1

CHAPTER

Technical Overview

When we mean to build, we first survey the plot, then draw the model.
—William Shakespeare, Henry IV Part II Act 1 Scene 2

1. Introduction

Building and interacting with 3D graphics is a "hands on" experience. There are many examples in this book to teach you how X3D works and to assist you in building your own projects. However, before creating your own Extensible 3D (X3D) graphics scenes, you should understand the background concepts explained here.

The book has an accompanying website at X3dGraphics.com. All examples plus links to other reference material and software are available on the website.

This chapter presents the ideas needed to understand how an X3D scene graph works. This information is used throughout the following chapters and is especially helpful when creating your own X3D scenes. This book assumes that you are interested in learning more about 3D graphics—prior knowledge is helpful, but not required. This chapter is best for people who already have some knowledge of 3D graphics and are ready to learn more of the technical background about how X3D works.

X3D uses a scene graph to model the many graphics nodes that make up a virtual environment. The scene graph is a tree structure that is directed and acyclic, meaning that there is a definite beginning for the graph, there are parent-child relationships for each node, and there are no cycles (or loops) in the graph. Each node has a single parent, except for the X3D root at the top (which has no further parent). The scene graph collects all aspects of a 3D scene in a hierarchical fashion that properly organizes geometry, appearance, animation, and event routing.

X3D is built on the Virtual Reality Modeling Language (VRML), which was first approved as an international standard in 1997. X3D adds Extensible Markup Language (XML) capabilities to integrate with other World Wide Web technologies.

X3D design features include validity checking of content, componentized browsers for faster downloads, flexible addition of new hardware extensions, a lightweight Core Profile, and better script integration compared to VRML. Numerous (more than 2000) example scenes demonstrate most 3D and animation aspects of these scene-graph specifications, and demonstrate syntax checking during autotranslation to VRML encodings. The web3d.org members-only website provide a challenging conformance and performance suite for demonstrating exemplar high-end content. Both XML-based and VRML-based file formats are valid ways to encode the information in an X3D scene. The relative benefits of each are explained and compared in this chapter.

If you are interested in learning how to author X3D scenes right away, you can skim (or even skip) this chapter and read Chapter 2, Geometry Nodes, Part I: Primitives. The background information contained here is not immediately necessary, because the authoring tools take care of headers and structure automatically. The X3D-Edit authoring tool is available to help you build scenes, available free on the book's website.

2. Concepts

This chapter begins with historical background on the development of X3D, presents a brief look at the X3D specifications, and then provides a detailed overview of how the X3D scene graph works. Relevant X3D concepts include scene-graph structure, file encoding, field and node types, and extensibility via profiles and components.

2.1. Historical background: VRML, ISO, and the Web3D Consortium

X3D is a scene-graph architecture and file-format encoding that improves on the VRML international standard (formally listed as ISO/IEC 14772-1:1997 but frequently called VRML 2 or VRML97). X3D uses XML to express the geometry and behavior capabilities of VRML. VRML is well known as a highly expressive 3D interchange format that is supported by many tools and codebases. In addition to expressing diverse geometry renderings and animation behaviors, X3D allows program scripting (in ECMAScript or Java) and node prototyping, which together provide excellent support for scene-graph extensions and new language functionality defined by authors.

The VRML effort began in 1994 when Mark Pesce and Tony Parisi called for the creation of a markup language for 3D graphics on the Web. Several candidates were considered through an open competition, and eventually OpenInventor by Rikk Carey and Paul Strauss at Silicon Graphics Inc. (SGI) was selected as the best basis for creating such a language. Originally called the Virtual Reality Markup Language, VRML 1.0 was quickly produced using an informal, open, consensus-based working-group effort. Several years later, a somewhat restructured (and much improved) VRML 2.0 successfully passed the rigorous scrutiny necessary for approval as International Standard 14772-1:1997, becoming known as VRML97.

To protect VRML as an open 3D-graphics standard, and further encourage the continuation of open development, more process and support was needed than a simple mailing list and web server. A broad cross-section of business companies, academic institutions, government agencies, and individual professionals joined together to form the nonprofit Web3D Consortium. This combination of friendly cooperation, hard work, formal organizational support, and an enthusiastic community have remained the hallmark of VRML and X3D evolution throughout many development cycles.

Despite intermittent industry engagement over the past decade, VRML has persisted and remains the most widely supported nonproprietary 3D-scene format in the world. VRML has now entered its third generation as X3D. Many lessons have been learned and many successful new capabilities have been integrated into this evolving international standard. X3D 3.0 was formally approved by the International Standards Organization (ISO) as ISO/IEC 19775 in 2004. Since that milestone, annual specification updates continue to track the cutting edge of industry capabilities. The Web3D Consortium (www.web3d.org) actively supports many working groups and an active community of interested users. Authors, developers, professionals, and enthusiasts continue to add to the long list of capabilities in X3D. Web3D Consortium membership is open to both organizations and individuals.

2.2. X3D browsers

X3D browsers are software applications that can parse (read) X3D scenes and then render (draw) them, not only showing 3D objects from varying viewpoints but also enabling object animation and user interaction. Sometimes referred to as players or viewers, X3D browsers are often implemented as plugins that work as an integrated part of a regular hypertext markup language (HTML) web browser (such as Mozilla Firefox or Internet Explorer). X3D browsers can also be delivered as standalone or embedded applications that present X3D scenes for user viewing.

Figure 1.1 shows a representative example of the software architecture typically used in a browser. The descriptions in this paragraph follow the blocks in a counterclockwise order (starting at the upper left). X3D scenes are usually files that are read (or written) by the browser. Parsers are used to read the various file-format encodings available. Nodes are then created and sent to a scene-graph manager, which keeps track of defined geometry, appearance, locations, and orientations. The scene-graph manager repeatedly traverses the scene-graph tree of X3D Nodes to draw output image frames at

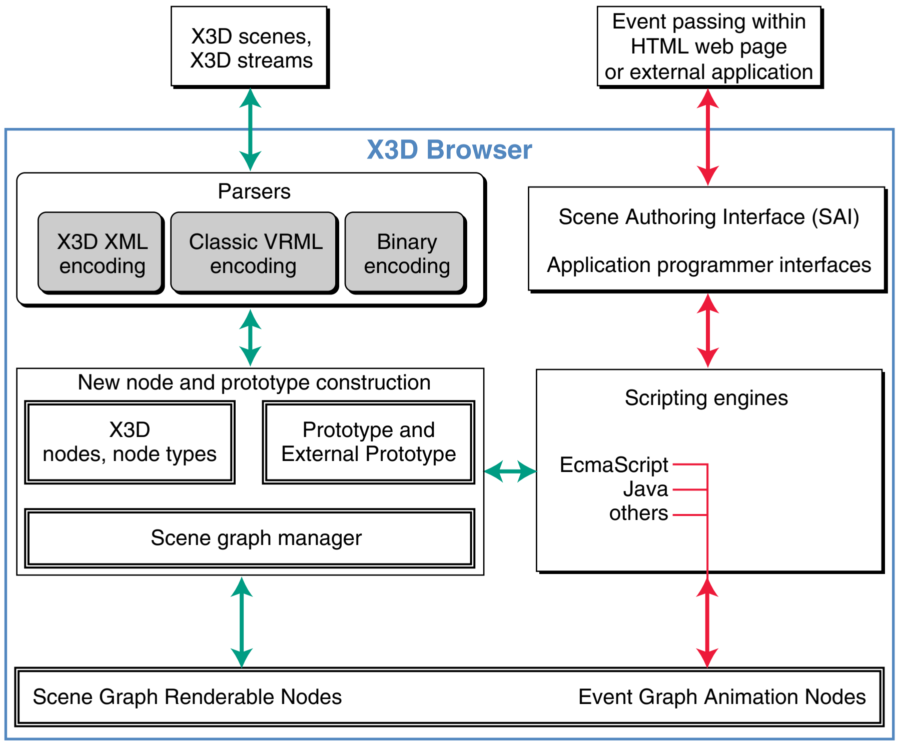

X3D scenes, X3D streams
Event passing within HTML web page or external application
X3D Browser
Parsers
X3D XML encoding
Classic VRML encoding
Binary encoding
Scene Authoring Interface (SAI)
Application programmer interfaces
New node and prototype construction
X3D nodes, node types
Prototype and External Prototype
Scene graph manager
Scripting engines
EcmaScript
Java
others
Scene Graph Renderable Nodes
Event Graph Animation Nodes

Figure 1.1. Example software architecture for an X3D browser.

a rapid rate. This process rapidly redraws precisely calculated perspective-based images as the user's point of view and objects of interest change. The event graph also keeps track of all animation nodes, which are computationally driven to generate and pass value-change events into the scene graph. Events received by the scene graph can change any aspect of the rendered image. Further extending the animation nodes are scripts, which can send or receive events and also generate (or remove) geometry in the scene. Scripts encapsulate programming code from other languages, usually either ECMAScript (formerly known as JavaScript) or Java. The Scene Authoring Interface (SAI) defines how these application programming interfaces (APIs) work, allowing authors to create scripting code that can work across different operating systems and different browser software. Finally (ending at the upper-right corner of the diagram), HTML web pages and external applications can also be used to embed X3D plugin browsers, which appear to users as live, interactive 3D images on the page.

The internals of different browsers vary. Nevertheless, the goal for all is to provide a consistent user experience for each X3D scene. That is one of the greatest strengths of X3D: defining how a software browser ought to draw 3D images and interact with users, rather than trying to tell browser-building programmers exactly how to write their high-performance software. This balance works well because authors can simply focus on building good X3D models (rather than difficult, nonportable software programs) and have confidence that the X3D scenes will work wherever they are displayed.

