COLLADA – Digital Asset Schema Release 1.3.1

Specification

August 2005

Publication date: August 2005

Sony Computer Entertainment Inc.
2-6-21 Minami-Aoyama, Minato-ku,
Tokyo 107-0062 Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

Table of Contents

About This Manual

The U.S. Research & Development department of Sony Computer Entertainment is working on software to enhance and facilitate the deployment of PlayStation® products.

For more information on SCE US R&D, please consult www.research.scea.com.

For more information on PlayStation®, please consult www.playstation.com.

This document describes the COLLADA schema specifications. The file extension chosen for this format is "**`.dae`**" (an acronym for Digital Asset Exchange). When an Internet search was completed, no preexisting usage was found.

The COLLADA schema is the result of a research project within Sony Computer Entertainment US Research & Development (SCE US R&D). COLLADA is a COLLAborative Design Activity lead by SCE US R&D that involves designers, developers, and interested parties from within Sony Computer Entertainment as well as third party companies such as Alias Systems Corporation, Avid Technology Inc., Autodesk Inc., 3Dlabs Inc., NVIDIA Corporation and ATI Technologies Inc.

For more information please visit www.collada.org. You can also contact us by e-mail at collada@collada.org.

The purpose of this document is to provide a specification for the COLLADA schema in sufficient detail as to enable software developers to create tools to process COLLADA resources. In particular, it is relevant to importers and exporters to digital content creation (DCC) applications, 3D interactive applications and tool chain, prototyping tools, real-time visualization application such as used in the video game and the movie industry.

This document covers the initial design and specifications of the COLLADA schema, as well as a minimal set of requirements for COLLADA exporters. A short example: a COLLADA instance document is presented in Appendix A.

Changes Since Last Release

Date	Author	Comments
15-August-2005	SCEA IDD	Added remaining reviewer content.
10-August-2005	SCEA IDD	Added new reviewer content; updated release number to 1.3.1 throughout.
4-May-2005	SCEA IDD	Made final formatting refinements; converted to .pdf
2-May-2005	SCEA IDD	Added new reviewer content. Created .doc proof for MBarnes' review.
29-Apr-2005	Mark C. Barnes	Gathered more edits and changes; provided to IDD.
22-Apr-2005	SCEA IDD	Implemented and formatted reviewer content. Checked all cross-references and links.
21-Apr-2005	Mark C. Barnes	Gathered edits and changes from reviewers; provided to IDD.
12-Apr-2005	Mark C. Barnes	Edits and changes from more bug reports. Merged in changes from Pip Stuart.
8-Apr-2005	Mark C. Barnes	Reviewed IDD changes; Corrected merge errata.
4-Apr-2005	SCEA IDD	Incorporated new content; updated the Index and About This Manual sections.
30-Mar-2005	Mark C. Barnes	Finished outstanding bug reports. Included changes from Rémi Arnaud.
29-Mar-2005	Mark C. Barnes	Added Chapter 5 from Pip Stuart
28-Mar-2005	Mark C. Barnes	Drafting 1.3.0 changes.

COLLADA Documentation

This document consists of the following chapters:

Chapter/Section	Description
Ch. 1: Design Considerations	Description of issues concerning the design
Ch. 2: Schema Overview	A general description of the schema and its design
Ch. 3: Schema Reference	A reference description of the schema in greater detail
Ch. 4: Common Profile	A description of the COLLADA COMMON profile
Ch. 5: Tool Requirements and Options	A description of COLLADA tool requirements for implementors
Appendix A: Cube Example	An example COLLADA instance document
Glossary	Definitions of terms used in this document
Index	

Other Sources of Information

Resources that serve as reference background material for this document include:

- Extensible Markup Language (XML) 1.0, 2nd Edition
- XML Schema
- XML Base
- XML Path Language
- XML Pointer Language Framework
- Extensible 3D (X3D™) encodings ISO/IEC FCD 19776-1:200x
- Softimage® dotXSI™ FTK
- NVIDIA® Cg Toolkit
- Pixar's RenderMan®

Audience

This document is public and available to anybody. The intended audience is programmers that want to create applications, or plug-ins for applications, that can utilize the COLLADA schema.

Prerequisites

Readers of this document should:

- Have knowledge of XML, XML Schema.
- Be familiar with shading languages such as NVIDIA® Cg or Pixar RenderMan®.
- Have a general knowledge and understanding of computer graphics and graphics API such as OpenGL®.

Typographic Conventions

Certain Typographic Conventions are used throughout this manual to clarify the meaning of the text:

Conventions	Description
Helvetica 45	Indicates descriptive text
`Courier New`	Indicates references to class, method and variable names
`Courier New bold`	Indicates file names
`> Courier New bold`	Indicates commands, which are preceded with a right angle bracket (>)
blue	Indicates a hyperlink

Developer Support

Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	E-mail: scea_support@ps2-pro.com Web: https://www.ps2-pro.com/ Developer Support Hotline: (650) 655-5566 (Call Monday through Friday, 8 a.m. to 5 p.m., PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees only in the PAL television territories (including Europe and Australia). You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
Attn: Development Tools Manager Sony Computer Entertainment Europe 13 Great Marlborough Street London W1F 7HP, U.K. Tel: +44 (0) 20 7859-5000	E-mail: scee_support@ps2-pro.com Web: https://www.ps2-pro.com/ Developer Support Hotline: +44 (0) 20 7911-7711 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT/BST)

This page intentionally left blank.

Chapter 1:
Design Considerations

This page intentionally left blank.

Introduction

Development of the COLLADA Digital Asset schema involves designers and software engineers from many companies in a collaborative design activity. In this chapter we review the more important design goals, thoughts and assumptions made by the designers.

Assumptions and Dependencies

During the course of the first phase of the design of the COLLADA Asset Schema, the contributors did a lot of discussion and necessary agreement. In particular, the following:

First of all, this is not a game engine format. We assume that COLLADA will be beneficial to authoring tool users and interactive application content creation pipelines. We assume that most interactive applications will use COLLADA in the production pipeline, but not as a final delivery mechanism. For example most games will use proprietary size optimized binary files.

We assume that the end users will want to quickly develop and test relatively simple content and test models that still include advanced rendering techniques such as vertex and pixel programs (shaders).

Additionally we assume that end users will be using the Microsoft Windows® and Linux® operating systems in their production environments. In the case of end user developers, we assume that they will program in the C/C++ languages, predominantly. This is what we will be focusing on for COLLADA example source code.

Goals and Guidelines

Design goals for the COLLADA Digital Asset schema include the following:

- To liberate digital assets from proprietary binary formats into a well-specified XML based open-source format.
- To provide a standard common format so that COLLADA assets can be used directly in existing content tool-chains, and facilitate this integration.
- To be adopted by as many digital content users as possible.
- To provide an easy integration mechanism that enables all the data to be available through COLLADA.
- To be a basis for common data exchange between 3D applications.
- To be a catalyst for digital asset schema design among developers and DCC, hardware, and middleware vendors.

Explanation, consequences and rational of goals:

- To liberate digital assets from proprietary binary formats.

  Digital Assets are the largest part of most 3D applications today.

  Developers have enormous investment in assets that are stored in opaque proprietary formats. Exporting the data out of the tool requires considerable investment to develop software for proprietary complex Software Development Kits. Even once this investment has been made, it is still impossible to modify the data outside of the tool and import it back later. It is necessary to permanently update the exporters with the ever-evolving tools, to the risk of seeing the data become obsolete.

  Hardware vendors need increasingly more complex assets to take advantage of new hardware. The data needed may exist inside a tool, but there is often no way to export this data from the tool.

Or exporting this data is a complex process that is a barrier for developers to be able to use advanced features, and a problem for hardware vendors to promote new products.

Middleware and tool vendors have to integrate with every tool chain to be able to be used by developers, which is an impossible mission. Successful middleware vendors have to provide their own extensible tool chain and framework, and have to convince developers to adopt it. That makes it impossible for game developers to use several middleware tools in the same project, just like it is difficult to use several DCC tools in the same project.

There are several decisions that have been arrived at from this goal such as:

- COLLADA will use XML.
  - XML provides a very well defined framework. Issues like character sets (ASCII, Unicode, shift-jis) are already covered by the XML standard making any schema that uses XML instantly internationally useful. XML is also fairly easy to understand given a sample instance document only and no documentation, something that is rarely true for other formats. There are XML parsers for nearly every language on every platform making the files easily accessible to almost any possible application.
- COLLADA will not use binary data inside XML.
  - Some discussion is often brought up around the topic of storing vertices and animation data in some kind of binary representation for ease of loading, speed and asset size. Unfortunately that goes counter to the object of being useful to the most number of teams as many languages do not easily support binary data inside XML files nor do they support manipulation of binary data in general. Keeping COLLADA a 100% text based representational format supports the most options. COLLADA does provide mechanisms to store external binary data and reference it from a COLLADA asset.
- The COLLADA common profile will expand over time to include as much common data as possible.
  - COLLADA currently supports polygon-based models in a common format. As we discuss new issues COLLADA will attempt to cover other areas as well. Shader effects, physics, parametric surfaces and so on.

