Informatics in Education, 2004, Vol. 3, No. 2, 191–218
© 2004 *Institute of Mathematics and Informatics, Vilnius*

Virtual SceneBean: a Learning Object Model for Collaborative Virtual Learning Environment

Jinan FIAIDHI
Department of Computer Science, Advance Technology and Academic Centre, ATAC 5015
Lakehead University
Thunder Bay, Ontario P7B 5E1, Canada
e-mail: jfiaidhi@mail1.lakeheadu.ca

Received: July 2004

Abstract. It is commonly agreed that a well-balanced mix of collaboration, training and simulation eventually produce a superior learner. Today's collaborative design and learning environments integrate variety of interactive objects as well as many technological aspects to achieve such balance. Unfortunately, the actual profit of the resulting learning systems is largely reduced by poorly represented interactive objects as well as poor interlinking between such objects. In particular, such objects appear isolated: they neither can be modified sufficiently (e.g., by choosing parameters or enhancing functionality) nor be interlinked properly with their context (e.g., by synchronizing with a guided tour or metadata). We are presenting in this article a model for representing virtual and 3D scenes as learning objects. The model utilizes notions and techniques based on Scene Graphs, X3D, Java3D, and SceneBeans. The prototype accompanied with a simple client-server protocol for exchanging and viewing the 3D SceneBeans. This research aims to extend this protocol by utilizing Sun JXTA primitives to link to the POOL of other learning objects repositories.

Key words: virtual learning environment, virtual/3D learning objects, SceneBeans, X3D, Java3D.

1. Introduction

Virtual learning environments (VLEs) can provide rewarding teaching and learning experiences. In terms of academic results, virtual learning environments can represent a more successful learning environment and have proven to be motivating contexts for learning. In these virtual environments the learning experience can be flexible, more accessible and inclusive. Not only are these environments often a more economically viable option, but they also allow specialist tuition and knowledge to transcend geographical boundaries (Brogan *et al.*, 1998).

Traditionally such environments are used to support distance and online learning. There are over 100 packages, available on the market, that have been developed by Universities or commercial companies, most common ones include: Blackboard, WebCT, WebFuse, CoSE, TopClass, WebEx, VNC, SCORM, and Tango. Although these packages offer course designers some encouraging built-in facilities (e.g., communication and collaboration tools, passive interactivity, administrative tools, evaluation tools, and helpful interfaces), they luck supplying some intrinsic capabilities for the learning materials reusability, active interactivity, widely acceptable indexing and metadata standard, interoperability, and of having an open source. These intrinsic capabilities are quite vital for the future eLearning applications (c.f., semantic web). However, many emerging standards (e.g., Learning Objects LOs standards) and technologies can be used currently to enrich and provide greater interactivity within any virtual learning environment. Indeed, the current technologies of VLEs have many innovative and exciting possibilities especially in the direction of virtual reality using 3D modeling, multisensory communication, and active/immersive interaction. Such technologies provides students with more learning opportunities way beyond those offered by the eLearning in its current state but careful modeling, planning and innovation will be required to ensure that the potential for the scope of delivery is reached (Carpenter and Anderson, 1996). According to Stephen McWilliam, V.P. of Astound (`www.astound.com`) a Canadian company specialized in Web conferencing, who offered statistics about the degree of retention of learned material as a function of the medium use:

Read	10%
Visual	20%
Audio	30%
Visual & Audio	70%

In other words, the greater the number of senses involved, the greater the retention. With virtual reality we may obtain a tremendous degree of involvement and engagement, and thus greater learning and retention. In this direction, the animated objects are needed to play the role of teachers or guides, team-mates or competitors, or just to provide a source of interesting motion in virtual environments. For a virtual environment to be compelling, the animated objects must have a wide variety of interesting behaviours and must be responsive to the actions of the user/learner as well as to comply to one of the learning objects standards. The difficulty of constructing such synthetic object currently hinders the development of these environments, particularly when reusability, learner-centered and reprogrammability are required by both the designer and the learner/user.

This article addresses a solution to this problem by providing a new model for VLEs based on the notions of SceneBeans and Learning Objects. This new model is designed to provide active student engagement with active learning. This research is part of our ongoing research to establish multimedia learning objects repository for the academia at LU as well as to establish a proper search engine for such learning objects (Fiaidhi and Mohammed, 2004; Fiaidhi *et al.*, 2004; Fiaidhi *et al.*, 2003a), and (Fiaidhi *et al.*, 2003b).

2. Related Research

Collaborative virtual learning environments (CLEs) traditionally were studied in classroom-based environment at first for tasks such as industrial team training, collaborative design and engineering, and multiplayer games (Singhal and Zyda, 1999). Moreover, much work in the area of enabling effective collaboration in CLEs has focused on developing the virtual reality metaphor to the point where it attempts to completely mimic collaboration in real environments (Benford *et al.*, 1995b; Capin *et al.*, 1998). In particular, much attention has been paid to user embodiment (Benford *et al.*, 1995; Era *et al.*, 1998; Snowdon and Tromp, 1997). However, much more recent work was focused on Web-Based CLEs (Jianhua *et al.*, 2001). Web-based CLE systems can be divided into two categories, one is *asynchronous* system, and another is *synchronous*. The influential asynchronous system includes First Class, CSILE/Knowledge Forum, Learning Space, WebBoard, and WebCT; synchronous system includes Conference MOOS, WebChat Broadcasting System, and Microsoft Netmeeting.

Although the above mentioned CLE research focused on interactive instructional visualization, not much of the research work focus on planning for change (i.e., flexible and easy to adopt to new and changing user requirements as well as to have reusable learning materials). This eventually means that CLEs systems must be built with open requirements out of reusable components conforming to a plug-in architecture. Although the component-based (CB) solutions developed to date are useful, they are inadequate for developers and instructors building directly CLE systems in which the components must respond to the meaning of the content as well as comply with certain widely acceptable standards. Hence it essentially means that the CLE system must be based on the IEEE learning object metadata notion as well as to be based on CB technology. In this direction only very few research attempts can be cited in the literature which address CLE as CB reusable systems (e.g., *multibook* CLE of the Technical University of Darmstadt (El Saddik *et al.*, 2000), the WebDAV-Collaborative Desk of the Institute of Telematics (Qu *et al.*, 2000) and *JASMINE* (Shirmohammadi *et al.*, 2003) as well as the *Java Multimedia Telecollaboration Kits* (Oliveira *et al.*, 2003) from Ottawa University. Indeed the late mentioned research tried to solve many issues related to CLE design flexibility based on CB technology and metadata standardization, but the issue that remain to be answered is how to model their basic visualization objects (i.e., active/flexible and virtual scenes) and how to structure their accompanying metadata. This article utilizes the notion of Scene Graph as a model for representing CLE's active and virtual scenes.

3. Describing Active Scene Using Scene Graph

Scene graph is a common model for storing and retrieving graphical scenes as used in computer graphics (see `http://en.wikipedia.org/wiki/Scene_graph`). A **scene graph** is a general data structure commonly used by vector-based graphics editing applications. Many graphical applications use the model of scene graph to model flexible shapes (e.g., AutoCAD, Adobe Illustrator and CorelDraw). The scene graph contains the pictorial data items that can be edited and displayed. Each node in a scene graph represents some atomic unit of the document, usually a shape such as an ellipse. However with scene graphs, shapes themselves can be decomposed further into other nodes. The simplest type of scene graph uses an array or linked list data structure, and displaying its shapes is simply a matter of linearly iterating the nodes one by one.

Other common operations, such as checking to see which shape intersects the mouse pointer (e.g., in a GUI-based applications) are also done via linear searches. For small scene graphs, this tends to suffice. Larger scene graphs cause linear operations to become noticeably slow and thus more complex underlying data structures are used, the most popular being a tree. In this case, the scene graph can contain smaller scene graphs as nodes, and formal type declarations of such structures often include themselves recursively as members. These "subscenegraphs", depending on the application, can be known to the user and even user-defined and editable. A common feature, for instance, is the ability to group related shapes into a compound shape which can then be moved, transformed, selected, etc. as easily as a single shape. An operation applied to a group automatically propagates its effect to all of its members. In many programs, associating a geometrical transformation matrix at each group level and concatenating such matrices together is an efficient and natural way to process such operations.

In 2D cases, scene graphs typically render themselves by starting at the tree's root node and then recursively drawing the child nodes. The tree's leaves represent the most foreground objects. Since drawing proceeds from back to front with closer objects simply overwriting farther ones, the process is known as employing the Painter's algorithm. In 3D systems, which often employ depth buffers, it is more efficient to draw the closest objects first, since farther objects often need only be depth-tested instead of actually rendered. Depending on the particulars of the application's drawing performance, a large part of the scene graph's design can be impacted by rendering efficiency considerations.

