The Design Principles of Object-Oriented VRML*

Sungwoo Park, Taisook Han
Distributed Computing Laboratory SERI**, Department of Computer Science KAIST***
{gladius,han}@compiler.kaist.ac.kr

Abstract

We present a new 3D scene description language, Object-Oriented VRML, to show how VRML can be extended to incorporate object-oriented design principles. Object-Oriented VRML supports the creation of objects in an object-oriented style. It is based on a safe object type system in which runtime type errors are not generated. It facilitates the efficient implementation of virtual environment systems by making event transmissions unnecessary in most situations. We give the syntax with examples and explain the design criteria.

1 Introduction

The Virtual Reality Modeling Language [1], VRML[1], is a file format for describing interactive three-dimensional virtual worlds. It has evolved from a simple 3D scene description language to a more sophisticated 3D modeling language capable of expressing behavior of nodes and supporting various interactions.

VRML is intended to be an object-oriented system. Object-oriented features of VRML are such as prototype extension mechanism, event passing between nodes, and the behavior associated with nodes. However, some of the language features violate the general object-oriented design principle. For example, adding behavior to a node is accomplished by exploiting `Script` nodes and routes that are not a part of the node; the behavior of the node is controlled not by it own elements but by other independent nodes, namely `Script` nodes in conjunction with routes. Further object-oriented extensions to VRML are possible in many ways. Some extensions to VRML are found in [2,3].

In this paper, we present a new object-oriented 3D scene description language, Object-Oriented VRML (OO-VRML), which is a derivative of VRML. OO-VRML supports the creation of objects in an object-oriented style as the name suggests. It also supports a safe object type system which does not generate runtime type errors. Event transmissions become unnecessary in most cases by employing a new kind of content for attributes. The primitive version of OO-VRML is found in [4].

Characters in OO-VRML files are shown in `fixed width font`. Reserved words are shown in **`bold fixed width font`**. Non-terminals are enclosed in angle brackets <> and shown in *italic font*. Regular expressions are used in the grammar specification. We use *[ ]* to denote an optional symbol, *[ ]** to denote zero or more occurrences, *[ ]+* to denote one or more occurrences of a symbol. *{}* is used to collect several symbols. / is used to denote a choice among several symbols.

2 Objects

Objects are fundamental components constituting OO-VRML files. They are abstractions of various real-world objects and concepts. An object itself cannot be represented in the virtual world; instead, it serves as the basis for instances, which are represented concretely in the virtual world. OO-VRML defines an object as a collection of attributes and eventIns with associated event handlers. The attributes and eventIns are specified by the object type; objects of the same type have the same set of attributes and eventIns.

2.1 Attributes

Attributes are parameters which distinguish one object from another of the same type. They correspond to data members of objects in general object-oriented systems. Every attribute has a content which should be of the attribute type. The content is given when the object is created. It is used to determine the content of the corresponding attribute of instances which are derived from the object. That is, the contents describe the method of specifying the values of attributes of instances. Since an object serves as the basis for instances, the contents of the attributes cannot be modified after the object is created.

OO-VRML provides three kinds of attribute: field, exposedField, and eventOut. Fields store various kinds of data which determine the characteristics of objects. They specify geometric properties of objects, control dynamic behavior, and establish relations with other objects. A field has a content which should be of the field type. ExposedFields are a special kind of field. They provide an access mechanism by which an exposedField may be related with fields of other objects through constraints. As with VRML, an exposedField `P` declares implicitly an eventIn `P.`**`SET`** and an eventOut `P.`**`CHANGE`** through which other objects communicate with `P` by means of events. Hence declaring an exposedField `P` by

`exposedField` <*type*> `P`

is equivalent to

`eventIn` <*type*> `P.`**`SET`**
`field` <*type*> `P`
`eventOut` <*type*> `P.`**`CHANGE`**

, where `P` can be accessed from the outside. When the eventIn `P.`**`SET`** receives an event, the value of the incoming event is assigned to the exposedField `P` by the event handler associated with `P.`**`SET`**; the functionality of the event handler may be redefined. When a value is assigned to the exposedField `P`, the eventOut `P.`**`CHANGE`** becomes activated and generates zero or more new events according to the number of eventIns contained in `P.`**`CHANGE`**.

---

* This work was supported by MICRK, Ministry of Information and Communication Republic of Korea.
** System Engineering Research Institute, 1 Ueun-dong Yusung-gu Taejon Korea 305-333
*** Korea Advanced Institute of Science and Technology, 373-1 Gusung-dong Yusung-gu Taejon Korea 305-701
[1] VRML stands for VRML 2.0 or VRML97 in this paper.

In both VRML and OO-VRML, eventOuts serve as gates through which outgoing events are transferred. The key difference is that eventOuts of OO-VRML have contents which are used in specifying the destinations of the outgoing events, while eventOuts of VRML only create such gates and the destinations are determined by routes associated with them. Conceptually both eventOuts and routes of VRML are incorporated into eventOuts in OO-VRML. The content of an eventOut is composed of references to eventIns and eventOuts of the same type as the eventOut. When a value is written to the eventOut, it is activated and sends an event constructed from the value to every eventIn connected with it.

2.2 EventIns and Event Handlers

EventIns in conjunction with event handlers correspond to methods or member functions in conventional object-oriented systems. An eventIn and its associated event handler may be regarded as the method declaration and the method body, respectively; when events are delivered to the eventIn, the event handler is invoked to process them. The value of incoming events is used for the argument of the event handler. During its execution, the event handler may assign new values to some attributes.

