Modelling Behaviour of Configurable VR Applications
Krzysztof Walczak
Creation of complex behaviour-rich and meaningful content is one of the main difficulties that currently limit wide use of virtual reality technologies in everyday applications. To enable widespread use of VR applications new methods of content creation must be developed. In this paper, we propose a novel approach to designing behaviour-rich virtual reality applications, called Flex-VR. The approach enables building configurable VR applications, in which content can be easily created and modified by domain experts or even common users without knowledge about VR design and computer programming. The VR content is configured from reusable programmable content elements, called VR-Beans. Appearance and behaviour of the VR-Beans are controlled by scripts programmed in a novel high-level language, called VR-BML (Behaviour Modelling Language). The language enables specification of generic behaviours of objects that can be dynamically composed into virtual scenes. The paper introduces the Flex-VR component and content models, describes the VR-BML language and provides an example of a Flex-VR application in the cultural heritage domain.
1. INTRODUCTION

Virtual reality technologies have reached the level of maturity that makes possible their use in a diversity of real-life applications. Architecture, cultural heritage, education, training and entertainment are notable examples of application domains that can largely benefit from the use of VR technologies – both immersive VR and desktop-VR.

Wider use of VR applications has been recently enabled by several factors: significant progress in hardware performance including cheap 3D accelerators present in almost every contemporary graphics card, development of platform-independent 3D content description standards, and availability of broadband multi-purpose communication networks that are able to deliver large amounts of data required by network-based VR applications. Increasing interest can be observed in exploitation of possibilities offered by lightweight desktop-VR applications implemented through 3D web content accessible on-line. Such applications do not require any special hardware or software investments and can be accessed both locally and remotely using a variety of devices.

However, despite the technical possibilities, the actual uptake of 3D applications on the web is still very low. One of the main problems that currently limit wide use of 3D applications on everyday basis is related to the creation of complex interactive behaviour-rich content.

Increasingly interest can be observed in exploitation of possibilities offered by lightweight desktop-VR applications implemented through 3D web content accessible on-line. Such applications do not require any special hardware or software investments and can be accessed both locally and remotely using a variety of devices.

However, despite the technical possibilities, the actual uptake of 3D applications on the web is still very low. One of the main problems that currently limit wide use of 3D applications on everyday basis is related to the creation of complex interactive behaviour-rich content.

Practical 3D web applications require enormous amounts of complex interactive content. Moreover – in most cases – the content must be created by domain experts (e.g., museum curators), who cannot be expected to have experience in computer programming or 3D content design. However, only involvement of domain experts guarantees sufficient amount of high-quality content, which may contribute to wider adoption of 3D applications in everyday use.

This motivation lead to the development of the Flex-VR approach to building VR applications, in which content can be relatively easily created and modified by domain experts and common users [12, 13]. In Flex-VR, the VR application content is dynamically configured from reusable programmable content elements, called VR-Beans. Appearance and behaviour of the VR-Beans are controlled by high-level behaviour scripts.

In this paper, we describe the main elements of the Flex-VR approach: the component model, the content model and a novel behaviour programming language, called VR-BML, which is used to program the behaviour scripts. The language enables specification of generic behaviours of objects that can be later dynamically composed into technically correct virtual scenes without additional programming.

The remainder of this paper is organized as follows. Section 2 provides an overview of the state of the art in programming of VR content behaviour. Section 3 contains an overview of the Flex-VR component model. In Section 4,
the Flex-VR content model is presented. Section 5 provides a detailed description of the VR-BML language. Section 6 provides an example of a practical application of the Flex-VR approach and the VR-BML language. Finally, Section 7 concludes the paper and indicates future works.

2. STATE OF THE ART

Significant research effort has been invested in the development of methods, languages and tools for programming behaviour of virtual reality content. These approaches can be classified into three main groups.

The first group constitute scripting languages for describing behaviour of virtual scenes. An example of a scripting language designed for programming VR interaction scenarios is MPML-VR (Multimodal Presentation Markup Language for VR) [9]. Another interesting example is VEML (Virtual Environment Markup Language) based on the concept of atomic simulations [3]. An extension to the VRML/X3D standards enabling definition of behaviour of objects, called BDL, has been described in [4]. Another approach, based on the concept of aspect oriented programming has been proposed by Mesing and Hellmich [8]. Ajax3D is a recently developed method of programming interactive web 3D applications, based on the combination of JavaScript and the Scene Authoring Interface (SAI) [10].

