Independent Scene Element Representation in the *basho* Virtual Environment Framework

Florian Mannuß, André Hinkenjann

Computer Graphics Lab
Bonn-Rhein-Sieg University of Applied Sciences (BRSU)
53754 Sankt Augustin
E-Mail: Andre.Hinkenjann@fh-brs.de

**Abstract:** We present *basho*, a lightweight and easily extendable virtual environment (VE) framework. Key benefits of this framework are independence of the scene element representation and the rendering API. The main goal was to make VE applications flexible without the need to change them, not only by being independent from input and output devices. As an example, with *basho* it is possible to switch from local illumination models to ray tracing by just replacing the renderer. Or to replace the graphical representation of the scene elements without the need to change the application. Furthermore it is possible to mix rendering technologies within a scene. This paper emphasises on the abstraction of the scene element representation.

**Keywords:** virtual environment framework, scene element representation, software engineering

1 Indroduction

Today a wide variety of VE-Frameworks, like Avango [Tra01], Lightning [BLRS98] and VR Juggler [CNBH+02] are available and VE-Applications have already advanced in many industrial areas as in automotive field and engineering. All of these frameworks embed the scene element representation into the framework and couple it with the rendering. Avango is based on a distributed scene graph that encapsulates and extends OpenGL Performer [RH94]. New scene elements, like sound, are added by deriving from the scene graph’s base class. In order to replace Performer with another scene graph the whole internal scene graph has to be rewritten. As a result all applications are bound to the functionality and rendering quality of Performer. Lightning and VR Juggler use managers to hold the different types of scene elements, like sound or graphics. Those managers are, too, responsible to create the output. They are exchangeable, but the application has to be modified. Also, all scene elements are coupled to its managers that act as renderers.

*basho* is designed to overcome the dependence between scene elements and APIs, like between a scene graph and the rendering API, by introducing an abstraction layer. All scene elements are described through this abstraction layer. They are not, like in the other three frameworks entities of the utilised APIs. Instead they try to represent an entire object.
in the virtual world. A table is not a loose collection of geometric shapes, materials and (if needed) a weight for physics simulation. It represents a single scene element. The shape, the material and the weight are only attributes of the table. In our terminology this table is a virtual object. Another result of using this abstraction layer is the separation of scene data and rendering. It is possible to exchange the renderer without the need to change the scene elements that store the data. In our context rendering is not bound to graphical rendering. It can be physics, behavioural or sound rendering, for example.

The following section gives an overview of the whole framework followed by section 3 describing the virtual object abstraction layer and section 4 the rendering abstraction. Section 5 examines the efficiency of the abstraction layer.

2 System Overview

Our framework consists of a Kernel managing the whole system. Besides this managing task the Kernel implements no functionality. All functionality is loaded at run-time through plug-ins to the Kernel. All plug-ins that have to be loaded are specified in a simple configuration file. Changing the renderer for example is done by changing the configuration file.

Inside the Kernel four managers are the storing and calling units for the different types of plug-ins as sketched in figure 1. A mainloop controls the entire system and calls the different managers.

The inputDeviceManager is responsible for managing the input devices, represented as plug-ins. Here input devices are seen as data sources, implemented using the VRPN library [THS+01] to access the devices. Data acquired through input devices is interpreted through an action stored in the actionManager. Actions can be responsible for moving the camera or interacting with virtual objects, for example. The action used to interpret the input data
can be specified through the input device that sends the data. Otherwise the active action in the actionManager is used. To set an action as active an action selector has to be used. The selected action can manipulate the scene administered by the sceneManager or some of the loaded renderers through sceneRenderer. The virtual scene stored in the sceneManager is composed of virtualObjects. All attributes a virtual object can aggregate are loaded via plug-ins. All renderers are managed through the sceneRenderer.

At the moment scene plug-ins that encapsulate Open SceneGraph (OSG) [BO] for graphics, OpenAL [Sof] for sound, ODE [Smi02] for physics and a scene plug-in for additional ray tracing data exist. Furthermore, renderers are available to process the scene data. The ray tracer is independent from the graphical scene plug-in and the OpenGL renderer uses directly the OSG rendering functionality and has to have access to the root node stored in the OSG scene plug-in. The same holds for the sound and physics renderer. Both utilise the built-in rendering functionality of OpenAL and ODE and need access to their data.

The managers called by the mainloop are the inputDeviceManager and the sceneRenderer. Further scene or application control is given through two user deriveable methods called before and after the sceneRender has been called.

