Insights on the Design of InTml

Abstract

This paper describes some details about the design of InTml, the interaction techniques markup language. We explain three main elements in its architecture: a simple mixed reality (MR) based component model, a communication model between components that allows fusion of multimodal information at a fine level of granularity, and an indirection mechanism for dataflows that is useful to keep state inside a dataflow. We also briefly discuss the advantages we have found in the use of formal methods, model driven development, and encapsulation mechanisms. The purpose of this description is to make explicit the design rationale of these mechanisms, which may be fruitful for other developments in our field.

1 Introduction

The field of mixed reality (MR) has a huge potential in the near future. Several display technologies are emerging, from large tiled displays to small flexible screens; 3D technologies are now in movie theaters and bundled with consumer-level graphic cards; and novel input devices such as the Wiimote are in wide usage. Novel interfaces such as the Microsoft Surface promise a revolution in the way people interact with information, beyond the WIMP paradigm. Moreover, novel CPU/GPU architectures promise more multithreaded computing power for richer MR experiences. All these possibilities may daunt both developers and users, since such a variety of technologies may create a Babel of solutions which may preclude integration and overall usability. However, it is also an opportunity to explore novel development schemes that tackle such a complex scenario.

Our interest is to facilitate the development of families of rich MR experiences, in which interaction with several input and output devices should be taken into account. However, in order to integrate hardware solutions, we require software technologies that hide part of their complexity and facilitate development. We require software environments that allow us to move applications from one platform to another by accordingly change devices, interaction techniques, and content. Our research around InTml (interaction techniques markup language) targets these goals. InTml (Figueroa, Bischof, Boulanger, Hoover, & Taylor, 2008; Figueroa, Green, & Hoover, 2002) is an architecture, a language, and a set of tools that allow developers to create applications that use novel input and output devices. It is designed to allow retargeting of such applications to new scenarios of usability and hardware. We took into consideration foundational ideas in the fields of MR and software engineering (i.e., dataflows, scene graphs, software product lines, component based development, model driven architectures) and we created a development
environment that fulfills our stated goals. The result is a set of tools that we are using for several developments in MR, and an open software infrastructure that can be used over a wide variety of configurations and applications, as has been reported elsewhere.

Although InTml has been described before, this paper emphasizes unique details in its development, which may be extrapolated to other systems. We use in this description the concept of design patterns (Gamma, Helm, Johnson, & Vlissides, 1994), which describes both a common problem and its proved solution in a particular domain. Our presentation tries to follow the de facto standard, in which each pattern is described with a name, a description of the problem it tackles, a description of the proposed solution, expected consequences or trade-offs, and known uses. Although we believe our patterns should be further evaluated in other scenarios and from different points of view such as performance and memory management, this presentation scheme allows us to clearly present and divide the main concepts we have used. In general, we believe it is an interesting way to present solutions to problems in our field that invites further discussion.

First in this paper, we present three design patterns, DeviceObjectBehavior, InTmlDataflow, and ComponentHolder. Later, we briefly describe how we have used the concepts of formal models, model driven development, and encapsulation mechanisms in this development. Finally, we present some conclusions and future work.

2 DeviceObjectBehavior

2.1 Intent

Organize an MR application in terms of just devices, objects, and behavior, for simplicity and encapsulation reasons.

2.2 Motivation

Any MR application has some graphic assets (i.e., graphical representations of widgets or objects in a scene), audio clips, haptic effects, input and output devices, and application logic. The way these elements are represented is platform-specific, with some common solutions. For example, there is usually a scene graph for graphic assets and a set of callbacks for event gathering from input devices. However, there are few recommendations in existing toolkits on how to structure code for interaction techniques or application control. Common architectures in WIMP architectures such as model-view-controller (MVC) (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996, p. 125) could be useful, but use of WIMP architectures in the MR field requires careful design and further development due to the lack of a standard set of view objects. Moreover, if changes to devices or interaction techniques should be accomplished, there is little support in current architectures to facilitate such an endeavor. Other abstract models have been presented, such as SSIML/AR (Vizthum, 2006), APRIL (Haller, Billinghurst, & Bruce, 2007, p. 145), and DART (Haller et al., p. 163). SSIML proposes a model with virtual, real, and hybrid objects. Our model is simpler in the sense that we just consider one type of object that may be any of these three types, since we have not found a necessity to differentiate them. APRIL’s model considers extra concepts such as story and interactions, which can be seen as more complex constructions than the ones we show here, and DART uses Director’s model, which was not aimed from the start at the field of MR.