In 3D video games such as Quake, for example, binary space partitioning (BSP) trees are heavily favored to minimize visibility tests. BSP trees, however, take a very long time to compute from design scene graphs, and must be recomputed if the design scene graph changes. Scene graphs for dense regular objects such as heightfields and polygon meshes tend to employ quadtrees and octrees, which are specialized variants of a 3D bounding box hierarchy. Since a heightfield occupies a box volume itself, recursively subdividing this box into eight subboxes (hence the 'oct' in octree) until individual heightfield elements are reached is efficient and natural. A quadtree is simply a 2D octree.

4. From Scene Graph to 2D SceneBeans

Originally SceneBeans are introduced as a java component-based 2D animation framework by Pryce and Magee during 2001 (Pryce and Magee, 2001). SceneBean *animation* encapsulates both a scene graph and the behaviours that animate the nodes of that graph. It acts as the manager for the behaviours encapsulated within it, routing commands and events. Most importantly, a SceneBean animation is also a scene graph node, since this means we can compose animations, applying transformation and further animation as required. SceneBeans conform to industrial standards: Java and XML, and because of its component nature, allows extensibility of the framework within its "domain-specific visual and behavioural components" (Magee *et al.*, 2000).

The SceneBeans framework provides programmers with a convenient programming model for creating and controlling interactive multimedia objects, and a useful set of components that can be plugged together within that framework. Indeed, it is not practical to expect learners to write Java programs in order to define animations for example for use in applications, and even for experienced programmers the edit/compile/debug cycle is slow and frustrating when fine-tuning animation parameters are involved. To simplify the authoring/description process XML format has been used by SceneBeans which requires a parser to translate the XML document into interactive multimedia or Animation objects. The XML document type definition (DTD) used by the SceneBeans parser is relatively minimal compared to DTDs for similar applications, such as the W3C's Scalable Vector Graphics (SVG) standard. The DTD does not prescribe a limited number of component types and their options, but instead describes compositions of components that the parser loads dynamically and manipulates generically through the JavaBeans APIs.

A SceneBean object is described using an XML metadata. The top-level of the description is called <animation> which contains five types of sub-elements: a single <draw> element defines the scene object to be rendered; <define> elements define named scene graph fragments that can be linked into the visible scene graph; <behaviour> elements define behaviours that animate the scene graph; <event> elements define the actions that the animation performs in response to internal events; and <command> elements name a command that can be invoked upon the animation and define the actions taken in response to that command. Both <draw> and <define> elements can contain the elements <primitive>, <transform>, <style> and <compose>.

SceneBeans defines object behaviour with the "behaviour" element and then animating the parameters of scene graph nodes with the "animate" tag. The behaviour tag is used to instantiate behaviour beans: the parser maps the algorithm of the behaviour to a Java class the same way as it does for scene graph nodes, although it searches a different set of packages. Like scene object nodes, param tags are used to configure behaviours by setting their Java Bean properties. SceneBeans is a framework for two-dimensional animations based on Java Beans. An animation in SceneBeans is defined as a "scene graph" – Java Beans components that form a directed a cyclical graph (DAG). A scene is defined in terms of compositions and transformations of geographic shapes. Animations are defined in XML documents and contain commands that instantiate and bind Java Beans components to each other.

Leaves of the DAG draw geographic primitives such as shapes (i.e., ellipses, polygons), text and images. Nodes differently from than leaves combine the primitives into complex scenes, either by composing more than one scene graph or by modifying the way a scene graph is rendered. Scene graphs are composed using Composite nodes. There are six types of Composite node: Layered, Switch, Union, Intersection, Subtraction and Difference. Layered nodes draw its subgraphs one above the other. Switch nodes draw its subgraphs at a time selected by a bean property. The other four types of node perform constructive area geometry operations on their subgraphs.

Two other types of node, Style and Transform, modify the way a graph is drawn. Style nodes set the fill and line properties used to draw their subgraphs. Transform nodes apply a positional transformation to their subgraphs. Instead of one type of Transform node that would require a transformation matrix as a parameter, SceneBeans use four types to facilitate the composition of animated transformations: Translate, Rotate, Scale and Shear.

Scene graph nodes, being Java Beans, expose modifiable properties at their interface. What is done in SceneBeans exclusively is that these properties control their appearance upon rendering. The Circle bean, for example, exposes a radius property that specifies the radius of the circle drawn, and the Translate bean exposes x and y properties that specify the translation to be applied to its subgraph.

SceneBeans provides components, termed behaviours and activities, which automate the animation of scene graphs instead of explicitly locating nodes in the graph. The animation is then easily accomplished by modifying bean properties in the graph before drawing each frame. A behaviour bean encapsulates a time-varying value and periodically announces events that contain the current value. Some types of behaviour are: DoubleBehaviour, which encapsulates a double-precision real number value, PointBehaviour for point values, ColorBehaviour for a color value, and others. A node in the scene graph is then animated by registering an event-listener to behaviour. The event-listener then routes announced changes of the behaviour's value to a property of the bean.

Behaviours can encapsulate any time-varying values such as the location of the mouse cursor or the size of a window. However when used within animations, most behaviours will periodically announce the value of a time-varying function. These behaviours implement the Activity interface, and are called activities or active behaviours.

The Animation class acts as a façade for a scene graph and the active behaviours that modify the graph. An Animation is a composite scene graph node that layers its subgraphs above one another allowing for it to be easily embedded into a larger scene graph. An animation manages all active behaviours that animate its subgraphs, and is itself an activity that can be managed (see Fig. 1).

Actually, each activity is managed by an Activity Runner that is responsible for invoking its perform Activity method. Activities and their runners can be organized as a hierarchy. The root activity runner is a thread that iteratively calculates the duration of each frame and passes the duration down to the activities that it manages. Intermediate nodes of the hierarchy act both as activities invoked by a higher Activity Runner and as

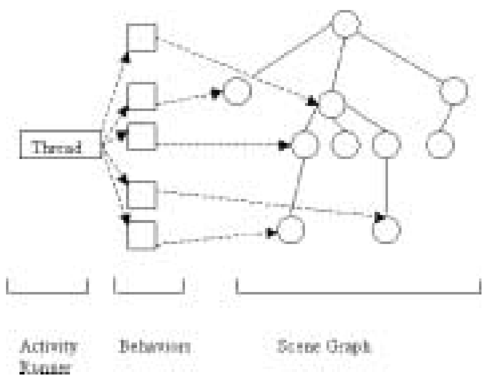


Fig. 1. SceneBeans execute behaviours, which in turn modify the properties of the Scene Graph.

Activity Runners for lower nodes. The Animation class provides a control interface for use by applications. Animations can send commands to and receive Animation Events from an animation. Commands and Animation Events are identified by textual names. Animations respond to commands or internal events by performing one of six types of actions:

- change the bean parameter values of scene graph nodes or behaviour beans;
- start active behaviours;
- stop active behaviours;
- reset the internal state of an active behaviour to the original values with which it was created;
- invoke a command of an embedded animation;
- announce an event to the application or animation in which it was embedded.

Complete control over an animation's behaviour and appearance is almost achieved using a combination of the above action types.

User interaction with the animation is handled by scene graph beans derived from the Input class. The Mouse Click and Mouse Motion classes handle mouse input. Mouse Click indicates that a portion of the scene graph can be clicked on by the mouse and announces Animation Events when the user presses or releases mouse buttons on the visible portions of that subgraph. Mouse Motion reacts to the user moving or dragging the mouse, feeding the location of the mouse pointer and the angle from the origin to the mouse pointer into other scene beans. Fig. 2 illustrates an XML metadata for an animated copter scene.

The <primitive> element on line 19 of Fig. 2 instantiates a bean that implements a Primitive interface. The type attribute defines the concrete type of bean, in this case a polygon. The SceneBeans parser maps the type name to a Java class by capitalizing the first letter of the type name and then searching a list of packages for a class with that name. The <param> element (lines 20 to 24) is used to set the values of the bean properties. Here, the <param> elements define the polygon to have four points, and give the coordinates to those points. The primitive on line 19 is defined as a rotor (line 20).

The plural "rotors" is defined on line 28 to be three singular rotors equal-angle apart from each other. One is pasted to the coordinates assigned. The second is rotated (2 * pi/3) degrees away from the original coordinates. The third is pasted in between at (4 * pi/3) degrees from the original coordinates.