This approach is sometimes summarized by the slogan "content is king," because achieving interoperability and consistency among X3D scenes is considered more important than the idiosyncracies of any single programming approach. The overall approach also works well because the software programmers who build X3D browsers (an immensely capable but notoriously opinionated group!) can compete on implementation performance and conformance, independently choosing which programming approaches work best, all while agreeing on consistently achieving X3D scene interoperability. Best of all, legacy X3D scene content doesn't "rust" or get bugs. Instead, good content can remain valid and useful indefinitely, without modifications, even as browsers continue to change and improve year after year.

An example scene and corresponding browser snapshot image are provided in the last section of this chapter.

2.3. X3D specifications

The X3D specifications are a highly detailed set of technical documents that define the geometry and behavior capabilities of Classic VRML using the Web-compatible tagsets of XML. Scene graphs, nodes, and fields (in X3D terminology) correspond to documents, elements, and attributes (in XML terminology). As part of the Web3D Consortium, the X3D working Group has designed and implemented the next-generation X3D graphics specification (www.web3D.org/x3d). Lots of feedback and modifications by an active user community improved these results throughout the review process.

It is particularly important to note that XML benefits are numerous: XML has customized metalanguages for structuring data, is easily read by both humans and computer systems, has validatable data constraints, and so on. XML is license free, platform independent and well supported (Bos 2001). Together these qualities ensure that the VRML ISO standard has been extended to functionally match the emerging family of next-generation XML-based Web languages. X3D is now part of that Web-compatible information infrastructure.

The original VRML specification was written to stand alone as a single document. While this made for a simpler reference document, the result was not easily modifiable. Growing from VRML97 to X3D 3.0 took many years of work. Because the X in X3D stands for Extensible, there are now multiple specification documents that govern the coherent evolution and diverse capabilities of X3D. Each can be developed and extended independently, allowing annual updates that document the stable growth of X3D.

It is interesting that the primary functionality of nodes and fields in X3D graphics are specified in a technology-neutral way that is independent of any particular file encoding or programming-language binding. The X3D Abstract Specification remains the governing reference on how the X3D scene graph works. In this way, each file-format encoding and language binding is expected to remain interoperable, compatible, and functionally equivalent with the rest.

Each of these specification documents has been developed through an open, collaborative process by volunteer working-group members in the nonprofit Web3D Consortium. Many of these participants are industry experts who are supported by their companies. Each year, proposed new functionality is implemented (at least twice), evaluated, and specified in formal draft specifications that go to the ISO for final review and ratification. Because all X3D development, implementation, and evaluation is done within the Web3D Consortium process, the overall process is relatively rapid for production of an International Standard. The annual update cycle keeps this work in step with commercial-product development and the always-growing capabilities of the 3D-graphics industry.

The various file encodings and language bindings consistently implement the common-core functionality of the X3D Abstract Specification. The Humanoid Animation (H-Anim) specification is also supported. Figure 1.2 (the honeycom diagram) illustrates how these specifications relate to each other.

The World Wide Web Consortium (W3C) Document Object Model (DOM) is another language-neutral API designed for processing XML documents. It uses string-based

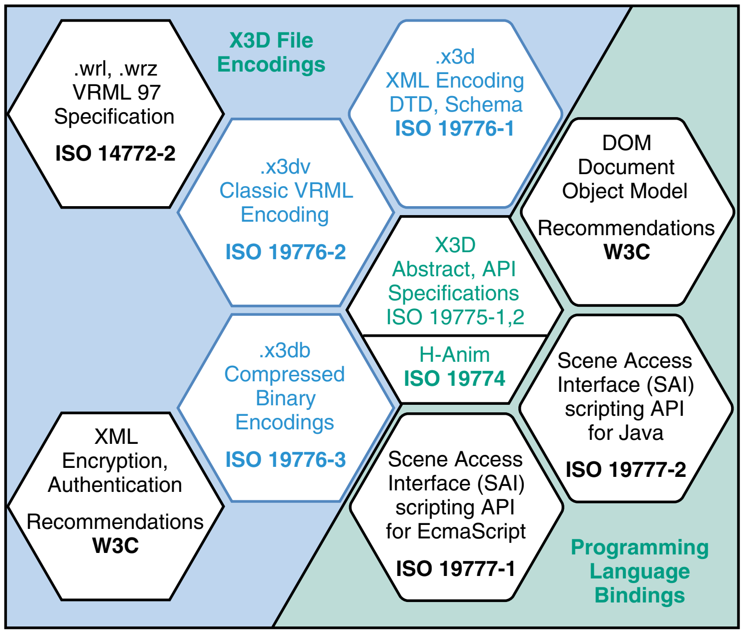


Figure 1.2. The family of X3D specifications includes multiple file encodings and programming-language bindings, all mapping to the same common functionality defined by the abstract specifications.

accessor methods to get and set both element and attribute values. It can also be used to build and modify X3D scenes that are written in the XML encoding. However, because string-based DOM performance is usually too slow for the demands of real-time, interactive 3D graphics, DOM is rarely used directly in combination with a rendering X3D browser. Nevertheless, DOM and other XML-related APIs can be quite useful for creating, reading, or modifying X3D scenes separately from an X3D browser.

Two more W3C Recommendations apply: XML Signature for digital signatures and XML Encryption. Together they are part of the XML Security area, and are compatible with the .X3db Binary Encoding. This rich set of functionality allows authors to reduce file size, optionally reduce geometric complexity (using either lossless or lossy algorithms), digitally sign content to prove ownership, and encrypt content to protect the source document.

Table 1.1 lists the pertinent specifications currently used in X3D and this book. They are available online (www.web3d.org/x3d/specifications), with X3D-Edit, and on the website accompanying this book. Installing the specifications on your working computer and referring to them in combination with this book is a good way to become familiar with the many details and options available in X3D.

Someday additional programming language bindings (perhaps for C++ and/or C#, perl, and so on) might be added to the X3D specification. Although this kind of work requires significant expertise, producing such an addition to the 19777 specification series of API language bindings is a relatively straightforward matter. The well-defined nature of X3D extensibility requires that all functionality align with already-standardized X3D Abstract Specification. The precise objects and methods of any new API are likely to resemble the similarly structured ECMAScript and Java language bindings. This means that the most difficult work (i.e., defining compatible 3D functionality) is already done, and any new language bindings merely have to define how they will implement well-specified X3D capabilities. Last but not least, any such programming-language interface must be independently implemented in at least two browsers before specification approval.

All of the X3D specifications are freely available in electronic form, updated online. Paper hardcopies or CDs can also be purchased directly from ISO. Serious X3D authors are advised to keep a copy of the specifications easily available to refer to when tricky questions arise. Copies of the X3D specifications, X3D tooltips, and other help files are linked on X3dGraphics.com for your convenience. Although this kind of reading can be difficult at first, the effort is worthwhile. Asking questions online about specification details is a good way to learn more. Perhaps someday you will even find yourself proposing an improvement to the X3D specifications on a Web3D mailing list. A large, friendly community of interest posing questions and solutions has made it possible for X3D to grow steadily since 1994.

2.4. Scene graph

Knowing some of the theory behind X3D is helpful. Scene graphs are a model-centric approach. Model geometry, size, appearance, materials, relative locations, and internal

Specification Number	Document Title	Description
ISO/IEC 14772	The Virtual Reality Modeling Language: Part 1 with Amendment 1 and Part 2,Known as VRML97	VRML 2.0 after formal specification review. Added EAI API. Superceded by X3D, which replaced EAI with SAI.
ISO/IEC 19774	Humanoid Animation (H-Anim)	Abstract definitions for H-Anim functionality, includes VRML encoding.
ISO/IEC 19775-1 & ISO/IEC 19775-1/Amendment 1	Extensible 3D (X3D)—Part 1: Architecture and base components	X3D Abstract Specification. Defines scene-graph functionality including nodes, components and profiles.
ISO/IEC 19775-2	Extensible 3D (X3D)—Part 2: Scene access interface (SAI)	X3D Abstract Specification of principles and semantics for SAI API.
ISO/IEC 19776-2 & ISO/IEC 19776-1/Amendment 1	Extensible 3D (X3D) encodings—Part 1: Extensible Markup Language (XML) encoding	XML encoding syntax for .x3d files.
ISO/IEC 19776-2 & ISO/IEC 19776-2/Amendment 1	Extensible 3D (X3D) encodings—Part 2: Classic VRML encoding	VRML encoding syntax for .x3dv files.
ISO/IEC 19776-3	Extensible 3D (X3D) encodings—Part 3: Binary encoding	Binary encoding syntax for .x3db files.
ISO/IEC 19777-1	Extensible 3D (X3D) language bindings—Part 1: ECMAScript	SAI functionality and syntax for script code written in ECMAScript.
ISO/IEC 19777-2	Extensible 3D (X3D) language bindings—Part 2: Java	SAI functionality and syntax for script code written in Java.

API, application programming interface; EAI, external authoring interface; H-Anim, humanoid animation; SAI, scene access interface; VRML, virtual reality modeling language; X3D, Extensible 3D; XML, Extensible markup language.

Table 1.1. X3D Specifications Summary Table

relationships are expressed as a directed acyclic graph (DAG). Data functionality is collected in nodes, which contain field parameters. Field parameters may contain simple-datatype values or further nodes. This approach allows program-independent, logical structuring of model data. The inclusion of viewpoint definitions makes it easy to add different user viewing perspectives.

It is interesting that scene graphs have been used by many 3D-graphic APIs, including OpenInventor and Java3D. Scene-graph concepts were derived from the notion of display lists, used in early 3D graphics. Scene-graph design is well suited for equivalent model representations as source code or file content. Thus, learning X3D

Brutzman-Ch01.qxd 2/16/07 12:27 PM Page 9

can be valuable, because it provides an introductory path for many approaches to 3D graphics. Web authors and students do not need a programming background to get started.

Several 3D browsers render and animate scene graphs in a straightforward way. Starting at the root of the tree, the scene graph is traversed in a depth-first manner. Traversal of transformation nodes modifies the location and orientation of the current coordinate system (i.e., the current relative reference frame). Traversing appearance or material properties modifies the rendering parameters for subsequent geometry. Traversing geometry nodes renders polygons. Traversal of USE nodes (instance references) can efficiently redraw previously created structure without compromising the acyclic nature of the DAG tree.