2.3 Applicability

Use this pattern in order to easily identify content objects, devices, or behavior in an application, for reusability or adaptation purposes.

2.4 Structure

The general structure of the DeviceObjectBehavior pattern is shown in Figure 1.

2.5 Participants

The DeviceObjectBehavior pattern has the following participants: Component, InteractiveObject,
Device, Behavior, CompositeComponent, and Application. Component abstractly represents any type of component. InteractiveObject represents an interactive object in a scene, with its set of modifiers and expected queries. Device represents a particular physical device, which may have input and output functionality. It provides methods for device configuration and methods for changing its output state. Behavior is a particular function or logic for a particular interaction technique. It has methods for setting parameters of such a behavior. CompositeComponent is a collection of components that represent a full interaction technique, a complex device, or a complex behavior, for example. Finally, Application is a particular type of composite component that represents an entire application. These classes are structured as in the classic Composite pattern, with InteractiveObject, Device, and Behavior as leafs. There might be more than one class in each concrete role in a particular implementation.

### 2.6 Collaborations

Each concrete subclass of DeviceObjectBehavior encapsulates the particular mechanisms and APIs to communicate with its corresponding physical device. Also, if the implementation of several devices share common mechanisms for accessing events, there should be an implicit collaboration scheme. Section 2.10 (Related Patterns) links to communication patterns between components. CompositeComponent can hide some behavior in its parts, for the sake of simplicity.

### 2.7 Consequences

The consequences of DeviceObjectBehavior are as follows.
Figure 2. Implementation of the DeviceObjectBehavior pattern.

- Changes in the scene graph technology do not affect the rest of the software. This can be used for migration from one scene graph to another or for supporting several scene graph technologies.
- Device identification. It is straightforward in this structure to identify and replace devices, for maintenance or retargeting purposes.
- This structure can be combined with several patterns for communication. The simplest way is by means of Observer, in order to propagate events from devices to other interested parties. MVC is another option that defines a more organized communication scheme, in which there is an inner model, interactive content classes fulfill the view role, and devices and behavior classes fulfill the control role. We also propose another mechanism with extra features, in the IntmlDataflow pattern.
- In order to apply local transformations to objects, this pattern requires that every object is preceded in the scenegraph by a transform node.

2.8 Implementation

A DeviceObjectBehavior implementation in a particular application is shown in Figure 2. There are two interactive object types, Ray and SimpleObject, two devices (Gamepad and Fob2Tracker), two behaviors (SceneLoader and RaySelection), and a composite component, SelectByRayIT.

Ray is the visible representation of a ray, which is useful for ray casting based selection. SimpleObject allows a 3D object to move in the space and show a bounding box, which is useful for selection. Gamepad models any gamepad with a variable number of buttons and axes. Fob2Tracker represents a flock of birds with two trackers. Although it is a particular device and cannot be generalized, it is very useful for end users since it allows easy identification of a device. SceneLoader is a component that loads a file with a scene graph and retrieves all named objects. RaySelection computes a selected object given a ray and a set of selectable objects. Finally, SelectByRayIT is a composite component that combines a Ray, a set of SimpleObjects, and a RaySelection component in order to allow ray casting selection.

1. The inner model could be a model for a simulation, or the basic scene graph for common MR applications.
As an example of multiple scene graphs, our C++ implementation uses an AbstractFactory in order to support both Performer and OpenSG. Our Java implementation uses Java3D as the scene graph, and both implementations connect to devices through VRPN.

2.9 Known Uses

All families of applications we have created follow the DeviceObjectBehavior pattern. For example, an application for learning about machine tools (Toro et al., 2009) was implemented in both a PC and a tiled wall environments, and we successfully replaced both devices and interaction techniques in the PC (mouse, keyboard, standard display, ray casting) for the ones in the tiled wall (wireless gamepad, tiled display, iteration over selectable pieces). We have also shown (Figueroa, Ferreira, & Castro, 2007) several ways in which these structures can be beneficial, that is, by separating the components that fulfill each application’s task or by implementing very different interaction techniques for the same task in different hardware environments.

