HEAD:

Understanding Scene Graphs

DEK

Using graph-based data structures to organize and manage scene contents

Bio

Aaron is chairman of Mantis Development, and teaches computer graphics and Internet/Web application development at Boston College. He can be contacted at aaron@mantiscorp.com.

NOTE TO AUTHORS:
PLEASE CHECK ALL URLs AND SPELLINGS OF NAMES CAREFULLY.
THANK YOU.

Byline

Aaron E. Walsh

Pullquotes

Before scene-graph programming models, we usually represented scene data and behavior procedurally

Scene-graph nodes represent objects in a scene

Scene graphs are data structures used to organize and manage the contents of hierarchically oriented scene data. Traditionally considered a high-level data management facility for 3D content, scene graphs are becoming popular as general-purpose mechanisms for managing a variety of media types. MPEG-4, for instance, uses the Virtual Reality Modeling Language (VRML) scene-graph programming model for multimedia scene composition, regardless of whether 3D data is part of such content. In this article, I'll examine what scene graphs are, what problems they address, and scene-graph programming models supported by VRML, Extensible 3D (X3D), MPEG-4, and Java 3D.

Scene Composition and Management

Scene graphs address problems that generally arise in scene composition and management. Popularized by SGI Open Inventor (the basis for VRML), scene-graph programming shields you from the gory details of rendering, letting you focus on what to render rather than how to render it.

As Figure 1 illustrates, scene graphs offer a high-level alternative to low-level graphics rendering APIs such as OpenGL and Direct3D. In turn, they provide an abstraction layer to graphics subsystems responsible for processing, eventually presenting scene data to users via monitors, stereoscopic goggles/glasses, projectors, and the like.

Before scene-graph programming models, we usually represented scene data and behavior procedurally. Consequently, code that defined the scene was often interspersed with code that defined the procedures that operated on it. The result was complex and inflexible code that was difficult to create, modify, and maintain—problems that scene graphs help resolve.

By separating the scene from the operations performed on it, the scene-graph programming model establishes a clean boundary between scene representation and rendering. Thus, scenes can be composed and maintained independent of routines that operate on them. In addition to making things easier, this lets you create sophisticated content using visual authoring tools without regard for how work is processed.

Listing One is VRML code for a scene consisting of a sphere that, when touched, appears yellow. As you can see, the objects and their behavior are represented at a high level. You don't know (or care) how the sphere is rendered—just that it is. Nor do you know or care about how the input device is handled by the underlying run-time system to support the "touch" behavior. Ditto for the light.

At the scene level, you concern yourself with what's in the scene and any associated behavior or interaction among objects therein. Underlying implementation and rendering details are abstracted out of the scene-graph programming model. In this case, you can assume that your VRML browser plug-in handles low-level concerns.

Nodes and Arcs

As Figure 2 depicts, scene graphs consist of nodes (that represent objects in a scene) connected by arcs (edges that define relationships between nodes). Together, nodes and arcs produce a graph structure that organizes a collection of objects hierarchically, according to their spatial position in a scene.

With the exception of the topmost root node (which defines the entry point into the scene graph), every node in a scene has a parent. Nodes containing other nodes are parent nodes, while the nodes they contain are the child nodes (children) of their parent. Nodes that can contain children are grouping nodes; those that cannot are leaf nodes. Subgraph structures in Figure 2 let a specific grouping of nodes exist as a discrete and independently addressed unit of data within the main scene-graph structure. Operations on the scene can be performed on all nodes in the graph, or they may be restricted to a particular subgraph (scenes can therefore be composed of individual nodes as well as entire subgraphs that may be attached or detached as needed).

Scene graphs in Figure 2 resemble tree data structures when depicted visually. Not surprisingly, trees are often used for scene-graph programming. The directed acyclic graph (DAG) data structure (also known as an "oriented acyclic graph") is commonly used because it supports node sharing at a high level in the graph (nodes in a DAG can have more than one parent) although typically at the expense of additional code complexity and memory consumption. In a DAG, all nodes in the graph have a directed parent-child relationship in which no cycles are allowed—nodes cannot be their own parent.

Graph Traversal

Scene-graph nodes represent objects in a scene. Scene graphs used for 3D content, for instance, usually support nodes that represent 3D geometric primitives (predefined boxes, cones, spheres, and so forth), arbitrarily complex polygonal shapes, lights, materials, audio, and more. On the other hand, scene-graph programming models for other forms of media might support nodes for audio/video content, timing and synchronization, layers, media control, special effects, and other functionality for composing multimedia.