The second group of solutions are integrated application design frameworks. Such frameworks usually include some complex languages and tools that extend existing standards to provide additional functionality, in particular, enabling specification of virtual scene behaviour. Interesting works include Contigra [5] and Behavior3D [6], which are based on distributed standardized components that can be assembled into 3D scenes during the design phase. However, this approach is based on the classical VRML/X3D dataflow paradigm and standard event processing making it difficult to specify more complex behaviours. Another solution, based on the use of distributed components accessible through web services has been proposed in Zhang and Graćanin [18].

The common motivation for developing new scripting languages and content design frameworks, as those described above, is to simplify the process of designing complex 3D presentations. However, even most high-level scripts and content description languages lead to complex code when they are used for preparing complicated 3D presentations. The third group of solutions try to alleviate this problem by using graphical applications for designing behaviour of 3D content. Virtools is a good example of this type of tools [7]. Recent research works in this field include those by Arjomandy and Smedley [2] and Vitzthum [11]. However, even if graphical specification of behaviours may be in some cases more intuitive than programming, users still must deal with complex diagrams illustrating how a scenario progresses and reacts to user interactions. Such diagrams are usually too difficult to be effectively used by non-programmers.
The approaches described above may be successfully used by 3D designers and programmers for preparing fixed 3D scenes. This, however, is not sufficient to enable widespread use of VR applications. Such applications require creation of large amounts of complex interactive content by domain experts, such as museum curators or architects. Therefore, a method is needed for efficient creation of interactive behaviour-rich 3D content going beyond the capabilities of content creation methods available today.

3. THE FLEX-VR COMPONENT MODEL

3.1. Overview

The Flex-VR approach uses a specific organisation of the VR content, called Beh-VR. In Beh-VR, a VR application is built of software components called VR-Beans. Technically, VR-Beans are objects, implemented as standard Script nodes conforming to a specific convention. Conformance to this convention enables combining arbitrary sets of VR-Beans into technically correct 3D scenes and provides means of inter-object discovery and communication. Beh-VR applications are fully compliant with existing 3D content description standards – VRML/X3D – and therefore can run in standard 3D browsers [17].

Each VR-Bean consists of at least one scenario script, an optional set of media objects, and an optional set of parameters (Figure 1). The scenario script is the main element controlling each VR-Bean. Scenarios are programmed in a high-level XML-based programming language called VR-BML (Behaviour Modelling Language). Scenarios describe what happens when the object is initialized, what actions are performed by the object and what are the responses of the object to external stimuli. In some cases, there may be several different scenarios controlling the VR-Bean depending on the presentation context.

A VR-Bean can have a number of media components, which are used for representing the VR-Bean in virtual scenes. Such media components may be

---

> Figure 1. A VR-Bean object consisting of scenarios, media components and parameters.
3D models (X3D/VRML), images, audio and video sequences, or texts. A VR-Bean can be also associated with a set of parameters. Parameters are characterized by a name, a type (integer, string, Boolean, etc.) and a value. Parameters can be read by scenario scripts and can be used in determining appearance or behaviour of objects.

3.2. Scene Structure

A Beh-VR scene is created dynamically by combining independent VR-Bean objects. Each VR-Bean object is controlled by a VR-BML script (Figure 2). A behaviour script may load any number of media components into the virtual scene, thus creating scene components — the geometrical and aural manifestation of the VR-Bean. The scene components may be created during the object initialization phase or later during the object lifetime. Objects may also freely change their representations at any time. A scenario can create and destroy scene components and can communicate with the scene components within a single VR-Bean by sending and receiving events to/from the components. Each object can control all scene components created by the object, but has no direct influence on other scene components.

Since the contents of a Beh-VR scene is generally not known at the design time, communication between objects becomes a critical element. Meaningful communication requires identification of objects present in the scene, well-defined roles of objects and existence of technical means of communication.

Identification of objects is possible due to a process of registration and discovery. Each object may be registered in an arbitrary number of hierarchically organized categories. The categories also define roles of objects in the scene. An object assigned to more than one category plays several different roles in the scene. For example, an object may be a status indicator and a game activator at the same time.

Figure 2. A Beh-VR structure and the resulting 3D scene.
Communication between objects is realized using two mechanisms: public values and method invocation. Public values are named public expressions that can use variables and events from a single VR-Bean. Each VR-bean can explicitly read public values, can be notified when such a value changes, and public values can be directly assigned to input events of scene components. Method invocation can be performed on single objects, lists of objects and the whole categories. A method consists of a list of VR-BML commands, which may change the state of a VR-Bean, alter its representation in the virtual scene, invoke other methods, etc. Each method may have any number of parameters. Formal specification of parameters is provided in the method declaration, while their actual values are set in a method call.