2.10 Related Patterns

We have used InTmlDataflow for the communication scheme between instances in this structure, either inside a CompositeComponent or an Application. We believe other schemes such as MVC or Observer can be used, with different results in the execution model.

3 InTmlDataflow

3.1 Intent

Provide ways to fuse events from devices or other components in a dataflow.2

2. Fusion is understood in this context as the integration of events that are complementary or redundant. An example is shown in the seminal work “Put-That-There” (Bolt, 1980).

3.2 Motivation

Let us assume there is an MR application with several devices, interactive objects, and behaviors, which are modeled by means of DeviceObjectBehavior. In addition, let us assume it is a requirement to update output devices at a rate at least as high as the one of the fastest device, and this application is aimed to a target machine with several CPU cores. Such a scenario requires ways to execute the application’s behavior at a high frame rate, and ideally to make use of the available CPUs. There should also be communication between these components in order to implement the application’s functionality, and coordination in order to ensure deterministic execution when components execute in parallel.

Callbacks tie events to their possible responses: a general query, changes in the application’s state, changes in the virtual object’s state, or changes in the current dialog with the user. However, this solution, although simple, has the following disadvantages.

• This code is called each time an event is generated, independent of the actual rate of the simulation or output devices. This does not account for cases with too many events, in which case it is necessary to discard or filter some of them.
• Code for event fusion, or the generation of complex events as a pattern of simpler events, has to be written by developers in a case-by-case scenario and across several callbacks.
• It is difficult to parallelize this type of code by receiving events in different threads, since application and object states change everywhere, and it is not possible to ensure a deterministic behavior.
• If developers would like to try a new input device, they will have to write code in a different callback in order to capture other events.
• If developers would like to change an interaction technique, they will have to look for the appropriate callbacks for adding or deleting behavior.
• If developers would like to change the way objects respond to events in the system, they will have to find the particular object’s changes in several callbacks.
In software engineering terms, this solution tightly couples the reception of input events and its response in terms of state changes in objects, interaction dialog, and control. This can be solved by decoupling the connections between these components. Looking at current solutions, we can use either Observer or MVC as communication patterns, although there are no provisions for multithreading of events and it could be difficult to create chains of connections between several components.

In terms of fusion, Touraine, Bourdot, Bellik, and Bolot (2002) present a central manager instead of our approach of fusion per component. There are more complex models such as the one in Latoschik (2005), which includes knowledge interpretation for event fusion, although this could also be done at each component.

Our solution is structured as a dataflow of components with possible loops, with a particular execution model that fulfills two main goals at each execution frame: all components see the same application state and each component has all the required input information when its time for execution arrives. These conditions allow us to execute in parallel some components in a deterministic way. The dataflow architectural pattern, or Pipes and Filters in Buschmann et al. (1996, p. 53), has been used in several systems for VR applications, such as VRML (Carey & Bell, 1997), X3D (Web3D Consortium, 2003), Performer (SGI, 2009), PMIW (Jacob, Deligiannidis, & Morrison, 1999), Lightning (Blach, Landauer, Rösch, & Simon, 1998), Avango (Kuck, Wind, Riege, & Bogen, 2008), or Virtools (Virtools 2007), to name just a few. However, there are some limitations in current solutions. With the exception of the first version of Avango, execution models do not guarantee a common application state for all components throughout a concurrent execution, although Avango is based on a shared total message ordering among all machines, which may affect performance. Loops in dataflows are usually not supported, or they are special cases that have to be treated with special care. Finally, with very few exceptions such as Avango and Lightning, execution models do not guarantee that a component has all required input events before its execution, which precludes event fusion.

3.3 Applicability

It is appropriate to use InTmlDataflow when your MR application should fuse information from several input devices, and you would like to run components in parallel in several CPUs.

3.4 Structure

The general structure of the InTmlDataflow pattern is shown in Figure 3.