Scene-graph programming models support a variety of operations through traversals of the graph data structure that typically begin with the root node (root nodes are usually the entry point for scene rendering traversals). Graph traversals are required for a number of operations, including rendering activities related to transformations, clipping and culling (preventing objects that fall outside of the user's view from being rendered), lighting, and interaction operations such as collision detection and picking.

Nodes affected by a given operation are visited during a corresponding traversal. Upon visitation, a node's internal state may be set or altered (if supported) so that it reflects the state of the operation at that point in time. Rendering traversals occur almost constantly with interactive and animated graphics because the state of affairs change as often as the user's viewpoint, necessitating continual scene-graph queries and updates in response to an ever-changing perspective. To increase performance, effect caching can be used so that commonly applied operations use cached results when possible.

Virtual Reality Modeling Language (VRML)

VRML is an international Standard for 3D computer graphics developed by the Web3D Consortium (formerly the VRML Consortium) and standardized by ISO/IEC. The complete specification for ISO/IEC 14772–1:1997 (VRML97) is available at http://web3d.org/.

An Internet and web-enabled outgrowth of Open Inventor technology developed by SGI (http://www.sgi.com/), VRML standardizes a DAG-based scene-graph programming model for describing interactive 3D objects and 3D worlds. Also intended to be a universal interchange format for integrated 3D graphics and multimedia, the VRML Standard defines nodes that can generally be categorized as:

- Geometry nodes that define the shape or form of an object.
- Geometric property nodes used to define certain aspects of geometry nodes.
- Appearance nodes.
- Grouping nodes that define a coordinate space for children nodes they may contain.
- Light-source nodes that illuminate objects in the scene.
- Sensor nodes that react to environmental or user activity.
- Interpolator nodes that define a piecewise-linear function for animation purposes.
- Time-dependent nodes that activate and deactivate themselves at specified times.
- Bindable children nodes that are unique because only one of each type can be bound, or affect the user's experience, at any instant in time.

Every VRML node has an associated type name that defines the formal name for the node—*Box, Fog, Shape,* and so forth. Each node may contain zero or more fields that define how nodes differ from other nodes of the same type (field values are stored in the VRML file along with the nodes and encode the state of the virtual world) in addition to a set of events, if any, that the node can send or receive. When a node receives an event, it reacts accordingly by changing its state, which might trigger additional events. Nodes can change the state of objects in the scene by sending events. A node's implementation defines how it reacts to events, when it may generate and send events, and any visual or auditory appearance it might have in the scene.

VRML supports a *Script* node that facilitates dynamic behaviors written in programming languages such as ECMAScript, JavaScript, and Java. *Script* nodes are typically used to signify a change in the scene or some form of user action, receive events from other nodes, encapsulate program modules that perform computations, or effect change elsewhere in the scene by sending events. External programmatic control over the VRML scene graph is possible via the External Authoring Interface (EAI). Currently awaiting final ISO standardization as Part 2 of the VRML97 Standard, EAI is a model and binding for the interface between VRML worlds and external environments.

All in all, the VRML Standard defines semantics for 54 built-in nodes that implementers, such as VRML browser plug-ins, are obligated to provide. In addition, VRML's PROTO and EXTERNPROTO statements (short for "prototype" and "external prototype") offer extension mechanisms for creating custom nodes and behaviors beyond those defined by the Standard.

VRML is a text-based language for which a variety of authoring and viewer applications and freely available browser plug-ins exist, making it popular for exploring scene-graph programming fundamentals. The file human.wrl (available electronically; see "Resource Center," page 5), for instance, defines the 3D humanoid in Figure 3, which is composed of primitive sphere and cylinder shapes. To view and examine this scene, open human.wrl file in your web browser (after installing a VRML plug-in such as Contact, http://blaxxun.com/, or Cortona, http://www.parallelgraphics.com/).

The scene graph in human.wrl relies heavily on *Transform*, a grouping node that contains one or more children. Each *Transform* node has its own coordinate system to position the children it contains relative to the node's parent coordinate system (*Transform* children are typically *Shape* nodes, *Group* nodes, and other *Transform* nodes). The *Transform* node supports transformation operations related to position, scale, and size that are applied to each of the node's children. To help identify the children of each *Transform* used in human.wrl, I've placed alphabetical comments (#a, #b, #c, and so on) at the beginning/ending braces of each *children* field.

As with Listing One, the nodes that compose human.wrl are named using VRML's DEF mechanism. After a node name has been defined with DEF (short for "define"), it can then be referenced elsewhere in the scene. Listing One shows how USE is combined with ROUTE to facilitate event routing; human.wrl illustrates how specific node instances can be reused via the USE statement. With Figure 3, the arm segments defined for the left side of the body are reused on the right. Likewise, the skin appearance DEF'ed for the body is USE'd for the skull.