4. THE FLEX-VR CONTENT MODEL

To fully exploit the benefits of configurable virtual reality applications, a high-level model of the VR application content is required. Such model enables efficient organization, manipulation and exchange of content between applications. The Flex-VR content model describes a VR application on a much higher level of abstraction than a typical content description language, such as VRML/X3D. The Flex-VR content model may be stored in a content database to provide persistence, high-performance data manipulation, multi-user access and transactional processing.

In the Flex-VR approach, the VR content is organized hierarchically. The hierarchy is built of presentation spaces (Figure 3).

Each presentation space may have any number of sub-spaces. The depth of the hierarchy is not limited. Presentation spaces may correspond to complete virtual environments, parts of environments, or may be used merely as containers for objects (e.g., containing alternative representations of the same object). The semantics of the sub-spacing relationship depends on the super-space. For example, it may denote spatial, temporal or logical
composition, alternative representations, scenario steps, etc. In some cases there may be no semantic connection of the super-space and the sub-spaces.

Presentation spaces are containers that may generally hold three types of elements: instances of content templates, instances of behaviour templates, and instances of content objects.

Template is a parameterized program used to generate representations of presentation spaces. There are two types of templates: content templates and behaviour templates. Content templates are used to generate the representation of the space and to select content objects that should be included in this representation. Simplest templates generate scenes by combining content objects. More complex templates may additionally include background elements such as a model of a room, environmental properties (e.g., a fog), etc.

Each of the included content objects may contain its own behaviour script. In some cases, however, it is useful to have the same (or similar) behaviour shared by a number of objects. To achieve this, an instance of a behaviour template, used to generate scripts implementing common object behaviour may be also included in the presentation space.

Templates may be encoded in any scripting language that is suitable for scene content generation (X-VRML, PHP, JSP, etc.) [15]. A template consists of a template implementation and specification of a set of template parameters, each of which has a name, a type and a default value. Parameters influence the way of execution of the template code.

A template instance is a template supplied with actual values of some of its formal parameters. Default values are used for parameters that do not have a value specified in the instance (Figure 4). A single template can have an arbitrary number of instances.

A content object is the basic element of the Flex-VR content model. Content objects may correspond to simple 3D objects, complex objects gathering several media components or VR-Bean objects with their own behaviour specification. A content object is independent of other content objects.

---

**Figure 4. Content model: a template and a template instance.**

\[ P \text{ – Parameter, } D \text{ – default value, } V \text{ – actual value} \]
A content object consists of a number (zero or more) of media components and content object metadata (Figure 5). Media components are representations of the content object in various media. Examples are 3D model, image, video sequence, audio sequence, and text. More than one media component of the same type may be associated with a content object. Media components are used to represent the content object in a virtual scene. Media components may be shared by content objects. Media components may be associated with media component metadata providing component description.

A content object instance is a content object assigned to a presentation space, optionally with a metadata description and a set of presentation properties. Similarly as with template instances, content objects can have multiple instances in the same or different presentation spaces.

A common problem in the design of real-life virtual reality applications is caused by different hardware and software environments, in which the applications must run (e.g., an immersive system versus a laboratory equipped in PC computers). Also, often the applications must be targeted at different groups of users (e.g., a group of school children playing with a system while visiting a museum versus an archaeologist trying to find some details about a specific cultural object). To enable reuse of the same generic content model at different target platforms and with different presentation methods, while keeping consistency of the designed presentation structure and the content, the notion of presentation domains has been introduced. A presentation domain corresponds to a target environment or a usage scenario for the virtual reality application.

Presentation domains form a hierarchy (Figure 6). An example of a domain can be “WEB.LOCAL”. “WEB” is the name of the parent domain, while “LOCAL” is the name of the sub-domain. “ANY” is an abstract super-domain of all domains, and has no practical implementations.

In the Flex-VR content model, in each presentation space there may be a separate template for each presentation domain. For example, a space with a virtual museum exhibition may contain three sets of content templates.
and behaviour templates, which enable presentation of the system (1) on a high-end immersive installation in the museum, (2) remotely over the Internet on a standard PC computer, and (3) locally in the museum on small portable computers in form of an interactive guide. The templates differ in the selection of the content objects, quality of the 3D models, interaction methods, etc.

To make the process of designing large virtual reality applications more efficient, the notion of template instance inheritance has been introduced. If a presentation space does not contain a template instance in a particular presentation domain, first a template instance in a super-domain is used – if it exists in the presentation domain. If not, a template instance from a higher-level presentation space is used – in the same domain or a super-domain.

