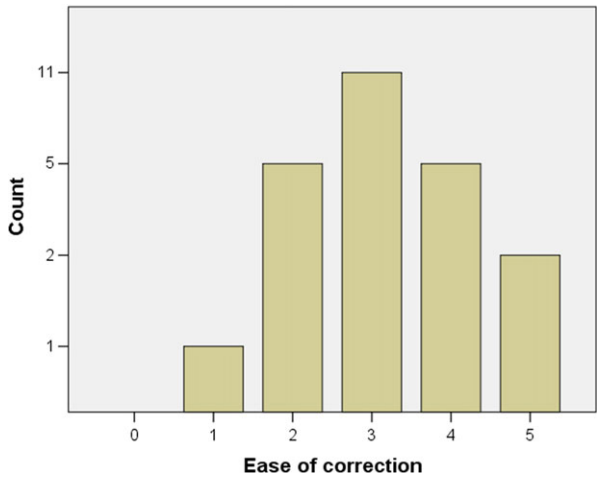

Figure 32. Ease of correcting problems.

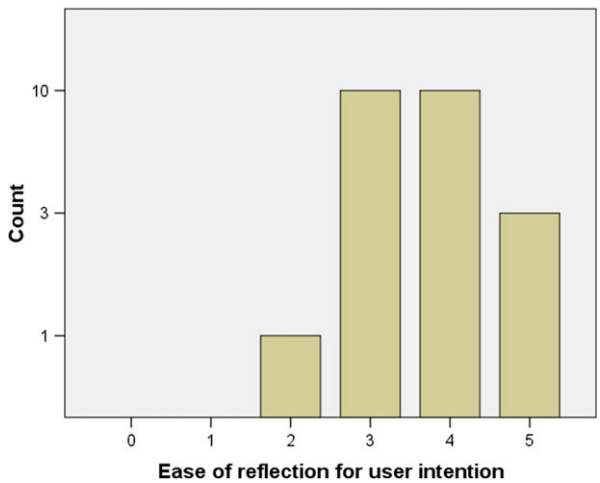

Figure 33. Ease of reflecting user intention.

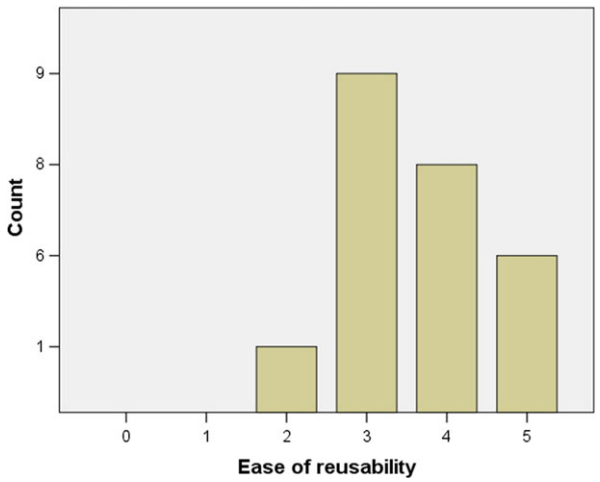

Figure 34. Ease of porting the final product.

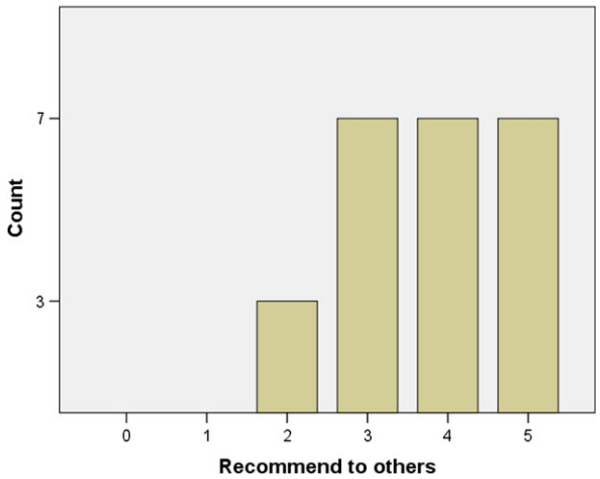

Figure 35. Preference for the proposed method.