Documents describe animated graphics by creating and naming behaviour beans with the <behaviour> element and then animating the parameters of scene graph nodes with the <animate> tag. In the example given in Fig. 3, a behaviour "rotor-spin" is defined to spin the rotors like a helicopter would spin them. The <behaviour> tag on line 4 is used to instantiate behaviour beans. The parser maps the algorithm of the behaviour to a Java class the same way it does for scene graph nodes – although it searches a different set of packages. Like scene graph nodes, <param> tags are used to configure behaviours by setting their Java Bean properties. Behaviours must be identified by an id attribute so they can be referred by an <animate> element within the scene graph. Animate elements create a binding between behaviour and a property of a bean so that the behaviour modifies the value of the property over time, therefore creating an animation.

```
<?xml version=“1.0”?>
<animation width=“256” height=“256”>
  <behaviour id=“rotor-spin” algorithm=“Loop” state=“${rotor_state=
                                                            stopped}”>
    <param name=“from” value=“0.0” />
    <param name=“to” value=“2*pi” />
    <param name=“duration” value=“1.0” />
  </behaviour>
  <command name=“start”>
    <start behaviour=“rotor-spin” />
  </command>
  <command name=“stop”>
    <stop behaviour=“rotor-spin” />
  </command>
  <define id=“rotor”>
    <primitive type=“polygon”>
      <param name=“pointCount” value=“4” />
      <param name=“points” index=“0” value=“(0, 0)” />
      <param name=“points” index=“1” value=“(-16, 96)” />
      <param name=“points” index=“2” value=“(0, 100)” />
      <param name=“points” index=“3” value=“(16, 96)” />
    </primitive>
  </define>
  <define id=“rotors”>
    <style type=“RGBAColor”>
      <param name=“color” value=“000000”/>
      <primitive type=“circle”>
        <param name=“radius” value=“12” />
      </primitive>
    </style>
    <transform type=“rotate”>
      <param name=“angle” value=“1.0” />
      <animate param=“angle” behaviour=“rotor-spin” />
      <style type=“RGBAColor”>
        <param name=“color” value=“888888”/>

        <paste object=“rotor” />
        <transform type=“rotate”>
          <param name=“angle” value=“2*pi/3” />
          <paste object=“rotor” />
        </transform>
        <transform type=“rotate”>
          <param name=“angle” value=“4*pi/3” />
          <paste object=“rotor” />
        </transform>
      </style>
    </transform>
  </define>
  <draw>
    <paste object=“rotors” />
  </draw>
</animation>
```

Fig. 2. An XML metadata of an animated copter scene.

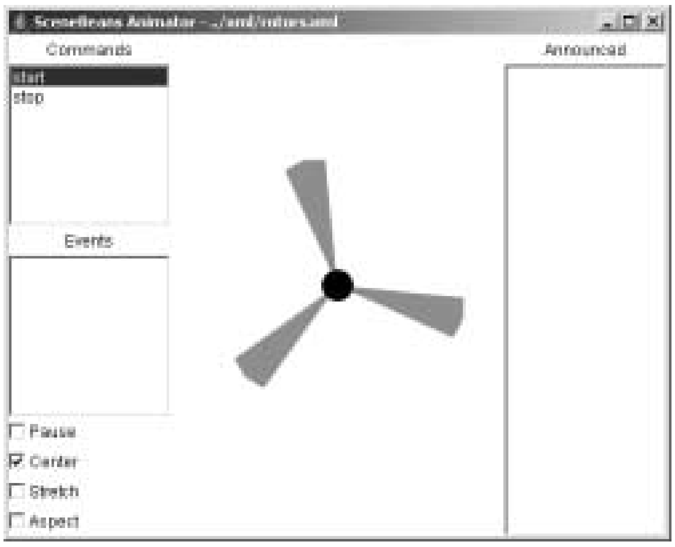


Fig. 3. SceneBeans interface for rotors.xml.

Commands that can be invoked upon an animation are introduced by <command> elements which contain one or more action elements of type <set>, <stop>, <start>, <reset>, <invoke> or <announce>. In Fig. 2, the "start" and "stop" commands on lines 10 and 14 respectively, start and stop the spinning of the rotors. Where they appear in the SceneBeans interface can be seen on the left side of Fig. 3.

In the example in Fig. 3, users can double-click any of the options in the Commands window. The "start" command starts the "rotor-spin" behaviour and makes the rotors spin like they would on an actual helicopter. The image in Fig. 3 is an altered version of the image in Fig. 3. It is enhanced to mimic the animation of the SceneBean. Double-clicking the "stop" command will stop the animation.

The <invoke> tag (not shown in Fig. 2 or 5) invokes another command defined by the animation. The <event> element (line 1 of Fig. 4) defines the action performed by the animation in response to an event fired by one of its constituent beans. Attributes of the <event> tag identify the source and name of the event. The body of the tag defines the actions performed in exactly the same way as the <command> tag.

When the copter-track.xml SceneBean is open, the event "flight.landed" is listed in the Events window (as seen in Fig. 6). When the event occurs, in other words when the "flight_path" behaviour has completed, the <announce> tag (line 4 of Fig. 5) announces the event to the user in the Announced window (also seen in Fig. 7). The announced events in the Announced window, as well as the event list in the Events window, are read-only and do nothing when clicked on. This differs from the Commands window where the commands can be invoked by double-clicking.

The Pause checkbox, located under the Events window (see Fig. 6), can be checked to pause the animation. It can be unchecked to resume the animation. The Center checkbox can be checked to center the origin of the animation to the middle of the window.

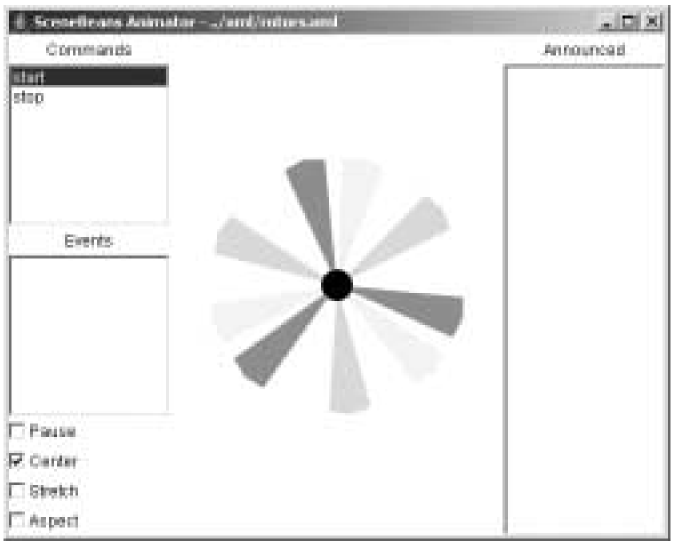


Fig. 4. SceneBeans interface for rotors.xml (image enhanced to mimic animation).

```
<event object="flight_path" event="finished">
  <stop behaviour="flight_path" />
  <set object="copter_angle" param="angle" value="0" />
  <announce event="flight.landed" />
</event>
```

Fig. 5. <event> element in copter-track.xml.

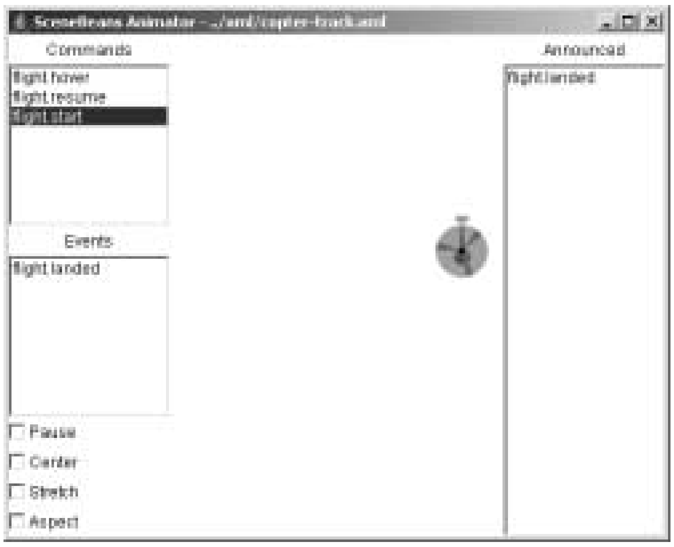


Fig. 6. SceneBeans interface for copter-track.xml.

```
<?xml version="1.0"?>

<animation width="256" height="256">
  <draw>
    <include src="copter.xml">
      <param name="rotor_state" value="started"/>
    </include>
  </draw>
</animation>
```

Fig. 7. A complete yet simple SceneBean document (started-copter.xml).

Unchecking it will position the origin at the top-left corner of the window (the default position). The Stretch checkbox can be checked to stretch or shrink the animation as the window is resized. It can be unchecked to fix the size of the animation as the window is resized. The Aspect checkbox can be checked to maintain the aspect ratio of the animation when it is resized with the window.

The size of the animations can be set by the author of the XML document by including "height" and "width" attributes in the <animation> tag at the root of the document. That is the only control the author and/or user has over the interface (other than the animation itself). The components mentioned earlier: the Commands, Events and Announced windows; and the Pause, Center, Stretch and Aspect checkboxes, are all positioned in the interface by the application and cannot be altered.

A SceneBean document can be as simple as containing just one <animation> element and one <draw> element. In the example in Fig. 6, the SceneBean metadata started-

Table 1

Available SceneBean components

Primitive SceneBeans	Circle	Ellipse
	Polygon	Rectangle
	Sprite	Text
	Null	
Compose SceneBeans	Layered	Difference
	Intersect	Subtract
	Switch	Union
Transform SceneBeans	Rotate	Scale
	Shear	Translate
Style SceneBeans	Font	RGBAColor
Input SceneBeans	MouseClick	MouseMotion
Behaviours	Bounce	ColorFade
	CopyPoint	Loop
	Move	MovePoint
	MultiTrack	RandomTimer
	Track	RelativeMove
	Timer	ToMove
	ConstantSpeed	Move

copter.xml contains just that, and includes another scenebean, copter.xml, into it. The reason for this is to demonstrate how the state of behaviour can be changed on inclusion. The state of the “rotor-spin” behaviour is set to “started” (line 6 of Fig. 7) when the SceneBean interface opens started-copter.xml.