In addition to enabling node sharing and reuse within the scene, DEF is handy for sharing VRML models with other programming environments. Human.wrl takes care to DEF a number of nodes based on the naming conventions established by the Web3D Consortium's Humanoid Animation Working Group (H-Anim; http://hanim.org/). As a result, the *Human_body, Human_r_shoulder, Human_r_elbow,* and *Human_skullbase* nodes are accessible to applications that support H-Anim semantics for these and other human-like structures. VRML Viewer (VView), http://coreweb3d.com/, does this.

Nodes are discrete building blocks used to assemble arbitrarily complex scenes. If you need lower-level application and plug-in plumbing, check OpenVRML (http://openvrml.org/) and FreeWRL (http://www.crc.ca/FreeWRL/). Both are open-source implementations that add VRML support to projects.

OpenVRML and FreeWRL are open-source VRML implementations hosted by SourceForge (http://sourceforge.net/). X3D is the official successor to VRML that promises to significantly reduce development requirements while advancing state-of-the-art for 3D on and off the Web.

Extensible 3D (X3D)

Extensible 3D (X3D; http://web3d.org/x3d/) enables interactive web- and broadcast-based 3D content to be integrated with multimedia while specifically addressing limitations and issues with the now-obsolete VRML Standard. X3D adds features and capabilities beyond VRML including advanced APIs, additional data-encoding formats, stricter conformance, and a componentized architecture that enables a modular approach to supporting the Standard (as opposed to VRML's monolithic approach).

X3D is intended for use on a variety of devices and application areas — engineering and scientific visualization, multimedia presentations, entertainment and education, web-page enhancement, and shared multiuser environments. As with VRML, X3D is designed as a universal interchange format for integrated 3D graphics and multimedia. But because X3D supports multiple encodings — including XML encoding — it should surpass VRML as a 3D interchange format.

X3D was designed as a content development and deployment solution for a variety of systems — number-crunching scientific workstations, desktop/laptop computers, set-top boxes, PDAs, tablets, web-enabled cell phones, and devices that don't have the processing power required by VRML. X3D also enables the integration of high-performance 3D facilities into broadcast and embedded devices, and is the cornerstone of MPEG-4's baseline 3D capabilities.

X3D's componentized architecture enables lightweight client players and plug-ins that support add-on components. X3D eliminates VRML's all-or-nothing complexity by breaking functionality into discrete components loaded at run time. An X3D component is a set of related functions consisting of various objects and services, and is typically a collection of nodes, although a component may also include encodings, API services, or other X3D features.

The X3D Standard specifies a number of components including a Core component that defines the base functionality for the X3D run-time system, abstract base-node type, field types, event model, and routing. The Core component provides the minimum functionality required by all X3D-compliant implementations, and may be supported at a variety of levels for implementations conformant to the X3D architecture, object model, and event model.

The X3D Standard defines components such as Time (nodes that provide the time-based functionality), Aggregation and Transformation (organizing and grouping nodes that support hierarchy in the scene graph), Geometry (visible geometry nodes), Geometric Properties (nodes that specify the basic properties of geometry nodes), Appearance (nodes that describe the appearance properties of geometry and the scene environment), Lighting (nodes that illuminate objects in the scene), and many other feature suites including Navigation, Interpolation, Text, Sound, Pointing Device Sensor, Environmental Sensor, Texturing, Prototyping, and Scripting components.

A number of proposed components are under consideration including nodes needed for geometry using Non-Uniform B-Splines (NURBS), for applying multiple textures to geometry using multipass or multistage rendering, to relate X3D worlds to real world locations, humanoid animation nodes (H-Anim), Distributed Interactive Simulation (DIS) IEEE 1284 communications nodes, and more. Since it is extensible, you can create your own components when X3D's predefined components aren't sufficient.

X3D also specifies a suite of implementation profiles for a range of applications, including an Interchange Profile for content exchange between authoring and publishing systems; an Interactive Profile that supports delivery of lightweight interactive animations; an Extensibility Profile that enables add-on components; and a VRML97 Profile that ensures interoperability between X3D and VRML97 legacy content.

By letting scenes be constructed using the Extensible Markup Language (XML), X3D scene graphs can be exposed via markup. This lets you weave 3D content into web pages and XML documents like that of Scalable Vector Graphics (SVG), Synchronized Multimedia Integration Language (SMIL), and other XML vocabularies.