The event handler associated with an eventIn must have the same name as the eventIn except for eventIns accompanied by exposedFields, whose event handlers may have the same name as the exposedFields, not as the eventIns. For example, an exposedField `P` declares implicitly an eventIn **`P.SET`**. In order to override the event handler associated with **`P.SET`**, we only have to attach a new event handler `P` to the object. An event handler **`P.SET`** is, of course, also a valid event handler for the exposedField `P`.

In VRML, the processing of incoming events is always done in a fixed way; their values are assigned directly to the relevant exposedFields without modification and new events are generated via the eventOuts connected with the exposedFields. Usually the complicated processing of events would be achieved in `Script` nodes, which are in themselves complete and independent objects. Hence, in order to modify the behavior of a node with respect to a certain eventIn, one should examine all `Script` nodes that are connected with the eventIn through routes. In other words, the behavior of an object may be controlled by other independent objects, `Script` nodes, in VRML.

In OO-VRML, event handlers carry out such works as `Script` nodes of VRML perform. They describe the actions taken when events are delivered to objects via eventIns. Since they are a part of objects, we may say that the behavior of an object is completely described by its own components, which makes every object behaviorally autonomous. This conforms to the design principles of object-oriented systems. With this object-oriented scheme for event handlers, we could build or reconstruct virtual worlds in an object-oriented style. Analyzing virtual worlds would also be easier than with VRML because we only have to concentrate on each object itself, not the relationships between objects.

2.3 Types for Attributes and EventIns

Field types are divided into two general categories: those for fields containing a single unit and those for fields containing multiple units. A single unit may be an integer, a vector, an image, and so on. A type specifier is composed of two parts: a category specifier and an element specifier. The category specifier **SF** and **MF** are used for single unit fields and multiple unit fields, respectively. The element specifier determines the type of units contained in a field. The category specifier and the element specifier are separated by a period. If the category specifier is omitted, the default category specifier **SF** is used.

A field should be of one of the following types. The right-hand side denotes the actual content structure of fields of the left-hand field type. A non-terminal with a field type in the angle brackets indicates a single unit of the specified field type.

SF.Bool *{* **TRUE** / **FALSE** *}*
SF.Color *<SF.Float> <SF.Float> <SF.Float>*
SF.Float *<single-precision floating point number>*
SF.Image *<SF.Int32> <SF.Int32> <SF.Int32> [<SF.Int32>]**
SF.Int32 *<32 bit integer value>*
SF.<*class name*> *<reference to an object>*
SF.Rotation *<SF.Float> <SF.Float> <SF.Float> <SF.Float>*
SF.String *<valid UTF-8 string enclosed in double quotes>*
SF.Time *<double-precision floating point number>*
SF.Vec2f *<SF.Float> <SF.Float>*
SF.Vec3f *<SF.Float> <SF.Float> <SF.Float>*
MF.Color *{* [*[<SF.Color>]**] *}*
MF.Float *{* [*[<SF.Float>]**] *}*
MF.Int32 *{* [*[<SF.Int32>]**] *}*
MF.Rotation *{* [*[<SF.Rotation>]**] *}*
MF.String *{* [*[<SF.String>]**] *}*
MF.Vec2f *{* [*[<SF.Vec2f>]**] *}*
MF.Vec3f *{* [*[<SF.Vec3f>]**] *}*
MF.<*class name*> *{* [*[<SF.<class name>>]**] *}*

The content structure of **MF.**<*type*> fields is composed of a list of units for **SF.**<*type*>. Object reference types **SF.**<*class name*> and **MF.**<*class name*> are used for fields whose contents give a reference or a collection of references to objects, respectively. All these objects must be of types which are convertible to the type of the object <*class name*>.

The semantics of eventOut types is that all the eventIns referenced from an eventOut, which is actually a multiple unit attribute, must be either of the same type as the eventOut or convertible to the eventOut type. The type of an eventIn specifies the type of events delivered to the eventIn. For example, the content of every event delivered to an **SF.Int32** eventIn must be convertible to an integer.

Compared with VRML, OO-VRML presents an enhanced model of object reference type by enabling an object reference type to accept only objects of the specified type. In VRML, **SFNode** and **MFNode** are employed on behalf of all node reference types, irrespective of their semantics. For instance, the definition of the `Shape` node of VRML would be as follows.

```
Shape {
        exposedField SFNode appearance NULL
        exposedField SFNode geometry NULL
}
```

According to the specification of the `Shape` node, the field `appearance` must contain an `Appearance` node which specifies the visual attributes to be applied to the geometry. However, the node reference type **`SFNode`** is irrelevant because it allows any node to be contained in the field `appearance`. In contrast, OO-VRML would write the definition of the object `Shape` as follows.

```
Shape {
        exposedField SF.Appearance appearance NULL
        exposedField SF.Geometry geometry NULL
}
```

The field `appearance` now accepts only references to **`SF.`**`Appearance` objects, and objects whose types are incompatible with the object reference type **`SF.`**`Appearance` cannot be used for the field `appearance`. Thus, the contents of object reference fields are always guaranteed to be semantically correct in OO-VRML.