- To provide a standard common format so that COLLADA assets can be used directly in existing content tool-chains, and facilitate this integration.
  - This goal is what gives us the “COMMON” profile. The goal is that if your tools can read a COLLADA asset and use the data presented by the common profile you should be able to use any DCC tool for content creation.
  - To facilitate the integration of COLLADA assets into tool chains we have come to the conclusion that we need to provide not only a schema and a specification, but also a well designed API (the COLLADA API) that helps integrate COLLADA assets in existing tool-chains. This new axis of development can open numerous new possibilities, as well as provide a substantial saving for developers. Its design has to be such as to facilitate its integration to specific data structure used by existing content tool-chains.
  - Development of numerous tools that can be organized in a tool-chain to create a professional content pipeline. COLLADA will facilitate the design of a large number of specialized tools, rather than a monolithic hard-to-maintain tool chain. Better reuse of tools developed internally, or externally will provide economic and technical advantages to developers and tools/middleware providers, and therefore strengthen COLLADA as a standard Digital Asset Exchange format.
- To be adopted by as many digital content users as possible.

- In order to be adopted, COLLADA needs to be useful to developers. For a developer to measure the utility of COLLADA to their problem, we need to provide them with the right information, and enable the measurement of the quality of COLLADA tools.
  - Provide a conformance test suite to measure the level of conformance and quality of tools.
  - Provide a list of requirements in the specification for the tool providers to follow in order to be useful to most developers. (These goals are specified in “Tool Requirements and Options”).
  - Collect feedback from users and add it to the requirements and conformance test suite.
  - Manage bug reporting problems and implementation questions to the public. This involves prioritizing bugs and scheduling fixes among the COLLADA partners.
  - Facilitate asset exchange and asset management solutions.
  - Engage DCC tool and middleware vendors to directly support COLLADA exporters, importers and other tools.
- Game developers win because they can now use every package in their pipeline.
- Tool vendors win because they have the opportunity to reach more users.
- Provide a command line interface to DCC tool exporters and imports so that those tasks can be incorporated into an automated build process.

- To provide an easy integration mechanism that enables all the data to be available through COLLADA.
  - COLLADA is fully extensible so it is possible for developers to adapt COLLADA to their specific needs.
  - Design the COLLADA API and the future designs of COLLADA to ease the extension process by making full use of XML schema capabilities and rapid code generation.
  - Encourage DCC vendors to make exporters and importers that can be easily extended.
  - If developers need functionality that is not yet ready to be in the COMMON profile, encourage vendors to add this functionality as a vendor-specific extension to their exporters and importers.
    - This applies to tools-specific information, such as undo stack, or to concepts that are still int he consideration for inclusion in COLLADA, but that are urgently needed, such as complex shaders.
  - Collect this information and lead the group to solve the problem in the COMMON profile for the next version of COLLADA.
  - Make COLLADA asset management friendly
    - For example, select a part of the data in a DCC tool and export it as a specific asset.
    - Enable asset identification and have the correct meta data.
    - Enforce the asset meta data usage in exporters and importers.
- To be a basis for common data exchange between 3D packages.
  - The biggest consequence of this goal is that the COLLADA common profile will be an ongoing exercise. At the point of this writing it covers polygon-based models, materials and shaders, some animations and DAG based scene graphs. In the future it will hopefully cover NURBS, subdivision surfaces and other more complex data types in a common way that makes exchanging that information between tools a possibility.

- To be a catalyst for digital asset schema design among developers and DCC, hardware, and middleware vendors.
    - There is a fierce competition between market segments: the DCC vendors, the hardware vendors, the middleware vendors, and between game developers. But all of them need to communicate to solve the digital content problems. Not being able to collaborate on a common Digital Asset format has a direct impact on the overall optimization of the market solutions.
        - Hardware vendors are suffering from the lack of features exposed by DCC tools.
        - Middleware vendors suffer because they lack compatibility between the tool chains.
        - DCC vendors suffer from the amount of support and specific development required to make developers happy.
        - Developers by the huge amount of investment necessary to create a working tool-chain.
    - None of the actors can lead the design of a common format, without being seen by the others as factoring a commercial or technical advantage into the design. No one can provide the goals that will make everybody happy. But it is necessary to have everybody's acceptance of the format. It is necessary for all major actors to be happy with the design of this format for it to have wide adoption, and be accepted.
        - Sony Computer Entertainment (SCE), because of its leadership in the videogame industry, was the right catalyst to make this collaboration happen. SCE has a history of neutrality toward tool vendors and game developers in its middleware and developer programs, and can bring this to the table, as well as its desire to raise the bar of quality and quantity of content for the next generation platforms.
    - The goal is not for SCE to always drive this effort, but to delegate completely this leadership role to the rest of the group when the time becomes appropriate.
        - Doing this too early will have the negative effect of partners who will feel that SCE is abandoning COLLADA.
        - Doing this too late will prevent more external involvement and long term investment from companies concerned that SCE has too much control over COLLADA.

Development Methods

The development approach and methodologies used to develop the COLLADA schema include the standard waterfall process of analysis, design, and then implementation. The analysis phase has included a fair amount of comparative analysis of current industry tools and formats.

During the design phase, the Microsoft Visual Studio® XML Designer tool is being used to iteratively develop and validate the schema for the format. In addition, we have been using XMLSPY® from ALTOVA GmbH. to validate files against the COLLADA schema and create drawings for the purpose of documentation.

Chapter 2: Schema Overview

This page intentionally left blank.

Introduction

The COLLADA schema is an XML database schema. The XML Schema language is used to describe the COLLADA feature set.

Concepts

XML

XML is the Extensible Markup Language. XML provides a standard language to describe the structure and semantics of documents, files or data sets. XML itself is a structure language consisting of elements, attributes, comments and text data.

Elements

An XML document consists primarily of elements. An element is a block of information that is bounded by tags at the beginning and end of the block. Elements can be nested producing a hierarchical data set if so desired.

Tags

Each XML element begins with a start tag. The syntax of a start tag includes a name surrounded by angle brackets as follows:

```
<tagName>
```

Each XML element ends with an end tag. The syntax of an end tag is as follows:

```
</tagName>
```

Between the start and end tags is an arbitrary block of information.

Attribute

XML elements can have attributes. Attributes are given within the start tag and follow the tag name itself. For example:

```
<tagName attribute=”value”>
```

Attributes

An XML element can have zero or more attributes. Each attribute is a name-value pair. The value portion of an attribute is always surrounded by quotes (“”). Attributes provide semantic information about the element they are bound on.

Name

The name of an attribute generally has some semantic meaning in relation to the element that it belongs to. For example:

```
<Array size=”5” type=”xs:float”>
  1.0 2.0 3.0 4.0 5.0
</Array>
```

Here we have an element named Array with two attributes, size and type. The size attribute tells us how large the array is and the type attribute tells us that the array contains floating-point data.

Value

The value of an attribute is always textual data during parsing.

Comments

XML files can contain comment text. Comments are identified by special markup of the form show below.

```
<!-- This is a XML comment -->
```

XML Schema

The XML Schema language provides the means to describe the structure of a family of XML documents that follow the same rules for syntax, structure and semantics. XML Schema is itself written in XML, making it simpler to use when designing other XML base formats.

Validation

XML by itself does not describe any one document structure or schema. XML provides a mechanism by which an XML document can be validated. The target or instance document provides a link to schema document. Using the schema document, an XML parser can validate the instance document’s syntax and semantics by the rules given in the schema document. This process is called validation.

Core Categories

There are many ways to design database schemas using XML. One approach is to create elements that represent specific constructs. For example:

“Specific Construct” Example 1

```
<polymesh>A specific type of mesh data</polymesh>
```

Another is to create elements that represent categories within the problem domain. Within the category, more specific information can be provided as a combination of attributes and child elements. For example:

“Categorical” Example 2

```
<geometry>
 <!-- This is geometry of some kind -->
 <polymesh>Its polygon mesh data</polymesh>
</geometry>
```

OR

“Categorical” Example 3

```
<geometry>
 <!-- This is geometry of some kind -->
 <mesh type=polygon>Its polygon mesh data</mesh>
</geometry>
```

OR

“Categorical” Example 4

```
<geometry type=polymesh>
 <!-- This geometry is polygon mesh data -->
</geometry>
```

The value of the categorical approach is subtle. If a tool is parsing the first (concrete) example looking for geometry, and does not know about the `<polymesh>` tag, it can skip the tag and do nothing. The same tool can parse either categorical example 2, 3 or 4 and determine that there is geometry that it should consider. But, since it does not recognize the polygon mesh type, it can issue a diagnostic message.