The file mountains3.x3d.txt (available electronically) is an X3D scene encoded in XML. In this case, the scene consists of a *NavigationInfo* node that specifies physical characteristics of the viewer's avatar and viewing model, and a *Background* node that specifies ground and sky textures, which create a panoramic backdrop for the scene. Because this scene is expressed in XML, the nodes that make up this scene graph are exposed through the Document Object Model (DOM), and the scene graph itself may be transformed into other formats as needed. In this way, XML-encoded X3D content is a convenient mechanism by which 3D content can be delivered to devices that don't yet support X3D. Figure 4, for instance, shows the X3D scene in human.wrl displayed in a VRML-enabled web browser. Here the XML file was transformed into VRML97 format, letting the scene be viewed using any VRML product. When the benefits of XML aren't required, an alternate data-encoding format (say, X3D's binary and VRML97 UTF-8 encodings) can be used.

MPEG-4

Developed by the Moving Picture Experts Group (MPEG; http://web3dmedia.com/web3d-mpeg/ and http://mpeg.telecomitalialab.com/), MPEG-4 is an ISO/IEC Standard for delivering multimedia content to any platform over any network. As a global media toolkit for developing multimedia applications based on any combination of still imagery, audio, video, 2D, and 3D content, MPEG-4 builds on VRML97 while embracing X3D. MPEG-4 uses the VRML scene graph for composition purposes, and introduces new nodes and features not supported by the VRML Standard. In addition, MPEG has adopted the X3D Interactive Profile as its baseline 3D profile for MPEG-4, thereby enabling 3D content that can play across MPEG-4 and X3D devices.

Recall from my article "The MPEG-4 Java API & MPEGlets" (*DDJ*, April 2002) that MPEG-4 revolves around the concept of discrete media objects composed into scenes. As such, it builds on scene-graph programming concepts popularized by VRML. MPEG-4 also introduces features not supported by VRML—streaming, binary compression, content synchronization, face/body animation, layers, intellectual property management/protection, and enhanced audio/video/2D.

MPEG-4's Binary Format for Scenes (BIFS) composes and dynamically alters MPEG-4 scenes. BIFS describes the spatiotemporal composition of objects in a scene and provides this data to the presentation layer of the MPEG-4 terminal. The BIFS-Command protocol supports commands for adding/removing scene objects and changing object properties in a scene. In addition, the BIFS-Anim protocol offers sophisticated object animation capabilities by allowing animation commands to be streamed directly to scene-graph nodes.

As a binary format, BIFS content is typically 10 to 15 times smaller in size than VRML content stored in plain-text format, and in some cases up to 30 times smaller. (VRML can also be compressed with GZip, although GZip's Lempel-Zip LZ77 compression isn't as efficient as binary compression, resulting in files around eight times smaller than the uncompressed VRML file.)

In its uncompressed state, BIFS content resembles VRML, although nonVRML nodes are often present in the BIFS scene graph. Listing Two, for instance, contains a snippet of the MPEG-4 uncompressed (raw text) ClockLet scene presented in my April article. If you're familiar with VRML, you'll recognize several 2D nodes not defined by the VRML Standard. *Background2D*, *Transform2D*, and *Material2D* are a few of the new nodes introduced by BIFS, which currently supports over 100 nodes.

In addition to new nodes, VRML programmers will notice the absence of the *#VRML V2.0 utf8* comment in the first line of every VRML97 file. ("Utf8" comments identify version and UTF-8 encoding information.) In MPEG-4, information like this is conveyed in object descriptors (OD). Similar in concept to URLs, MPEG-4 ODs identify and describe elementary streams and associate these streams with corresponding audio/visual scene data.

As Figure 5 illustrates, a media object's OD identifies all streams associated with that object. In turn, each stream is characterized by a set of descriptors that capture configuration information that can be used; for instance, to determine what resources the decoder requires or the precision of encoded timing information. Stream descriptors can also convey Quality of Service (QoS) hints for optimal transmission.

MPEG-4 scene descriptions are coded independently from streams related to primitive media objects, during which identification of various parameters belonging to the scene description are given special attention. In particular, care is taken to differentiate parameters that improve object coding efficiency (such as video coding motion vectors) from those that are used as modifiers of an object (such as parameters that specify the position of the object in the scene) so that the latter may be modified without actually requiring decoding of the media objects. By placing parameters that modify objects into the scene description instead of intermingling them with primitive media objects, MPEG-4 lets media be unbound from its associated behavior.

In addition to BIFS, MPEG-4 supports a textual representation called Extensible MPEG-4 Textual format (XMT). As an XML-based textual format, XMT enhances MPEG-4 content interchange while providing a mechanism for interoperability with X3D, SMIL, SVG, and other forms of XML-based media.