Although VRML does not enforce correct contents upon node reference types in the language semantics, most VRML systems perform a semantics check on node reference types. Hence the new model of object reference type in OO-VRML would not bring a significant improvement on the type safety of attributes over VRML node reference types if only constants were used for the contents of attributes. However OO-VRML introduces a new kind of attribute content, namely paths, in addition to constants. Paths make it inevitable to employ such an enhanced model of object reference type as to support robustly the type safety of attributes containing paths. Paths are explained in detail in Section 3.2.2.

2.4 Creating New Objects

Attributes of objects are divided into two groups: internal attributes and external attributes. External attributes are those whose contents may be specified at the time of creating objects. Objects of the same type have the same set of external attributes and they are differentiated upon their characteristics by the contents of the external attributes. For this reason, every object usually has at least one external attribute. Internal attributes serve to maintain the internal operations of objects. Objects of the same type have the same set of internal attributes. Furthermore each internal attribute has the same content in these objects; it cannot be assigned a new content when objects are created. In order to catch the behavior of an object, one only has to understand the purpose of each external attribute and the operation of each event handler; internal attributes are not necessarily required to be examined.

A new object is created from an existing source object. The grammar for creating objects is as follows.

<object> ::=
[**`DEF`** *<new object identifier>] [* **`MULTIPLE`** / **`SINGLE`** *] <source object identifier>*
{
[
<field identifier> <field content> |
<eventOut identifier> <eventOut content>
*]**
} *<handler declaration>*

The new object may be given its own name by the keyword **`DEF`**. The properties **`SINGLE`** and **`MULTIPLE`** are used when instances are derived from the object, that is, it is instantiated. The default property is **`SINGLE`**. The property **`SINGLE`** or **`MULTIPLE`** of the source object is inherited to the new object if not overridden. The new object may override event handlers of the source object by presenting new definitions for them. New event handlers with corresponding eventIns, however, cannot be added to the new object.

The source object and the new object have the same set of attributes and eventIns. The new object may override the contents of external attributes of the source object. If a new content is not provided for an external attribute, its inheritance mode, which is explained in Section 3.1, is employed in order to specify how its content is obtained. The new content must be of a correct type. Otherwise, it is replaced by an undefined content. Internal attributes cannot be provided with new contents; they continue to have the same contents as the internal attributes of the source object.

The two objects have the same object type mutually. For example, if a new object `B` is created from a source object `A`, both **`SF.`**`A` and **`SF.`**`B` are legal object reference types for `A` as well as `B`. Note that the name of any object can be used for object types such as **`SF.`**`Obj` and **`MF.`**`Obj`. Thus creating an object supplied with a name provides an additional way to refer to a certain object type, but does not introduce a new object type.

In the viewpoint of OO-VRML, every field of nodes of VRML is regarded as an external attribute in that it may be given a new content overriding the default content. That is, every field is available to VRML users and there is no hidden field. All fields are directly employed in controlling the behavior of nodes. In contrast, objects of OO-VRML may have internal attributes which are invisible to users but still take part in determining the behavior of objects. An external attribute may be used for the contents of internal attributes or other external attributes. In this way, OO-VRML supports a more sophisticated mechanism for creating objects than VRML does for creating nodes.

3 Attributes

OO-VRML provides fields and exposedFields to specify the properties of objects as VRML does for nodes. However, the contents of fields and exposedFields of OO-VRML may contain paths in addition to constants, while only constants are available in VRML. Moreover, OO-VRML treats even eventOuts as attributes, a kind of data member of objects, which have contents. In this section, we explore the differences between VRML and OO-VRML in assigning contents to attributes.

3.1 Inheritance Modes for Attributes

When a new object is created from a source object, it is usually provided with specific contents for its attributes so that it may have

different properties from other objects of the same type. If an attribute is not given a content, its inheritance mode in combination with the content of the attribute in the source object determines its content in the new object. The inheritance mode controls how the content of the attribute in the source object is exploited for the attribute of the new object. Therefore OO-VRML does not maintain default contents for attributes as VRML does for fields; the content of an attribute in a source object does not serve as the default content for the same attribute of all objects created from the source object.

OO-VRML supports three inheritance modes: inherit, copy, and local. Suppose that a new object `B` is created from a source object `A` and that the content of an attribute `f` is not specified in `B`. If the inheritance mode of the attribute `f` is copy, which is the default inheritance mode, the content of `f` of the object `A` is copied to `f` of the object `B`. In this case, the attribute has the same content in the two objects. If the inheritance mode is inherit, the attribute `f` of the object `B` has a path which references the same attribute of the source object `A`. The inheritance mode inherit presents the result that the content of an attribute in a source object is shared by the same attribute of other objects created from the source object. The final inheritance mode is local. In this case, the content of the attribute `f` in the object `B` is **`UNDEFINED`**, which is a special content. **`UNDEFINED`** is converted to a constant according to the attribute type. In another point of view, the content of an attribute with the inheritance mode local becomes never available to other objects; every object must provide a unique content of its own for such an attribute.

Every attribute has an inheritance mode whether it is internal or external. However, the actual inheritance mode of internal attributes is always copy regardless of their nominal inheritance modes. For this reason, objects of the same type have the same contents for all the internal attributes, and only the contents of external attributes of an object determine its properties different from other objects of the same type. This strategy for internal attributes gives natural results in creating objects. Likewise the inheritance mode of eventOuts is always copy and cannot be specified otherwise.