In order to run SceneBeans, users only need a version of Java to be installed on their computer. They need not know how to program Java. Authors of SceneBean documents must know a minimal amount of XML (e.g., the structure of XML documents) and the different components and their properties available for use in SceneBeans. The available SceneBean Components are listed in Table 1. To create a new animation or edit an existing one, only a text-editor is needed to input the XML code. Then all a user has to do is open the XML document in the SceneBeans environment.

5. Extending the 2D SceneBean Model into a 3D Virtual SceneBean

The importance of scene graphs as a powerful tool for modeling any scene including virtual reality or 3D scenes, let most of the notable software venders to adopt it for their graphics APIs (e.g., Java3D, VRML, and OpenGL/Direct3D). Particularly Java3D is getting more popularity as it combines the vast knowledge of the collaborated companies which includes venders Intel, Silicon Graphics, Apple, and Sun. Java3D has been designed to be a platform-independent API concerning the host’s operating system (PC/Solaris/Irix/HPUX/Linux) and graphics (OpenGL/Direct3D) platform, as well as the input and output (display) devices. The implementation of Java3D is built on top of OpenGL, or Direct3D. The high level Java3D API allows rapid application development which is very critical, especially nowadays.

A Java 3D scene graph consists of a collection of Java 3D node objects connected in a tree structure. These node objects reference other scene graph objects called *node component objects*. All scene graph node and component objects are subclasses of a common SceneGraphObject class. The SceneGraphObject class is an abstract class that defines methods that are common among nodes and component objects. Scene graph objects are constructed by creating a new instance of the desired class and are accessed and manipulated using the object’s set and get methods.

Once a scene graph object is created and connected to other scene graph objects to form a subgraph, the entire subgraph can be attached to a virtual universe—via a high-resolution Locale object-making the object *live* Prior to attaching a subgraph to a virtual universe, the entire subgraph can be *compiled* into an optimized, internal format. The Java 3D renderer incorporates all graphics state changes made in a direct path from a scene graph root to a leaf object in the drawing of that leaf object. The View object is the central Java 3D rendering object for coordinating all aspects of viewing. All viewing parameters in Java 3D are either directly contained within the View object or within objects pointed to by a View object. Java 3D supports multiple simultaneously active View objects, each of which can render to one or more canvases (see Fig. 8).

The basic idea of adopting Java3D within flexible component based architecture can be easily done within the framework of Java Beans. The viewing of the 3D scene will

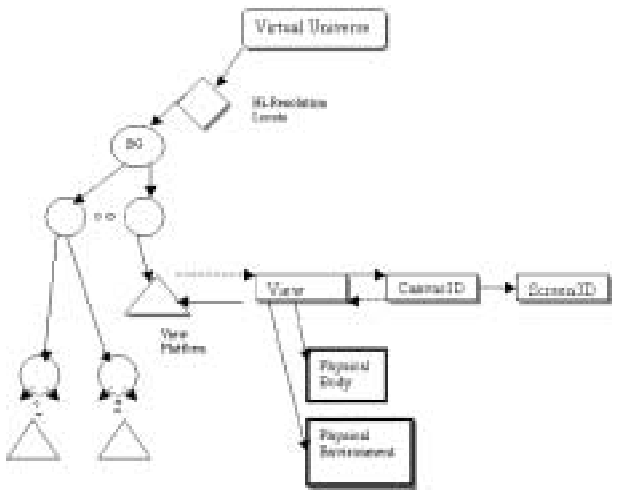


Fig. 8. Viewing a Scene using Java3D.

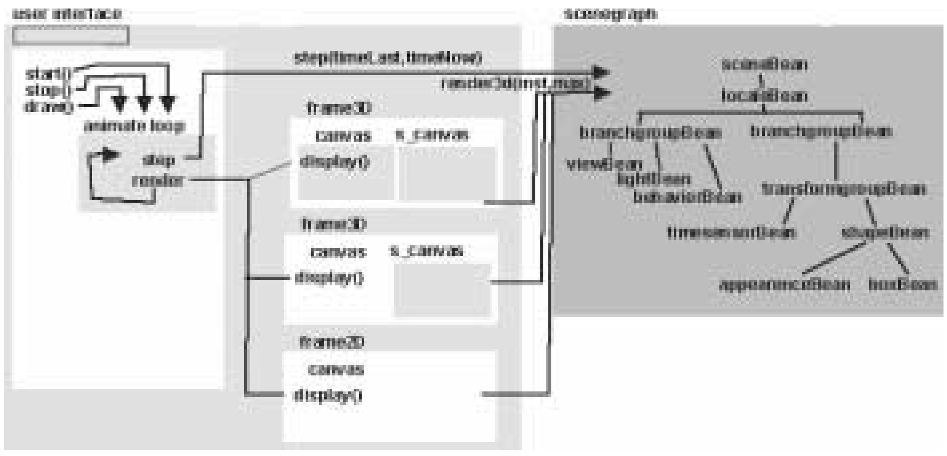


Fig. 9. Rendering a Java3D frame within the framework of Java Beans.

be the responsibility of these beans. A bean interface will then represent an animate loop that runs continuously which alternately calls two other methods (e.g step then render). Step and render are both called on sceneBean, which is the root of the scene graph, and are then called on the subnodes so that they are called on all nodes in the scene graph. The reason that they are separate methods and not all done in the same method is that if there are many views we may have to render the scene many times before stepping on to the next frame. Fig. 9 illustrates the primitive idea of rendering Java3D scene within the framework of Java beans.

Such primitive implementation of the rendering process based on Java beans was introduced recently by Martin Baker at his web page (`www.martinb.com`). However, his rendering APIs supports only VRML type events which is not the standard used by the

WC3 for modeling the description of learning objects, if those scenes required to represent a learning material. The events at his model are defined by a route node which specifies the 'to' and 'from' nodes and properties. When the render loop is started as described above, the scenegraph is called to allow the route nodes to setup the corresponding events. However, one can use the standard WC3 X3D or WJ3D (a 3D version of XML) instead of VRML in describing Java3D scene. Fig. 10 illustrates how a sphere scene is described in X3D, VRML and Java3D. This will require changing the way Martin Baker describe and parse the beans events.

The final step required indeed to transfer the X3D or XJ3D description into a standard visual learning object metadata is to embed the 3D scene description within an acceptable

A simple Sphere with appearance in VRML

```
Group \{
    children [
          Shape {
               geometry Sphere {}
               appearance Appearance {
                    material Material {
                          diffuseColor 1 0 0
                    }
               }
         }
    }
}
```

A simple Sphere with appearance in X3D

```
<Group>
    <Shape>
         <Sphere/>
         <Appearance>
             <Material diffuseColor='1 0 0'/>
         <Appearance>
    <Shape>
<Group>
```

A simple Sphere with appearance in Java3D

```
BranchGroup group = new BranchGroup group();

     Sphere sphere = new Sphere();

     Appearance appear = new Appearance();
          Material material = new Material ();
          material. setDiffuseColor (new Color3f(1.0 f, 0.0 f, 0.0 f));

     sphere.setAppearance(appear);

group.addChild(sphere);
```

Fig. 10. Describing a sphere scene in VRML, X3D, and Java3D.

standard such as the CanCore (widely used by the academia environment in Canada) (see (Richards *et al.*, 2002)). Fig. 11 illustrates how the sphere scene is expressed within the CanCore format.

CanCore Guidelines Version 2.0 was released in late 2003 to reflect the version of LOM that had been approved as a standard by the IEEE. It is also important to note that the IMS specification will soon be revised and made identical to the IEEE standard. The LOM data model defines 76 elements covering a wide variety of characteristics. CanCore reduces the number of elements to 61 where 46 of them are defined as active elements. Active elements are those that can have values assigned by record creators and systems. Those elements that are not active are at a higher level in the hierarchy. The hierarchy begins with nine main categories that contain sub-elements. These main categories (General, LifeCycle, Meta-Metadata, Technical, Educational, Rights, Relation, Annotation and Classification) contain all other elements in the LOM. The place of an element in the hierarchy is indicated by decimals in the element number and name (e.g., 2.3.2: LifeCycle.Contribute.Entity).