Java 3D

Java 3D is a collection of Java classes that define a high-level API for interactive 3D development. As an optional package (standard extension) to the base Java technology, Java 3D lets you construct platform-independent applets/applications with interactive 3D graphics and sound capabilities.

Java 3D is part of Sun's Java Media APIs multimedia extensions (http://java.sun.com/products/java-media/). Java 3D programs are created using classes in the javax.media.j3d, javax.vecmath, and com.sun.j3d packages. Java 3D's primary functionality is provided by the javax.media.j3d package (the core Java 3D classes), which contains more than 100 3D-graphics-related classes. Alternatively, the javax.vecmath package contains a collection of vector and matrix math classes used by the core Java 3D classes and Java 3D programs. A variety of convenience and utility classes (content loaders, scene-graph assembly classes, and geometry convenience classes) are in com.sun.j3d.

Unlike scene-graph programming models, Java 3D doesn't define a specific 3D file format. Instead, it supports run-time loaders that let Java 3D programs support a range of 3D file formats. Loaders currently exist for VRML, X3D, Wavefront (OBJ), AutoCAD Drawing Interchange File (DXF), Caligari trueSpace (COB), Lightwave Scene Format (LSF), Lightwave Object Format (LOF), 3D-Studio (3DS), and more. You can also create custom loaders.

Java 3D uses a DAG-based scene-graph programming model similar to VRML, X3D, and MPEG-4. Java 3D scene graphs are more difficult to construct, however, owing to the inherent complexity of Java. For each Java 3D scene object, transform, or behavior, you must create a new object instance using corresponding Java 3D classes, set the fields of the instance, and add it to the scene. Figure 6 shows symbols visually representing aspects of Java 3D scenes in scene-graph diagrams like those in Figures 7 and 8.

Although complex, Java 3D's programmatic approach is expressive: All of the code necessary to represent a scene can be placed in a central structure, over which you have direct control. Altering Java 3D node attributes and values is achieved by invoking instance methods and setting fields.

The Java 3D term "virtual universe" is analogous to scene or world and describes a 3D space populated with objects. As Figure 7 illustrates, Java 3D scene graphs are rooted to a *Locale* object, which itself is attached to a *VirtualUniverse* object. Virtual universes represent the largest possible unit of aggregation in Java 3D, and as such can be thought of as databases. The *Locale* object specifies a high-resolution coordinate anchor for objects in a scene; objects attached to a *Locale* are positioned in the scene relative to that *Locale*'s high-resolution coordinates, specified using floating-point values.

Together, *VirtualUniverse* and *Locale* objects comprise scene-graph superstructures. Virtual universes can be extremely large and can accommodate more than one *Locale* object. A single *VirtualUniverse* object, therefore, can act as the database for multiple scene graphs (each *Locale* object is the parent of a unique scene graph).

The Java 3D renderer is responsible for traversing a Java 3D scene graph and displaying its visible geometry in an on-screen window (an applet canvas or application frame). In addition to drawing visible geometry, the Java 3D renderer is responsible for processing user input.

Unlike modeling languages such as VRML, rendering APIs such as Java 3D typically give you complete control over the rendering process and often provide control over exactly when items are rendered to screen. Java 3D supports three rendering modes—immediate, retained, and compiled retained—which correspond to the level of control you have over the rendering process and the amount of liberty Java 3D has to optimize rendering. Each successive rendering mode gives Java 3D more freedom for optimizing program execution.

Java 3D lets you create customized behaviors for objects that populate a virtual universe. Behaviors embed program logic into a scene graph and can be thought of as the capacity of an object to change in response to input or stimulus. *Behavior* nodes, or objects, can be added to or removed from a scene graph as needed. Every *Behavior* object contains a scheduling region that defines a spatial volume used to enable the scheduling of the node. The file HelloUniverse.java (available electronically) shows how a simple rotation *Behavior* can be applied to a cube shape in Java 3D. In this case, the rotation *Behavior* makes the cube spin on the y-axis.

Java 3D supports a unique view model that separates the virtual world from the physical world users reside in. Although more complicated than view models typically employed by other 3D APIs, Java 3D's approach lets programs operate seamlessly across a range of viewing devices: A Java 3D program works just as well when viewed on a monitor as when viewed through stereoscopic video goggles. The *ViewPlatform* object represents the user's viewpoint in the virtual world while the *View* object and its associated components represent the physical (Figure 7). Java 3D provides a bridge between the virtual and physical environment by constructing a one-to-one mapping from one space to another, letting activity in one space affect the other.