3.2 Contents of Attributes

The contents of attributes are built on constants and paths. Various types of attribute contents are constructed with constants and paths.

3.2.1 Constants

Constants are fundamental expressions used for contents of attributes. A constant is an expression whose value is determined statically and does not depend on contexts or environments. OO-VRML supports seven types of constants. The counterparts of all these constants are found in VRML as well.

- boolean constants
- 32bit integers
- single-precision floating point numbers
- double-precision floating point numbers
- UTF-8 strings
- references to objects

An object reference `X` is regarded as an **`SF.`**`X` constant.

- **`NULL`**

`NULL` is used for empty multiple unit attributes, in which case it may be used interchangeably with **`[]`**. It is also used to denote null references to objects.

3.2.2 Paths

Paths are another kind of fundamental expression used for contents of attributes. A path used in an object references an attribute either in the same object or in other objects. Unlike constants, the evaluation result of a path depends on contexts or environments.

Paths begin with an object specifier. An object specifier is one of the following: **`SELF`**, **`SOURCE`**, **`PARENT`**, **`CURRENT`**, and object references. It is followed by a series of attribute pointers or the object specifier **`PARENT`**. In cases that a path does not begin with an object specifier, **`SELF`** is employed as the default object specifier. Each element of a multiple unit attribute can be accessed by indexing the attribute, while eventOuts cannot be indexed. Only integer constants may be used for indices.

<path> ::=
{ <object identifier> | <attribute pointer> | **`SELF`** *|* **`SOURCE`** *|* **`PARENT`** *|* **`CURRENT`** *}*
[**`.`** *{ <attribute pointer> |* **`PARENT`** *}]**
<attribute pointer> ::=
<attribute identifier> [`[` *<SF.Int32>* `]` *]*

The object specifier **`SELF`** references the object in which the path appears. It is attached to every path which begins with not an object specifier but an attribute pointer. For example,

```
DEF A Transform {
        center 0 0 0
        translation center
}
```

the final content of the attribute `translation` becomes **`SELF`**`.center`, and **`SELF`** in this path references the object `A`. Object references are another kind of object specifier. An object reference `Obj` references the object with its identifier `Obj`. An object reference may not appear in the middle of paths. The object specifier **`SOURCE`** is used only for attributes whose inheritance mode is inherit. When an attribute `f` with the inheritance mode inherit is not given a new content, its content is set to **`SOURCE`**`.f`, where **`SOURCE`** references the source object. Therefore, the object specifier **`SOURCE`** is not available to OO-VRML users. The object specifier **`PARENT`** is the only object specifier which may appear in the middle of paths. **`PARENT`** in an object `Obj` references the parent object which makes it necessary to instantiate `Obj`; the parent object must have an attribute which references the object `Obj`. For example,

```
DEF Child ClassChild {
        q PARENT.p
```

```
    }
    DEF Parent ClassParent {
            p 10
            child Child
    }
```
the attribute `q` of the object `Child` comes to reference the attribute `p` of the object `Parent` because **PARENT** in `Child` points to `Parent` in the above context; instantiating the object `Parent` entails the instantiation of the object `Child`. Several attributes in different objects may reference one common object. In such cases, **PARENT** in the common object may not be fixed to reference a certain object and the evaluation result of **PARENT** in an object may vary depending on the context in which the object is instantiated.

CURRENT is a special object specifier. It is used to directly import the contents of attributes in the source object. An attribute may contain a path beginning with **CURRENT**, but the path is converted to a new content. Consider the following example.
```
    DEF Q P {
            children [ CURRENT.child R ]
    }
```
The content of the attribute `children` of the object `Q` depends on both the inheritance mode and the content of the attribute `child` of the object `P`, the source object of `Q`. If the inheritance mode is copy, **CURRENT**`.child` is converted to the actual content of the attribute `child` of the object `P`, that is, the content of `child` of `P` is directly copied to into the attribute `children` of the object `Q`. If the inheritance mode is local, **CURRENT**`.child` is converted to **UNDEFINED** because the attribute `child` of the object `P` should not be available to any other object. If the inheritance mode is inherit, **CURRENT**`.child` is converted to **SOURCE**`.child`; the conversion conforms to the design principle of the inheritance mode inherit. In this example, **SOURCE** in the object `Q` points to the object `P`. Hence the attribute `child` of the object `P` is to be used for the attribute `children` of the object `Q`. In this way, any path beginning with **CURRENT** is converted to another appropriate content.

The design principle of the object specifier **CURRENT** should be distinguished from that of the inheritance mode inherit. The inheritance mode inherit is employed only for attributes whose contents are not provided. In contrast, a path beginning with **CURRENT** is converted to a new attribute content and the conversion is controlled primarily by the inheritance mode of the attribute which follows the object specifier **CURRENT**. This kind of path is useful for multiple unit attributes whose contents are constructed by appending additional contents to their contents in the source object.

Paths beginning with object specifiers **SOURCE** and **CURRENT** cannot contain more than one attribute pointer. This type of path is called single level paths, which contain only one attribute pointer. Multilevel paths take more than one attribute pointer. Object specifiers such as **SELF**, **PARENT**, and object references can be used to construct multilevel paths.