The overall structure of a Flex-VR presentation is shown in Figure 7. A presentation designer builds Flex-VR presentations by forming a hierarchy of presentation spaces and creating – in the presentation spaces – instances of content templates, instances of behaviour templates and instances of content objects, and by setting values of presentation properties.
To simplify creation of complex VR presentations consisting of objects with different roles and behavior, a presentation space may be associated with a content pattern (Figure 8). A content pattern defines a tree of content object categories, communication channels between the categories (method signatures, public values) and default implementation of methods and actions (possibly empty). Objects assigned to a category inherit methods and actions from the pattern, but can also override the default implementation with some object-specific implementation.

With the use of a content pattern, a designer may create a behaviour-rich presentation by simply assigning object instances to particular categories. A single object instance may be assigned to several categories. The role and behavior of a content object in the presentation depends on the categories it is assigned to.

Each category contains objects with a specific role and behavior and therefore it implements a specific interface. In the example presented in Figure 8, the category User Interface/Out implements the display(text) method. The category interface is specified by a pattern designer (a programmer), but the assignment of particular objects to this category is performed by a presentation designer (a domain expert). For example, there may be no objects implementing this method assigned to this category (text will not be displayed), there may be one object (e.g., a text window), but there may be also more objects (e.g., a 3D text object and a 2D banner).

Some methods have default implementations in the pattern. For example, the default rotate() method may rotate all scene components created by a content object by the same angle. Some objects may override the default implementation. For example, the Skeleton object may override this method to rotate several components around a common axis.
In the example presented in Figure 8 there are two objects assigned to the category User Interface/Out: Text Window and Sound Player. When an object calls a method User Interface/Out/display(text) the method is called in both objects. However, only the Text window object has a non-empty implementation of this method. The Sound Player object has no implementation of this method, and therefore it inherits only an empty implementation from the content pattern. This solution brings some small performance overhead but frees the presentation designer from any programming tasks.

5. THE VR-BML LANGUAGE

5.1. Overview

The VR-BML language is based on XML, which is the de-facto standard for creating new languages in the domain of multimedia and particularly 3D content. Most new 3D content standards such as X3D, 3D XML, and COLLADA are based on XML. XML is easy to interpret, existing software may be used for parsing and processing the program code. The structure of XML programs may be verified against their formal specification in form of XML Schema.

The VR-BML language consists of a list of commands, a specification of program structure in the form of XML Schema and expression grammar for attribute values. There are two main types of commands: structuring commands and instruction commands. Structuring commands provide the necessary structure of a VR-BML script, which enables to correctly interpret the instruction commands. Examples of structuring commands are: Scenario, Initialize, Method and Action.

Instruction commands are VR-BML commands that perform some operations, such as loading an X3D model into the scene (Load), moving a component to a specific location (Move), and calling a method (Call). Most commands require parameters, which may be provided as constant values or expressions containing references to variables, public values, and events generated by scene components.

VR-BML uses a hybrid approach based on both declarative programming for high-level elements (e.g., event actions) and imperative programming for low-level elements (e.g., algorithm details). This hybrid approach enables VR-BML to take best of the two worlds enabling the programmer to concentrate on important elements, and leave the common elements to the Flex-VR framework.

5.2. VR-BML Scenarios

A scenario script contains three sections (Listing 1): the initialization section, which describes what happens when the object is loaded into the scene, a number of action statements, which describe what actions are performed by the object as a result of changes or events in the scene (time, scene
properties, user interactions), and a number of methods that can be explicitly called by the same or other objects in the scene. Scenario scripts can inherit methods and actions from scenario classes that form inheritance hierarchies.

Listing 1. VR-BML scenario structure.

```xml
<Scenario extends="...">

  <Initialize>
    ...
  </Initialize>

  <Action name="..." cond="..." time="...">
    ...
  </Action>

  <Method name="..." param="..." wait="...">
    ...
  </Method>

  ...
</Scenario>
```

The main commands used to build VR-BML scenarios are described below.

*Scenario (extends)*

The *Scenario* command is the main element of each VR-BML scenario script. Each *Scenario* command is executed separately in parallel to other scenarios (e.g., through the use of threads). The *Scenario* element can contain only three types of elements: *Initialize*, *Action* and *Method* (see Listing 1).

*ScenarioClass (name)*

The *ScenarioClass* command is used to create classes of scenarios. The structure of a scenario class is the same as the structure a scenario. It may contain the initialization section, a number of action statements and a number of method definitions.