The Java 3D program HelloUniverse.java (available electronically) is a slightly modified version of Sun's HelloUniverse program. Figure 7 is a corresponding scene-graph diagram. The content branch of HelloUniverse consists of a *Transform-Group* node that contains a *ColorCube* shape node. A rotation *Behavior* node animates this shape by changing the transformation on the cube's *TransformGroup*.

The content branch of this scene graph is on the left side of Figure 7, while the right side illustrates aspects related to viewing the scene. The SimpleUniverse convenience utility manages the view branch so that you don't have to handle these details unless you want that level of control.

Acknowledgments

Thanks to my *Java 3D Jump-Start* coauthor Doug Gehringer of Sun Microsystems, and Mikael Bourges-Sevenier, my coauthor for *Core Web3D* and *MPEG-4 Jump-Start*. Thanks also to Steve Wood, Michelle Kim, Jeff Boston of IBM, and Alex MacAulay of Envivio.

DDJ

(Listings begin on page xx.)

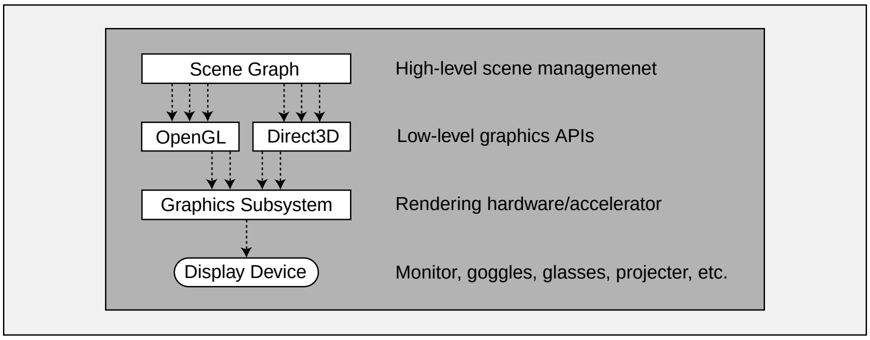

Figure 1: *Scene-graph programming models shield you from underlying graphics APIs, and graphics rendering and display devices.*

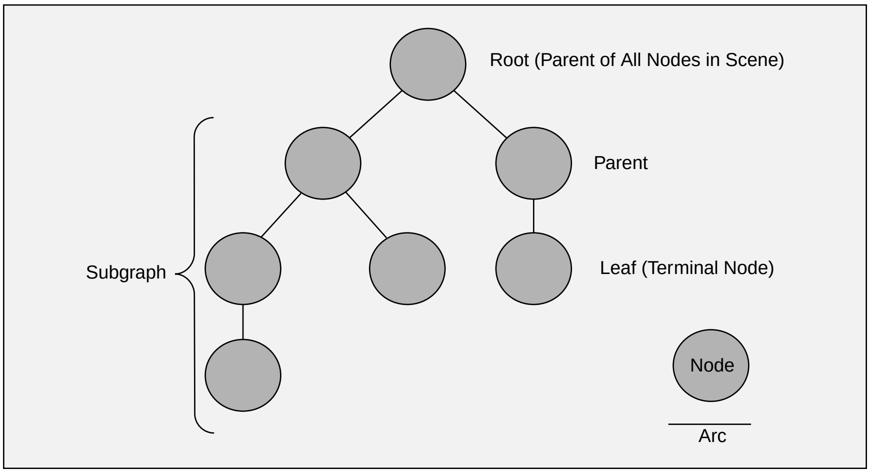

Figure 2: *Scene graphs consist of nodes connected by arcs that define node relationships.*

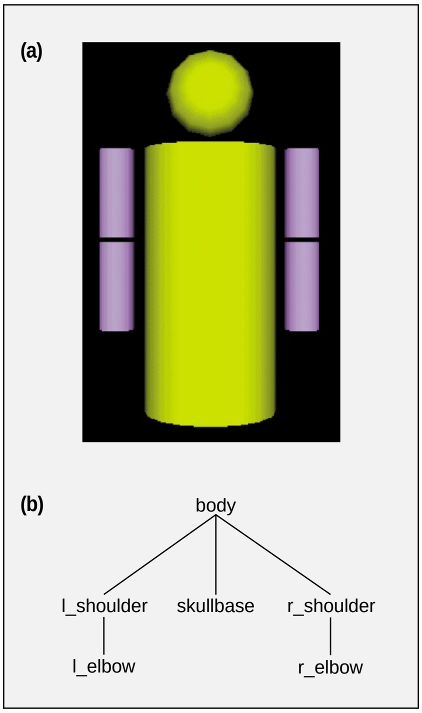

Figure 3: *(a) VRML humanoid; (b) Corresponding scene-graph diagram based on human.wrl.*