Given the enhanced model of object reference types, OO-VRML makes it straightforward to examine the validity of a path and obtain all possible attribute types represented by the path. An object specifier is always resolved to a set of objects. For example, **PARENT** used in an object presents all other objects that entail the instantiation of the object. **SELF** and **SOURCE** present the object itself and the source object, respectively. Hence it is statically known whether these objects contain the attribute designated by the attribute pointer following the object specifier. If the attribute is of an object reference type, it is also statically known which attributes are expected to follow if any. If an unexpected attributes is encountered, the path becomes invalid and is replaced by **UNDEFINED**. This process goes on until the last attribute pointer in the path is examined. In this manner, all possible attribute types represented by the path are collected and its type correctness can be checked.

As an example, consider a path expression `Obj.f.g.h` where `Obj` is an object reference. The type of the object referenced by `Obj` is determined statically. Hence, it is also statically determined whether the object has the attribute `f` or not. If the attribute exists, it must be of an object reference type which contains the attribute `g`. The attribute `h` which appears last in the path does not need to be of an object reference type. However, it must have the attribute type of which the path is expected to be. Thus it can be checked if the path is valid or not.

The contents of eventOuts are composed of only paths, excluding constants. These paths reference either an eventOut or an eventIn. Suppose that such a path references an eventIn. In this case, the attribute pointer which appears last in the path is not related with an attribute. Rather it is related with an eventIn, which is not an attribute. This shows the only exception to the definition of paths, in which paths should reference only attributes. In another point of view, however, eventIns may be looked upon as a special kind of attribute with no content. Then the above definition of paths would be applied to those for eventOuts without any difficulties.

VRML employs routes in order to deliver the value of one exposedField to others. The value is transmitted via routes according to the VRML event transmission mechanism. In this way, the relationships between exposedFields are established. In OO-VRML, a path establishes a relationship between two independent attributes. In this sense, paths are said to declare one-way equality constraints between attributes. In most cases, event transmissions become unnecessary if paths are substituted for them. Without paths, much more event transmissions would be inevitable. The support of paths in OO-VRML is a significant enhancement over VRML with respect to both the language semantics and the system performance.

3.2.3 Accessibility

Since paths can reference attributes across objects, OO-VRML provides a data hiding mechanism which prevents some attributes of an object from being accessed by other objects. ExposedFields can be read by other objects, but they cannot be written to. Unlike exposedFields, fields and eventOuts may not be accessed by other objects. However, eventOuts accompanying exposedFields are accessible. For example, if an exposedField `P` is given, `P.`**CHANGE** may be referenced by other objects. Note that the accessibility of attributes is irrelevant to whether they are external or internal.

EventIns of an object may be accessed by other objects. Since eventIns correspond to the member function of objects in object-oriented systems, the references to eventIns are available only to eventOuts; they cannot be used for the contents of fields or exposedFields. Inside an object, all the attributes and eventIns are freely accessible.

The only exception to this accessibility policy is found in fields which are referenced by path expressions beginning with the object

specifier **SOURCE**. Given a path **SOURCE.**`f`, `f` may be either a field or an exposedField. Since such a path is not given by OO-VRML users but generated by the OO-VRML system, the accessibility policy about the object specifier **SOURCE** does not weaken the data hiding mechanism.

3.2.4 Specifying Contents of Attributes

The content of a single unit attribute is a constant, a path, or a fixed number of constants and paths. If a path is given, it must be a reference to another single unit attribute of the same type. Parentheses may be inserted to avoid ambiguous expressions. The following example demonstrates three cases for the contents of **SF.Color** fields.

```
A {
        p 0 0 0          # SF.Color
        q p              # SF.Color
        r 0              # SF.Float
        s ( r 0 0 )      # SF.Color
}
```

The content of a multiple unit attribute is a path, **NULL**, or a list of contents for single unit attributes. It may also contain as one of its elements a reference to another multiple unit attribute of the same type. For example,

```
DEF A Transform {
        children [ P Q R ]      # exposedField
}
DEF B Transform {
        children [ A.children S ]
}
```

the content of the attribute `children` of the object `B` is `[ P Q R S ]` in the current context. If a path is given for the content, it must be a reference to another multiple unit attribute of the same type as with single unit attributes.

4 Classes

Object-oriented systems are divided into two broad categories: object-based systems and class-based systems. In object-based systems, new objects are created by modifying or extending the characteristics of existing objects. This kind of system does not maintain classes separately from objects. In class-based systems, a class introduces a new domain of objects which have common properties. An object is created by instancing a class, which determines the type of the object. Thus, classes and objects are two disparate concepts which are not compatible.

OO-VRML is a hybrid system incorporating both approaches. It supports classes as well as objects, but the semantics of classes is different from what is expected in class-based systems. OO-VRML is regarded as an object-based system with respect to object creation. New objects are created from source objects. Classes are a kind of object and not distinguished from objects. Thus, OO-VRML presents conceptually only one mechanism to create objects, which manages classes in the same way as objects. OO-VRML is also regarded as a class-based system with respect to object typing. The type of an object specifies its general characteristics, and only classes introduce new object types. Every object created from a class has the same type as the class. Interfaces provide a mechanism to add new object reference types to field types.

4.1 Defining Classes

Defining a class is actually creating a new object from a source object, which is called superclass. Ordinary objects as well as classes may be used for the superclass. New attributes and event handlers can be added, the type of the attributes in the superclass can be refined, and the inheritance mode of the attributes may be modified.