3 Virtual Objects

A virtual scene is composed of virtualObjects. VirtualObjects try to imitate objects of the real world. Therefore a virtualObject is not only a node in a scene graph. It stores all attributes that are needed to describe an object in the virtual world. For example, if the virtualObject describes a table as in figure 2, then the geometry attribute stores all geometry data needed to describe this table, here the table top and the table legs. The material of the table is stored in the material attribute. Also, a physical attribute (the weight of the table) is added which does not belong to the scene graph storing the graphical appearance of the table.

All attributes that can be added to a virtualObject must be derived from the base class virtualObjectAttribute and must be stored in plug-ins that are loaded at run-time. By storing all attributes in plug-ins and by the fact that all attributes encapsulate the underlying storage of the data, independence of the underlying data storage is achieved. It is possible to store the graphical data in Open SceneGraph or OpenGL Performer and the behaviour of the virtualObject does not change at all. Since the sceneManager is able to load any number of scene plug-ins the behaviour of virtualObjects can be extended in any way. On the other hand only plug-ins that are needed must be loaded.

All virtualObjects in the virtual world are identified by a unique name. Their attribute objects are instead identified by a unique number. virtualObjects by itself are empty. They only have a name and the ability to store attributes. All functionality is aggregated through attributes. They are therefore containers for attributes that belong to the same entity in the virtual world. Attributes of virtualObjects cannot be accessed directly. They are completely encapsulated and can only be accessed indirectly through the getAttributeData(...) and setAttributeData(...) methods of their storing virtualObject. Both, virtualObject and the
derived attribute act as a proxy [GHJV96] to the scene data, e.g. the geometry of the table.

Furthermore, virtualObjects can be grouped using a virtualObjectContainer to build complex entities in the virtual world. For the table of figure 2, table top and legs have the same material. If the legs are made of metal and the top of glass, two virtualObjects are needed. Creating the virtualObject “Table”, these two objects are grouped in one virtualObject-Container. As seen in figure 3, a virtualObjectContainer object can store any number of virtualObjects and is stored in the sceneManager as virtualObject. A virtualObjectContainer itself can store attributes. In the example the weight of the table would be stored as an attribute of the container object.

3.1 Creating virtualObjectAttributes

For creating virtualObjects and their attributes two methods are available. Either the sceneManager calls a sceneLoaderInterface in order to let the plug-in load its scene data. The implemented sceneLoaderInterface is responsible to load the scene data and create the virtualObjects and their attributes out of the loaded data. Furthermore, virtualObjects can be created via the sceneManager and attributes are added by hand. Because of the design principle that the system or the VE-Application has to be independent of the loaded plug-ins an attribute object cannot be created directly through the new command. If this would be possible the system would be bound to the the plug-in that implements this attribute.
Figure 3: Class diagram of the virtualObject. Every virtualObjectAttribute has to register itself automatically with one prototype at the virtualObjectAttributeFactory. If a new virtualObjectAttribute has to be created the virtualObject asks the factory to create it. The registration and object creation is symbolised through the dotted lines. A scene plug-in consists of one sceneLoaderInterface and an arbitrary number of virtualObjectsAttribute classes.

To avoid this, attributes are added to the virtualObject indirectly through the virtualObjectAttributeFactory using a prototype factory pattern [GHJV96]. The factory knows all loaded attributes and can identify them by their name, since all attributes of a scene plug-in have to register themselves with their name and one prototype object at this factory after the plug-in has been loaded. The factory uses this attribute prototype to clone the desired attribute and adds it to the virtualObject, either with a user specified identifier or an automatically assigned one. Adding an attribute to virtualObject is done by calling

\[
\text{void virtualObject::addAttribute( attrName, attrData, attrIdentifier)}
\]

The virtualObject asks the virtualObjectAttributeFactory to create the attribute object from the prototype with the specified name attrName.

### 3.2 Accessing virtualObjectAttributes

Scene plug-ins of the same type have to be exchangeable. Therefore the material attribute of a graphical scene plug-in for example has to have the same functionality as the one of another graphical scene plug-in and it has to be transparent to the user which plug-in is loaded. In order to achieve this goal two things are needed. First an independent description of the attributes functionality is needed. Through this description it is possible to get/set data of the attribute independently of the plug-in providing the attribute. In our case an enumeration is used for this independent description. All elements that can be accessed in the attribute are listed in this enumeration. Second a data type class hierarchy is needed to be able to pass the different data types through the set/get interface.
namespace objMaterial
{
    enum properties
    {
        // dataCapsuleEl::Colour
        DIFFUSE_COLOUR = 0,
        AMBIENT_COLOUR,
        SPECULAR_COLOUR,
        // dataCapsuleEl::Double &
        SHININESS,
        ...
    };

    enum faceEnum
    {
        FRONT = 10,
        BACK,
        FRONT_AND_BACK
    };

    // Access the face type in the data map.
    // dataCapsuleEl::Integer is the storage data type.
    const std::string Face("face");
};

Figure 4: Implementation independent attribute description (part of the material description). The comments describe the set/get data type.