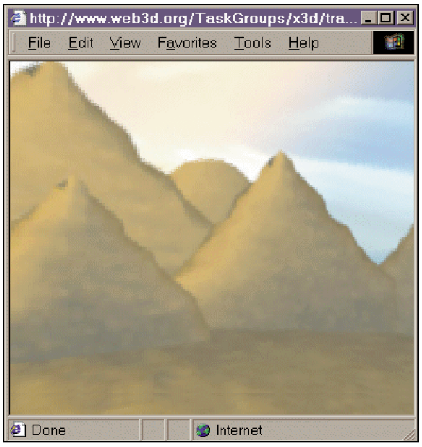

Figure 4: *VRML scene generated from the X3D code in human.wrl. Universal Media images courtesy of Gerardo Quintieri (http://web3dmedia .com/UniversalMedia/).*

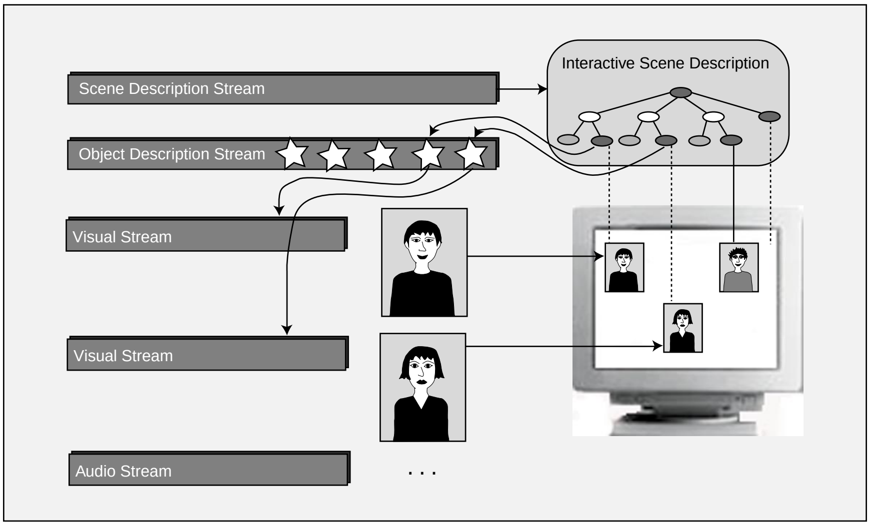

Figure 5: *MPEG-4 media streams are composed at the terminal according to a scene description.*