Behaviors are an essential aspect of scene graphs. A ***behavior*** is defined as changing the value of some field within some node of the scene graph. This usually means generating numbers to animate the size, shape, position, or orientation of an object. Behaviors are powerful and general, because any parameter in the scene graph is usually manipulatable in one way or another. Data producers (such as sensor nodes monitoring the user, linear-interpolation functions, or scripts) are connected to other 3D nodes in the scene graph via ROUTEs. Thus a ***ROUTE*** passes events, which are time-stamped behavior values, produced by the field of one node and sent to the field of another node. This technique is presented in Chapter 7, Event Animation and Interpolation.

Behavior ROUTEs make up an event-routing graph that connects nodes and fields within the encapsulating scene graph. One way to visualize these relationships is print out a hardcopy of a pretty-print HTML version of a scene (landscape printing is usually best), and simply draw ROUTE lines from each target node to each destination node. Then annotate the beginning and end of each line with the name and data type of the source and target fields. The result is a copy of the event graph superimposed on the scene graph. This is an excellent learning technique for understanding behavior relationships within an X3D scene.

Taken together, the rules governing this animation-behavior framework are commonly referred to as the ***X3D event model***. More rules regarding parallelism are provided in the X3D specification for browser builders. Nevertheless these special cases are not often a worry for X3D authors.

The overall design of X3D to augment the geometry of a scene graph with interpolators, animations, and behaviors is an excellent architectural example of a declarative simulation. The defining scene declares the various implicit interrelationships between objects, and the actual simulation sequence of events is determined simply by user-provided interactions and by author-defined timing of event production. Resultant behavior typically remains unaffected by differences in processor speed on different viewing computers, because animated events are synchronized with wall-clock time rather than processor-clock time. Addition of scripting nodes can further integrate explicit step-by-step imperative algorithms, compatibly operating imperatively (for short time intervals) within the declarative scene-graph framework.

Declarative simulations are powerful. The author can focus on the constraints and relationships between different parts of the scene graph, and desired behaviors emerge from those connections. This is much different (and usually much simpler) than writing an imperative program using a graphics programming language, in which every detail (draw this triangle, draw that triangle) must be directed and handled exhaustively. Experienced 3D-graphics programmers are often pleasantly surprised at the power and capability of X3D, packed into a much simpler modeling methodology.

2.5. File structure

The X3D scene graph is usually presented in a file, using either the .x3d XML encoding or the .x3dv Classic VRML encoding. The same-graph structure is well defined and remains consistent in each encoding. Each encoding imposes its own requirements on the syntax and layout of the common representation of information.

The top-level structure of X3D files is:

- File header
- X3D header statement
- Profile statement
- Component statement (optional, multiple)
- META statement (optional, multiple)
- X3D root node (implicit in Classic VRML encoding)
- X3D scene graph child nodes (multiple)

Each structural file element is described in more detail in the following sections.

2.5.1. File header

The X3D file header contains the primary setup information about the X3D scene. There are no renderable nodes in this portion of the file. The header contains required and optional information about the scene capabilities. The file header comprises the following statements: XML and X3D headers, profile, component, and meta.

Table 1.2 shows the proper definitions for a file that includes the Immersive profile, multiple components, and example meta tags. The .x3d XML encoding is similar to XHTML in that most of the header information is contained in a `<head>` tag. Complete details are provided in example scene, HeaderProfileComponentMetaExample.x3d. The following sections contain information on each part of the file header.

2.5.2. X3D header statement

The X3D header statement identifies the file as an X3D file. The specific format and position is encoding dependent. The information in the header statement is the X3D identifier, X3D version number, and the text encoding. X3D uses the (universal text format UTF-8) character encoding, which supports essentially all electronic alphabets for different human languages. The allowed versions of X3D include 3.0 and 3.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.1//EN"
         "http://www.web3d.org/specifications/x3d-3.1.dtd">
<X3D version="3.1" profile="Immersive"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xsd:noNamespaceSchemaLocation=
    "http://www.web3d.org/specifications/x3d-3.1.xsd">
      <head>
        <component name='DIS' level='1'/>
        <component name='Geospatial' level='1'/>
        <component name='H-Anim' level='1'/>
        <component name='NURBS' level='4'/>
        <meta name='filename'
          content='HeaderProfileComponentMetaExample.x3d'/>
      </head>
      <Scene>
          <!--Scene graph nodes are added here-->
      </Scene>
 </X3D>
```

```
#X3D V3.1 utf8
PROFILE Immersive
# No HEAD statement is provided in ClassicVRML Encoding
COMPONENT DIS:1
COMPONENT Geospatial:1
COMPONENT H-Anim:1
COMPONENT NURBS:4
META "filename" "HeaderProfileComponentMetaExample.x3d"
# Scene graph nodes are added here
```

Table 1.2. Comparison of XML (.x3d) and ClassicVRML (.x3dv) header syntax

The XML encoding matches general XML header requirements, starting with the `<?xml?>` declaration. Thus all .x3d scenes must first be well-formed XML: properly formed open, close, and singleton tags, single- or double-quoted attributes, and so on. There are two available XML-based mechanisms to further validate the correctness of the .x3d file. Each is optional but recommended. A Document Type Definition (DTD as indicated by a DOCTYPE statement) and an XML Schema reference. Schema information appears in the document root `<X3D>` tag. Version number is a required part of the X3D root declaration in all encodings.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.1//EN"
   "http://www.web3d.org/specifications/x3d-3.1.dtd">
<X3D profile="Immersive" version="3.1"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
   xsd:noNamespaceSchemaLocation=
   "http://www.web3d.org/ specifications/x3d-3.1.xsd">
```

The character-encoding is identified in the `<?xml?>` declaration and Universal Text Format (UTF-8) is the most commonly used. The X3D identifier and version number are also included in the optional (but recommended) DOCTYPE and Schema references. Using a later version number in the DOCTYPE or Schema reference than the X3D root node requires is allowed, because each new version of X3D maintains backwards compatibilty.

For the ClassicVRML encoding, the header is as follows.

```
#X3D V3.1 utf8
```

This must be the first statement in a ClassicVRML encoded X3D file. Unlike the XML encoding, no external validation references are provided, because browsers are assumed to be capable of properly parsing the VRML-based scene syntax.

Like comment statements, header information is generally not available for run-time access after an X3D scene has been loaded into memory and begun running. Persistent information (such as metadata) can be stored within the scene using the strictly datatyped, persistent metadata nodes.

This book is based on ISO/IEC 19775 Parts 1 and 2, ISO/IEC 19775 Amendment 1, ISO/IEC 19776 Parts 1 and 2, ISO/IEC 19776 Parts 1 and 2 Amendment 1, and ISO/IEC 19777 Parts 1 and 2. These are known collectively as X3D 3.0 and X3D 3.1. The version number of X3D changes with each amendment release of the specification. The X3D 3.0 specification was approved by ISO in 2005. X3D 3.1 was approved in 2006, correcting some minor problems in X3D 3.0 and adding significant new functionality.

As additional amendments are developed and approved, the X3D version number is incremented accordingly. The Web3D Consortium and ISO continue working to produce extension updates, completing an amendment about every 18 months. As this book goes to press, amendment 1 (X3D version 3.1) is approved, amendment 2 (X3D version 3.2) is undergoing formal review, and proposed amendment 3 extensions are undergoing collaborative Web3D working-group implementation and evaluation.

2.5.3. Profile statements

There are a number of profiles defined for X3D. Each profile is targeted for a common market or commonly used set of functionality. Profiles exist in order to enable browser builders to achieve intermediate levels of support for X3D, rather than trying to implement a large specification at once. Profiles also assist authors, because they can target their created scenes to use the functional vocabulary supported in a given profile. This means that content is well defined and more likely to be widely portable. Profiles help conversion programs translate between different file formats. Finally, profiles help extend the reach of X3D to smaller, lightweight devices such personal digital assistants and cell phones.

The minimalist X3D profile is called Core. It includes only the basic X3D definitions (such as ROUTEs) and the metadata nodes, but does not include any geometry, appearance, or animation capabilities. The Core profile is a base level so that an author can specify the special functionality required for a virtual environment using component statements. Several intermediate profiles are also provided: Interchange, Interactive, CADInterchange and Immersive. The CADInterchange profile is designed to support web export and interoperability for computer-aided design (CAD) formats using X3D. An MPEG-4Interactive profile (similar to the Interactive profile) has been approved for use with Motion Picture Experts Group (MPEG-4) audio, video, and multimedia content.

The Full profile includes everything defined in all related X3D specifications. It provides a means for referring to every X3D component without naming any of them.

Typically each profile is a superset of the preceding profile, shown in Figure 1.3 (known as the "onion-layers" diagram). Details about each profile follow.

Core: The Core profile is not designed for regular use. Rather, it provides the absolute minimal definitions required by an X3D browser (essentially just the metadata nodes). Advanced authors can build minimally defined scenes by explicitly specifying the component and levels required in the scene.

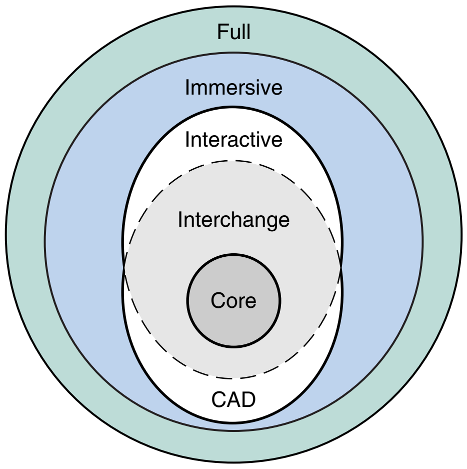


Figure 1.3. Multiple X3D profiles providing increasing sophistication allow more efficient browser support for lightweight content.