The COLLADA schema defines a core set of categorical elements that cover the problem domain of computer game development needs, focusing on graphics. The purpose of the categories is:

- To identify coarse sections of data to the processing tools, allowing them to quickly recognize data of interest;
- To provide a basis for future extensions, without introducing undue complexity and confusion to tools; (Adding new elements to one category will not affect a tool that processes data in a separate category.)
- To represent the distinct contributions from different participants in the content creation pipeline such as texture artist, modeler, animator, and level designer. The categories define a first level of document modularity within the schema.

The categorical elements are:

Categorical Element	Description
animation	The base category for animation information
camera	The base category for camera information
controller	The base category for controls
geometry	The base category for geometric information
image	The base category for image information

`library`	The base category for content modularity
`light`	The base category for lights
`material`	The base category for material information
`scene`	The base category for scene graph information
`texture`	The base category for texture information

Address Syntax

COLLADA uses two mechanisms to address elements and values within an instance document, as follows:

- The *url* and *source* attributes of many elements use the URI addressing scheme that locates instance documents and elements within them by their *id* attributes.
- The *target* attributes of the animation elements use a COLLADA defined addressing scheme of *id* and *sid* attributes to locate elements within an instance document. This can be appended with C/C++ style structure member selection syntax to address element values.

The *id* attributes are addressed using the URI fragment identifier notation. The XML specification defines the syntax for a URI fragment identifier within an XML document. The URI fragment identifier must conform to the XPointer syntax. As COLLADA only addresses unique identifiers with URI, the XPointer syntax used is called the shorthand pointer. A shorthand pointer is the value of the *id* attribute of an element in the instance document.

In a *url* or *source* attribute, the URI fragment identifier is preceeded with the hash (“#”) character. In a *target* attribute there is no hash character as it is not a URI. For example the same **`<source>`** element is address as follows using each notation:

```
<source id=”here” />
<input source=”#here” />
<skin target=”here” />
```

The *target* attribute syntax is formed of several parts:

- The first part is the ID of an element in the instance document.
- Then follows one or more sub-identifiers. Each is preceded by a literal “/” character as path separator. The sub-identifiers are taken from a child of the element identified by the first part. For nested elements, multiple sub-identifiers may be used to identify the path to the targeted element.
- The final part is optional. If this part is absent, all member values of the target element are targeted (e.g. all values of a matrix). If this part is present it may take one of two forms:
  - The name of the member value (field) indicating symbolic access. This notation consists of:
    - A literal “.” character indicating member selection access.
    - The symbolic name of the member value (field). The Common Glossary in chapter 4 documents values for this field under the common profile.
  - The cardinal position of the member value (field) indicating array access. This notation consists of:
    - A literal “(“ character indicating array selection access.
    - A number of the field, starting at zero (for the first field).
    - A literal “)” character closing the expression.

The array access syntax can be used to express fields in one-dimensional vectors and two-dimensional matrices only.

Here are some examples of the *target* attribute syntax:

```
<channel target="here/trans.X" />
<channel target="here/trans.Y" />
<channel target="here/trans.Z" />
<channel target="here/rot.ANGLE" />
<channel target="here/rot(3)" />
<channel target="here/mat(3)(2)" />

<node id="here">
  <translate sid="trans"> 1.0 2.0 3.0 </translate>
  <rotate sid="rot"> 1.0 2.0 3.0 4.0 </rotate>
  <matrix sid="mat">
    1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0
    9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0
  </matrix>
</node>
```

Each **`<channel>`** element targets one component of the **`<translate>`** element's member values denoted by X, Y, and Z. Likewise the **`<rotate>`** element's ANGLE member is targeted twice using symbolic and array syntax respectively.

For increased flexibility and concision, the target addressing mechanism allows for "skipping" XML elements. It is not necessary to assign id and/or sid attributes to all "in-between" elements.

For example, you can target the Diffuse color of a material/shader without adding sid attributes for **`<shader>`**, **`<technique>`**, **`<pass>`** and **`<program>`**. Actually, some of these elements don't even allow for **`id`** and **`sid`** attributes.

It is also possible to target the Diffuse color of that material in multiple techniques without the need to create extra animation channels for each targeted technique (techniques are "switches": One or the other is picked on import, but not both, so it still resolves to a single target).

For example:

```
<channel source="#RedDiffuseSampler" target="RedPlastic/Diff" />
...
<material id="RedPlastic">
  <shader>
    <technique profile="COMMON">
      <pass>
        <program url="PHONG">
          <param sid="Diff" name="DIFFUSE" type="float3"> 1 1 1 </param>
        </program>
      </pass>
    </technique>

    <technique profile="Cg">
      <pass>
        <program url="#MyCgPhong">
          <param sid="Diff" name="Diffuse" type="float3"> 1 1 1 </param>
        </program>
      </pass>
    </technique>
  </shader>
```

```
    </material>
```

Notice that the same **`sid="Diff"`** attribute is used even though the name of the parameter is different in each technique ("DIFFUSE" in the COMMON technique versus **`"Diffuse"`** under the Cg technique). This is valid to do.

Without allowing for "skipping", targeting elements would be a brittle mechanism and require long attributes and potentially many extra animation channels.

Of course you may still use separate animation channels if the targeted parameters under different techniques require different values.

This page intentionally left blank.

Chapter 3: Schema Reference

This page intentionally left blank

Introduction

In this chapter, the COLLADA schema syntax is described feature by feature. Each XML element in the schema has the following sections:

Section	Description
Introduction	Name and purpose of the element
Concepts	Background and rationale for the element
Attributes	Attributes applicable to the element defined by the schema
Elements	Element constraints and relationships defined by the schema
Remarks	Information concerning the usage of the element
Example	Example usage of the element

COLLADA

Introduction

The COLLADA schema is XML based; therefore, it must have a “document root element” or document entity to be a well-formed XML document.

Concepts

The <**`COLLADA`**> element declares the root of the document that comprises some of the content in the COLLADA schema.

Attributes

The <**`COLLADA`**> element has the following attributes:

`version`	**`xs:string`**

The **`version`** attribute is the COLLADA schema revision with which the instance document conforms. Required.

Related Elements

The <**`COLLADA`**> element relates to the following elements:

Occurrences	One time
Parent elements	No parent elements
Child elements	**`asset, library, scene`**
Other	**`xml:base`**, **`xmlns`**

Remarks

The <**`COLLADA`**> element is the document entity (root element) in a COLLADA instance document.

The document root must contain one **`<asset>`** element.

The document root may contain zero or more **`<library>`** elements.

The document root may contain zero or one **`<scene>`** element.

These child elements must occur in the following order if present:

1. The **`<asset>`** element.
2. The **`<library>`** elements.
3. The **`<scene>`** element.

The XML Schema namespace attribute **`xmlns`** applies to this element to identify the schema for an instance document.

The XML Base attribute **`xml:base`** applies to this element to identify the base URI for an instance document.

Example

The following example outlines an empty COLLADA instance document whose schema version is “1.1.0”.

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2004/COLLADASchema" version="1.1.0">
  <asset />
  <library />
  <scene />
</COLLADA>
```

scene

The scene embodies the entire set of information that can be visualized from the contents of a COLLADA resource.

Introduction

The `<scene>` element declares the base of the scene hierarchy or scene graph. The scene contains elements that comprise much of the visual and transformational information content as created by the authoring tools.

Concepts

The hierarchical structure of the scene is organized into a scene graph. A scene graph is a directed acyclic graph (DAG) or tree data structure that contains nodes of visual information and related data. The structure of the scene graph contributes to optimal processing and rendering of the data and is therefore widely used in the computer graphics domain.

Attributes

The `<scene>` element has the following attributes:

id	**xs:ID**
name	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the `<scene>` element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is a text string containing the name of the `<scene>` element. Optional attribute.

Related Elements

The `<scene>` element relates to the following elements:

Occurrences	One time
Parent elements	**COLLADA**
Child elements	**lookat**, **matrix**, **perspective**, **rotate**, **scale**, **skew**, **translate**, **boundingbox**, **node**, **extra**
Other	None

Remarks

There is one `<scene>` element declared under the `<COLLADA>` document (root) element. The scene graph is built from the `<scene>` element and its child elements.

The `<scene>` element may contain any number of permissible child elements. The `<scene>` elements form the topology of the graph.

The **<scene>** element forms the root of the scene graph topology. As such, it can have many of the same child elements as the **<scene>** element.

Element	Description
lookat	Allows the scene to express a lookat transform
matrix	Allows the scene to express a matrix transform
perspective	Allows the scene to express a perspective transform
rotate	Allows the scene to express a rotational transform
scale	Allows the scene to express a scale transform
skew	Allows the scene to express a skew transform
translate	Allows the scene to express a translation transform
boundingbox	Allows the scene to express a bounding box
node	Allows the scene to define hierarchy
extra	Allows the scene to define extra information

The **<scene>** element represents a context in which the child transform elements are composed in the order that they occur. All the other child elements are affected equally by the accumulated transform in the scope of the **<scene>** element.

The transform elements transform the coordinate system of the **<scene>** element. Mathematically, this means that the transform elements are converted to matrices and post-multiplied in the order they are specified to compose the coordinate system.

Example

The following example shows a simple outline of a COLLADA resource containing a **<scene>** element with no child elements. The name of the scene is “world”.

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2004/COLLADASchema.xsd" version="1.1.0">
  <scene name=”world” id=”world01” >
  </scene>
</COLLADA>
```

node

Nodes embody the hierarchical relationship of elements in the scene.