3.5 Participants

Component represents any type of component with three responsibilities: collect information, compute output events based on input information, and send output events to interested components. StatelessComponent is a component whose state remains the same as it computes its output. In other words, given certain input information and an initial state, it computes the requested output information. Behavior classes in the DeviceObjectBehavior pattern are usually stateless components. A statefulComponent may change its internal state in its computation. In other words, given certain input information and an internal state, it computes a new internal state (different from the previous one) and output information. InteractiveObject classes in the DeviceObjectBehavior pattern are usually stateful components. SourceComponent represents a stateless source of information from the outside. Device classes in the DeviceObjectBehavior pattern are usually source components. InputPort and OutputPort represent a

3. Virtools’ dataflow is indeed a control flow, since a unique connection between components defines order of execution, and information is implicitly shared between components.

4. We consider here an initial state in order to allow configuration parameters.
component’s input and output ports, that allow it to receive events from or send events to other components. They inherit from the abstract class `Port`, which can collect several events and delete them. `Application` is a collection of interconnected components.

### 3.6 Collaborations

An application with the `InTmlDataflow` pattern runs the algorithm in Algorithm 1.

Algorithm 1: Dataflow’s Execution

1. `preprocess();` // Order components for execution
2. `// It computes the possible threads of execution`  
3. `While( !exit )`  
4. `dataGathering();` // Execute `dataGathering` from source components  
5. `execStatelessDataflow();` // execute threads of stateless components  
6. `execStatefulComponents();` // Manage state changes  
7. `rendering();`

If the dataflow is not executed in one step, it is possible to delay the `rendering` operation until all components have been executed. The number of steps required to run all components at least once depends on the number of cycles and objects and their connections. Other characteristics are the following.

- Connections between component’s ports define a dataflow.
- Information through ports may be typed, in order to facilitate initial checking of the dataflow.
- The execution of the first four methods in an application (preprocess, dataGathering, execStatelessDataflow, execStatefulComponents) define a step. Any component executes at most once in each step, and it is possible to require several steps in order to execute all components in the dataflow.
- The application precomputes an order between components in the dataflow so each component is executed at most once in any time frame.
- Loops in the dataflow with any type of components are allowed, although they require at least two execution frames in order to reach all connections.

---

**Figure 3.** Class structure for the InTmlDataflow pattern.
• The execStatelessDataflow operation may execute several chains of interconnected stateless components. Once the chain of execution reaches a stateful component it stops, and the propagation of events resumes in execStatefulComponents. Each chain of interconnected stateless components may run in different threads.

• The execStatefulComponents operation may involve some kind of coordination between stateful components, if they are somehow related, that is, in a scene graph.

• The execution of the methods in an application may be pipelined, if there are available CPUs.

• All unprocessed information in components involved in a step is discarded after they run. This allows a mechanism for handling just the right amount of information.

3.7 Consequences

The consequences of the InTmlDataflow pattern are as follows.

• An application can define a common rate of execution, given the execution speed of its operations.

• Any component can fuse the information in all its input ports before it executes. This code can be written just once in the collect() method at the Component class, so it can be reused for all subtypes.

• Components are isolated between them, so they are independent. Ports allow a common mechanism for communication. No shared memory mechanism is used.

• The execution of a component does not depend on the number of events received in its input ports, as is common in other dataflow models.

• The particular technologies for accessing physical devices are encapsulated.

• Instances from SourceComponent and StatelessComponent can be executed in parallel, depending on their connections in the dataflow.

• All stateless components executing in a step see the same state of the application, which is represented in stateful components.

• It is trivial to identify and replace a source of information, an interaction technique, or a behavior, since they correspond to a component in this model.

3.8 Implementation

Our implementations of InTmlDataflow add input and output ports to the class Filter in Figure 2. In the C++ implementation, InputPort and OutputPort are templates, so it is possible to check type correspondence of connections at compile time. The Java implementation uses generic collections in these two classes, so there is no type checking between connections. The execution in both environments has the following features and limitations:

• Cycles are identified and broken. Special components are created in order to propagate events between execution steps.

• Components in the dataflow are organized by means of the topological sort algorithm (Cormen, Leiserson, Rivest, & Stein, n.d., p. 549). This is a sequential order of components \((c_1, \ldots, c_n)\), in which it is ensured that all components after \(c_i\) do not have output connections to it.

• There is only one thread of execution, for the components in the topological sort, and we are working on a parallel version in the C++ based environment.

• The source components are physical input devices and a timer. They collect information by means of implementation dependent APIs and produce events through their output ports.

• The only stateful components are interactive objects. They currently represent independent leaves in a scene graph.