Many of the elements can be repeated to accommodate multiple values (e.g., multiple

```
<lom>
  <general>
    <identifier>
      <catalog>CAREO</catalog>
      <entry>632844</entry>
    </identifier>
    <identifier>
      <catalog>URI</catalog>
      <entry>http://www.lakeheadu/∼jfiaidhi/3dsphere.html</entry>
    </identifier>
    <title>
      <string language="en">Describing a 3D Sphere </string>

    </title>
    <language>en</language>
        <description>
      <Shape>
        <Sphere>
             <Appearance>
                  <Material diffuseColor="1 0 0">
             </Appearance>
        </Sphere>
      </Shape>
    </description>
    ...
    <structure>
      <source>LOMv1.0</source>
      <value>Hierarchical</value>
    </structure>
  </general>
  ...
</lom>
```

Fig. 11. Describing a 3D Scene using the CanCore Standard – an example.

authors, versions). Actually, CanCore was developed in Phase I of the POOL project by the collaboration of Canadian researchers searching for a level of sufficient specificity to enable the efficient search of learning objects. CanCore has sufficient flexibility in its protocol that not all fields need be completed, thus developers can ignore many fields that may be inappropriate for their purposes. The POOL protocol expands the JXTA P2P communication protocol by building in more control for distributed searches and provides for flexibility in metadata schemas used for queries and responses (Hatala and Richards, 2002).

6. Essential Steps for Creating Simple Virtual/Java3D SceneBeans

In this section we will introduce the basic strides/steps required for creating simple Virtual/3D SceneBeans in Java programming environment:

Stride 1*: *Download and Install Java3D SDK

The Java3D software development kit (SDK) must be installed on your system before you can create 3D programs in Java. The most recent JDK implementation can be obtained from the Java Developer Connection (JDC) website. The Java3D implementation that we use is built on top of Direct3D.

Stride 2*: *Create the Source .java File

A Java3D program file has the same attributes as any other Java application. No new syntax or file naming scheme must be learned.

Stride 3*: *Import Java and Java3D Classes

This is an essential step in any Java program. Besides a few standard Java classes, a Java3D program must import class files that enable the creation of 3D universes. Table 2 lists and describes the classes required for the Java file.

Stride 4*: *Create the 3D Canvas and Container Layout

Java3D renders virtual scenes on a special component called **javax.media.j3d.Canvas3D**. This component is an extension of the Abstract Window Toolkit (AWT) **java.awt.Canvas** class. While **Canvas3D** is also extensible, it is sufficient for most 3D applications. Only a few lines of code are required to create a **Canvas3D** component and add it to a container. The following code shows how to create a 3D canvas.

```
public void layoutComponents() {
   setLayout(new BorderLayout());
   canvas3D = new Canvas3D(null);
   add("Center", canvas3D);
}
```

The first line in the method simply sets the layout so that components are added to specific regions of the container (e.g., “East”, “West”, or “Center”). The second line creates the **Canvas3D** object. There is only one constructor method in **Canvas3D**, and that method requires a **java.awt.GraphicsConfiguration** object as a parameter.

Table 2

Classes required to be imported into Java Source Program

Class	Package	Description
MainFrame	com.sun.j3d.utils.applet	Convenience class that enables a Java3D applet to be run as an application.
ColorCube	com.sun.j3d.utils.geometry	Convenience class used to create a multi-color cube.
SimpleUniverse	com.sun.j3d.utils.universe	Convenience class used to create a simple Java3D universe that is ready for viewing.
TextureLoader	com.sun.j3d.utils.image	Convenience class used to load textures.
Applet	java.applet	Used to create Java applet programs.
Frame	java.awt	Used to create a framed (windowed) Java application.
BorderLayout	java.awt	Used to layout components in an AWT *Container* (such as a *Frame* or *Applet*).
BranchGroup	javax.media.j3d	The root of a scene graph branch.
Background	javax.media.j3d	The background color or image that fills the window of each new rendering frame.
Canvas3D	javax.media.j3d	An extension of java.awt.Canvas that enables basic 3D rendering capabilities.
TransformGroup	javax.media.j3d	A group node that contains a transform.
Point3d	javax.media.j3d	Double precision floating point x, y, z coordinates.
Transform3D	javax.media.j3d	An abstraction that encapsulates a row-major, 4x4 double precision floating point matrix. Used for rotations, translations, and other transformations.

GraphicsConfiguration is an abstract class that encapsulates information about a particular graphics device. The last line simply adds the 3D canvas to the current container. For further information, complete tutorials on creating Java3D scenes, look at `http://www.acm.org/crossroads/xrds5-3/ovp53.html`.

Stride 5: Create the Scene Graph

Java3D uses a *scene graph* for rendering purposes. To do that we need to perform the followings:

Stride 5-1: Create a Branch Group Object

A **BranchGroup** object represents the root of a particular scene graph. You need to create it in a separate method using

sceneGraph = BranchGroup().

Once the root of a scene graph has been defined, a scene graph can be constructed and readied for rendering.

Stride 5-2: Create the Background

The default background of the 3D scene is null and void (black). However, to make the scene a little more interesting, you can add an image to the background. As an example, the following *createBackground()* method contains the code required to add a background image to the scene.

```
public void createBackground() {
   BoundingSphere boundingSphere =
      new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0);
   TextureLoader backgroundTexture =
      new TextureLoader(backgroundImage, this);
   Background background =
      new Background(backgroundTexture.getImage());
   background.setApplicationBounds(boundingSphere);
   sceneGraph.addChild(background);
}
```

The first object created is the **javax.media.j3d.BoundingSphere**. The **BoundingSphere** object specifies the spherical region in which the background will be active. There are two parameters: The sphere center location and the sphere radius. In this example, the **BoundingSphere** is located at x, y, z coordinates 0.0, 0.0, 0.0, respectively, and has a radius of 100.0 meters. Java3D uses the right hand coordinate system, meaning that the x-axis points toward the right, the y axis points up, and the z-axis points out of the screen. The **com.sun.j3d.utils.image.TextureLoader** object is used to load the background image specified by the user at start-up. The specification for the **TextureLoader** constructor is as follows:

TextureLoader(java.lang.String fname, java.awt.Component observer)
where **fname** is the name of the file that contains the texture and **observer** is the image observer. An image observer receives asynchronous image update notifications during image construction.

The background is constructed with a **javax.media.j3d.Background** object. The **Background** constructor expects a **javax.media.j3d.ImageComponent2D** object. The **ImageComponent2D** class represents a 2D image and can be applied to 3D objects in addition to backgrounds. The **ImageComponent2D** object is obtained by calling the **TextureLoader.getImage()** method. Once the background image has been obtained, the application bounds of the background is set with the **Background.setApplicationBounds()** method. This is necessary to specify the region in which the texture is to be applied. Finally, the background is added to the scene graph with the **BranchGroup.addChild()** method, which is inherited from **javax.media.j3d.Group**.

Stride 5-3: Create a 3D Transformation

A 3D transformation is the movement of a 3D point from one location to another. Rotations, scales, translations, and reflections are all examples of transformations. In Java3D, a 3D transformation is represented by the

javax.media.j3d.Transform3D class. The scene graph group node that contains a **Transform3D** object is an instance of **javax.media.j3d.TransformGroup**.

The transformation performed in the following code segment simply rotates the cube a specified number of degrees about the x axis:

```
public void createTransformation() {
   // Create a 3D transformation that will
   // rotate the cube a specified number of degrees on the x axis
   Transform3D transform3D = new Transform3D();
   transform3D.rotX(xRotationAngle);
   // Create the transform group node and add the transformation
   // to the node. Add the transformation group to the scene graph.
TransformGroup transformGroup=new TransformGroup(transform3D);
   sceneGraph.addChild(transformGroup);
   // Add a colored cube to the transform group node.
   transformGroup.addChild(new ColorCube(cubeScale));
}
```

The first two lines of code create a Java3D transformation that will be used to rotate a 3D object about the x-axis. The method used to perform the rotation, **javax.media.j3d.Transform3D.rotX()**, accepts a single double precision floating point number that specifies the angle in radians. The next two lines create a transform group node with the specified 3D transformation. Once this group node has been properly initialized, it is added to the scene graph.

The last line of code simply adds the colored cube to the transform group node. The transform group node now has two children: the 3D transformation object and a colored cube.

Stride 5-4: Compile the Scene Graph

After the scene graph has been constructed, it can be compiled. This is done by simply calling the **BranchGroup.compile()** method.

Stride 6*: *Create the UniverseRF

After creating the scene graph, you are now ready to create a Java3D universe. In this direction you need to create a method like **createSimpleUniverse()** method to construct the universe and add it to the scene graph.

```
public void createSimpleUniverse() {
   SimpleUniverse simpleUniverse = new SimpleUniverse(canvas3D);
   simpleUniverse.getViewingPlatform().setNominalViewingTransform();
   simpleUniverse.addBranchGraph(sceneGraph);
}
```

This segment of source code readies the scene graph for rendering. First, a **javax.javamedia.j3d.SimpleUniverse** object is created. This constructor method accepts a **Canvas3D** object that will be used for rendering. Next, the viewing distance is set by calling the **com.sun.j3d.utils.universe.ViewingPlatform.setNominalViewingTransform()** method. Finally, the scene graph is added to the universe by calling the method.