<class> ::=
[**MULTIPLE** / **SINGLE** *]* **CLASS** *<class identifier>* **EXTENDS** *<superclass identifier>*
[**IMPLEMENTS** *<interface identifier> [, <interface identifier>]*]*
[
<external declaration>
]
<body>
<handler declaration>

A class definition is composed of four parts: header, external declaration, body, and handler declaration. The header states the name of the new class, or the subclass. It also specifies the superclass and optional interfaces which the subclass should implement. The properties **MULTIPLE** and **SINGLE** specify whether or not the new class can be instantiated more than once. The default option is **SINGLE**, which implies that it can be instantiated at most once. The property **SINGLE** or **MULTIPLE** in the superclass is inherited to the subclass if not overridden. More than one interface may be given, and the new class must satisfy the specifications of each interface.

The subclass inherits all attributes and event handlers of the superclass. The external declaration enumerates all the external attributes of the subclass with their contents. The external attributes are either those inherited from the superclass or new attributes introduced in the external declaration for the first time. The body assigns new contents to internal attributes of the subclass which are external in the superclass. The handler declaration may add new event handlers or override the event handlers of the superclass.

The relationship between the subclass and the superclass on the attributes is shown in Figure 1. Every attribute is first introduced to a class as an external attribute. Once the attribute becomes an internal attribute, it cannot revert to an external attribute in any subclass. External attributes can still remain external in subclasses. In this way, a class can refine the behavior of its superclass by introducing additional new external attributes and exploiting them in specifying the contents of internal attributes.

4.2 External Declaration

The external declaration is composed of a list of specifications on external attributes and eventIns. Since the basic behavior of a class is controlled by its external attributes, the external declaration contains all parameters that distinguish the objects created from the class. Every external attribute must appear in the external declaration, where all the properties of each external attribute are given. If it is a new attribute added to the class, its inheritance mode, type, and content should be specified. Otherwise, that is, if it is shown to be an external attribute in the superclass by the keyword **`TRANSPARENT`**, its inheritance mode, type, and content may be overridden. New eventIns added to the class appear in the external declaration with their types. The old eventIns of the superclass can refine their types in the external declaration.

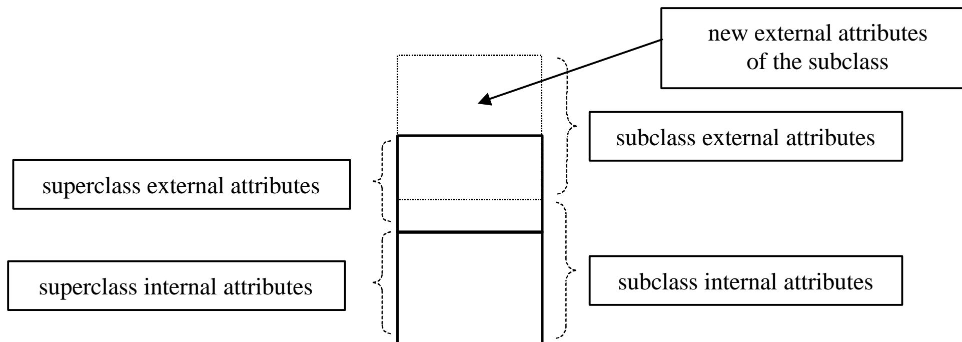

Figure 1. The relationship between the subclass and the superclass on the attributs

The syntax for the external declaration has three elements: the element for fields and exposedFields, the element for eventOuts, and the element for eventIns.

<external declaration> ::=
[
[**`TRANSPARENT`** *] [<inheritance mode>] {* **`field`** / **`exposedField`** *} <field type>*
<field identifier> [<field content>] |
[**`TRANSPARENT`** *]* **`eventOut`** *<eventOut type> <eventOut identifier> [<eventOut content>] |*
`eventIn` *<eventIn type> <eventIn identifier>*
*]**
<inheritance mode> ::=
`INHERIT` / **`LOCAL`** / **`COPY`**

Consider a field or exposedField declaration. If the keyword **`TRANSPARENT`** is not given, the field or exposedField is a new external attribute. In this case, a content must be provided for the attribute. The inheritance mode is one of local, copy, and inherit. It is optional and the default inheritance mode is copy.

If the keyword **`TRANSPARENT`** is given, the field or exposedField is not a new external attribute but an external attribute of the superclass. In this way, OO-VRML achieves the mechanism by which a subclass directly inherits external attributes from its superclass without changing them into internal attributes. The inheritance mode is optional, and it is not altered if a new inheritance mode is not specified. The field or exposedField must be of a subtype of its field type in the superclass. That is, the field type can only be refined. It cannot be set to another completely different field type; hence, if its field type in the superclass is not an object reference type, the same field type must be used again in the external declaration. The field content is optional. If a new content is not given, the content of the attribute is determined according to its inheritance mode.

A field or exposedField in the external declaration contains constants and valid paths of the correct type. The paths should reference either external attributes in the external declaration or attributes in other objects which are permitted to be accessed; paths in the external declaration cannot reference internal attributes. The object specifiers **`SOURCE`** and **`CURRENT`** used in the external declaration have the same semantics as they have in creating objects. Suppose that a class `B` is defined from a superclass `A`. Defining the class `B` is actually creating an object from the superclass `A`. Hence **`SOURCE`** in the class `B` references the superclass `A`. The superclass `A` is also employed to resolve paths beginning with **`CURRENT`** into new attribute contents.