• Event fusion is allowed in a method called collect(), which is called for each component after all events in a time frame have been collected and before its main behavior is executed. Within this method, developers can handle redundant or complementary events.
• We implemented in the C++ version three generic policies to deal with multiple events from one source in a time frame: pick any, which is the simplest but indeterministic policy, pick average, which computes the average of all input events, and pick median.

3.9 Known Uses

InTmlDataflow is at the core of all implementations of InTml in a native language, that is, C++, Java, or ActionScript. We have used fusion in order to correlate inputs from several devices, that is, keyboards and trackers, or gamepads and trackers, among others.

3.10 Related Patterns

InTmlDataflow has been used together with the DeviceObjectBehavior pattern. It is possible that the execution of the entire InTml dataflow as it is defined up to this point does not meet the special requirements in throughput of a particular output device. In this case, a decoupling technique for slow components such as the one in Shaw, Liang, Green, and Sun (1992), plus an interpolation and prediction schemes for missing output values, could provide a solution that allows the entire dataflow to run at the fastest frame rate required.

4 ComponentHolder

4.1 Intent

Allow dynamic changes in a dataflow, in order to keep references to important components. In particular, dataflows are allowed to refer to a dynamically changed object.

4.2 Motivation

In the ComponentHolder pattern, an MR application may require ways to refer to the current and previous selected objects, in order to modify a particular characteristic, that is, their visibility, their bounding boxes’ visibility, or their color. In the context of components in an InTmlDataflow, it is desirable to count with a way to refer to such objects, so that behaviors can just send messages to them in order to modify their characteristics. However, any interactive object could be selected at a particular time in the interaction, and the dataflow in InTmlDataflow does not provide ways to refer to the current selection in just one point.

Dataflows in other systems also lack a way to refer to the current selected object, and workarounds should be used. For example, it is possible to create several instances of prototypes in VRML in order to represent a selected object and the manipulation tools around it, but this requires redundancy, a mechanism for activating one instance at a time, and more memory load. Other systems may use dynamic rerouting of the dataflow, but it is a complex and error prone technique. Finally, other systems may use a hybrid solution, in which the selected object is defined as a shared state value outside of the dataflow, which precludes modularization.

4.3 Applicability

In the context of the InTmlDataflow pattern, use the ComponentHolder pattern as an indirection mechanism for components. In other words, use it to refer to a concept that can be fulfilled by a particular component in the dataflow during execution.

4.4 Structure

ComponentHolder is an extension of a Component in InTmlDataflow. The classes that are involved in the new behavior are shown in Figure 4. Its structure is similar to the Decorator pattern, with ComponentHolder as Decorator. 5

5. We chose to classify ComponentHolder as a Decorator instead of a Proxy since ComponentHolder can add behavior to its contained object. For example, in our C++ implementation, ComponentHolder delays all input events one frame in order to ensure that those events will be directed to the right object, in the case of changes of the contained object.
4.5 Participants

In the ComponentHolder pattern, StatefulComponent, InputPort, OutputPort, and Application are defined as they are in InTmlDataflow, with the exception that Application can now include component holders. A ComponentHolder is a decorator for a stateful component. It contains a component in an application, and it has two special ports: an input port setComp that allows changes for the contained component (comp), and an output port compChanged that reports changes to the contained component. Other input and output ports should be mapped to the input and output ports of the contained component.

4.6 Collaborations

The collaborations for the ComponentHolder pattern are as follows.

- ComponentHolder’s input and output ports are defined at the creation of an application. There should be an implementation dependent method for mapping these ports to the ones of the current contained component.
- The standard operation of a component holder happens when its contained component (comp) is not changed, that is, there are no events in its setComp input port. In this case, it propagates events in its input ports to the mapped ones in comp, and it propagates events in comp’s output ports to the components connected to its output ports, once comp is executed. In terms of the methods in Application, propagation of events in input ports happens in execCHInputs(), while propagation of events in output ports is done in execCHOOutputs().
- When comp has to be changed, a component holder first replaces the mapping between its ports and
the contained component by releasing the previous associations and creating new ones to the new contained component. Then it executes its standard operation.

- The execution of component holders in an application is apart from other components, between stateless components and stateful ones. In this way, events received during a step of execution are directed to the new contained component.

### 4.7 Consequences

The consequences of the ComponentHolder pattern are as follows.