Interchange: This is the base profile for exchanging geometric models between various 3D applications. It is designed to be easy to export and import. All of the basic geometry (including primitives, triangles, and polygons), appearance (material and texture), and keyframe animation nodes are included in this profile.

Interactive: This profile is slightly bigger than the Interchange profile, and adds most of the nodes necessary for users to interact with a scene.

MPEG-4Interactive: This profile was designed specifically for the MPEG-4 multimedia specification's need for 3D graphics definitions, and is an appropoximate match to the Interactive profile. It is not discussed further in this book.

CADInterchange: This is a specialized profile that supports the import of CAD models. It includes most (but not all) nodes from the Interchange profile and includes a handful of new nodes for CAD. The details of this profile are not addressed in this book.

Immersive: This profile most closely matches VRML97. The Immersive profile includes everything defined in Interactive plus several advanced capabilities and a number of new nodes: 2D geometry, environmental effects, event utilities, and so on. It is a superset of VRML97, but without the extra functionality of VRML specification ISO/IEC 14772/Amendment 1.

Full: This profile includes all nodes defined in the X3D specification and further extends the Immersive profile. The Full profile incorporates all X3D capabilities including Distributed Interactive Simulation (DIS), Humanoid Animation (H-Anim), GeoSpatial, Non-Uniform Rational B-spline Surfaces (NURBS) and other advanced components.

A profile statement is required in all X3D scenes. The format of the profile statement depends on the file encoding. The Classic VRML file encoding uses a profile statement format of `PROFILE Immersive` while the XML file encoding embeds the information in a `profile="Immersive"` attribute in the document's root X3D tag.

The approval of new nodes as a part of annual amendments means that most profiles can continue to evolve. The Core profile is always expected to remain at the minimum possible set. The Full profile is always the total of the capabilities provided in a particular version of X3D. Changes for each major or minor version number (for example 3.0, 3.1, or 3.2) always correspond to well-defined profiles.

2.5.4. Component statements

Each of the preceding profiles is made up of a collection of components. Each component is divided into levels that describe increasing capability. Every X3D node is part of a single component, and has the same or better features at each succeeding component level.

As an addendum to a profile declaration in a scene file, the component statement informs the browser that a scene needs further support for functionality at the specified component at the specified level. Components at a specific level usually do not need to be listed in the header of a scene, because they are usually included in the scene's declared profile. In some cases, however, defining component details is useful. Adding component statements provides authors finer-grained control of the browser support needed to support the scene.

Normally an author's request for a given node capability in a browser is made as part of the encompassing profile statement. Nevertheless, there are two main reasons for using the component statement to provide further additions to the supported profile.

1. The stated profile does not support one or more of the nodes included in a scene. The browser may not support Full profile, or the author may prefer to allow a lighter-weight browser to handle the content running on the user's machine.
2. The author is constructing a minimal environment for the scene, perhaps as part of a library or collection of complementary objects. By specifying a lower profile along with a few necessary components, more browsers may be able to present the scene. Similarly, more parent scenes can include the child scene without requiring a higher profile.

There are 24 components in X3D 3.0, each with multiple levels. In 2005, X3D version 3.1 added four more, providing a total of 28 components. Table 1.3 lists the various components and corresponding levels provided by each of the profiles.

Thus, in most cases, it is simply a good practice to pick a sufficient profile that covers all of the nodes in a scene. Immersive, Interactive, and Interchange are the most commonly used profiles. One authoring option is to specify the Full profile for every scene, but that isn't always a practical solution, because browser support for the Full profile is often limited. This situation is commonly encountered when working with components outside the Immersive profile, such as the GeoSpatial or H-Anim nodes. (Note: these advanced nodes are not discussed in this book.)

2.5.5. Meta statements

Meta statements provide information about the X3D scene. These annotations are frequently used to provide author, copyright, reference, and other information. Each meta statement contains a name-value pair, that is, the name of the metadata item and the corresponding content for that one piece of information. Name-value string pairs are quite powerful and can be used to capture nearly any type of information.

The head and meta statements are directly patterned after the Extensible Hypertext Markup Language (XHTML). This approach simplifies conventions for use and maximizes consistency for web authors.

The meta statements contained in the header are not typed, meaning that each name content value is a string. Meta tags can only appear in the scene header, and are different than the various metadata nodes, which are only allowed to appear within the scene itself.

Example file newScene.x3d provides a number of default meta tags. It is the default scene used by X3D-Edit. Use it as a starter when beginning a new scene, and change the file name and meta values to record information about the work. Excerpts follow in Figure 1.4, including further references about defining metadata values consistently.

Components	Interchange Profile Supported Levels	CAD Interchange Profile Supported Levels	Interactive Profile Supported Levels	Immersive Profile Supported Levels	Full Profile Supported Levels
CADGeometry (X3D 3.1)		2			2
Core	1	1	1	2	2
Cube map environmental texturing (X3D 3.1)					3
Distributed interactive simulation (DIS)					1
Environmental effects	1		1	2	3
Environmental sensor			1	2	2
Event utilities			1	1	1
Geometry2D				1	2
Geometry3D	2		3	4	4
Geospatial					1
Grouping	1	1	2	2	3
Humanoid animation					1
Interpolation	2		2	2	3
Key device sensor			1	2	2
Lighting	1	1	2	2	3
Navigation	1	2	1	2	2
Networking	1	1	2	3	3
NURBS					4
Pointing device sensor			1	1	1
Programmable shaders (X3D 3.1)					1
Rendering	3	4	2	3	4
Scripting				1	1
Shape	1	2	1	2	3
Sound				1	1
Text				1	1
Texturing	2	2	2	3	3
Texturing3D (X3D 3.1)					2
Time	1		1	1	2

CAD, Computer-aided design; NURBS, non-uniform rational B-spline; X3D, Extensible 3D.

Table 1.3. X3D Profiles, Components, and Corresponding Support Levels

```
<head>
  <meta name='filename' content='*enter FileName
     WithNoAbbreviations.x3d here*'/>
  <meta name='description' content='*enter description here,
     short-sentence summaries preferred*'/>
  <meta name='author' content='*enter name of original author here*'/>
  <meta name='translator' content='*if manually translating
     VRML-to-X3D, enter name of person translating here*'/>
  <meta name='created' content='*enter date of initial version *'/>
  <meta name='translated' content='*enter date of translation here*'/>
  <meta name='revised' content='*enter date of latest revision *'/>
  <meta name='version' content='*enter version here, if any*'/>
  <meta name='reference' content='*enter reference citation or
     relative/online url here*'/>
  <meta name='copyright' content='*enter copyright information here*
    Example: Copyright (c) Web3D Consortium Inc. 2007'/>
  <meta name='drawing' content='*enter drawing filename/url here*'/>
  <meta name='image' content='*enter image filename/url here*'/>
  <meta name='movie' content='*enter movie filename/url here*'/>
  <meta name='photo' content='*enter photo filename/url here*'/>
  <meta name='keywords' content='*enter keywords here*'/>
  <meta name='permissions' content='*enter permission statements or
     url here*'/>
  <meta name='warning' content='*insert any known warnings, bugs or
     errors here*'/>
  <meta name='url' content='*enter online url address for this file
     here*'/>
  <meta name='generator' content='X3D-Edit,
    http://www.web3d.org/x3d/content/README.X3D-Edit.html'/>
  <meta name='license' content='../../license.html'/>
  <!--Additional authoring resources for meta-tags:
  http://www.w3.org/TR/html4/struct/global.html#h-7.4.4
  http://dublincore.org/documents/dces
  http://vancouver-webpages.com/META
  http://vancouver-webpages.com/META/about-mk-metas2.html
  Additional authoring resources for language codes:
  ftp://ftp.isi.edu/in-notes/bcp/bcp47.txt
  http://www.loc.gov/standards/iso639-2/langhome.html
  http://www.iana.org/numbers.html#L
  -->
</head>
```

Figure 1.4. Example meta tags in newScene. x3d.

Metadata produces big dividends over time, particularly if different or competing versions of a scene are available. Think of meta tags as a way to communicate with others about the scene, to help yourself keep track of content in the future, and to support library archive tools that keep track of many example scenes. Meta tags are used throughout the examples provided with this book for just such purposes.

2.5.6. Scene graph body

Following the head of the X3D document is the scene. Putting together a scene graph is the focus of the rest of this book.

To further explain the technical fundamentals underlying the X3D nodes that make up the scene graph, field types and node types are presented next.

2.6. Field types

The data for each node is stored in the fields of the node. Fields can contain a single value or multiple values for each data type. All fields are built from the fundamental X3D data types for boolean, integer, single-precision floating point, double-precision floating point, and strings. Simple field types may have one or multiple values, and can further be arrays of such values. Field types are also provided for a singleton node (SFNode) and arrays of nodes (MFNode). There are restrictions or structures placed on some of the elements in various fields.

The X3D field-naming convention starts with two letters designating a single-valued field (SF) or multiple-valued field (MF). Next comes the name of the data type of the field, for example, SFString, MFVec3f, and so on. In the XML encoding, no square brackets are needed. The type names are consistent for both text-based encodings (XML and ClassicVRML). In the Classic VRML encoding, an additional requirement is that multiple-valued fields (MFs datatypes) must be enclosed in square brackets. For example, a 4-tuple SFRotation field value is expressed as `"0 1 0 1.57"` in .x3d files or `[0 1 0 1.57]` in .x3dv files.

For the node-definition tables appearing throughout the book, a specific notation is used. Default values are listed as typed literal values from the specification. Ranges of values are listed using standard mathematical notation: A square bracket means that the range includes the bounding value, while a parenthesis means that the range does not include the bounding value. For example [0,1] means all numbers between 0 and 1, including 0 and 1. Similarly, (0,1] means all numbers between 0 and 1, including 1 only. As another example, $[0, +\infty)$ means all non-negative numbers (zero inclusive).