EventOuts are a special kind of attribute. The semantics of eventOuts in the external declaration is exactly the same as that of fields and exposedFields except that the inheritance mode is not stated; the inheritance mode of eventOuts is always copy. EventIns of OO-VRML correspond to methods or member functions in object-oriented systems. They are accumulated in subclasses and accessible from the outside of classes. EventIns are used in the external declaration either when they are first added to classes or when their types are refined. If an eventIn is not given its associated event handler, every event sent to it is ignored.

An exposedField `P` in the external declaration accompanies implicitly the declaration of an eventOut `P.`**`CHANGE`** and an eventIn `P.`**`SET`** of the same type as `P`. The eventOut `P.`**`CHANGE`** has the content **`NULL`** by default. It may be given a new content in the external declaration. Both the eventOut `P.`**`CHANGE`** and the eventIn `P.`**`SET`** can refine their types in the external declaration.

All attributes and eventIns of a class must have unique names; they share a common name space within the class. Hence, the names of the new attributes and eventIns in the external declaration must be chosen so as to cause no name conflict.

4.3 Body

External attributes of a superclass turn into internal attributes in its subclasses unless they are explicitly declared external attributes again in the subclasses. The contents of these internal attributes are specified in the body.

<body> ::=
{
[
<field identifier> <field content> |
<eventOut identifier> <eventOut content>
*]**
}

As in the external declaration, internal attributes in the body may contain paths. The difference is that paths in the body can reference these new internal attributes as well as those attributes which paths in the external declaration can reference.

4.4 Handler Declaration

A handler declaration consists of zero or more event handler declarations.

<handler declaration> ::=
*[<event handler declaration>]**
<event handler declaration> ::=
HANDLE *<eventIn identifier>* {
*[<field type> <field identifier> <field content>]**
url *<SF.String>*
*[<field type> <field identifier> <field content>]**
}

In an event handler declaration, a new event handler is defined or an old event handler of the superclass is redefined. An event handler is composed of three components: an associated eventIn, local fields, and behavior code. It enters into execution when the associated eventIn receives events. If the class does not have the eventIn, the event handler declaration is ignored. Local fields are conceptually the same as the fields of classes except that they have no inheritance mode and that their contents may have only constants. Paths cannot be used for the local fields. The local fields are available only to the event handler to which they belong. The field **url** specifies a URL (Uniform Resource Locator) for the behavior code. It must be an **SF.String** constant. When the event handler is invoked, the behavior code is executed. The behavior code may access external attributes and local fields. However, it cannot access internal attributes. The above rules for handler declarations are applied in the same manner to handler declarations in object creation.

When a new object is created from a source object, the two objects have the same set of event handlers. If the new object does not override any event handler of the source object, their behavior for any eventIn is also the same. An object maintains its own local fields for each event handler; the local fields cannot be shared.

4.5 Core Attributes

Core attributes of a class are the external attributes of its topmost superclass which is provided by the OO-VRML system. The topmost superclass is said to be the basis class of the class. If the contents of all core attributes are specified, the class can be instantiated. The objects created from the class have the same set of core attributes as the class. Non-core attributes are usually exploited in order to determine the contents of core attributes. Consider the following example.

```
CLASS Cube EXTENDS Box [
        exposedField SF.Float extent 2
] {
        size ( extent extent extent )
}
```

Since the class `Box` is provided by the OO-VRML system, it is the basis class of the class `Cube`. The class `Cube` has two attributes: an external attribute `extent` and an internal attribute `size` which is a core attribute. The content of the external attribute `extent` is used for the core attribute `size`.

The division criterion for internal attributes and external attributes is irrelevant to core attributes; a core attribute may be internal or external. For instance,

```
CLASS TransformGroup EXTENDS Group [
        TRANSPARENT exposedField MF.Transform children
] {
}
```

the attribute `children` of the class `TransformGroup` is an external attribute and still a core attribute. The only change in the attribute `children` is that its type is refined from **MF.**`Node` to **MF.**`Transform`.

4.6 Interfaces

An interface is used to state explicitly the properties of classes. An interface is a collection of specifications on attributes and eventIns. When a class satisfies all these specifications, it is said to implement the interface. A class can implement several interfaces if there arise no conflicts between the interfaces. An interface declaration consists of zero or more fields, exposedFields, eventOuts, and eventIns with their types. Attributes in an interface declaration do not have contents.

<interface> ::=
INTERFACE *<interface identifier>*
[
[
{ **field** */* **exposedField** *} <field type> <field identifier> |*
eventOut *<eventOut type> <eventOut identifier> |*
eventIn *<eventIn type> <eventIn identifier>*

*]**
]

Suppose that a class A implements an interface B. For an attribute f in the interface B, the class A has the same attribute f of a correct type. It does not matter whether the attribute f of the class A is internal or external; both internal attributes and external attributes are employed in implementing interfaces. Likewise, for an eventIn g in the interface B, the class A also has an eventIn g of a correct type.

An interface introduces a new object type. For example, if an interface A is declared, **SF**.A and **MF**.A are valid types for attributes and eventIns. Classes implementing the interface A are regarded as objects of the type **SF**.A. Since a subclass inherits all the attributes and eventIns of its superclass and the types of the attributes and eventIns can be only refined, it also implements every interface that the superclass does.