- Component holders look like other components in an application. They are placeholders for any concept, that is, the current selection.
- An application’s dataflow is dynamically changed every time a contained component is changed, since new connections are made to such a component.
- Stateful components can flow through the application’s dataflow. In this sense, components become information that can be manipulated by an application.

### 4.8 Implementation

In the ComponentHolder pattern, both C++ and Java implementations define an ObjectHolder class as a Filter’s subclass, as shown Figure 2. An ObjectHolder creates ports when a new connection is requested, and mapping between its port and the ones in the contained component are solved by matching port names. In current implementations, execCHOutputs is done inside execStatelessDataflow, with a provision that allows ObjectHolders to postpone output events for the next step.

### 5 General Comments

We would like to point out three additional comments related to our development: the use of formal models in order to clearly describe the structure and operation of our model, the use of model driven development and its tools as a foundational development process, and the use of mechanisms for hiding complexity during design.

InTml defines a model for rich MR experiences, and it defines what kind of entities an application may have and how such entities behave. Although we still have to implement a wider set of applications in order to evaluate the convenience of the entities for the entire MR domain, it is important to know in advance how such an abstract machine will behave. This is more important if it is desirable to have several implementations in different programming languages, as is the case here, since the formal model provides a common blueprint for several implementations. We defined a model in Z (Spivey, 1992), a formal language that allows for the description of the structure and dynamic behavior of a model. Although we did not have at that time a theorem proof at hand, it was possible to evaluate and tune our formal model by means of expert peer reviews. For example, we validated dynamic elements such as the dataflow execution against desired features, and we could also envision novel behavior that could be implemented in the future, such as dynamic changes to the dataflow. Such a model has been the foundation for new implementations and new generic ideas in the implementation.

Model driven development (MDD; Stahl & Veolter, 2006) is a methodology for application development that concentrates on the development of domain specific models, in which solutions to requirements are described in a more natural way than by using a general purpose programming language. Concepts in the domain model are known to users in such a domain, and for that reason they are more understandable and easy to combine in a solution. Such solutions can be later translated into particular execution environments, and it is possible to have several implementations by means of additional translators from an abstract model. InTml becomes our MR specific language, in which solutions to requirements are expressed. Moreover, due to the popularity of MDD, it is possible to find tool support for the implementation of enriched IDEs.
(integrated development environments). For example, we have used tools such as eclipse (Eclipse Foundation, http://www.eclipse.org), its Graphical Modeling Framework (GMF; Eclipse Foundation, 2009), and openArchitectureWare (openArchitectureWare.org, 2008) in order to create a visual programming environment that allows developers to instantiate components or create new ones. Currently, these models can be executed in three different implementations, in Java, C++, or ActionScript, and other implementations can be created by the support of these tools.

Finally, mechanisms for hiding complexity are very useful for designers and improve designers productivity. They allow the creation of complex applications without looking at all the complexity at once. In our case, we used two mechanisms: composite components, in order to hide reusable subsets of interconnected components, and task views, in order to show just the subset of components within a dataflow that were related to a particular problem. Although more work has to be done in order to support such concepts in our visual programming environment, they allow designers to easily understand an application as the sum of its views and hide complexity inside composite components.

6 Conclusions and Future Work

We have shown as design patterns the most important design decisions we used in InTml, the interaction techniques markup language: a structure that clearly identifies components that represent content, devices, and behavior in an MR application; a specially designed execution model for a dataflow of these components that takes into account parallel execution and fusion of events; and a controlled mechanism for dynamic changes in the dataflow, which is aimed to selectable objects. We have also mentioned the importance of formal methods, MDD techniques, and encapsulation mechanisms in our research. With this development we have created a domain specific language that allows us to develop rich MR experiences, in which users are surrounded by several input and output devices. We hope this description of the main architectural decisions by means of design patterns and general lessons allows our community to better understand the decisions we have made, and use these ideas in other contexts.

We plan to continue the development of InTml as a common language for MR development in different hardware platforms, and in this direction we aim to create mechanisms to guide developers in the use and implementation of InTml components, a common library of components for the hardware platforms in our lab, more implementations in target platforms such as X3D and Virtools, and demo applications for both components and hardware platforms. We hope this future development will allow us to identify guidelines and best practices for each hardware platform.

References