Individual string values are enclosed in quotation marks. If array values are *n-tuples* (meaning arrays with 2, 3, or 4 numbers), the value is quoted with spaces separating the data elements (for example, `translation="0 0 0"`). Multiple-valued data-type arrays contain zero or more copies of the single-value data type. Note that commas can only appear between each n-tuple field value, not within them. The uninitialized value for multiple-valued data-type arrays is always the empty list, which is an array containing no elements.

Table 1.4 lists abbreviations, names, and example values for each simple field type.

Field-type names	Description	Example values
SFBool	Single-field boolean value	true or false (X3D syntax), TRUE or FALSE (ClassicVRML syntax)
MFBool	Multiple-field boolean array	true false false true (X3D syntax), [TRUE FALSE FALSE TRUE] (ClassicVRML syntax)
SFColor	Single-field color value, red-green-blue	0 0.5 1.0
MFColor	Multiple-field color array, red-green-blue	1 0 0, 0 1 0, 0 0 1
SFColorRGBA	Single-field color value, red-green-blue alpha (opacity)	0 0.5 1.0 0.75
MFColorRGBA	Multiple-field color array, red-green-blue alpha (opacity)	1 0 0 0.25, 0 1 0 0.5, 0 0 1 0.75 (red green blue, varying opacity)
SFInt32	Single-field 32-bit integer value	0
MFInt32	Multiple-field 32-bit integer array	1 2 3 4 5
SFFloat	Single-field single-precision floating-point value	1.0
MFFloat	Multiple-field single-precision floating-point array	–1 2.0 3.14159
SFDouble	Single-field double-precision floating-point value	2.7128
MFDouble	Multiple-field double-precision array	–1 2.0 3.14159
SFImage	Single-field image value	Contains special pixel-encoding values, see Chapter 5 for details
MFImage	Multiple-field image value	Contains special pixel-encoding values, see Chapter 5 for details
SFNode	Single-field node	<Shape/> or Shape {space}
MFNode	Multiple-field node array of peers	<Shape/><Group/><Transform/>
SFRotation	Single-field rotation value using 3-tuple axis, radian angle form	0 1 0 1.57
MFRotation	Multiple-field rotation array	0 1 0 0, 0 1 0 1.57, 0 1 0 3.14
SFString	Single-field string value	"Hello world!"
MFString	Multiple-field string array	"EXAMINE" "FLY" "WALK" "ANY"
SFTime	Single-field time value	0
MFTime	Multiple-field time array	–1 0 1 567890

Table 1.4. X3D Field Types

Field-type names	Description	Example values
SFVec2f/SFVec2d	Single-field 2-float/2-double vector value	0 1.5
MFVec2f/MFVec2d	Multiple-field 2-float/2-double vector array	1 0, 2 2, 3 4, 5 5
SFVec3f/SFVec3d	Single-field vector value of 3-float/ 3-double values	0 1.5 2
MFVec3f/MFVec3d	Multiple-field vector array of 3-float/ 3-double values	10 20 30, 4.4 –5.5 6.6

VRML, Virtual reality modeling language; X3D, Extensible 3D.

Table 1.4. (Cont'd.) X3D Field Types

2.7. Abstract node types

A major improvement in the design of the X3D language over VRML97 is the addition of strong typing of nodes, making the object-oriented nature of X3D nodes much more consistent. The X3D specification accomplishes strong node typing in two ways, first by defining field interfaces for child-node content and second by defining required simple-field attributes corresponding to each functional type of node. Although this aspect of the X3D architecture is not usually evident to authors, good language design leads to more predictable models and more consistent behavior. Strictly defined node typing gives the following benefits to X3D nodes implementing the same node type:

- Allowed child-node content is identical
- Simple-type field attributes are identical and have consistent default values
- Validation capabilities are improved
- Common APIs are the same
- Definitions and operations are easier to remember and adapt

New nodes (defined either by author prototypes or in future versions of X3D) with matching node types con be substituted for other nodes correctly and consistently. Thus, node types directly support Extensibility, the X in X3D.

Script programming uses node types and is covered in Chapter 9, Event Utilities and Scripting. Application programming interfaces (APIs) are more important when you are learning how to program X3D scene graphs directly. That is a big subject, suitable for another book. Most scene content preparation goes into the production of files, so file encodings are discussed in the next section.

2.8. File encodings: XML, ClassicVRML, and Compressed

There are three available encodings that can be used to format X3D files. In each case, the functionality of the displayed X3D scene remains consistent and independent of the encoding used. This is quite valuable, because any X3D encoding for a single given scene can be treated as visually and functionally equivalent. This is the same principle previously illustrated by Figure 1.2: an X3D scene renders and behaves consistently at run time, regardless of the file format (or programming API) used to create it.

The first file format is XML, which is plaintext and uses the .x3d filename extension. The second is also plaintext and based on the Classic VRML syntax of curly brackets and square brackets, using the .x3dv filename extension. The third is the compressed-binary format, which includes both geometric polygon/property compression as well as binary-data compression, using the .x3db filename extension. Gzip compression of .x3d or .x3dv files is another allowed approach, appending .gz as an additional extension to the original filename and extension, but the .x3db compressed binary encoding provides superior results.

X3D files that refer to other files (by using Inline and Anchor nodes) can legally refer to linked X3D scenes that have been saved with a different encoding than the master scene. Thus, .x3d, .x3dv, and .x3db file references are all legal in any combination of scenes. Hopefully all three encodings are supported by each user's browser as well.

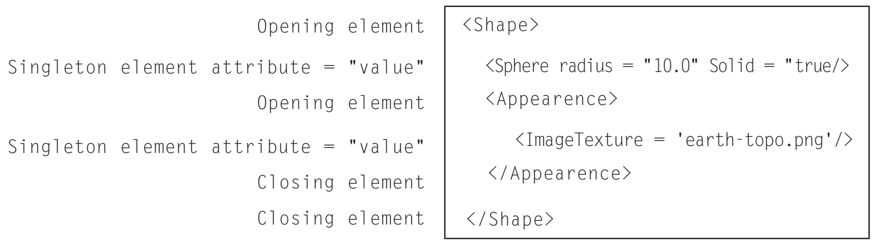
Some browsers also support VRML97, which uses the .wrl filename extension for plaintext files and .wrz/.wrl.gz for gzip-compressed files. Nevertheless, it is good practice to upgrade VRML97 scenes to X3D for maximum compatibility and interoperability. Multiple VRML97-to-X3D translators are provided on the book's website.

2.8.1. Extensible Markup Language (XML) encoding: .x3d files

XML encoding is one of the biggest improvements in X3D. This section describes the benefits of using an XML encoding as the file format for X3D.

XML-based files are usually called XML documents. Figure 1.5 shows an example of an XML document fragment.

Each element (sometimes referred to as a tag) is surrounded by angle brackets. Elements that contain other elements start with an opening tag (no slash) and finish with an ending tag (leading slash). Standalone elements that have no child elements can either finish with a trailing slash, such as `<Sphere radius="10.0" solid="true"/>` or simply a matched pair of opening and closing elements, such as `<Sphere radius="10.0" solid="true"></Sphere>`. Intervening whitespace between elements is usually insignificant, so source-file formatting and layout is flexible. These simple XML rules for structuring data provide a lot of descriptive power.


Opening element
Singleton element attribute = "value"
Opening element
Singleton element attribute = "value"
Closing element
Closing element

```
<Shape>
  <Sphere radius = "10.0" Solid = "true/>
  <Appearence>
    <ImageTexture = 'earth-topo.png'/>
  </Appearence>
</Shape>
```

Figure 1.5. XML documents have a consistent tree structure for elements, attributes, and values.

2.8.1.1. XML motivations

There are many reasons to use XML. Foremost is that XML is the basis of nearly every data language used on the World Wide Web. If 3D graphics are to become a "first-class citizen" on the Web, embedded in Web pages and interacting with clients, servers, and users of every type, then it is clear that an XML encoding is necessary; the .x3d encoding of X3D scenes provides that capability.

"XML in 10 Points" originally by Bert Bos (2001) (available at www.w 3. org /XML/ 1999/XML-in-10-points) provides an excellent overview of XML benefits and potential. An adapted summary follows.

1. *Structured data*. XML is a set of rules for designing text formats to structure data. XML is not a programming language.

2. *Similar to HTML*. Like HTML, XML makes use of tags (words surrounded by angle brackets) and attributes (of the form `name="value"`). HTML defines the functionality of each tag for presentation purposes, but XML is more general and allows for the definition of rules to govern tag and attribute names, relationships, and semantics that may be customized by the cognizant application.

3. *XML is text, but is not meant to be read*. Programs that produce structured data often store that data on disk, using either a binary or text format. One advantage of a text format is that it allows people to look at the data without the program that produced it. Text formats also allow developers to debug applications more easily. Rules for legal XML are strict, and prevent propagation or use of broken XML files. The *draconian parse* rule requires an application to stop and report any errors.

4. *XML is verbose by design*. Because XML is a text format and uses tags to delimit data, XML files are usually larger than comparable binary formats. That was a conscious decision by the designers of XML. The advantages of a text format are evident (see point 3), and the disadvantages can usually be compensated at a different level.

5. *XML is a family of technologies*. Numerous XML-support languages (themselves written in XML) extend the functionality of XML consistently for many uses. XLink describes hyperlinks. XPointer is a syntax for pointing to any specific part of an XML document. The Cascading Stylesheets (CSS) language is applicable to XML as it is to HTML. The Extensible Stylesheet Language for Transformations (XSLT) is used for rearranging, adding and deleting tags and attributes. The DOM is a standard set of string-based function calls for manipulating XML (and HTML) files from a programming language. XML Schemas help developers precisely define the structures of specialty XML-based formats.

6. *XML is new, but not that new*. Development of XML started in 1996 and has been a W3C recommendation since February 1998. Before XML there was Standard Generalized Markup Language (SGML), developed in the early 1980s,

an ISO standard since 1986, and widely used for large documentation projects. The development of HTML started in 1990. The designers of XML simply took the best parts of SGML, guided by the experience of using HTML, and produced something that is no less powerful than SGML and vastly more regular and simple to use.