- *Analysis results of open questions*

Q) Because the code was created for given circumstances, it is unclear how it would perform when applied to new circumstances. (**A:** *Due to the fact that the mission given in the user study was limited, it seems like there was insufficient time to entirely understand the I-Type. It will be possible to improve our understanding by directly developing a new application.*)

Q) The handler message cannot concretely know or adjust the perception unit layer's manipulation definition and conditions, so it seems that there is a need for a means of communication between the perception unit layer and application layer developers. (**A:** *This seems to be the opinion of developers who have experience in developing entire applications. This can be resolved by explicitly explaining to the developer that it is possible to adjust each of the algorithm parameters for the perception unit and the inference unit that are linked to the used I-Type.*)

Q) We think it will be easier to use if we can confirm what type of mutual connection there is with the resulting code's hierarchy. (**A:** *In the user study, we mainly perform feedback-related coding on the source template for the I-Type, however, we were unable to confirm the interaction model we designed using the DMITool. This will be understood if we demonstrate the results of saving the relationship between the perception unit and the inference unit as XML.*)

Q) Even if providing feedback is easy, it is still inadequate for implementing the desired motion. (**A:** *An API for handling virtual object motion is also provided, but this question seems to have arisen because this was not mentioned in the experiments.*)

Q) It seems that the entry barrier for using the I-Typed DMML is high. (**A:** *Developers who are accustomed to C/C++ or Java programming may experience difficulties when first learning visual programming techniques. It seems they require a bit more understanding of the development process, which creates applications through interaction modeling.*)

Q) The existing code that was developed has not been modified and new code was created, which makes it difficult to finish task 1 and perform task 2. (**A:** *The current DMITool has a problem in that it is possible to create a code template and write user code after designing the interaction model, but when a new I-Type is added to the interaction model, the code template is created again such that the existing code is not saved. This can be resolved by supporting a code saving feature in the DMITool.*)

Q) It would be good if the user feedback API list was also implemented in the interface so that there was no need to create any separate code. (**A:** *This is a good option, which can make the user development process easier. Visual programming for feedback is not supported in the current DMITool. It would be good to add visual, sound and haptic feedback features.*)

5. CONCLUSION

In this article, we propose a new method for modeling interactions with directly manipulable virtual objects and a DSL we call the I-Typed DMML, which allows for a more intuitive development of interactive applications. It is a type of DSL that transforms the I-Type-based model for DMIs into code, for programmers to develop interactions that designers intended.

The proposed method allows users to select I-Types and generate app configuration in the first step, generate application code templates in the second step, and implement feedbacks in the last step to develop interactive applications. The DMI engine supporting the I-Typed DMML is independent of the graphics rendering engine and the physics engine. It has a three-layer structure and a common interface to each unit for mediation and enhanced extensibility. The feasibility of the proposed method was shown through the development of three 3D virtual interactive applications, namely, *EduToy*, *VirtualOffice* and *VirtualChess*. We assessed the usability of the system by testing it on different user groups by developing these 3D DMI applications.

Currently in the DMI engine, all units run adequately within 33 ms, which is the same as the 30 fps data input frequency of an RGBD camera that captures the hand pose without any delay. However, due to the sequential execution of the units in the current method, if the number of units used in the application is too large or if the processing time increases within a specific unit, a propagation delay may occur. In future work, we would like to modify the FIFO structures for unit execution into parallel structures using a scheduler. This can reduce the delays propagated by units and affect the task completion time. Furthermore, we would like to test the method on widely used HMDs instead of 3D display environments. Using popular HMD devices removes the restrictions in user movements and sets users free to interact with virtual objects. Finally, we would like to enable bimanual interactions and interpersonal interactions using our I-Type-based method.

SUPPLEMENTARY MATERIAL

Supplementary data is available at *Interacting with Computers* online.

FUNDING

Global Frontier R&D Program on 'Human-centered Interaction for Coexistence' funded by the National Research Foundation of Korea grant funded by the Korean Government (MSIP) (2011-0031425).

REFERENCES

3Gear. (2017) http://threegearsystems.blogspot.com/ (visited in April 2017).

Ammar, L.B. and Mahfoudhi, A. (2013, June) Usability driven model transformation. Human System Interaction (HSI), 2013 the 6th International Conference on, pp. 110–116. IEEE.

Anzalone, D., Manca, M., Paternò, F. and Santoro, C. (2015, June) Responsive task modelling. Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 126–131. ACM.

Bevan, N. (2000) ISO and industry standards for user centered design. Retrieved November 23, 2010.

Blouin, A. and Beaudoux, O. (2010, June) Improving modularity and usability of interactive systems with Malai. Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 115–124. ACM.

Bullet physics. (2017) http://bulletphysics.org/wordpress/ (visited in April 2017).

CreativeZone/shutterstock.com. (2016) http://www.shutterstock.com/ko/video/clip-4436051.

Csisinko, M. and Kaufmann, H. (2010) Vital-the virtual environment interaction technique abstraction layer. Proceedings of the IEEE Virtual Reality 2010 Workshop: Software Engineering and Architectures for Realtime Interactive Systems, pp. 77–86.