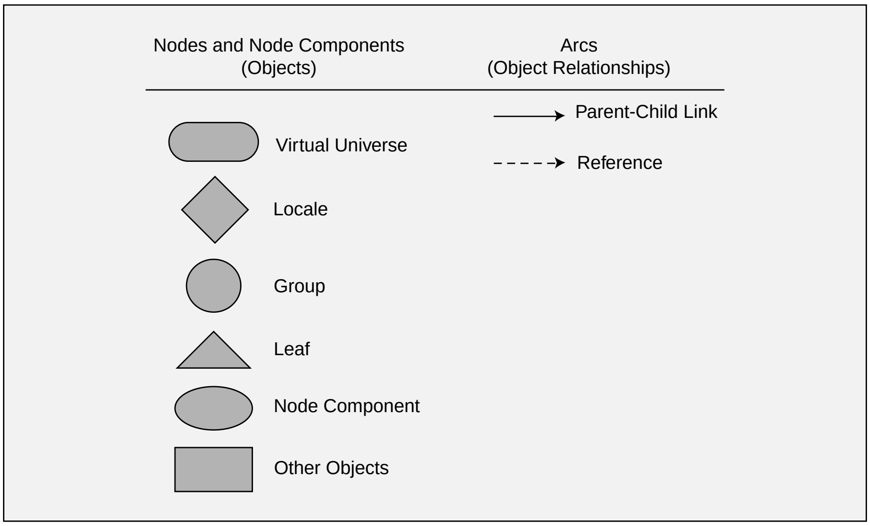

Figure 6: *Symbols commonly used to visually depict Java 3D scene graphs.*

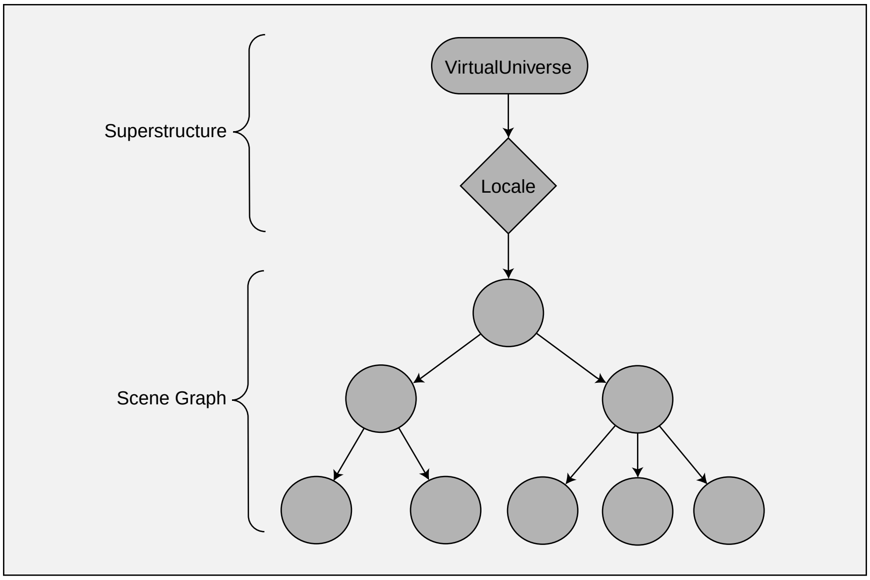

Figure 7: *Java 3D scene-graph nodes are rooted to a* Locale *object that is in turn rooted to a* VirtualUniverse *object.*

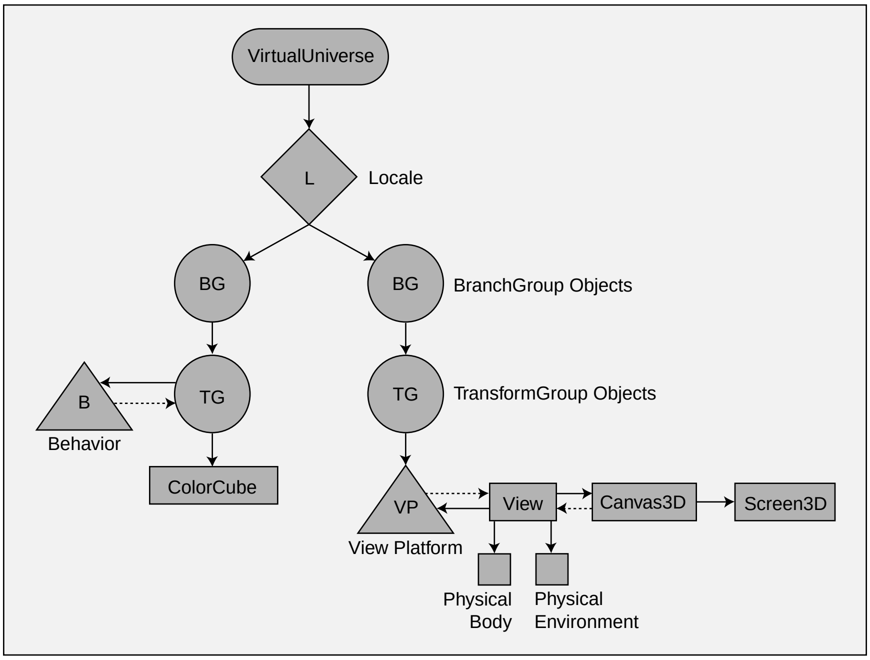

Figure 8: *HelloUniverse scene-graph diagram (right branch provided by the SimpleUniverse utility).*

Listing One

```
#VRML V2.0 utf8
Group {
 children [
  Shape {
   geometry Sphere {}
   appearance Appearance {material Material{}}
  }
  DEF TOUCH TouchSensor { }    # define sensor
  DEF LIGHT DirectionalLight { # define light
    color 1 1 0   # R G B
    on FALSE      # start with light off
  } ]
ROUTE TOUCH.isOver TO LIGHT.set_on
}
```

Listing Two

```
Group {
  children [
    Background2D {
      backColor 0.4 0.4 0.4
      url []
    }
    Transform2D {
      children [
        Transform2D {
          children [
            DEF ID0 Shape {
              appearance Appearance {
                material Material2D {
                  emissiveColor 0.6 0.6 0.6
                  filled TRUE
                  transparency 0.0
                }}
              geometry Rectangle {size 20.0 20.0}
            }
          ]
          center 0.0 0.0
          rotationAngle 0.0
          scale 1.0 1.0
          scaleOrientation 0.0
          translation 0.0 204.0
        }
```

DDJ