The initialization section defined in a class is executed before initialization sections of any scenarios or subclasses that inherit from this class. The methods and action statements are inherited by subclasses and scenarios, but can be overridden by method definitions and action definitions in subclasses and scenarios. Methods and actions are
unambiguously identified by their name – there is no method overloading based on the parameter sets.

Initialize

The Initialize command forms the initialization section of a VR-Bean. The initialization section is executed once for each VR-Bean object when the object is created. The initialization section typically contains all elements that are needed to build the VR-Bean and to set-up its initial state. Typically, the section contains commands for loading the scene representation of a VR-Bean, registering the VR-Bean in the virtual scene, and setting initial presentation attributes. The Initialize command may contain all non-structuring commands of the VR-BML language (e.g., Load, Set, Register and Activate).

Method (name, params)

Each scenario may contain an arbitrary number of Method statements. Each Method statement defines a method – i.e. a fragment of VR-BML code, which is executed when explicitly called by the same or another VR-Bean.

The name of the method is provided in the name attribute, while formal parameters of the method are provided in the params attribute. Each parameter has a default value, which is used when a method call does not specify value of this parameter.

The result parameter provides the name of a variable that will hold the value returned by the method. The returned value is defined using the optional Return command inside the method code. Example method definition is the following:

```xml
<Method name="move" params="dst=0,0,0" result="result">
  ...
  <Return value="{2*distance}"/>
</Method>
```

Action (name, cond, count, time)

Each scenario may also contain an arbitrary number of Action statements. Each Action statement represents an action – i.e. a fragment of VR-BML code, which is executed asynchronously when a specific condition is satisfied.

Execution of an Action command starts when the condition defined in the condition parameter is satisfied. The maximum number of executions is defined in the count parameter. Count equal –1 means that there is no limit. If the condition is still true, the action is triggered again after the time provided in the time parameter. If the time parameter is equal –1, the action is repeated immediately after the previous iteration. Example of use:
5.3. Content Patterns

Content patterns can be assigned to presentation spaces. Patterns define the overall structure of Flex-VR presentation spaces simplifying the design of particular presentations. A content pattern defines the following:

- a list of categories of objects that can be assigned to this presentation space together with their default scenario definitions;
- a list of subspaces of the presentation space together with their default content patterns.

The overall structure of a content pattern definition is presented in Listing 2, while the meaning of particular commands is explained below.

**Pattern (name)**

The `Pattern` command defines the overall content pattern. It has just one parameter: `name`, which enables this pattern to be referenced by presentation spaces. The `Pattern` element may contain two types of elements: `Category` and `Space`.

![Listing 2. Content pattern structure.](image)

```xml
<PATTERN name="...">

  <CATEGORY name="...">
  <SCENARIO extends="...">
    <METHOD name="...">
      ...
    </METHOD>
    ...
  </SCENARIO>
  </CATEGORY>

  <CATEGORY name="...">
  <SCENARIO>
    ...
  </SCENARIO>
  </CATEGORY>

  <SPACE name="..." translation="..." ...
  pattern="...">

</PATTERN>
```
**Category (name)**

The Category command defines a category of presentation objects that can be assigned to a presentation space associated with this pattern.

**Space (name, translation, rotation, scale, pattern)**

The Space command defines a subspace of the presentation space. The subspace is described by a name, transformation parameters: translation, rotation and scale relative to the current presentation space and default content pattern. The pattern can be referenced by name from a library of existing patterns or defined directly in the Space element.

In the following sections the VR-BML instruction commands, which can be used in the scenario and pattern definitions are described.

### 5.4. General Purpose Commands

**Set (name, value)**

The Set command implements assignment of a value to a variable. The command has two parameters: name and value. Example of use:

```xml
<Set name="i" value="10"/>
```

**Call (target, method, param, wait)**

The Call command may be used to call methods of the same VR-Bean or other VR-Beans, enabling synchronous communication between VR-Beans. The target parameter specifies the target VR-Bean(s). This may be a single VR-Bean, a list of VR-Beans or a category of VR-Beans. If the target parameter is omitted, the this value is used by default. The method parameter provides the name of the method to call. The param attribute provides the actual values of the method call parameters. VR-BML uses explicit naming of parameters in method calls, which results in more self-documenting code. Finally, the wait parameter specifies whether execution of the current VR-BML thread should be suspended until the execution of the method finishes or not. If wait is true, the next VR-BML command will be executed when the method finishes, if wait is false, execution of the VR-BML commands continues while the method is being executed. Example of use:

```xml
<Call target="{$co[$i]}" method="resize" param="size=2"/>
```