Duval, T., Blouin, A. and Jézéquel, J.M. (2014, March) When model driven engineering meets virtual reality: feedback from application to the Collaviz framework. Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2014 IEEE 7th Workshop on, pp. 27–34. IEEE.

Figueroa, P., Green, M. and Hoover, H.J. (2002, February) InTml: a description language for VR applications. Proceedings of the Seventh International Conference on 3D Web Technology, pp. 53–58. ACM.

GISMO's Tech. report (2017) https://github.com/tommens/gismo/blob/master/techreport.pdf (visited in February 2018).

GISMO's Wiki (2017) https://github.com/tommens/gismo/wiki/GISMO (visited in February 2018).

Hololens (2018) https://www.microsoft.com/en-us/hololens/ (visited in April 2018).

Jung, B., Lenk, M. and Vitzthum, A. (2015) Structured development of 3D applications: round-trip engineering in interdisciplinary teams. Comput. Sci. Res. Dev., 30, 285–301.

Khaddam, I., Mezhoudi, N. and Vanderdonckt, J. (2015, March) Adapt-first: a MDE transformation approach for supporting user interface adaptation. Web Applications and Networking (WSWAN), 2015 2nd World Symposium on, pp. 1–9. IEEE.

Leap motion (2017) https://www.leapmotion.com/ visited in April 2017.

Lenk, M., Vitzthum, A. and Jung, B. (2012, August) Model-driven iterative development of 3D web-applications using SSIML, X3D and JavaScript. Proceedings of the 17th International Conference on 3D Web Technology, pp. 161–169. ACM.

MacKenzie, I.S. (2012) Human-Computer Interaction: An Empirical Research Perspective. Elsevier, Newnes.

Martinie, C., Palanque, P. and Winckler, M. (2011, September) Structuring and composition mechanisms to address scalability issues in task models. IFIP Conference on Human-Computer Interaction, pp. 589–609. Springer, Berlin, Heidelberg.

Mauney, D., Howarth, J., Wirtanen, A. and Capra, M. (2010, April) Cultural similarities and differences in user-defined gestures for touchscreen user interfaces. CHI'10 Extended Abstracts on Human Factors in Computing Systems, pp. 4015–4020. ACM.

Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H. and Wimmer, M. (2014, September) ProMoBox: a framework for generating domain-specific property languages. International Conference on Software Language Engineering, pp. 1–20. Springer, Cham.

Oculus Rift (2017) https://www3.oculus.com/en-us/rift/ (visited in April 2017).

Paternò, F. (2004) ConcurTaskTrees: an engineered notation for task models. The Handbook of Task Analysis for Human-Computer Interaction, pp. 483–503.

Perception Neuron. (2017) https://neuronmocap.com/ (visited in April 2017).

Physics Abstraction Layer. (2017) http://www.adrianboeing.com/pal/index.html (visited in April 2017).

Pleuss, A., Wollny, S. and Botterweck, G. (2013, June) Model-driven development and evolution of customized user interfaces. Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 13–22. ACM.

Popp, R., Falb, J., Raneburger, D. and Kaindl, H. (2012, June) A Transformation Engine for Model-driven UI Generation. Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 281–286. ACM.

Raneburger, D. (2010, June) Interactive model driven graphical user interface generation. Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 321–324. ACM.

Raneburger, D., Kaindl, H. and Popp, R. (2015, June) Model transformation rules for customization of multi-device graphical user interfaces. Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 100–109. ACM.

Raneburger, D., Popp, R., Kaindl, H., Falb, J. and Ertl, D. (2011, June) Automated generation of device-specific WIMP UIs: weaving of structural and behavioral models. Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 41–46. ACM.

Raneburger, D., Popp, R. and Vanderdonckt, J. (2012, June) An automated layout approach for model-driven WIMP-UI generation. Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 91–100. ACM.

Romuald, D. and Mens, T. (2015, June) GISMO: a domain-specific modelling language for executable prototyping of gestural interaction. Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 34–43. ACM.

Schmidt, D.C. (2006) Model-driven engineering. Comput. IEEE Comput. Soc., 39, 25.

Valkov, D., Bolte, B., Bruder, G. and Steinicke, F. (2012, March). Viargo—a generic virtual reality interaction library. Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2012 5th Workshop on, pp. 23–28. IEEE.

Vitzthum, A. and Pleuß, A. (2005, March) SSIML: designing structure and application integration of 3D scenes. Proceedings of the Tenth International Conference on 3D Web Technology, pp. 9–17. ACM.