Introduction

The **<node>** element declares a point of interest in the scene. A node denotes one point on a branch of the scene graph. The **<node>** element is essentially the root of a sub graph of the entire scene graph.

Concepts

Within the scene graph abstraction, there are arcs and nodes. Nodes are points of information within the graph. Arcs connect nodes to other nodes. Nodes are further distinguished as interior (branch) nodes and exterior (leaf) nodes. We use the term “node” to denote interior nodes. Arcs are also called paths.

Attributes

The **<node>** element has the following attributes:

id	**xs:ID**
name	**xs:NCName**
type	One of: **JOINT, NODE**

The **id** attribute is a text string containing the unique identifier of the **<node>** element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is a text string containing the name of the **<node>** element. Optional attribute.

The **type** attribute indicates the type of the **<node>** element. The default value is “NODE”. Optional attribute.

Related Elements

The **<node>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**node, scene**
Child elements	**lookat**, **matrix**, **perspective**, **rotate**, **scale**, **skew**, **translate**, **boundingbox**, **instance**, **node**, **extra**
Other	None

Remarks

The **<node>** elements form the basis of the scene graph topology. As such they can have a wide range of child elements, including **<node>** elements themselves.

Element	Description
lookat	Allows the node to express a lookat transform
matrix	Allows the node to express a matrix transform
perspective	Allows the node to express a perspective transform
rotate	Allows the node to express a rotational transform
scale	Allows the node to express a scale transform
skew	Allows the node to express a skew transform

Element	Description
`translate`	Allows the node to express a translational transform
`boundingbox`	Allows the node to express a bounding box
`instance`	Allows the node to instantiate another copy of an object
`node`	Allows the node to recursively define hierarchy
`extra`	Allows the node to define extra information

The **`<node>`** element represents a context in which the child transform elements are composed in the order that they occur. All the other child elements are affected equally by the accumulated transform in the scope of the **`<node>`** element.

The transform elements transform the coordinate system of the **`<node>`** element. Mathematically, this means that the transform elements are converted to matrices and post-multiplied in the order they are specified to compose the coordinate system.

Example

The following example shows a simple outline of a **`<scene>`** element with two **`<node>`** elements. The names of the two nodes are “earth” and “sky” respectively.

```
<scene name=”world”>
  <node name=”earth”>
  </node>
  <node name=”sky”>
  </node>
</scene>
```

boundingbox

Introduction

The **`<boundingbox>`** element declares an axially aligned bounding box that encompasses the geometric extent of a node in the scene graph.

Concepts

The geometric extent of a scene graph is often a time consuming calculation. This calculation can be computed and saved as a bounding volume. Bounding volumes can be any geometric shape. Simpler shapes like boxes and spheres are common because they take less computing time to calculate.

Algorithms that process the scene graph can use the bounding volume information to accelerate some operations like field-of-view culling and intersection testing. Simpler shapes enable quicker but less accurate results because the shapes don't conform tightly to the actual geometry in the sub graph. More complex shapes like cylinders and polytopes offer greater accuracy but require more computation.

Attributes

The **`<boundingbox>`** element has the following attributes:

`sid`	**`xs:NCName`**

The **`sid`** attribute is a text string containing the sub-identifier of the **`<boundingbox>`** element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

Occurrences	Number of elements defined in the schema
Parent elements	**`node, scene`**
Child elements	**`min, max`**
Other	None

Remarks

The **`<boundingbox>`** element contains additional elements that describe extents of the box in the local coordinate system. These elements are as follows:

- The **`<min>`** element contains three floating-point values for the minimum corner of the box.
- The **`<max>`** element contains three floating-point values for the maximum corner of the box.

The sequence of child elements must be in the order: **`min, max`**.

If the **`<boundingbox>`** is transformed by **`<rotate>`** elements then the resulting bounding box is an axial aligned box that encompasses the original bounding box.

Example

Here is an example of a **`<boundingbox>`** element that encompasses the geometry of a **`<node>`** element.

```
<node>
  <boundingbox>
  <min>-10.0 –10.0 –10.0</min>
  <max>-5.0 10.0 20.0</max>
  </boundingbox>
  <instance url=”#some_geometry” />
</node>
```

camera

Cameras embody the eye point of the viewer looking into the scene.

Introduction

The `<camera>` element declares a view into the scene hierarchy or scene graph. The camera contains elements that describe the camera's optics and imager.

Concepts

A camera is a device that captures visual images of a scene. A camera has a position and orientation in the scene. This is the viewpoint of the camera as seen by the camera's optics or lens.

The camera optics focuses the incoming light into an image. The image is focused onto the plane of the camera's imager or film. The imager records the resulting image.

Attributes

The `<camera>` element has the following attributes:

id	**xs:ID**
name	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the `<camera>` element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

Related Elements

The `<camera>` element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**library**
Child elements	**technique**
Other	None

As a child of the `<camera>` element, the `<technique>` element is restricted to having only **optics** and **imager** child elements.

Remarks

The `<camera>` element contains one or more `<technique>` elements that describe its operating parameters for optics and imaging.

The camera technique element must contain an **optics** element and zero or more **imager** elements. The **optics** and **imager** elements each contain a `<program>` element.

For simple cameras, a generic technique need only contain an **optics** element that describes the field of view and viewing frustum using canonical parameters.

Example

Here is an example of a **<camera>** element that describes a perspective view of the scene with a 45-degree field of view.

```
<camera name=”eyepoint”>
  <technique profile="COMMON">
    <optics>
      <program url="PERSPECTIVE">
        <param name="YFOV" type="float" flow="IN">45.0</param>
          <param name="ZNEAR" type="float" flow="IN">0.1</param>
          <param name="ZFAR" type="float" flow="IN">32767.0</param>
      </program>
    </optics>
    <imager>
  <program />
  </imager>
  </technique>
</camera>
```

optics

Introduction

Optics represents the apparatus on a camera that projects the image onto the image sensor.

Concepts

Optics are composed of one or more optical elements. Optical elements are usually categorized by how they alter the path of light:

- Reflective elements – for example, mirrors (e.g., the concave primary mirror in a Newtonian telescope, or a chrome ball, used to capture environment maps).
- Refractive elements – lenses, prisms.

A particular camera optics might have a complex combination of the above. For example, a Schmidh telescope contains both a concave lens and a concave primary mirror and lenses in the eye-piece.

A variable focal-length “zoom lens” might, in reality, contain more than 10 lenses and a variable aperture (iris).

The commonly used “perspective” camera model in computer graphics is a simple approximation of a “zoom lens” with an infinitely small aperture and the field-of view specified directly (instead of its related value, the focal length).

To allow for representing the large variety of optics, the **`<optics>`** element supports multiple, profile-specific representations via **`<technique>`** elements.

The **`<technique>`** contains a **`<program>`** that describes the optics.

Related Elements

The **`<optics>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`camera`**
Child elements	**`technique`**
Other	None

Remarks

The COMMON profile defines the following optics programs:

- PERSPECTIVE – standard perspective projection, specified via a field-of-view and near and far Z-clipping planes.

  PERSPECTIVE optics have the following parameters:

Name	Type	Description	Default value
XFOV	float	Field of view in the X direction in degrees	N/A *
YFOV	float	Field of view in the Y direction in degrees	N/A *
ZNEAR	float	the “near” clpping plane depth value	N/A
ZFAR	float	the “far” clpping plane depth value	N/A

* The field of view may be specified separately for the X and Y directions. If only one is specified, the other should be derived using the aspect ratio of the rendering view-port (to avoid image distortion).