7. *XML leads HTML to XHTML*. One important XML application is a document format: W3C's XHTML, the successor to HTML. XHTML has many of the same elements as HTML. The syntax has been changed slightly to conform to the rules of XML. A document that is "XML-based" inherits its syntax from XML, which restricts and strengthens it in certain ways.
8. *XML is modular*. XML allows you to define a new document format by combining and reusing other formats. To eliminate name confusion when combining formats, XML provides a namespace mechanism. XML Schema is designed to mirror this support for modularity at the level of defining XML document structures, making it easy to combine two schemas to produce a third schema that covers a merged document structure.
9. *XML is the basis for Semantic Web*. XML provides an unambiguous syntax for W3C's Resource Description Framework (RDF), a language for expressing metadata (that is, information about information). To communicate knowledge, whether in XML/RDF or in plain English, both people and machines need to agree on what words to use. A precisely defined set of words that describes a certain area of life (from "shopping" to "mathematical logic") is called an *ontology*. RDF, ontologies, and the representation of meaning so that computers can help people do work are all topics of the Semantic Web Activity.
10. *XML is license free, platform independent, and well supported*. By choosing XML as the basis for a project, you gain access to a large and growing community of tools (one of which may already do what you need!) and engineers experienced in the technology. Opting for XML is a bit like choosing Structured Query Language (SQL) for databases: You still have to build your own database and your own programs and procedures that manipulate it, but there are many tools available and many people who can help you. Because XML is license free, you can build your own software around it without paying anybody. The large and growing support means that you are not tied to a single vendor. XML is not always the best solution, but it is always worth considering.

As an example case in point, the use of XML provided multiple major benefits in the development of the X3D-Edit authoring tool. Because the X3D tooltips are captured in XML, they are automatically embedded in the interface and are available as separate HTML web pages. Multilingual support means that tooltips are provided in Chinese, English, French, German, Italian, Portuguese and Spanish. The HTML pretty printer uses XSLT to read and redisplay .x3d scenes. Similarly, separate stylesheets can convert .x3d scenes into equivalent ClassicVRML .x3dv or VRML97 .wrl files. The example-scene catalogs are produced automatically by a Java program and a multidocument XSLT stylesheet that can automatically and rapidly produce hundreds of HTML pages. Another XSLT stylesheet even converts Extrusion crossSection fields into 2D plots using Scalable Vector Graphics (SVG). Future work will probably produce X3D scenes from server-based data and make X3D library archives automatically accessible via Web services. Many new opportunities are available because of the XML encoding for X3D.

2.8.1.2. XML design for X3D

A lot of work went into mapping the VRML scene graph into XML, producing the X3D XML encoding, and creating .x3d files. The following design patterns governed the construction of X3D's XML tagset.

- X3D nodes are expressed as XML elements.
- X3D simple-type fields are expressed as XML attributes. Default values can be safely omitted from .x3d model documents.
- DEF and USE names for nodes are captured as attributes and given XML types ID and IDREF respectively. This ensures that only one DEF ID is allowed per scene, and also ensures that all USE IDREF values only refer to an already-defined DEF ID name.
- Parent-child relationships between X3D nodes are captured as corresponding parent-child relationships between XML elements.
- Because contained nodes are themselves X3D fields of type SFNode/MFNode, the name of the field relationship is listed in a containerField attribute. Default containerField values for each node are correct in most cases, so overriding containerField values is infrequent. This approach makes the X3D tagset terser, easier to read, and less prone to error.
- Only X3D simple-type fields with accessType initializeOnly or inputOutput are saved as XML attributes. Other native fields that have accessType inputOnly or outputOnly are transient and only named when used with ROUTE connections. Thus no attributes exist for fields such as isActive, set_value or value_changed.
- Prototype definitions are explicitly declared using ProtoDeclare/ProtoInterface/ProtoBody and ProtoInstance elements and name attributes. New XML elements are not produced for newly defined prototypes. This approach eliminates many potential errors, and ensures strong validation checking of parent-child relationships for all nonprototype X3D nodes in all scenes.
- 'field' elements are used for defining Script and ProtoInterface fields. 'fieldValue' elements are used to provide overriding default values for fields during ProtoInstance creation.
- X3D node types are defined as XML Schema complexType definitions. These match the X3D interface hierarchy, capture strong typing of node relationships, and collect common attributes shared among node types. These features also provide functional

consistency between scene-graph content and the X3D Scene Authoring Interface (SAI) application programming interface (API).

Together these design rules have provided excellent results. The classical scene graph first developed as part of VRML97 is now fully elaborated in XML in ways that make sense for both languages. Furthermore, X3D graphics now become much more compatible with Web-based technologies. New capabilities such as server-side 3D and animation driven by Web services or chat also become feasible. As shown by the rapidly growing number of X3D scenes and software applications, this progress in deploying 3D content as a first-class interactive media type is a significant development for the World Wide Web.

2.8.1.3. XML validation

The original XML Recommendation by the World Wide Web Consortium (W3C) mandates default rules for "well-formedness" of XML documents. These rules also apply to .x3d scenes and include proper syntax for elements (corresponding to X3D nodes) and attributes (corresponding to X3D fields). Thus, even without prior knowledge of X3D constructs, these rules enable XML tools to detect a large number of potential syntax problems in documents.

After well-formedness checking, XML validity provides even stronger forms of error checking for XML documents. Special validation rules are provided to the XML processor that define the vocabulary of elements, attributes, and allowed parent-child relationships. One such set of rules is a DTD, and another is an XML Schema. DTDs provide strong checking of elements but weak checking of attributes, because attribute values are treated as simple text strings. XML Schemas provide strong checking of both elements and attribute values, and are defined using an object-oriented internal design that supports consistent extensibility.

Two primary mechanisms are provided for validating XML scenes: the X3D DTD and the X3D Schema. An X3D scene in the .x3d format, which is an XML document, can include references to either the DTD or the Schema (or both, preferably). Such validation capabilities are provided automatically in many XML-capable applications, making the production and modification of 3D content much less error-prone than in the past.

In addition, two stylesheets written in XML Stylesheet Language for Transformations (XSLT) are provided. The stylesheets convert .x3d files into VRML97 (X3dToVrml97.xslt) or Classic VRML (X3dToX3dvClassicVrml.xslt) encodings. These paired stylesheets also apply many X3D-specific consistency rules to provide extensive correctness checks. The X3D-based quality tests go beyond XML validation to provide hint, warning, and error diagnostic messages about the X3D scene graph, thus providing even more quality control of authored scenes. Detecting errors while authoring is invaluable, because scene-graph problems can otherwise go unnoticed until the scene fails for a user (who deserves a zero-defect result).

Table 1.5 shows which validation checks are provided with each mechanism. X3D-Edit (provided with this book) provides DTD and X3dToVrml stylesheet support. X3D Schema validation is also highly recommended and catches most authoring errors.

Scene graph validity checks	X3D DTD	X3D Schema	X3DToVrml stylesheets	Xj3D, Flux open-source browsers
Legal node (element) names	x	x	x	x
Legal field (attribute) names	x	x	x	x
Correct X3D profile/component level, matching nodes in the scene			x	x
Allowed parent-child relationships between nodes	x	x	x	x
All DEF attribute names are unique and properly named, all USE attribute names exactly match DEF names	x	x	x	x
All USE elements are only referenced after their corresponding DEF elements are defined			x	x
Properly overridden containerField value matches an allowed child-node field name			x	x
Legal field values are provided for attributes that match the defined simple type (SFInt32, MFFloat, etc.)		x	x	x
Mismatched interface names, types, or accessType values among Prototype, Script, field, fieldValue, and IS/connect definitions			x	x
Malformed MFString values for a URL field			x	x
Only ROUTE to/from DEF node names	x	x	x	x
Only ROUTE to/from legitimate target field names				x

URL, Uniform resource locator; VRML, virtual reality modeling language; X3D, Extensible 3D.

Table 1.5. Capabilities Comparison for Validation of X3D Scene Graphs

Browsers (such as Xj3D) often provide excellent debugging feedback. These can be the most authoritative tools to debug the operation of an X3D scene. Unfortunately, most browsers suffer from (at least) two common pitfalls: error messages are often vague or misdirecting, and errors are often not caught until the scene is delivered to the user. Better error checking during scene production is preferred by most X3D authors, to ensure their efforts to build great content aren't rendered useless. It is a good authoring practice to frequently check scenes with a strictly validating browser while creating them.

It is worth considering the big picture in this context. In 1975, Dr. Niclas Wirth wrote a book entitled *Algorithms + Data Structures = Programs*. Since then, several decades of software-engineering theory and effort have been devoted to algorithm-related issues, especially structured and object-oriented programming. However, studies have shown that a surprisingly large amount of development and debugging time is usually spent on reading and verifying (or recovering from errors in) input data. The pertinent expression for this situation is preventing "garbage in, garbage out" (GIGO). Since the original development of SGML and advent of XML, much more work has been devoted to structuring, describing, and strengthening data. This practical emphasis on standards and best practices for structured data, in addition to all the work on structured programming, is an exceptionaly productive trend that continues to provide big payoffs. The importance of data integrity and preventing garbage in, garbage out cannot be overestimated.

Building X3D scenes using the XML-based .x3d encoding and gaining the benefits of strong validation is the best way to author 3D graphics for the Web.

2.8.2. ClassicVRML encoding: .x3dv files

As discussed earlier, VRML has been quite successful since it was first named an international standard in 1997. The ClassicVRML encoding is for the X3D alternative files that use the same "squiggly and square bracket" format used for describing a scene graph. All of the functionality of X3D is fully supported, regardless of which encoding is used to save an X3D file.