Stride 7*: *Compile and Run the Java Program.

7. Developing a Collaborative Learning Environment for 3D Virtual SceneBean

The two previous sections describe how a virtual scene can be described in Java3D/X3D and the steps required to program the virtual scene within the J2SE Java environment. However, exchanging such virtual scenes within collaborative and distributed environment requires yet another enabling environment. Such situation is served best by using a peer to-peer computing environment. Each peer is *acquainted* with a small number of other peers with whom it can exchange information and services. Because of the ubiquitous nature of such collaborative architecture, student/instructor communication requires two different channels for exchanging information on 3D Scene (or any multimedia): the XML like channel (e.g., X3D or XJ3D) that can be used to display the definition of the multimedia objects as well as to convey the instant messages between the student and the instructor, and the 3D SceneBean channel that transfers the Java Bean files via the TCP/IP Channel which is used by most of the ubiquitous devices connected on the internet (see Fig. 12).

However, in our implementation the XML like channel is based on a simple Client-Server messenger mode as developed by Deitel *et al.* (2001). Clients, or users of the system, are people who need to communicate with other users in the system. Consider a case where user 1 needs to communicate with user 2. For communication to be established, both users need to be logged on to the server. The messenger displays all the users who are currently log on to the server. User 1 can then choose to send message to user 2. When user 1 types in a message for user 2, the message is tagged with XML like representation and sent to the server. The XML like message also contains the destination (user 2) of the message and its source (user 1). The server then reroutes the message to the respective user based on the destination information provided in the message. The XML like server (MessengerServer) uses a ServerSocket, object to wait for clients to connect. When a client connects, the server creates a new UserThread object to manage the client's

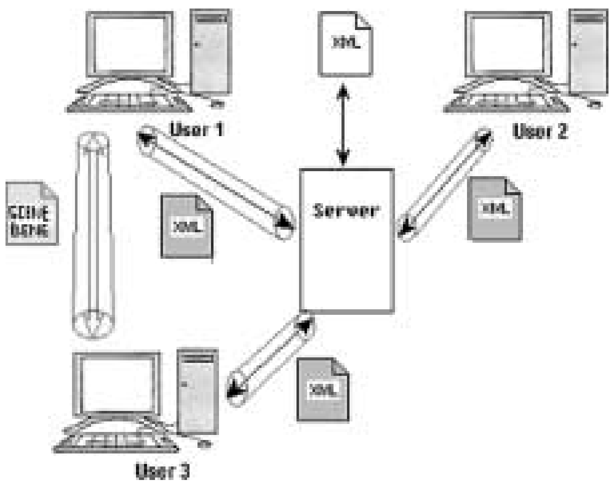


Fig. 12. The general architecture of the 3D SceneBean collaborative environment.

socket and streams. The MessengerServer object uses a vector to store all UserThread instances. The MessengerServer also uses a Document object users, consisting of the names of on-line users stored in individual user elements.

The XML like client has two main functions. First, it registers the user with the server by sending an XML like document that contains the user's name and ID. It then updates its current list of logged-on users with the new information it receives from the server. During the session, that is, the period during which the user is logged on, it has to update this list whenever a new user logs in. All such information is exchanged in the form of XML like (e.g., X3D). The second function of the client is to convert the text typed in by the user into XML-like messages, tagging them appropriately to identify the source and destination of each message, and to send them to the server. The client also has to parse the XML like messages received from the server and display them to the user. Fig. 13 illustrates the UML Diagrams for the XML like communication channel.

On the other hand, the 3D SceneBean channel is based on Java Socket programming. However, there are two Java communication protocols that utilize socket programming: datagram communication and stream communication. Our Image transfer channel is built upon a stream socket programming. The stream socket is protocol that is based on TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions. Creating a socket like Socket MyClient can be done simply using

MyClient = new Socket (“Machine name”, PortNumber);

where Machine name is the machine you are trying to open a connection to, and PortNumber is the port (a number) on which the server you are trying to connect to, is running. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users (that is, super user or root). These port numbers are reserved for standard services, such as email, FTP, and HTTP. When selecting a port number for your server, select one that is greater than 1,023! The programming techniques for this type of messenger are well-known and we refer the reader to (Mahmoud, 1996). In case of user1 wants to send an Image File to user2, user1 must use the collaborative infrastructure GUI and press “Send SceneBean” to the user name of User2. Before sending the actual image file, the user needs to use the XML messenger to notify the other user.

<TransferRequir to = “receiver” from = “sender”>
Waiting for file transfer: (filename) </TransferRequir>

If user2 accept the file transfer, an accept message will be created and sent back to user1:

<Accept to = “receiver” from = “sender”> (IP address of user2) </Accept>

At the same time user2 create a SeverSocket and waiting for incoming connection. User1 can get the IP address of user2 from the above message. Then user1 creates a Socket and connect to user2 by the IP got from message. After all above success, file

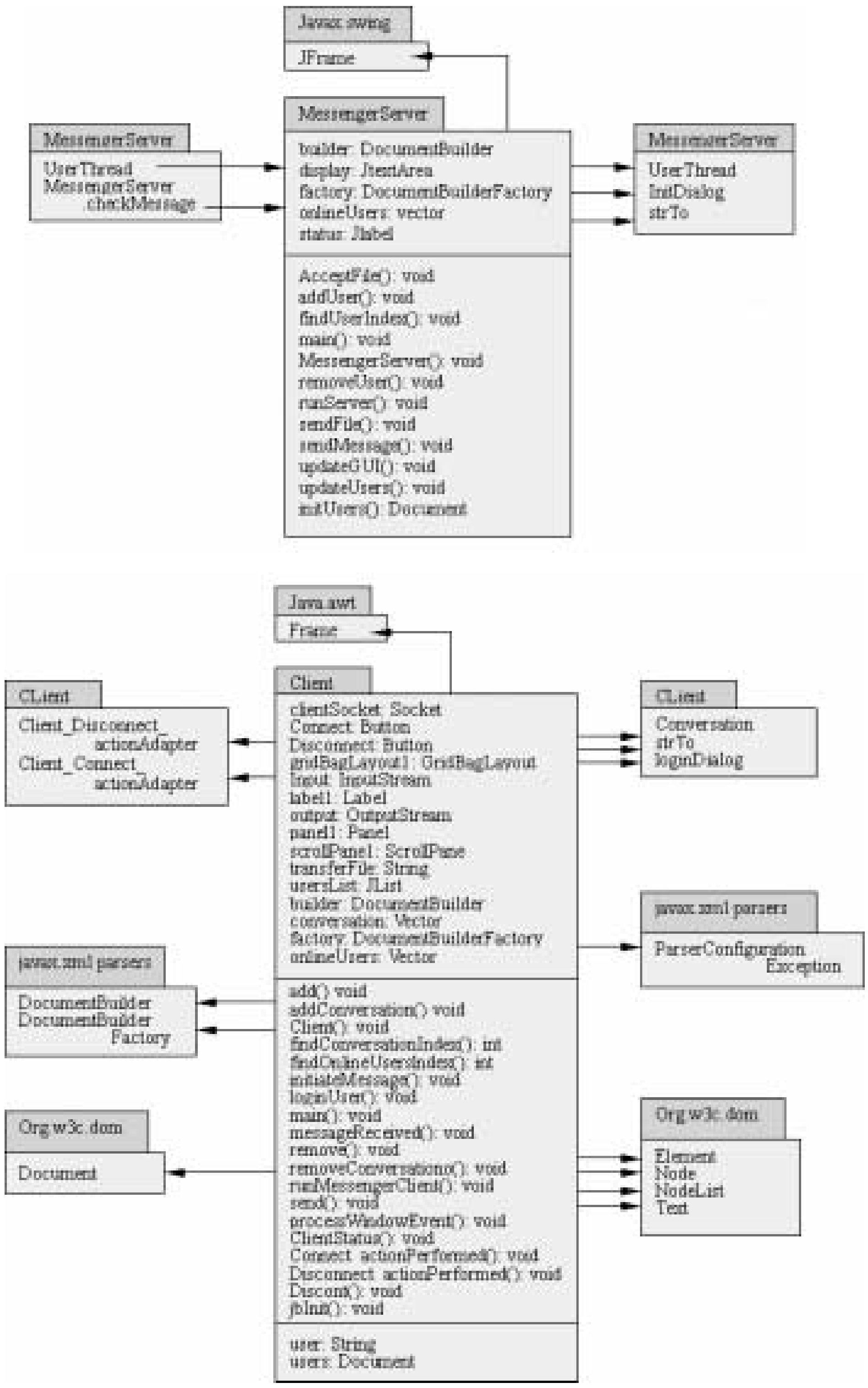
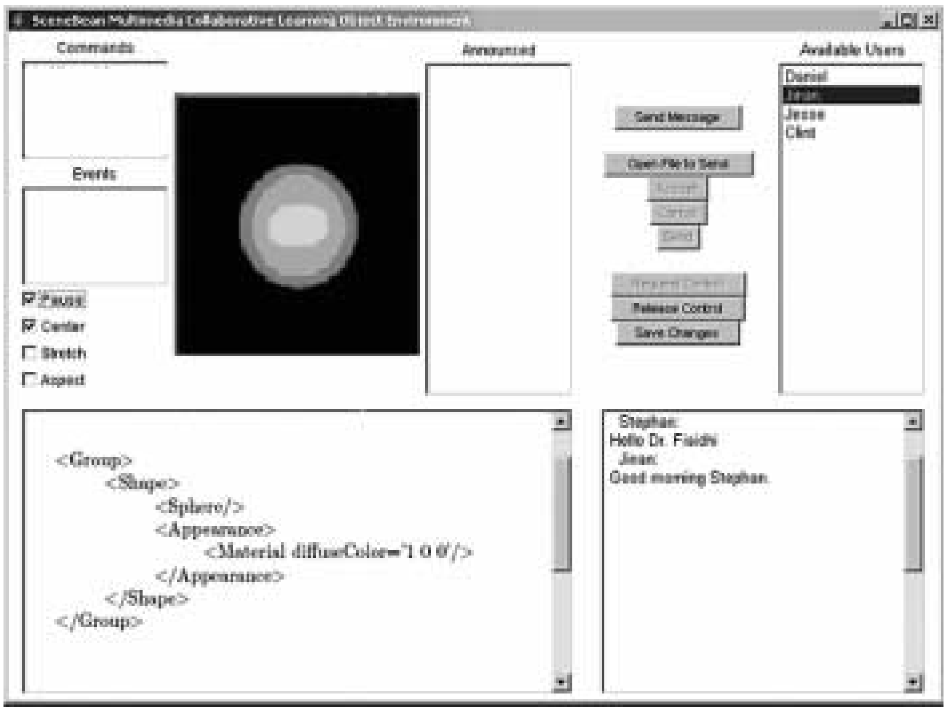