- ORTHOGRAPHIC – “parallel” projection, specified via 4 corners:

Name	Type	Description	Default value
LEFT	float	Distance of the left edge of the view-port from the center of the projection	-1.0
RIGHT	float	Distance of the right edge of the view-port from the center of the projection	1.0
TOP	float	Distance of the top edge of the view-port from the center of the projection	1.0
BOTTOM	float	Distance of the bottom edge of the view-port from the center of the projection	-1.0

Example

Here is an example of a **`<camera>`** element that describes a perspective view of the scene with a 45-degree field of view.

```
<camera name=”eyepoint”>
  <technique profile="COMMON">
    <optics>
      <program name="PERSPECTIVE">
        <param name="YFOV" type="float">45.0</param>
        <param name="ZNEAR" type="float">0.1</param>
        <param name="ZFAR" type="float">32767.0</param>
      </program>
    </optics>
  </technique>
</camera>
```

imager

Introduction

Imagers represent the image sensor of a camera (for example film or CCD).

Concepts

The optics of the camera projects the image onto a (usually planar) sensor.

The **`<imager>`** element defines how this sensor transforms light colors and intensities into numerical values.

Real light intensities may have a very high dynamic range. For example, in an outdoor scene, the Sun is many orders of magnitude brighter than the shadow of a tree. Also, real light may contain photons with an infinite variety of wavelengths.

Display devices use a much more limited dynamic range and they usually only consider 3 wavelengths within the visible range: Red, Green and Blue (primary colors). This is usually represented as 3, 8-bit values.

An image sensor therefore performs two tasks:

- Spectral sampling
- Dynamic range remapping

The combination of these is called tone-mapping, and is performed as the last step of image synthesis (rendering).

High-quality renderers – such as ray-tracers – represent spectral intensities as floating-point numbers internally and will store the actual pixel colors as float3s, or even as arrays of floats (multi-spectral renderers), then perform tone-mapping to create an 24-bit RGB image that can be displayed by the graphics hardware and monitor.

Many renderers can also save the original high dynamic range (HDR) image to allow for “re-exposing” it later.

Related Elements

The **`<imager>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`camera`**
Child elements	**`program`**
Other	None

Remarks

The **`<imager>`** element is optional. The COMMON profile omits it and the default interpretation is:

- Linear mapping of intensities
- Clamping to the 0 ... 1 range (in terms of an 8-bit per component frame-buffer, this maps to 0...255)
- R,G,B spectral sampling

Multi-spectral renderers need to specify an **`<imager>`** element to at least define the spectral sampling.

Example

Here is an example of a **<camera>** element that describes a realistic camera with a CCD sensor.

```
<camera name=”eyepoint”>
  <technique profile="MyFancyGIRenderer">
    <optics>
      <program url="#RealisticZoomLens">
        <param name="FocalLength" type="float">180.0</param>
        <param name="Aperture" type="float">5.6</param>
...
      </program>
    </optics>
    <imager>
      <program url="#RGB_CCD_And_Analog_Digital_Converter">
        <param name="ShutterSpeed" type="float">200.0</param>
<!- "White-balance"   -->
        <param name="RedGain" type="float">0.2</param>
        <param name="GreenGain" type="float">0.22</param>
        <param name="BlueGain" type="float">0.25</param>

        <param name="RedGamma" type="float">2.2</param>
        <param name="GreenGamma" type="float">2.1</param>
        <param name="BlueGamma" type="float">2.17</param>

        <param name="BloomPixelLeak" type="float">0.17</param>
        <param name="BloomFalloff" type="Name">InvSquare</param>
      </program>
    </imager>
  </technique>
</camera>
```

instance

Introduction

The **`<instance>`** element declares the instantiation of a node, an object or the contents of another COLLADA resource.

Concepts

The actual data representation of an object may be stored once. However, the object can appear in the scene more than once. The object may be transformed in various ways each time it appears. Each appearance in the scene is called an instance of the object.

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A non-unique (shared) instance shares some or all of its data with other instances of that object. Changes made to one shared instance affect all the other instances sharing that data.

When the mechanism to achieve this effect is local to the current scene or resource it is called instancing. When the mechanism to achieve this effect is external to the current scene or resource it is called external referencing.

Attributes

The **`<instance>`** element has the following attributes:

`url`	**`xs:anyURL`**

The **`url`** attribute indicates the URL of the location of the object to instance. Required attribute.

When the **`url`** attribute indicates a **`<geometry>`** element then the instantiation does not resolve any **`<controller>`** elements that may target the **`<geometry>`** element.

When the **`url`** attribute indicates a **`<controller>`** element then the instantiation resolves the **`<controller>`** element and its target, such as a **`<geometry>`** element.

Related Elements

Occurrences	Number of elements defined in the schema
Parent elements	**`node`**
Child elements	No child elements.
Other	None

Remarks

The **`url`** attribute refers to a local instance using a relative URL fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the id of the element to instantiate.

The **`url`** attribute refers to an external reference using an absolute or relative URL when it contains a path to another resource.

COLLADA does not dictate the policy of data sharing for each instance. This decision is left to the run-time application.

Example

Here is an example of an **`<instance>`** element that refers to a locally defined **`<geometry>`** element identified by the id "cube". The instance is translated some distance from the original.

```
<library>
  <geometry id="cube" />
</library>

<node>
  <node>
    <translate>11.0 12.0 13.0</translate>
    <instance url="#cube" />
  </node>
</node>
```

library

Introduction

The `<library>` element declares a module of elements of a single category.

Concepts

As data sets become larger and more complex they become harder to manipulate within a single container. One approach to manage this complexity is to divide the data into smaller pieces organized by some criteria. These modular pieces can then be stored in separate resources as libraries.

Attributes

The `<library>` element has the following attributes:

id	**xs:ID**
name	**xs:NCName**
type	One of **ANIMATION, CAMERA, CODE, CONTROLLER, GEOMETRY, IMAGE, LIGHT, MATERIAL, PROGRAM, TEXTURE**

The **`id`** attribute is a text string containing the unique identifier of the `<library>` element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of the element. Optional attribute.

The **`type`** attribute indicates the category of child elements contained. Required attribute.

Related Elements

The `<library>` element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**COLLADA**
Child elements	**animation, camera, code, controller, geometry, image, light, material, program, texture**
Other	None

Remarks

The `<library>` element can have zero or more child elements that conform to the category of its type attribute. A library whose **`type`** attribute is “MATERIAL” must contain only `<material>` elements, for example.

Example

Here is an example of two **<library>** elements with one being lights and the other containing materials.

```
<library type=”LIGHT”>
  <light />
  <light />
</library>
<library type=”MATERIAL”>
  <material />
  <material />
  <material />
</library>
```

animation

Introduction

The **`<animation>`** element categorizes the declaration of animation information. The animation contains elements that describe the animation's key-frame data and sampler functions.

Concepts

Animation describes the transformation of an object or value over time. A common use of animation is to give the illusion of motion. A common animation technique is called key-frame animation.

A key-frame is a sampling of data at a known point in time. Using a set of key-frames and an interpolation algorithm, intermediate values are computed for times in between the key-frames, producing a smooth set of output values over the interval between the key-frames. A key-frame animation engine performs these computations on the key-frame data.

Attributes

The **`<animation>`** element has the following attributes:

`id`	**`xs:ID`**
`name`	**`xs:NCName`**

The **`id`** attribute is a text string containing the unique identifier of the **`<animation>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

Related Elements

The **`<animation>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`library`**
Child elements	**`source, sampler, channel`**
Other	None

Remarks

An **`<animation>`** element contains the elements that describe animation data. The actual type and complexity of the data is left to the child elements to represent in detail.

The **`<animation>`** element must contain one or more **`<source>`** elements.

The **`<animation>`** element must contain one or more **`<sampler>`** elements.

The **`<animation>`** element must contain one or more **`<channel>`** elements.

The sequence of child elements must be in the order: **`<source>, <sampler>, <channel>`**.

Example

Here is an example of an empty **<animation>** element with the allowed attributes.

```
<library type=”ANIMATION”>
  <animation name="walk” id=”Walk123”>
    <source />
    <source />
    <sampler />
    <channel />
    <channel />
  </animation>
</library>
```

channel

Introduction

The `<channel>` element declares an output channel of an animation.

Concepts

As an animation transforms value over time, those values are directed out to channels. The animation channels describe where to store the transformed values from the animation engine. The channels target the data structures that receive the animated values.

Attributes

The `<channel>` element has the following attributes.

id	**xs:ID**
name	**xs:NCName**
source	**xs:anyURL**
target	**xs:token**

The **id** attribute is a text string containing the unique identifier of the `<channel>` element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

The **source** attribute indicates the location of the sampler using a URL expression. Required attribute.

The **target** attribute indicates the location of the element bound to the output of the sampler. This text string is a path-name following a simple syntax described in Address Syntax. Required attribute.

Related Elements

The `<channel>` element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**animation**
Child elements	None
Other	None

Remarks

The **target** attribute addressing syntax is described in the section Address Syntax.