**If-Then-Else (cond)**

The If-Then-Else statement is a typical statement found in many programming languages. The If element specifies a condition (a Boolean
expression), which may be met or not. The element may contain only two
other elements: Then and Else. If the condition is met, the contents of the
Then element is executed, otherwise the contents of the Else element is
executed. Example of use:

```
<If cond="{$co!=null}">
  <Then>
    ...
  </Then>
  <Else>
    ...
  </Else>
</If>
```

The Print command enables to print a value on a system console (e.g., a web
browser console). The Print command is mostly useful for testing and
debugging purposes. Example of use:

```
<Print value="Position={$posValue}"/>
```

5.5. Loop Commands

Every (time)

The Every command is a loop that may be used to repeat execution of a
fragment of VR-BML code in given constant time intervals. The interval is
provided in the time attribute in milliseconds. Example use:

```
<Every time="{1000*moveTimeInterval}">
  <Call ...>
</Every>
```

ForEach (list, var)

The ForEach command is a loop, which is repeated once for every object in
a list provided as the command parameter list. The elements of the list are
being taken sequentially and assigned to a variable with the name provided
in the var parameter. The ForEach command is very useful to execute
specific operations on lists of objects that are retrieved dynamically. For
example, all objects with the creation date outside a given time scale should
disappear from the virtual museum exhibition room. Example of use:

```
<ForEach list="{$objList}" var="currObject">
  <Call target="{$currObject}" method="" ... />
</ForEach>
```
For (name, from, to, step)

The For command implements a typical for loop. The loop control variable is defined by name, while the starting, ending and step values are defined by the from, to, and step parameters, respectively.

While (cond)

The While command implements a typical while loop, repeated as long as a condition is satisfied.

5.6. Object Discovery Commands

Since the contents of a Beh-VR scene is configured after the behaviour scripts are programmed, there must be a mechanism of discovering what objects are in the scene. This is achieved using the process of registration and discovery.

Register (category, name)

The Register command enables to register the current VR-Bean object in a category. Categories form a hierarchical catalogue, enabling VR-Beans to discover what other VR-Beans are in the scene and what are their roles.

The Register command has two parameters: category and name. The category parameter specifies a comma-separated list of categories, where the VR-Bean should be registered. The name parameter specifies the name by which the VR-Bean will be registered in the category.

```xml
/Register category="objects/cultural/artistic, objects/animated" name="Mortar"/>
```

Retrieve (category, var)

The Retrieve command is complementary to the Register command and enables retrieving the list of VR-Beans which are registered in specific categories. The Retrieve command has two parameters: category and var. The category parameter provides the list of categories. If a category is followed by a “+” sign, all sub-categories of this category are also taken into account. The var parameter provides the name of a variable, which will contain the list of retrieved VR-Beans. Example of use:

```xml
.Retrieve category="obj/cultural+" var="obj"/>
```

5.7. Communication Commands

Publish (comp, public, value)

The Publish command enables definition of a public value – either as a constant or as an expression using variables and events originating from a
component. The Publish command is useful for creating objects that somehow control the virtual scene. For example, a scale slider may be created and its scale value may be published using the Publish command. Whenever the setting of the slider changes, the public value is automatically changed. The public value is not created directly from the component’s event but it re-calculated using expression specified in the value parameter. Names of events in the expression are denoted starting with the tilde (~) sign. The public parameter specifies the name of the public value. Example of use of the Publish command:

\[
\text{<Publish comp="slider" public="scale_of_objects" value="\{(~\text{translation\_out}+5)/10\}"/>}
\]

Assign (comp, public, event)

The Assign command may be used to directly assign a public value to an event controlling particular component of the VR-Bean. For example, a public value scale_of_objects can be assigned to the set_scale event of components of all presentation objects. In this way, by changing the public value, the scale of all presentation objects in the scene can be changed.