Figure 4 shows an example for such a description. In this case it is a part of the description for the material attribute. As seen in the enumeration properties, the material attribute itself is a composition of different properties. This is valid for all attributes. Here the properties are DIFFUSE, AMBIENT, SPECULAR_COLOUR and SHININESS. With this definition the properties data can be accessed through its storing virtualObject. The two functions are

\[
\text{void virtualObject::setAttributeData( sendReceiveData *attrDescr );}
\]

and

\[
\text{dataElSPtr::Data virtualObject::getAttributeData( sendReceiveData *attrDescr );}
\]

sendReceiveData is the storage class to specify the attribute and its desired property.

In order to describe the usage of the attribute description an analogy to set/get-method calls of an object can be used. A get call can be written in general form as

\[
<\text{return-value}> \text{object.getMethod( <Parameter list> );}
\]

Specifying the get-method is done through <Parameter list>. object corresponds to the attribute identifier used in the virtualObject. The property of the attribute in this analogy is the member function name. Parameters specifying the properties are similar to...
the parameter list. If the material attribute with identifier 2 is of interest and only the front
face has to be gathered from the virtualObject with name “Table”, the get call could look like

\[
\text{virtualObjectAttribute } *\text{attr} = \text{“get attribute with identifier 2 from virtualObject <Table>”}
\]
\[
\text{colour-value} = \text{attr->getDiffuseColour( faceEnum::FRONT );}
\]

Translating this method call in basho set/get terminology:

\[
\text{virtualObject } *\text{table} = \text{sceneManagerObj->getVirtualObject(“Table”);}
\]
\[
\text{sPtrSendReceiveData } \text{attrDescr;}
\]
\[
\text{attrDescr->setAttributeIdentifier( 2 );}
\]
\[
\text{attrDescr->setAttributePropertyIndentifier( objMaterial::DIFFUSE_COLOUR );}
\]
\[
\text{attrDescr->addAttributePropertyParameter( objMaterial::Face, objMaterial::FRONT );}
\]
\[
\text{dataElSPtr::Data } \text{dCol} = \text{table->getAttributeData( attrDescr );}
\]

After getAttributeData(...) was called the virtualObject searches the attribute with the
given identifier. If found, getAttributeProperty(...) of the attribute is called. This method is
responsible to implement the properties specified for this attribute type, like the material in
figure 4. setAttributeProperty(...) is the method for the set data-calls.

All data passed to or from an attribute property is always of base type of the data class
hierarchy. At its destination the base object has to be casted into the right type. These
casting operations should be dynamic casts to ensure that the data is of the right type. It is
obvious, that the attribute description must contain a textual description of the data types
used for storing the data. Otherwise, it is impossible to cast into the right type. In figure 4
the data type is explained in a comment.

With the construction of the data class hierarchy and the attribute description the get-
and set-methods are independent of the plug-in. An attribute does not need to implement
the complete attribute description. Only supported properties need to be implemented. If
an unsupported property is requested the attribute should ignore it and return NULL as
data value to indicate a failed request.

All objects of data class hierarchy are reference counted [Mey96] and in conjunction
with smart pointers [Mey96] they provide, if used correctly, an automatic garbage collection
mechanism.

3.3 Connecting virtualObjects

virtualObjects can be connected with each other using a special connection object. Possible
connections are between virtualObjects or between attributes of virtualObjects as illustrated in
figure 5. If single attributes of virtualObjects are connected, only changes of these attributes
are transmitted.
Figure 5: Possible virtualObject connections. a) shows a simple connection between two virtualObjects. As seen in b), two or more virtualObjects can be connected to one virtualObject. c) shows a connection with one constraint. Here one virtualObject is connected to two virtualObjects. This connection can result in undefined state cases, because both input virtualObjects can overwrite their data passed to the connected object. A reasonable appliance is, that one of the two input objects is used by the constraint object to control the connection. d) shows two virtualObjects connected only through one of their attributes.

Connections form a directed graph. Unfortunately, such a graph can build cycles. If cycles are not cut, an infinite recursion occurs. Two methods are available to handle cycles. First, the cycle can get cut. Second, the cycle is kept, but is interrupted after the first pass and will be continued after the actual frame.

Additionally every connection can own a constraint object. This object can cancel the connection, if certain preconditions are not fulfilled. Constraint objects can directly manipulate the data passed through them. This can be used, for example to add an offset to the data.