Example

Here is an example of a **<channel>** element that targets the translate values of an element whose id is "Box".

```
<animation>
  <channel id="Box-Translate-X-Channel" source="#Box-Translate-X-Sampler"
target="Box/Trans.X"/>
  <channel id="Box-Translate-Y-Channel" source="#Box-Translate-Y-Sampler"
target="Box/Trans.Y"/>
  <channel id="Box-Translate-Z-Channel" source="#Box-Translate-Z-Sampler"
target="Box/Trans.Z"/>
</animation>
```

sampler

Introduction

The **`<sampler>`** element declares an N-dimensional function.

Concepts

Animation function curves are represented by 1-D **`<sampler>`** elements in COLLADA. The sampler defines sampling points and how to interpolate between them. When used to compute values for an animation channel, the sampling points are the animation key-frames.

Sampling points (key-frames) are input data sources to the sampler. Animation channels direct the output data values of the sampler to their targets.

Attributes

The **`<sampler>`** element has the following attributes.

`id`	**`xs:ID`**
`name`	**`xs:NCName`**

The **`id`** attribute is a text string containing the unique identifier of the **`<sampler>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

Related Elements

The **`<sampler>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`animation`**
Child elements	**`input`**
Other	None

Remarks

The **`<input>`** element must appear one or more times.

Sampling points are described by the **`<input>`** elements that refer to source elements. The semantic attribute of the **`<input>`** element can be one of, but is not limited to: INPUT, INTERPOLATION, IN_TANGENT, OUT_TANGENT, and OUTPUT.

COLLADA recognizes the following interpolation types: LINEAR, BEZIER, CARDINAL, HERMITE, BSPLINE, STEP.

A **`<sampler>`** element must contain an **`<input>`** element with a semantic attribute of INTERPOLATION in order to be complete. COLLADA does not specify a default interpolation type. If an interpolation type is not specified the resulting **`<sampler>`** behavior is application defined.

Example

Here is an example of a **`<sampler>`** element that evaluates the X-axis values of a key-frame source element whose id is “Box-Trans-X”.

```
<animation>
  <sampler id=" Translate-X-Sampler">
    <input semantic=”INPUT” source=”#Box-Trans-X-Time” />
    <input semantic=”OUTPUT” source=”#Box-Trans-X” />
  </sampler>
</animation>
```

controller

Introduction

The **`<controller>`** element categorizes the declaration of generic control information.

Concepts

A controller is a device or mechanism that manages and directs the operations of another object.

Attributes

The **`<controller>`** element has the following attributes:

id	**xs:ID**
name	**xs:NCName**
target	**xs:IDREF**

The **id** attribute is a text string containing the unique identifier of the **`<controller>`** element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

The **target** attribute indicates the ID of the geometry element bound to the controller. Required attribute.

Related Elements

The **`<controller>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**library**
Child elements	**skin**
Other	None

Remarks

A **`<controller>`** element contains the elements that describe control data. The actual type and complexity of the data is left to the child elements to represent in detail.

The **`<skin>`** element must occur exactly one time.

Example

Here is an example of an empty **`<controller>`** element with the allowed attributes.

```
<library type=”CONTROLLER”>
  <controller name="skinner” id=”skinner456” target=”SomeBox”>
    <skin />
  </controller>
</library>
```

skin

Introduction

The `<skin>` element contains vertex and primitive information sufficient to describe blend-weight skinning.

Concepts

For character skinning, an animation engine drives the joints (skeleton) of a skinned character. A skin mesh describes the associations between the joints and the mesh vertices forming the skin topology. The joints influence the transformation of skin mesh vertices according to a controlling algorithm.

A common skinning algorithm blends the influences of neighboring joints according to weighted values.

The classical skinning algorithm transforms points of a geometry (for example vertices of a mesh) with matrices of nodes (sometimes called joints) and averages the result using scalar weights. The affected geometry is called the *skin*, the combination of a transform (node) and its corresponding weight is called an *influence* and the set of influencing nodes (usually a hierarchy) is called a *skeleton*.

“Skinning” involves two steps:

- pre-processing, known as “binding the skeleton to the skin”
- running the *skinning algorithm* to modify the shape of the skin as the *pose* of the skeleton changes

The results of the pre-processing, or “skinning information” consists of the following:

- bind-shape: also called “default shape”. This is the shape of the skin when it was *bound* to the skeleton. This includes positions (required) for each corresponding `<mesh>` vertex and may optionally include additional vertex attributes.
- influences: a variable-length lists of node + weight pairs for each `<mesh>` vertex.
- bind-pose: the transforms of all influences at the time of binding. This per-node information is usually represented by a “bind-matrix”, which is the *local-to-world* matrix of a node at the time of binding.

In the skinning algorithm, all transformations are done relative to the bind-pose. This relative transform is usually pre-computed for each node in the skeleton and is stored as a *skinning matrix*.

To derive the new (“skinned”) position of a vertex, the skinning matrix of each influencing node transforms the bind-shape position of the vertex and the result is averaged using the blending weights.

The easiest way to derive the skinning matrix is to multiply the current local-to-world matrix of a node by the inverse of the node's bind-matrix. This effectively cancels out the bind-pose transform of each node and allows us to work in the common *object space* of the skin.

The binding process usually involves:

- storing the current shape of the skin as the bind-shape
- computing and storing the bind-matrices
- generating default blending weights, usually with some fall-off function: the farther a joint is from a given vertex, the less it influences it. Also, if a weight is 0, the influence can be omitted.

After that, the artist is allowed to hand-modify the weights, usually by “painting” them on the mesh.

Attributes

The **<skin>** element has the following attributes:

id	**xs:ID**
name	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the **<skin>** element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

Related Elements

The **<skin>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**controller**
Child elements	**source, vertices**
Other	None

Remarks

The **<source>** element must occur one or more times. The **<source>** elements provide the bulk of the skin's vertex data.

The **<vertices>** element must occur exactly one time. The **<vertices>** element describes the skin vertices that are formed from the vertex data.

The skinning information is stored outside of the affected geometry, under the **<skin>** element. This de-coupling allows skeletons to "skin" multiple geometries and different instances of a **<geometry>** element to be modified by different skeletons.

- Bind-shape: represented with a **<source>,** the same way as **<mesh>** vertex positions.
- Influences / joints: a **<combiner>** represents the variable-length influence lists for each vertex.
- Bind-pose: to avoid inverting matrices on import, COLLADA stores the *inverse* of the bind-matrices.

The matrices are per-node (they are the same for each vertex that is influenced by a certain node). This is represented by two matching **<array>**s (one for the nodes and one for the matrices) and combined by the **<joints>** element. Note that any **<node>** can be an influence, not only joints. The name of the **<joints>** element was chosen for familiarity reasons.

There must be a one-to-one correspondence between the vertices in **<skin>** and the **<mesh>** vertices. Naturally, if a mesh is exported in its bind-shape (the skeleton was not moved after binding), the bind-shape positions in the **<skin>** will be the same as the vertex positions under the targeted **<mesh>.**

Also, the skeleton's transforms will result in matrices that, after inverting, will match the inverse-bind matrices in the **<skin>.**

The sequence of child elements must be in the order: **<source>**, **<vertices>**.

Example

Here is an example of an empty **<skin>** element with the allowed attributes.

```
<skin name="humanoid">
  <source id="Pos" />
  <source id="Joints_and_Weights" />
  <vertices />
```

```
</skin>
```

combiner

Introduction

The **`<combiner>`** element declares the aggregation of input streams.

Concepts

The **`<combiner>`** element aggregates input data streams into arrays of data structures. The resulting data structures provide a logical organization of the input streams for higher-level elements.

Attributes

The **`<combiner>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**

The **`count`** attribute indicates the number of value elements. Required attribute.

Related Elements

The **`<combiner>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`technique`**
Child elements	**`input`**, **`v`**
Other	None

A **`<combiner>`** element contains a sequence of **v** elements, where "v" stands for value. Each **v** element describes the values for an arbitrary number of data structures.

Each **v** element contains indices that reference into the **`<input>`** elements. These indices are position dependent and reference the **`<accessor>`** elements according to the order of the **`<input>`** elements. Here is an example of this: The first index references the first *unique* **`<input>`** element; the second index references the second *unique* **`<input>`** element, and so on. This is a simple form of compression that reduces the number of indices required in each **v** element. The **`<input>`** elements are uniquely identified by their *idx* attribute values.

A complete sampling of an aggregate value is completed by gathering one value from each input using the associated index in the **v** element.

The sequence of child elements must be in the order: **`<input>, v`**.

Remarks

The **`<input>`** element must occur two or more times. There must be at least two things to combine.

The **v** element must occur one or more times.