The Assign command is similar to the ROUTE statement in VRML/X3D/MPEG-4 standards, with the main difference, that ROUTEs must be established from the source to all receiving objects, which are not known when the virtual scene is being dynamically composed. Example of use:

\[
\text{<Assign comp="comp1" event="set_scale" value="scale_of_objects"/>}
\]

Link (comp, event, target, method)

The Link command enables attaching events coming from VR-Bean components to method calls in the same or other VR-Beans. Since the Beh-VR scenes are configured dynamically, the list of target VR-Bean objects may not be known. Therefore the Link command enables specifying either a list of concrete VR-Beans or a category. The target parameter may be a list of objects, a list of categories, or may be left empty (not specified). If target is empty, the current VR-Bean is used as the target. The Link command has the following parameters:

- \text{comp} – is the name of the component, which is the source of events;
- \text{event} – is the name of the event activating the method;
- \text{target} – represents the list of target VR-Beans or the list of target VR-Bean categories. If not specified, the current VR-Bean is used;
- \text{method} – specifies the name and the parameters of the method being called.
Example of use of the Link command:

```xml
<Link comp="s1" event="touchTime"
method="moveTo(0,0,0)"/>
```

5.8. Component Management Commands

The following commands are used to manage components representing VR-Beans in a virtual scene.

Load (uri, comp, active)

The Load command enables loading components into the Beh-VR scene. The uri parameter provides location of the component – this may be a remote object accessed by the http:// protocol, a local file system component accessed by file:// or an object retrieved from a database. The comp parameter specifies the name that will be assigned to the newly loaded component. The active parameter specifies, whether the component should be automatically activated after loading or should stay as inactive until it is activated with the Activate command. Example of use:

```xml
<Load uri="{$path}/oid={$o}" comp="object_{$o}" active="false"/>
```

Activate (comp, active)

The Activate command activates a component that has been loaded (using the Load command) into the Beh-VR scene. Because the loading process may include content generation, transmission over the network and parsing, it may take relatively long time. Therefore, to allow Beh-VR applications to switch on and switch off particular components in an efficient way, the Activate command may be used. Example of use:

```xml
<Activate comp="wheel" active="true"/>
```

SetPosition (comp, value, relative)

The SetPosition command is used to position a component in the 3D space. The command has three parameters: comp, value and relative. The comp parameter specifies the name of the component to be positioned. The value parameter provides the new position of the component, while the relative parameter indicates whether the new position is absolute or relative to the current position. Example of use:

```xml
<SetPosition comp="object_1" value="0,10,0"/>
```
SetScale (comp, value, relative)

The SetScale command enables to set a new scale factor to a component. The command has three parameters: comp, value and relative with the meaning similar as in the SetPosition command.

SetOrientation (comp, axis, angle, relative)

The SetOrientation command enables to rotate an object in the three-dimensional space. The comp parameter specifies the name of the component to be rotated. The axis and angle parameters specify the axis and angle by which the component should be rotated.

GetPosition (comp, var)

The GetPosition command enables retrieval of the current component's position in the scene. The position is assigned to a variable whose name is provided in the var parameter.

GetScale (comp, var)

The GetScale command enables retrieval of the current scale of the component.

GetOrientation (comp, varAxis, varAngle)

The GetOrientation command enables retrieval of the current cumulative orientation of the component in space in the axis-angle form.

Send (comp, event, value)

The Send command enables to send an event to a component. The comp parameter specifies the name of the component, to which the event should be sent. The event parameter specifies the name of the event, and the value parameter specifies the value of the event to be sent. Example of use of the Send command:

```
<Send comp="plate" event="set_color" value="0.2 0.2 0.8"/>
```

Read (comp, event, var)

The Read command enables retrieving the current value of an output field of a component. The names of the component and the field are provided in the comp and event parameters, respectively. The var parameter specifies the name of the variable that will hold the retrieved value. Example of use:

```
<Read comp="door" event="isOpen" var="doorOp"/>
```
Unload (comp)

The *Unload* command is complementary to the *Load* command and causes removal of the object from the Beh-VR scene (regardless whether the component is currently active or not).

5.9. Animation Commands

*MoveTo (comp, position, relative, time, wait)*

The *MoveTo* command is one of the animation commands in VR-BML. Using this command it is possible to animate a component from its current location to a new location. The *comp* parameter specifies the component to be animated. The *position* and *relative* parameters provide the target location. The *time* parameter specifies how long (in milliseconds) the animation should last. Finally, the *wait* parameter specifies whether execution of the current VR-BML thread should be suspended until the animation finishes or not. Example of use:

```xml
<Move comp="o1" position="0,2,0" time="500" wait="false"/>
```

*RotateTo (comp, axis, angle, relative, time, wait)*

The *RotateTo* command enables to animate rotation of a component. The *comp* parameter specifies the name of the component to be animated. The *axis* and *angle* parameters specify how the component should be rotated. The meaning of *relative*, *time* and *wait* parameters is analogous to the corresponding parameters in the *MoveTo* command.