4 Rendering

The sceneRenderer is closely coupled with the sceneManager but not with the other two managers as seen in figure 1. This is useful, because the render plug-ins access the virtualObjects through the sceneManager. However, rendering plug-ins cannot be interconnected. The only connection is the indirect connection through the virtualObjects all renderers can manipulate.

Being able to configure the order in which the renderers are called is important. If for example a render which manipulates the virtualObjects is called after the graphics renderer, the manipulation is delayed one frame. Therefore a physics renderer should always be called before the graphics rendering.
Graphic renderers are not connected directly to the sceneRenderer. They are instead plug-ins of a display abstraction rendering plug-in using Open Producer [Bur]. This display abstraction is responsible for creating the output window, the camera, calling all renderers and for merging the rendering results by comparing the depth components of every pixel. The ability to use more renderers in one display abstraction leads to the possibility of mixed rendering. Here two or more renderers are calculating one scene. Implementing every renderer as one thread, it is possible to do this rendering on different CPUs in parallel.

5 virtualObject-layer efficiency

In order to stay independent of the APIs to represent the virtual scene an abstraction layer was introduced. When requesting or setting data of a virtualObjectAttribute all data is stored in a temporary data structure. This results in creating the data structure and copying the data into it. When directly using a scene graph for example these two steps are not needed. However, if we interact with the scene only transformations need to be send and the overhead is minimal. In case of rendering the overhead vanishes if both, the renderer and the scene use the same API. This is the case for the Open SceneGraph scene and renderer plug-in. In this case the renderer has direct access the root node of the scene graph. When using a distributed ray tracer instead, all geometric data has to be encapsulated in those temporary data structures and sent to the ray tracing servers, leading to a start up overhead depending on the scene size. In the following frames only the deltas have to be send.

Figure 6: Initial test scene of size 1024x768 with about 2000 triangles. The glider got cloned to become a 40k and 400k triangle scene.

Figure 7: Fraction of the initial framerate and call time for different numbers of get-calls a frame. Initial framerates for 2000, 40k and 400k triangles are 560, 190 and 26 frames/s. Every get-call returned 512 surface normals.

In order to measure the time overhead needed to get data from virtualObjectAttributes
the following test case was constructed. Using the scene of figure 6 with about 2000 triangles, while utilising Open SceneGraph (OSG) in the scene and render plug-in, the time needed to render the scene without any setter/getter calls was measured first and compared with the time needed by the OSG build in viewer. The frame rate was the same for both: 560 frames/s on an ATI Radeon 9700 graphics card with an Intel Xeon as CPU. The glider of the scene was cloned in order to produce scenes with 40,000 and 400,000 triangles. Measured frame rates were 190 and 26 frames/s.

Then all 512 per vertex normals were requested from the glider object 1, 10, 50 and 100 times a frame. Every getter-call returned an array with 512 vector objects for all three scenes leading to 512, 5120, 25600 and 51200 vector objects that have to be created each frame. Creating those objects consumes nearly all time of this operation and the time needed to find the requested \textit{virtualObjectAttribute} can be neglected. All results were averaged over 1000 frames.

As seen in figure 7 the framerate drops significantly with an increasing number of getter-calls in the case of the 2000 triangle model. Since the time needed to request the surface normals stays constant for all scenes, the framerate loss for the bigger scenes is less significant as for the small scene. Comparing the results for 100 getter-calls the speed loss for the 2000 triangle model is 35\%, for the 40k model it is 18\% and for the 400k model it is only 4\%.

All measurements are made using getter-calls. However, data has to be set, too. Since the setter-calls follow the same logic as the getter-calls, the time needed is similar.

6 Conclusion and Future Work

We presented a lightweight and extensible VE framework that is independent of the actual representation of the scene element data and the rendering. In order to gain this independence we introduced an abstraction layer that describes the elements in the virtual world. Those virtual objects try to rebuild objects of the real world and are therefore a composition of different attributes. Attributes are the proxy to the underlying data stored in a third party API. Using a prototype factory to create attributes for virtual objects ensures that a VE application is independent of any scene plug-in storing the attributes encapsulating a certain API. Additionally, by providing a pair of set/get methods to access all parameters of the attributes a ”textual” description of the attribute parameters has been introduced together with an extendable data type class hierarchy.

\textit{basho} is still under development and there are many areas to extend the framework. Work has to be done to extend \textit{basho} to distribution and collaboration, for example by using the set/get method interface of the virtualObject to distribute changes to other \textit{basho} instances. Additionally, connection objects can be used to implement distribution.

At the moment scripting functionality is integrated with the goal to be able to do runtime configuration and to write entire applications and plug-ins as scripts. Furthermore, we plan to use this scripting facility to develop user interfaces.
References