Example

Here is an example of a **<combiner>** element that builds a structure of joints and their blend weights.

```
<combiner count=”2”>
  <input source="#joints" />
  <input source="#weights" />
  <v> 0 0  1 1 </v>
  <v> 2 2  3 7  5 12 </v>
</combiner>
```

joints

Introduction

The **<joints>** element declares the association between joint nodes and attribute data.

Concepts

The **<joints>** element associates joint nodes in the scene hierarchy with bind-position matrices in a skin controller. The joint nodes represent the joints of a character skeleton. The skeleton can be resused by different skin controllers. Information particular to a specific skin controller is stored with the skin including the position of the joints at the time the skin was bound them.

Attributes

The **<joints>** element has the following attributes.

id	**xs:ID**
name	**xs:NCName**
count	**xs:nonNegativeInteger**

The **id** attribute is a text string containing the unique identifier of the **<joints>** element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

The **count** attribute indicates the number of value elements. Optional attribute.

Related Elements

The **<joints>** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**technique**
Child elements	**input**
Other	None

Remarks

The **<input>** element must occur two or more times because there must be at least two things associated with each other.

The **<input>** element must *not* have the **idx** attribute when it is the child of a **<joints>** element.

Example

Here is an example of a **<joints>** element that associates joints and their bind positions.

```
<skin>
  <joints id="count="2">
    <input semantic="JOINT" source="#joints" />
    <input semantic="BIND_MATRIX" source="#bind-positions" />
  </joints>
</skin>
```

geometry

Geometry describes the visual shape and appearance of an object in the scene.

Introduction

The `<geometry>` element categorizes the declaration of geometric information. Geometry is a branch of mathematics that deals with the measurement, properties, and relationships of points, lines, angles, surfaces, and solids. The `<geometry>` element contains a declaration of a mesh or a NURBS.

Concepts

There are many forms of geometric description. Computer graphics hardware has been normalized, primarily, to accept vertex position information with varying degrees of attribution (color, normals, etc.). Geometric descriptions provide this vertex data with relative directness or efficiency. Some of the more common forms of geometry are listed below:

- B-Spline
- Bezier
- Mesh
- NURBS
- Patch

This is by no means an exhaustive list.

Attributes

The `<geometry>` element has the following attributes:

id	**xs:ID**
name	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the `<geometry>` element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is a text string containing the name of the `<geometry>` element. Optional attribute.

Related Elements

The `<geometry>` element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**library**
Child elements	**mesh, extra**
Other	None

Remarks

A `<geometry>` element contains the elements that describe geometric data. The actual type and complexity of the data is left to the child elements to represent in detail.

The `<mesh>` element must occur exactly one time.

The `<geometry>` element can contain zero or more `<extra>` elements.

Example

Here is an example of an empty **`<geometry>`** element with the allowed attributes.

```
<library type=”GEOMETRY”>
  <geometry name="cube” id=”cube123”>
  </geometry>
</library>
```

mesh

Introduction

The **`<mesh>`** element contains vertex and primitive information sufficient to describe basic geometric meshes.

Concepts

Meshes embody a general form of geometric description that primarily includes vertex and primitive information.

Vertex information is the set of attributes associated with a point on the surface of the mesh. Each vertex includes data for attributes such as:

- Vertex position
- Vertex color
- Vertex normal
- Vertex texture coordinate

The mesh also includes a description of how the vertices are organized to form the geometric shape of the mesh. The mesh vertices are collated into geometric primitives such as polygons, triangles, or lines.

Attributes

The **`<mesh>`** element has the following attributes:

`id`	**`xs:ID`**
`name`	**`xs:NCName`**

The **`id`** attribute is a text string containing the unique identifier of the **`<mesh>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is a text string containing the name of the **`<mesh>`** element. Optional attribute.

Related Elements

The **`<mesh>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`geometry`**
Child elements	**`source, vertices, lines, linestrips, polygons, triangles, trifans, tristrips`**
Other	None

Remarks

The **`<source>`** element must occur one or more times as the first elements of **`<mesh>`**. The **`<source>`** element may also occur 0 or more times after the **`<vertices>`** element but before the primitive elements. The **`<source>`** elements provide the bulk of the mesh's vertex data.

The **`<vertices>`** element must occur exactly one time. The **`<vertices>`** element describes the mesh-vertex attributes and establishes their topological identity.

To describe geometric primitives that are formed from the vertex data, the **`<mesh>`** element may contain zero or more of the following primitive elements:

- The **`<lines>`** element contains line primitives.
- The **`<linestrips>`** element contains line-strip primitives.
- The **`<polygons>`** element contains polygon primitives.
- The **`<triangles>`** element contains triangle primitives.
- The **`<trifans>`** element contains triangle-fan primitives.
- The **`<tristrips>`** element contains triangle-strip primitives.

The **`<vertices>`** element under **`<mesh>`** is used to describe mesh-vertices. Polygons, triangles, and so forth index mesh-vertices, not positions directly. Mesh-vertices must have at least an **`<input>`** element with a **`semantic`** attribute whose value is POSITION.

For texture coordinates, COLLADA's right-handed coordinate system applies therefore an "ST" coordinate of [0,0] maps to the lower-left texel of a texture image, when loaded in a professional 2D texture viewer/editor.

The sequence of child elements must be in the order: **`<source>`**, **`<vertices>`**, **`<source>`**, **`primitive elements`**, where the second **`<source>`** element is optional and primitive elements is any combination of **`<lines>`**, **`<linestrips>`**, **`<polygons>`**, **`<triangles>`**, **`<trifans>`**, or **`<tristrips>`**.

Example

Here is an example of an empty **`<mesh>`** element with the allowed attributes.

```
<mesh name="cube" id="cube123">
  <source id="box-Pos" />
  <vertices id="box-Vtx" />
</mesh>
```

lines

Introduction

The `<lines>` element declares the binding of geometric primitives and vertex attributes for a `<mesh>` element.

Concepts

The `<lines>` element provides the information needed to bind vertex attributes together and then organize those vertices into individual lines.

The vertex array information is supplied in distinct attribute arrays of the `<mesh>` element that are then indexed by the `<lines>` element.

Each line described by the mesh has two vertices. The first line is formed from first and second vertices. The second line is formed from the third and fourth vertices and so on.

Attributes

The `<lines>` element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`material`	**`xs:anyURL`**

The **`count`** attribute indicates the number of line primitives. Required attribute.

The **`material`** attribute refers to the name of a `<material>` element, using a URL expression, bound to the lines. Optional attribute.

If the **`material`** attribute is not specified then the lighting and shading results are application defined.

Related Elements

The `<lines>` element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	`mesh`
Child elements	`input, p, param`
Other	None

Remarks

A `<lines>` element contains a sequence of **p** elements, where "p" stands for primitive. Each **p** element describes the vertex attributes for an arbitrary number of individual lines.

Each **p** element contains indices that reference into the `<source>` elements. These indices are position dependent and reference the contents of the `<source>` elements according to the order of the `<input>` elements. Here is an example of this: The first index references the first *unique* `<input>` element; the second index references the second *unique* `<input>` element, and so on. This is a simple form of compression that reduces the number of indices required in each **p** element. The `<input>` elements are uniquely identified by their *idx* attribute values.

A complete sampling of a single vertex is completed by gathering one value from each input using the associated index in the **p** element.

The **`<input>`** element may occur zero or more times.

The **p** element may occur zero or more times.

The **`<param>`** element may occur zero or more times.

The sequence of child elements must be in the order: **`<param>`, `<input>`, `p`**.

Example

Here is an example of the **`<lines>`** element: collating three **`<input>`** elements into one line, where the last two inputs are using the same index.

```
<mesh>
  <source name=”position” />
  <source name=”texcoord” />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <lines count=”1”>
    <input semantic=”VERTEX” source=”#verts” idx=”0” />
      <input semantic=”TEXCOORD” source=”#texcoord” idx=”1” />
      <input semantic=”TEXCOORD” source=”#texcoord” idx=”1” />
    <p>0 0 1 1</p>
  </lines>
</mesh>
```

linestrips

Introduction

The **`<linestrips>`** element declares a binding of geometric primitives and vertex attributes for a **`<mesh>`** element.

Concepts

The **`<linestrips>`** element provides the information needed to bind vertex attributes together and then organize those vertices into connected line-strips.

The vertex information is supplied in distinct attribute arrays of the **`<mesh>`** element that are then indexed by the **`<linestrips>`** element.

Each line-strip described by the mesh has an arbitrary number of vertices. Each line segment within the line-strip is formed from the current vertex and the preceding vertex.

Attributes

The **`<linestrips>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`material`	**`xs:anyURL`**

The **`count`** attribute indicates the number of line-strip primitives. Required.

The **`material`** attribute refers to the name of a **`<material>`** element, using a URL expression, bound to the line-strips. Optional.

If the **`material`** attribute is not specified then the lighting and shading results are application defined.