The ClassicVRML encoding for X3D matches the syntax for VRML97, with two expected differences: All of the new X3D nodes are also supported, and the first-line header changes as follows.

```
#VRML V2.0 utf8
```

becomes

```
#X3D V3.0 utf-8
```

For the ClassicVRML encoding, comment statements begin with a # character. A node's field values and children nodes are surrounded by curly brackets. Multiple-field (MF) values must be surrounded by square brackets if more than one value is present in the array. Fields with default values (such as a radius of 1 for a Sphere) can be omitted. DEF names precede a node's definition, and USE nodes replace a node's definition. The following example illustrates these rules.

```
# two simple shapes
DEF MyExample Transform {
translation 0 10 0
children [
  Shape {
    Sphere { radius 3 }
  }
```

```
      Shape {
        USE SomeOtherBox # declared earlier in the scene
        }
      ]
    }
```

Some other rules are worth mentioning. Whitespace is generally ignored when not with quotation marks (meaning within SFString and MFString fields). All text after a # comment character is ignored, so subsequent node or field definitions must start on a new line. ProtoDeclare and ExternProtoDeclare constructs are represented by PROTO and EXTERNPROTO, respectively. ProtoInstance has no corresponding term in Classic VRML, instead the name of the prototype is used solely by itself (just like a predefined X3D node). Empty node definitions are represented by NULL. There are no ClassicVRML equivalents for the X3D and head elements in the XML encoding nor is there a corresponding scene element. The XML-based profile, component, and meta elements are respectively represented as PROFILE, COMPONENT, and META statements in the ClassicVRML encoding.

One significant syntax change from VRML97 to X3D nomenclature is field accessType naming, as shown in Table 1.6.

Note that while the names for accessType values changed between VRML97 and X3D, the functionality remains unchanged. One troublesome accessType restriction in VRML97 has been relaxed: Script nodes can now define fields with `accessType="inputOutput"` in X3D.

Maintaining backwards compatibility with preexisting VRML97 content and authoring-tool exporters is an important design requirement for X3D. Because the creation of 3D models can be time consuming and expensive, most authors do not want to re-create or convert work that is already complete. Thus the ClassicVRML encoding ensures that legacy content remains available. Retaining the VRML97 approach as ClassicVRML (with the corresponding addition of new X3D nodes) also helps existing software tools upgrade to X3D capabilities more easily.

The ClassicVRML syntax does have some drawbacks. Any text editor can be used to create such files, but few provide VRML-aware shortcuts or special support.

VRML97 Name	X3D Name	X3D Specification abbreviation
eventIn	inputOnly	[in]
eventOut	outputOnly	[out]
field	initializeOnly	[ ]
exposedField	inputOutput	[in,out]

VRML, Virtual reality modeling language; X3D, Extensible 3D.

Table 1.6. Naming Conventions for accessType Values

Mismatched brackets or unpaired quote marks can be difficult (and sometimes maddening) to debug or isolate. Misspelled node or field names, bad values, and any number of other errors are not caught until run time. The threat of “garbage in” is always lurking. Thus all .x3dv scenes need to be tested as thoroughly as possible by an author before they are distributed to other users.

All of the examples in this book are provided in both .x3d and .x3dv form (as well as a pretty-print .html version of the .x3d form). It is a benefit of using XML source for the master versions that all of the .x3dv versions are produced automatically by converting the .x3d originals, guaranteeing consistency among the different forms of each example. Because it is encoded as XML, the .x3d version has numerous advantages, including ease of translation. Nevertheless, the ClassicVRML syntax is fully functional, precisely specified, and will doubtless remain in use for a long time. Some authors still prefer it.

Many good VRML 2.0 (VRML97) books have been published over the past decade, and the examples and explanations within them are still quite usable, though they are not up to date with all of the new features in X3D. In addition to presenting new nodes in X3D, this book compares both syntaxes while emphasizing the benefits of XML.

2.8.3. Binary encoding: .x3db files

The primary goals for the X3D Binary Encoding are smaller X3D files, faster scene loading at run time, and network streaming of 3D data. Numerous technical requirements are derived from these goals. Applied design efforts have shown that all three of these goals can be accomplished simultaneously. Two basic kinds of size-reduction techniques are used in combination: information-theoretic compression to minimize data duplication (similar to zip/gzip) and geometry-based compression, in which polygons, colors, interpolators, and so on are combined, compressed, and rearranged.

Although some of this work is still undergoing technical improvements, a draft X3D Binary Encoding specification standard is approved by the Web3D Consortium. X3D compression algorithms include the Fast Infoset (FI) standard for XML-based compression of .x3d documents. Both lossless and lossy techniques are provided, allowing authors to either maintain identical results (compared to the original) or produce visually acceptable substitutions. Lossless compression is always the default setting, although some highly detailed scenes benefit from the removal of extraneous polygons or insignificant digits. The primary two technologies currently used are ISO-standard FI data reduction of .x3d XML-based encodings, and (optionally) a large number of patented Java3D geometric-compression algorithms. Sun Microsystems Inc. has provided a royalty-free license for use of the Java3D compression technology to be applied in X3D scenes. Other options are available using XML-based security standards for authentication (digital signature) and encryption. Unlike many other approaches, all of this technology can be used for X3D without licensing fees. Thus X3D use is royalty free.

Figure 1.6 is an example algorithm that shows how all these technologies are - combined to produce .x3db files. Optional steps are outlined using dashed lines. Note that by using the Web design principle of least constraints, implementations may use other algorithms as long as the transformation converting one form to another form produces consistent and interoperable results.

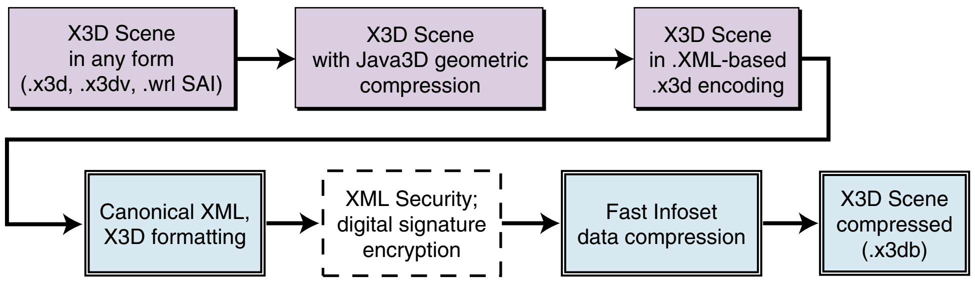

X3D Scene in any form (.x3d, .x3dv, .wrl SAI)
X3D Scene with Java3D geometric compression
X3D Scene in .XML-based .x3d encoding
Canonical XML, X3D formatting
XML Security; digital signature encryption
Fast Infoset data compression
X3D Scene compressed (.x3db)

Figure 1.6. Processing chain to start with any X3D content, apply geometric compression (if desired), add XML security features (if desired), and produce .x3db compressed binary encoding.

Most remarkable about this work is that so many optimizations can be performed in combination. Typical file-size reductions range from 10%–25% of the original, consistently beating gzip reduction. Parsing speedups when reading data at run-time often run 5–10 times faster, which can be a significant improvement for large (multimegabyte) scenes.

The following list of requirements is taken from the original Request for Proposals by the Web3D Consortium to define an X3D binary encoding with compression. It lists the many capabilities achieved.

1. *X3D Compatibility*. The compressed binary encoding shall be able to encode all of the abstract functionality described in X3D Abstract Specification.
2. *Interoperability*. The compressed binary encoding shall contain identical information to the other X3D encodings (XML and ClassicVRML). It shall support an identical round-trip conversion between the X3D encodings.
3. *Multiple, separable data types*. The compressed binary encoding shall support multiple, separable media data types, including all node (element) and field (attribute) types in X3D. In particular, it shall include geometric compression for the following.
   - *Geometry*—polygons and surfaces, including NURBS
   - *Interpolation data*—spline and animation data, including particularly long sequences such as motion capture (also see Streaming requirement)
   - *Textures*—PixelTexture, other texture and multitexture formats (also see Bundling requirement)
   - *Array Datatypes*—arrays of generic and geometric data types
   - *Tokens*—tags, element and attribute descriptors, or field and node textual headers
4. *Processing Performance*. The compressed binary encoding shall be easy and efficient to process in a run-time environment. Outputs must include directly

typed scene-graph data structures, not just strings which might then need another parsing pass. End-to-end processing performance for construction of a scene-graph as in-memory typed data structures (i.e., decompression and deserialization) shall be superior to that offered by gzip and string parsing.

5. *Ease of Implementation.* Binary compression algorithms shall be easy to implement, as demonstrated by the ongoing Web3D requirement for multiple implementations. Two (or more) implementations are needed for eventual advancement, including at least one open-source implementation.

6. *Streaming.* The compressed binary encoding will operate in a variety of network-streaming environments, including http and sockets, at various (high and low) bandwidths. Local file retrieval of such files shall remain feasible and practical.

7. *Authorability.* The compressed binary encoding shall consist of implementable compression and decompression algorithms that may be used during scene-authoring preparation, network delivery and run-time viewing.

8. *Compression.* The compressed binary encoding algorithms will together enable effective compression of diverse datatypes. At a minimum, such algorithms shall support lossless compression. Lossy compression alternatives may also be supported. When compression results are claimed by proposal submitters, both lossless and lossy characteristics must be described and quantified.

9. *Security.* The compressed binary encoding will optionally enable security, content protection, privacy preferences and metadata such as encryption, conditional access, and watermarking. Default solutions are those defined by the W3C Recommendations for XML Encryption and XML Signature.

10. *Bundling.* Mechanisms for bundling multiple files (for example X3D scene, Inlined subscenes, image files, audio file, etc.) into a single archive file will be considered.

11. *Intellectual Property Rights (IPR).* All technology submissions must follow the predeclaration requirements of the Web3D Consortium IPR policy in order to be considered for inclusion. To date, the Web3D Consortium has only accepted royalty-free (RF) technologies for use in Web3D standards.