5 Locales

An OO-VRML file can describe more than one independent virtual world, which is called locale. Locales are composed of objects, classes, and interfaces. Each locale maintains a local coordinate system. With respect to the coordinate system, it does not share any portion of the world with other locales. An object is visible, or can be used as a source object in object creation, only in the locale to which it belongs. **COMMON** blocks, a special kind of locale, contains objects visible in all other locales.

<oovrml> ::=
[
COMMON { *[<element >]** } /
[**DEF** *<locale identifier>]* **LOCALE** { *[<element>]** }
] /*
*[<element>]**
<element> ::=
<object> | <class> | <interface>

The locale created finally in an OO-VRML file is the main locale, which becomes active first when the OO-VRML system runs. If no locale is created, the objects constructs a main locale with no name.

6 Instantiating Objects

A VRML file describes a virtual world by enumerating nodes, which are the fundamental element of VRML. The nodes construct the scene graph for the virtual world. The values of fields are always constants. Therefore, nodes can be directly used in rendering the virtual world. The fundamental element of OO-VRML is objects, which are the counterpart of nodes of VRML; both nodes of VRML and objects of OO-VRML represent real world objects and concepts. In contrast to nodes of VRML, objects of OO-VRML cannot be directly used in rendering the virtual world because the contents of their attributes may be paths as well as constants. They are first instantiated to derive their instances. The resultant instances are then employed to render the virtual world by the OO-VRML system.

Every instance has a basis object from which it is derived. It has the same set of attributes and eventIns as the basis object. The attributes have the same contents as those of the basis object except that object references in constants and paths are converted into instance references. As the values of attributes are calculated from their contents, the OO-VRML system continues to render the virtual world.

An object can derive more than one instance, depending on its property. If it has the property **SINGLE**, only one instance is derived. If it has the property **MULTIPLE**, more instances can be derived. An object is instantiated to create new instances when it is referenced by attributes of other objects. The attribute must be of an object reference type. If the object has the property **SINGLE** and it has no instance yet, a new instance is derived. If it has already derived an instance, this instance is used for the object afterwards. Hence, only one instance is maintained for an object with the property **SINGLE**. An object with the property **MULTIPLE** creates new instances whenever it is referenced in other objects. The instances derived from the same object are differentiated by their environments.

The environment of an instance stores the context where it is derived. It is composed of three instance pointers, or environment variables, **SELF**, **PARENT**, and **SOURCE**. They are used for resolving object specifiers **SELF**, **PARENT**, and **SOURCE** in paths. The instance pointer **SELF** references the instance itself. The instance pointer **PARENT** references the most recent instance of the parent object which entails the instantiation of the basis object. It may be null if the basis object is not referenced by other objects but instantiated by itself. The instance pointer **SOURCE** references the most recent instance of the source object. For example,

```
DEF P Q {
}
```

the instance pointer **SOURCE** in the instance of the object P references an instance of the object Q. When the object P is instantiated, the source object Q may have more than one instance if it has the property **MULTIPLE**. In such a case, its most recent instance is employed for the instance pointer **SOURCE**.

After the environment of every instance is computed, an attribute whose content contains paths can calculate its value. The structure of paths is built by a sequence of object specifiers and attribute pointers. Therefore, a path can be evaluated to a specific value by substituting instance pointers of environments for object specifiers in it. Since paths declare one-way equality constraints between attributes, they are semantically re-evaluated at every iteration of the rendering process by the OO-VRML system, where various optimization techniques could be applied. In contrast to paths, constants are evaluated only once when objects are instantiated; the evaluation results may be replaced with new constant values by event handlers. In this way, the OO-VRML system calculates the value of each attribute and renders the virtual world.

The idea of instancing in VRML allows a node to be shared by other nodes. In OO-VRML, the instantiation mechanism allows the definition of an object with the property **MULTIPLE** to be shared in deriving its instances. The instance derived from an object with the property **SINGLE** can also be shared by other instances. Thus, the two mechanisms can be considered to have the same design purpose basically.

7 Conclusion

We have shown how VRML can be extended to incorporate object-oriented design principles. The new language, OO-VRML, is an extension of VRML toward object-orientation. It also adopts other concepts such as paths and locales to support the efficient implementation of virtual environment systems. We have developed a prototype of the OO-VRML system. We are testing the utility of the language by running the OO-VRML system on a number of examples. The future works include the development of a full OO-VRML system containing a browser. The support of general constraints for the contents of attributes is also to be considered.

Acknowledgments

The main design ideas of OO-VRML should be ascribed to VRML Object-Orientated Extensions Working Group. We would like to thank the members of the working group, especially Stephan Diehl and Jean-Francis Balaguer.

References

[1] The Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997. http://www.vrml.org/Specifications/VRML97, 1997.

[2] Stephan Diehl. VRML++: A Language for Object-Oriented Virtual Reality Models. In Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems, Beijiing, China, 1997.

[3] Stephan Diehl. Extending VRML by One-Way Equational Constraints. 1997.

[4] Sungwoo Park and Taisook Han. Object-Oriented VRML for Multi-user Environments. In Proceedings of the Second Symposium on the Virtual Reality Modeling Language, Monterey, USA, 1997.

[5] VRML Object-Oriented Extensions Working Group. Discussion mails. http://www.cs.uni-sb.de/RW/users/diehl/ooevrml/, 1997.