Fig. 13. The UML diagrams of the XML like Messenger.



```
<Group>
    <Shape>
        <Sphere/>
        <Appearance>
            <Material diffuseColor='1 0 0'/>
        </Appearance>
    </Shape>
</Group>
```

```
 Stephan:
Hello Dr. Fraidhi
 Jinan:
Good morning Stephan.
```

Fig. 14. The main screen of our 3D SceneBean Collaborative Learning Environment.

transfer will start. Each side will be acknowledged when file transfer finished. When server redirects it to user2, the button "receive file" is enabled. User2 can choose to receive file or ignore it. Fig. 14 shows our 3D SceneBeans collaborative environment main screen.

Using this environments, users can actively interact with the 3D SceneBean through choosing the right SceneBean attributes and even to reprogram the SceneBean by changing the behavioural events as described by the accompanying XML like description. Moreover, the environment provides also a mean for exchanging instant messages among users, which will aid the process of a full comprehension of the 3D scene. However, our developed collaborative environment does not support pure P2P collaboration and has no protocol for accessing major other learning objects repositories.

In this direction we are currently introducing such P2P collaboration environment via incorporating the Sun Microsystems JXTA APIs. also to link the developed repository with the POOL of learning objects as used within the academia paradigm in Canada by utilizing the Sun Microsystems JXTA API primitives. Basically JXTA protocol will enable peers to communicate directly and exchange/interact-with Virtual SceneBeans via pipes as well as having the following capabilities:

1. Discovery of
    - Pipes. An application is able to search for a named pipes created by other Peers.

   - Groups. An application is able to discover a JXTA group and join it.
   - Contents. Applications are able to discover application specific contents.
2. Create
   - Pipes. An application is able to create pipes – both point-to-point and propagate pipes.
   - Groups. An application is able to create peer groups to limit the scope of discovery.
   - Contents. Application specific contents.
3. Join Groups. An application is able to join a given group as per JXTA spec.

Other main advantage of using JXTA is that it will enable us to connect directly to the major pool of learning object repositories such as the POOL (Hatala and Richards, 2002) which is also based on JXTA. Fig. 15 illustrates our vision of such protocol extension based on JXTA.

8. Conclusions and Future Research

In this article we extended our research work on ScenBeans (Fiaidhi *et al.*, 2004) to develop a collaborative tool to exchange virtual/3D learning objects. The 3D SceneBean metadata is based on X3D and comply with the Canadian Core Learning Resource Metadata Protocol (CanCore).

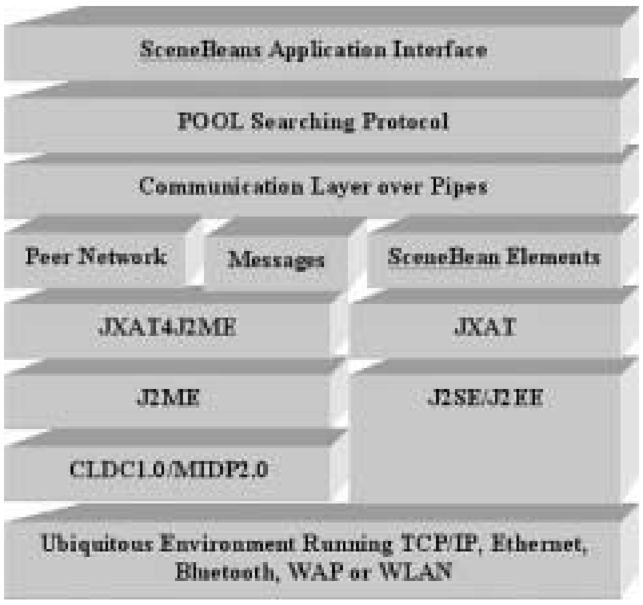

SceneBeans Application Interface
POOL Searching Protocol
Communication Layer over Pipes
Peer Network
Messages
SceneBean Elements
JXAT4J2ME
JXAT
J2ME
J2SE/J2EE
CLDC1.0/MIDP2.0
Ubiquitous Environment Running TCP/IP, Ethernet, Bluetooth, WAP or WLAN

Fig. 15. Extending the SceneBean Messenger to include JXTA APIs.

The actual scenes are described using Java3D APIs and encapsulated as Java beans. A special interface is used for viewing and exchanging the 3D scenes which forms the core of our collaborative learning environment.

The communication protocol developed for the purpose of exchanging 3D Scene Beans as well as the instant messages between users is a simple Client-Server XML like messenger which work on top of the TCP/IP protocol.

Our goal is develop variety of 3D learning objects that can be used for training our NOMS medical school students on endoscope training operations as well as to form a special 3D medical learning objects repository.

The Northern Ontario Medical School utilizes an infrastructure that is shared by two campuses (Lakehead and Laurentian). This infrastructure platform utilizes the SGI Onyx 350 that supports advanced visualization, high-performance computing, and storage. Actually, SGI Onyx 350 provides a supercomputer architecture with truly scalable CPUs, huge shared memory, and scalable I/O bandwidth to handle a NOMS's big data sets. All of the data can be visualized, enabling the creation of designs and analysis of problems that would be impossible by solely using the desktop UNIX®or Windows®systems. Fig. 16 illustrates the available NOMS infrastructure.

The SGI Onyx 350 four-way composite Infinite Performance graphics channel allows interactive visualization of enormous data sets, delivering up to 141 million triangles per second and 3.8 billion pixels per second to a single display. Indeed as the main advantage of Java technology for the development of collaborative/distributed environments is been platform independence. This feature indeed will enable us to transfer our initial work on virtual SceneBeans and its collaborative environment from the MS Windows into the SGI Onyx platform. With Onyx visualization performance we expect the performance of the Virtual SceneBean model and its collaborative environment to exceeds the

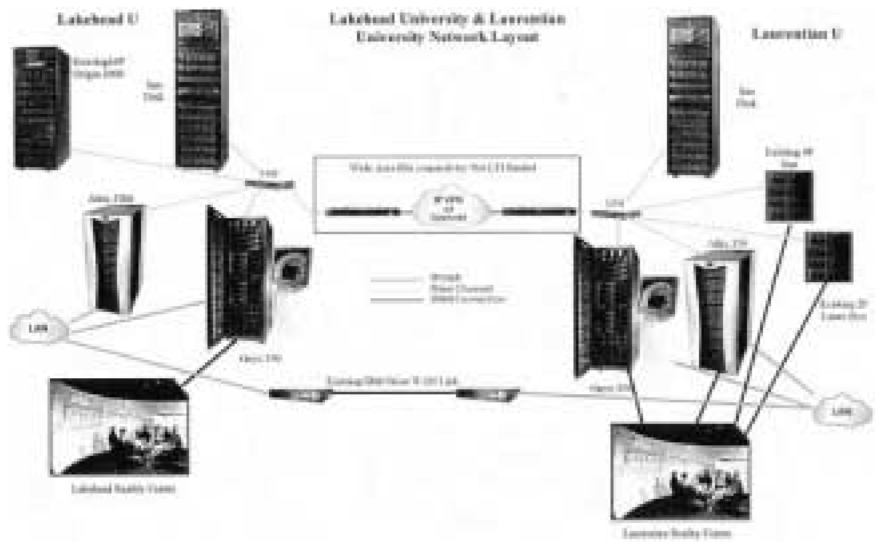

Lakehead U
Lakehead University & Laurentian
University Network Layout
Laurentian U
LAN
LAN
Lakehead Reality Centre
Laurentian Reality Centre

Fig. 16. The available NOMS virtual visualization infrastructure.

performance of a collection of notable java tools used for multimedia collaboration, e.g., jSTREAMING
(`http://jStreaming.com/~jauvane/H263Decoder/JDK1.1`),
JETS (Shirmohammadi *et al.*, 1997),
JMF (`http://java.sun.com/products/java-media/jmf/index.jsp`) and
JASMINE (Shirmohammadi *et al.*, 2003).