*ScaleTo (comp, value, relative, time, wait)*

The *Scale* command is an animation command which enables to set a new scale factor to a component through an animation. The command has five parameters: *comp*, *value*, *relative*, *time*, and *wait* (see *MoveTo* command).

5.10. Sensor Commands

*PlaneSensor (comp, orientation, active)*

The *PlaneSensor* command is one of four sensor commands available in VR-BML. The *PlaneSensor* command is an easy way to add interactivity to a virtual scene by enabling a component to be moved by a user on a two-dimensional plane. The name of the component is provided in the *comp* parameter. The *orientation* parameter provides the orientation of the two-dimensional plane in space. The *active* parameter specifies whether the plane sensor should be activated or de-activated. Example of use:

```xml
<PlaneSensor comp="object_1" orientation="0,1,0,0" active="true"/>
```
**SphereSensor (comp, active)**

The SphereSensor command enables to add interactivity to a virtual scene by enabling a user to freely rotate a component in space. The *comp* parameter specifies which component should be allowed to be rotated. The *active* parameter specifies whether the sphere sensor should be activated or de-activated. Example of use:

```
<SphereSensor comp="object_1" active="true"/>
```

**CylinderSensor (comp, orientation, active)**

The CylinderSensor command enables a user to rotate a component in space around an axis. The *comp* parameter specifies the component. The *orientation* parameter provides the orientation of the rotation axis in space. The *active* parameter specifies whether the sensor should be activated or de-activated. Example of use:

```
<CylinderSensor comp="object_1"
    orientation="0,1,0,0" active="true"/>
```

**TouchSensor (comp, method, active)**

The TouchSensor command enables to activate a component as a touch-sensitive element. The *comp* parameter provides the name of the component to become a touch sensor. The *method* parameter specifies the method invocation, which should be executed when the component is touched. The *active* parameter specifies, whether the touch sensor should be activated or de-activated. Example of use of the TouchSensor:

```
<TouchSensor comp="object_1"
    method="playSound(id=1)" active="true"/>
```

### 6. APPLICATION OF FLEX-VR TO CULTURAL HERITAGE

The Flex-VR approach has been successfully applied in a virtual museum system called ARCO – Augmented Representation of Cultural Objects [1]. The system allows museum staff to setup virtual exhibitions from existing components by creating presentation spaces and by assigning content patterns, templates and content objects to the spaces [16].

The process of designing a virtual exhibition content in the ARCO system is presented in Figure 9. The content model consists of one presentation space (Granary) based on a content pattern (Exhibition). A content template is assigned to this space. The template contains a 3D environment model and a set of commands to create VR-Beans corresponding to content objects.
The content pattern defines categories of content objects and provides default implementation of category behaviour scripts. The pattern defines three categories: Objects, Scenario and User Interface. The categories are represented as folders in the application allowing a designer to easily assign objects to the categories.

A presentation designer (e.g., a curator in a museum) can create different presentations based on the same content pattern by using different content templates and by assigning different sets of content objects to the categories, and setting their parameters and presentation properties (e.g., positions, sounds).

In Figure 10 an example view of a virtual exhibition based on the content model is presented. A user navigating in a 3D reconstruction of an old granary can examine a collection of virtual museum objects. The user can interact with the objects and can activate them by pressing buttons.
Activation of each object is programmed in the pattern, but behaviour of each object upon activation may be different as specified in the object’s behaviour script. For example, the Plane object disassembles to show how it is constructed, the Bells object plays recorded bell sound, while the Wooden Statue object starts a voice description explaining particularities of the piece of art.

The second example (Figure 11) shows an exhibition based on the same template (Granary) indented to explain how the granary itself was constructed. There are no objects assigned to the space and template parameters are set to remove parts of the roof and show details of the complex construction.

7. CONCLUSIONS

The presented method of programming and composition of behaviour-rich 3D content simplifies the design of interactive VR applications in two ways. On the one hand it simplifies programming behaviour of virtual objects and scenes by enabling the use of the high-level VR-BML language instead of the low level elements such as sensors, interpolators, and routes. This language can be used by experienced users.

On the other hand, by providing the capability of dynamic content configuration, the Flex-VR approach enables users without programming experience to create complex interactive 3D content by combining predefined content patterns and collections of objects. Therefore, the Flex-VR approach can be used as a basis for building simple to use VR authoring applications for domain experts.

Future works include extending the VR-BML language with more advanced features, such as non-linear animation, complex objects and object groups. The authoring application will be also extended to support visual arrangement of objects.
References


Department of Information Technology
Poznan University of Economics
walczak@kti.ue.poznan.pl