Streaming requirements include incremental loading of large geometry and interpolator data, incremental modifications for scene behaviors, and scene subgraph replacement to permit progressive improvements in level-of-detail fidelity. Some of these capabilities are already provided in X3D. For example, proper design of related X3D scenes allows an author to use the Inline node to selectively load scenes of increasing complexity. This approach allows a simple viewing experience to be quickly delivered and to interact with the user while additions to the scene continue to download in the background.

Though not yet in widespread use, binary encoding is expected to significantly increase the size, performance, and security of X3D scenes that can be effectively used over the Web. This is an important capability to check for in new X3D products.

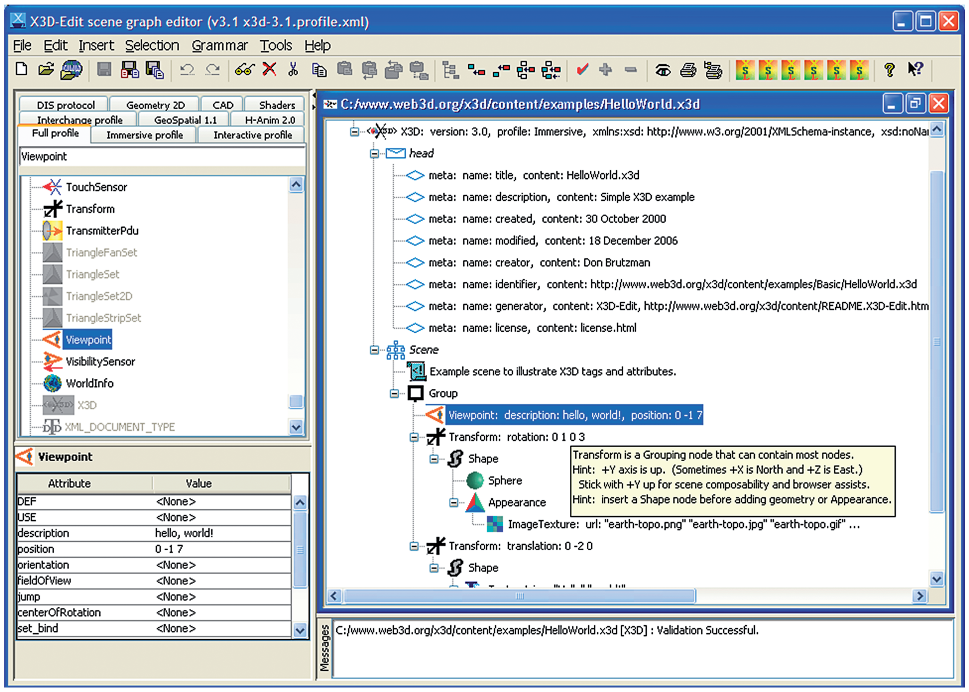
The Xj3D open-source browser provides the primary test implementation of .x3db scenes, with other commercial X3D browsers expected to add this functionality as they continue to build out their supported X3D feature sets.

2.9. Hello World example using X3D-Edit and an X3D browser

A good example can help illustrate the many concepts presented here. Many computer languages are compared by showing a Hello World program that provides a simple example of how the language works. The following example, HelloWorld.x3d, uses the X3D-Edit authoring tool to display the X3D scene graph and produce both .x3d and .x3dv versions.

Figure 1.7 shows the X3D-Edit display. Tables 1.7 and 1.8 show the XML (.x3d) and ClassicVRML (.x3dv) encodings, respectively.

X3D-Edit is available free for any use and included in the online distribution for this book. It is written in Java and runs on all major operating systems. An autoinstaller simplifies initial setup. Features include explanatory popup tooltips for all nodes and fields, DTD-based validation checking, and multilingual versions of tooltips. VRML97 import/export and browser-based, pretty-print HTML display are also provided.



Attribute	Value
DEF	<None>
USE	<None>
description	hello, world!
position	0 -1 7
orientation	<None>
fieldOfView	<None>
jump	<None>
centerOfRotation	<None>
set_bind	<None>

```
X3D: version: 3.0, profile: Immersive, xmlns:xsd: http://www.w3.org/2001/XMLSchema-instance, xsd:noNa
  head
    meta: name: title, content: HelloWorld.x3d
    meta: name: description, content: Simple X3D example
    meta: name: created, content: 30 October 2000
    meta: name: modified, content: 18 December 2006
    meta: name: creator, content: Don Brutzman
    meta: name: identifier, content: http://www.web3d.org/x3d/content/examples/Basic/HelloWorld.x3d
    meta: name: generator, content: X3D-Edit, http://www.web3d.org/x3d/content/README.X3D-Edit.htm
    meta: name: license, content: license.html
  Scene
    Example scene to illustrate X3D tags and attributes.
    Group
      Viewpoint: description: hello, world!, position: 0 -1 7
      Transform: rotation: 0 1 0 3
        Shape
          Sphere
          Appearance
            ImageTexture: url: "earth-topo.png" "earth-topo.jpg" "earth-topo.gif" ...
      Transform: translation: 0 -2 0
        Shape
```

Transform is a Grouping node that can contain most nodes.
Hint: +Y axis is up. (Sometimes +X is North and +Z is East.)
Stick with +Y up for scene composability and browser assists.
Hint: insert a Shape node before adding geometry or Appearance.

```
C:/www.web3d.org/x3d/content/examples/HelloWorld.x3d [X3D] : Validation Successful.
```

Figure 1.7. X3d-Edit is an XML-based Java application for creating and launching X3d scenes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
   "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D profile="Immersive" xmlns:xsd="http://www.w3.org/2001/
   XMLSchema-instance"
   xsd:noNamespaceSchemaLocation="http://www.web3d.
   org/specifications/x3d-3.0.xsd" version="3.0">
  <head>
   <meta name="filename" content="HelloWorld.x3d"/>
   <meta name="description" content="Simple X3D example"/>
   <meta name="url" content="http://www.web3d.org/x3d/
      content/examples/HelloWorld.x3d"/>
   <meta name="generator" content="X3D-Edit,
      http://www.web3d.org/x3d/content/README.X3D-Edit.html"/>
  </head>
  <Scene>
   <!-Example scene to illustrate X3D tags and attributes.—>
   <Group>
   <Viewpoint description="hello, world!" orientation="0 1 0 1.57"
      position="6-1 0"/>
   <NavigationInfo type='"EXAMINE" "ANY"'/>
   <Shape>
     <Sphere/>
     <Appearance>
      <Image Texture url='"earth-topo.png" "earth-topo-small.gif"
        "http://www.web3d.org/x3d/content/examples/earth-topo-
         small.gif" ‘/>
     </Appearance>
     </Shape>
     <Transform rotation="0 1 0 1.57" translation="0 -2 1.25">
     <Shape>
        <Text string='"Hello" "world!"'/>
         <Appearance>
           <Material diffuseColor="0.1 0.5 1"/>
         </Appearance>
        </Shape>
     </Transform>
  </Group>
  </Scene>
</X3D>
```

Table 1.7. Example HelloWorld scene using XML (.x3d) syntax

```
#X3D V3.0 utf8
#X3D-to-ClassicVRML XSL translation autogenerated by X3dToVrml97.xslt
#http://www.web3d.org/x3d/content/X3dToVrml97.xslt
PROFILE Immersive
META "filename" "HelloWorld.x3d"
META "description" "Simple X3D example"
META "url" "http://www.web3d.org/x3d/content/
   examples/HelloWorld.x3d"
META "generator" "X3D-Edit, http://www.web3d.org/x3d/content/README.
   X3D-Edit.html"
# Example scene to illustrate X3D tags and attributes.
Group {
  children [
    Viewpoint {
      description "hello, world!"
      orientation 0 1 0 1.57
      position 6-10
  }
  NavigationInfo {
    type ["EXAMINE" "ANY"]
  }
  Shape {
    geometry Sphere {
  }
    appearance Appearance {
      texture ImageTexture {
      url ["earth-topo.png" "earth-topo-small.gif"
      "http://www.web3d.org/x3d/content/examples/earth-topo-small.gif"]
    }
   }
  }
  Transform {
    rotation 0 1 0 1.57
    translation 0 –2 1.25
    children [
      Shape {
      geometry Text {
        string ["Hello" "world!"]
      }
      appearance Appearance {
```

Table 1.8. Example HelloWorld scene using ClassicVRML (.x3dv) syntax

```
          material Material {
            diffuseColor 0.1 0.5 1
          }
        }
      }
    ]
  }
 ]
}
```

Table 1.8. (Cont'd.) Example HelloWorld scene using ClassicVRML (.x3dv) syntax

The best way to learn X3D is by using it, so now is a good time to install an X3D viewer and X3D authoring tool on your computer. Almost all of the examples in this book were created using X3D-Edit. Loading, modifying, and improving (or breaking!) examples is a great way to make all of these capabilities part of your repertoire.

3. Summary

3.1. Key ideas

X3D has a long history of successful development, starting with the Virtual Reality Modeling Language (VRML) as early as 1994, adding the tremendous capabilities of the Extensible Markup Language (XML), and integrating many additional technologies along the way.

Building scenes is more like creating a Computer-Aided Design (CAD) model or authoring web-page content than programming. Using the .x3d encoding lets XML validation ensure that scene content is free of syntax errors, enabling authors to focus on the 3D models of interest. This is more productive for authors than worrying about how to achieve consistent results in different hardware and software environments.

A lot of technical detail is summarized in this chapter, but it is mostly background information about how X3D actually works. New X3D authors can skip ahead and learn how to create scenes without needing to know the underlying details. Periodic review of this chapter later can be quite helpful while mastering X3D.

3.2. Next chapters

Each of the following chapters explains a different set of nodes that together can be used to make up the X3D scene graph.

Chapter 2 (Geometry Primitives) shows how to construct simple geometric shapes to build a basic X3D scene. Chapter 3 (Grouping Nodes) shows how to group different shapes together and move them to different locations in virtual space.

Reference

Bos, Bert, and W3C Communications Team, *XML in 10 points*, web page, November 2001. Available at www.w3.org/XML/1999/XML-in-10-points.