The Virtual SceneBean and its collaboration applications presented in this article show the positive potential of the combination of Java and the internet for collaborative work. The active multimedia representation in X3D, its packaging in a Java Bean, the Virtual SceneBean interface, and the SceneBean messenger represent land marks for a simple and useful technology for multimedia telecollaboration. However, it is worth mentioning that there is a giant investment for developing a similar technology in Canada as expressed by the LORNET project (`www.lornet.org`). We are intending to compare our approach to the LORNET approach once both projects reached mature stages.

Acknowledgment

This research is part of the author CFI LUCARTT04 grant on establishing LU Virtual Reality Learning and Training Centre**.** The author would like to thank Mr. S. Sisko and Mr. Dan Gaudette, graduate students of the Department of Computer Science, for developing variety of learning objects using SceneBeans.

References

Brogan, D.C., R.A. Metoyer and J.K. Hodgins (1998). Dynamically simulated characters in virtual environments. *IEEE Computer Graphics and Applications*, **15**(5), 58–69.

Benford, S., J. Bowers, L.E. Fahlen, C. Greenhalgh and D. Snowdon (1995a). User embodiment in collaborative virtual environments. In *CHI'95 Proceedings*. ACM Press, pp. 242–249.

Benford, S., C. Greenhalgh and S. Massive (1995b). Acollaborative virtual environment for teleconferencing. *ACM Transactions on Computer-Human Interaction*, **2**(3), 239–261.

Capin, T.K., I.S. Pandzic, D. Thalmann and N.M. Thalmann (1998). Realistic avatars and autonomous virtual humans in VLNET networked virtual environments. In J. Vince and R. Earnshaw (Eds.), *Virtual Worlds on the Internet*. IEEE Computer Society, Los Alamitos, pp. 157–173.

Deitel, H.M., P.J. Deitel, T.R. Nieto, T.M. Lin and P. Sadhu (2001). *XML How to Program*. Deitel & Associats Inc.

El Saddik, A., *et al.* (2000). A component-based construction kit for algorithmic visualization. In *Proceedings IDPT*. Springer-Verlag, N.Y.

Era, T., K. Kauppinen, A. Kivimäki and M. Robinson (1998). Producing identity in collaborative virtual environments. In *Proceeding of the ACM Symposium on Virtual Reality Software and Technology (VRST'98)*. Taipei, Taiwan, pp. 35–42.

Fiaidhi, J., S. Mohammed and S. Sisko (2004). SceneBeans: a tool for constructing collaborative multimedia learning objects. In *The 9th Western Canadian Conference on Computing Education WCCCE*. BC, Canada.

Fiaidhi, J., and J. Mohammed (2004). Design issues involved in using learning objects for teaching a programming language within a collaborative eLearning environment. *International Journal of Instructional Technology & Distance Learning*, **1**(3), 39–53.

Fiaidhi, J., K. Passi and S. Mohammed (2004). Developing a framework for learning objects search engine. In *4th International Conference on Internet Computing (IC04)*. Las Vegas, Nivada, USA.

Fiaidhi, J., S. Mohammed, J. Jaam and A. Hasnah (2003a). Standard framework for search hosting via ontology based query expansion. In *The 7th World Multiconference on Systemics, Cybernatics, and Informatics*. Orlando, Florida, USA.

Fiaidhi, J., S. Mohammed and K. Faisal (2003b). Developing standards for collaborative eLearning systems. *International Journal of Applied Science and Computations*, **10**(1), 1–10.

Hatala, M., and G. Richards (2002). POOL, POND and SPLASH: a Canadian infrastructure for learning object repositories. In *5th IASTED Int. Conference on Computers and Advanced Technology in Education* (*CATE 2002*). Cancun, Mexico.

Jianhua, Z., L. Kedong and K. Akahori (2001). Modeling and system design for web-based collaborative learning. In *Proceedings of the 2nd International Conference on Information Technology based Higher Education and Training*. Kumamoto, Japan.

Magee, J., J. Kramer, B. Nuseibeh, D. Bush and J. Sonander (2000). Hybrid model visualization in requirements and design: a preliminary investigation. In *Proceedings of 10th International Workshop on Software Specification and Design* (*IWSSD-10*). San Diego, USA.

Mahmoud, Q.H. (1996). Sockets programming in Java: a tutorial. *JavaWorld Online Journal*, December. `http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets_p.html`.

Oliveira, J.C., M. Hosseini, S. Shirmohammadi, A. El Saddik, F. Malric, S. Nourian, N.D. Georganas (2003). Java Multimedia Telecollaboration. *IEEE Multimedia Magazine*, **10**(3), 18–29.

Pryce, N., and J. Magee (2001). *SceneBeans*: *A Component-Based Animation Framework for Java*. Technical Report, 2001, Department of Computing, Imperial College.

Qu, T.E., and C. Meinel (2000). Implementation of a WebDAV-based collaborative distance learning environment. In *ACM SIGUCCS 2000 Proceedings*. Richmond, Virginia.

Richards, G., R. McGreal R. and N. Friesen (2002). Learning objects repositories for teleLearning: the evolution of POOL and CanCore. In *IS2002 Proceedings of the Information Science & IT Education Conference*. Cork, Ireland.

Singhal, S., and M. Zyda (1999). *Networked Virtual Environments*: *Design and Implementation*. Addison Wesley, New York.

Snowdon, D., and J. Tromp (1997). Virtual body language: providing appropriate user interfaces in collaborative virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (*VRST'97*), pp. 37–44.

Shirmohammadi, S., A. El Saddik, N.D. Georganas and R. Steinmetz (2003). JASMINE: A Java tool for multimedia collaboration on the internet. *Journal of Multimedia Tools and Applications*, **19**(1), 5–28.

Shirmohammadi, S., and N.D. Georganas and R. Steinmetz (1997). JETS: Java enabled teleCollaboration system. In *Proceedings IEEE Multimedia Systems'97*. Ottawa.

J.A.W. Fiaidhi is a professor of computer science at Lakehead University. She received her graduate degrees in computer science from Essex University, UK (1983), and PhD from Brunel University, UK (1986). She served also as faculty member at University of Technology, Philadelphia University, Applied Science University and Sultan Qaboos University. Dr. Fiaidhi's research interests include learning objects, XML search engine, multimedia learning objects, recommender systems, software forensics, Java watermarking, and collaborative eLearning systems, software complexity. Dr. Fiaidhi is one of Canada Information Systems Professional (I.S.P.), member of the British Computer Society (MBCS), member of the ACM SIG Computer Science Education, and member of the International Forum of Educational Technology.

Virtualioji SceneBean: mokomųjų objektų modelis, skirtas virtualiai mokytis bendradarbiaujant

Jinan FIAIDHI

Sutariama, jog tinkamai suderinus bendradarbiavimą, mokymą bei tam tikrų sąlygų imitavimą besimokančiojo imlumas smarkiai padidėja. Šiuolaikinė bendradarbiavimu pagrįsta veikla ir mokymosi aplinkos apima ne tik daugelį interaktyvių objektų, bet ir nemažai technologinių aspektų, kurie leidžia pasiekti reikiamą suderinamumą. Deja, reali mokomųjų sistemų teikiama nauda neretai ribojama nepakankamos interaktyvių objektų ir jų tarpusavio sąsajų reprezentacijos. Dažnai tokie objektai pateikiami izoliuotai: viena vertus, jų negalima deramai modifikuoti (keisti jų parametrų ar didinti funkcionalumo), kita vertus, ir jų susietumas su kontekstu nėra deramas (pavyzdžiui, objektas ir parodomasis demonstravimas ar metaduomenys yra nepakankamai sinchronizuoti). Šiame straipsnyje aptariama, kaip virtualų ar trimatį vaizdą racionaliau panaudoti mokymui. Čia pristatomas modelis remiasi tokiomis priemonėmis kaip Scene Graphs, X3D, Java3D bei SceneBeans. Šiam prototipui reikalingas paprasčiausias vartotojo stoties protokolas, leidžiantis atsisiųsti ir naudoti 3D SceneBeans. Tyrimu siekiama išplėsti protokolo galimybes panaudojant Sun JXTA priemones susieti POOL ir kitas mokymosi objektų saugyklas.