Related Elements

The **`<linestrips>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`mesh`**
Child elements	**`input, p, param`**

Remarks

A **`<linestrips>`** element contains a sequence of **p** elements, where “p” stands for primitive. Each **p** element describes the vertex attributes for an arbitrary number of connected line segments.

Each **p** element contains indices into the **`<source>`** elements. These indices are position dependent and reference the contents of the **`<source>`** elements according to the order of the **`<input>`** elements. The first index references the first *unique* **`<input>`** element; the second index references the second *unique* **`<input>`** element, and so on. This is a simple form of compression that reduces the number of indices required in each **p** element. The **`<input>`** elements are uniquely identified by their *idx* attribute values.

A complete sampling of a single vertex is completed by gathering one value from each input using the associated index in the **p** element.

The **`<input>`** element may occur zero or more times.

The **p** element may occur zero or more times.

The **`<param>`** element may occur zero or more times.

The sequence of child elements must be in the order: **`<param>`, `<input>`, `p`**.

Example

Here is an example of the **`<linestrips>`** element that describes two line segments with three vertex attributes, where all three inputs are using the same index.

```
<mesh>
  <source name=”position” />
  <source name=”normals” />
  <source name=”texcoord” />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <linestrips count=”1”>
    <input semantic=”VERTEX” source=”#verts” idx=”0” />
    <input semantic=”NORMAL” source=”#normals” idx=”0” />
    <input semantic=”TEXCOORD” source=”#texcoord” idx=”0” />
    <p>0 1 2</p>
  </linestrips>
</mesh>
```

polygons

Introduction

The **`<polygons>`** element declares the binding of geometric primitives and vertex attributes for a **`<mesh>`** element.

Concepts

The **`<polygons>`** element provides the information needed to bind vertex attributes together and then organize those vertices into individual polygons. (A polygon is a closed plane figure bounded by straight lines or a closed figure on a sphere bounded by arcs of great circles per Merriam-Webster's 10th edition Dictionary).

The vertex array information is supplied in distinct attribute arrays of the **`<mesh>`** element that are then indexed by the **`<polygons>`** element.

The polygons described can contain arbitrary numbers of vertices. Ideally, they would describe convex shapes, but they may be concave as well. The polygons may also contain holes.

Attributes

The **`<polygons>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`material`	**`xs:anyURL`**

The **`count`** attribute indicates the number of polygon primitives. Required attribute.

The **`material`** attribute refers to the name of a **`<material>`** element, using a URL expression, bound to the polygons. Optional attribute.

If the **`material`** attribute is not specified then the lighting and shading results are application defined.

Related Elements

The **`<polygons>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`mesh`**
Child elements	**`input, p, param`**
Other	None

Remarks

A **`<polygons>`** element contains a sequence of **p** elements, where "p" stands for primitive. Each **p** element describes the vertex attributes for an individual polygon.

A hole in the polygon is indicated by an **h** element that acts as a contour separator. Any indices following the **h** element describe the hole, up to the next **h** element, or the close of the **p** element. Consequently, **h** elements are the last data inside a **p** element.

Each **p** element contains indices that reference into the **`<source>`** elements. These indices are position dependent and reference the contents of the **`<source>`** elements according to the order of the **`<input>`**

elements. The first index references the first *unique* **`<input>`** element; the second index references the second *unique* **`<input>`** element, and so on. This is a simple form of compression that reduces the number of indices required in each **p** element.

A complete sampling of a single vertex is completed by gathering one value from each input using the associated index in the **p** element.

The winding order of vertices produced is counter-clockwise and describe the front side of each polygon.

If the primitives are assembled without vertex normals then the application may generate per-primitive normals to enable lighting.

The **`<input>`** element may occur zero or more times.

The **p** element may occur zero or more times.

The **`<param>`** element may occur zero or more times.

The sequence of child elements must be in the order: **`<param>`, `<input>`, `p`**.

Tangents and bi-normals

Many operations need an exact orientation of a surface point. The normal vector partially defines this orientation, but it is still leaves the "rotation" about the normal itself ambigous.

One way to "lock down" this extra rotation is to also specify the surface tangent at the same point.

Assuming that the type of the coordinate system is known (e.g. right-handed), this fully specifies the orientation of the surface, meaning that we can define a 3x3 matrix to transforms between *object-space* and *surface space*.

The tangent and the normal specify two axes of the surface coordinate system (two columns of the matrix) and the third one, called *bi-normal* may be computed as the cross-product of the tangent and the normal.

COLLADA supports two different types of tangents, because they have different applications and different logical placements in a document:

- texture-space tangents: used for normal-mapping (bump-mapping). These specify, in object-space the (3D) direction of increasing S texture-coordinate, so they are inherently tied to the texture coordinates and are specified with the texture-coordinate set to which they apply.

  For example, a TEXCOORD <input> for a <polygons> element that has texture-space tangents will provide 5 values: S, T, TANGENT.X, TANGENT.Y and TANGENT.Z:

```
    <source id="MeshTexcoords0">
...
    <array id="sphere-0-TexCoord0-array" type="float" count="10">
 0.5  0.5 -0.980785 -0  0.195
 0  1 -0.980785 -0  0.19509</array>

    <technique profile="COMMON">
      <accessor source="#sphere-0-TexCoord0-array" count="2" stride="5">
        <param name="S" type="float"/>
        <param name="T" type="float"/>
        <param name="TANGENT.X" type="float"/>
        <param name="TANGENT.Y" type="float"/>
        <param name="TANGENT.Z" type="float"/>
      </accessor>
    </technique>
```

- standard (geometric) tangents: used for anisotropic shaders. These are independent of texturing (anisotropic shaders don't necessarily need textures), so they are specified via a separate <input> element with a ”TANGENT” semantic attribute:

```
    <source id=”MeshTangents0”>
    <source id=”MeshTangents1”>
...
    <polygons material="GroovySurface">
     ...
      <input semantic="TANGENT" source="#MeshTangents0" idx=”2”/>
```

Example

Here is an example of a **<polygons>** element that describes a single square. The **<polygons>** element contains two **<source>** elements that contain the position and normal data, according to the **<input>** element semantics. The **p** element index values indicate the order that the input values are used.

```
<mesh>
  <source name="position" />
  <source name="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <polygons count=”1” material=”#Bricks”>
    <input semantic="VERTEX" source="#verts" idx=”0” />
    <input semantic="NORMAL" source="#normal" idx=”1” />
    <p>0 0  2 1  3 2  1 3</p>
  </polygons>
</mesh>
```

triangles

Introduction

The **`<triangles>`** element declares the binding of geometric primitives and vertex attributes for a **`<mesh>`** element.

Concepts

The **`<triangles>`** element provides the information needed to bind vertex attributes together and then organize those vertices into individual triangles.

The vertex array information is supplied in distinct attribute arrays that are then indexed by the **`<triangles>`** element.

Each triangle described by the mesh has three vertices. The first triangle is formed from the first, second, and third vertices. The second triangle is formed from the fourth, fifth, and sixth vertices, and so on.

Attributes

The **`<triangles>`** element has the following attributes.

count	**xs:nonNegativeInteger**
material	**xs:anyURL**

The **`count`** attribute indicates the number of triangle primitives. Required attribute.

The **`material`** attribute refers to the name of a **`<material>`** element, using a URL expression, bound to the triangles. Optional attribute.

If the **`material`** attribute is not specified then the lighting and shading results are application defined.

Related Elements

The **`<triangles>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**mesh**
Child elements	**input, p, param**
Other	None

Remarks

A **`<triangles>`** element contains a sequence of **p** elements, where "p" stands for primitive. Each **p** element describes the vertex attributes for an individual triangle.

Each **p** element contains indices that reference into the **`<source>`** elements. These indices are position dependent and reference the contents of the **`<source>`** elements according to the order of the **`<input>`** elements. The first index references the first *unique* **`<input>`** element; the second index references the second *unique* **`<input>`** element, and so on. This is a simple form of compression that reduces the number of indices required in each **p** element. The **`<input>`** elements are uniquely identified by their *idx* attribute values.

A complete sampling of a single vertex is completed by gathering one value from each input using the associated index in the **p** element.

The winding order of vertices produced is counter-clockwise and describe the front side of each triangle.

If the primitives are assembled without vertex normals then the application may generate per-primitive normals to enable lighting.

The **`<input>`** element may occur zero or more times.

The **p** element may occur zero or more times.

The **`<param>`** element may occur zero or more times.

The sequence of child elements must be in the order: **`<param>`, `<input>`, p**.

Example

Here is an example of a **`<triangles>`** element that describes two triangles. There are two **`<source>`** elements that contain the position and normal data, according to the **`<input>`** element semantics. The **p** element index values indicate the order that the input values are used.

```
<mesh>
  <source name="position" />
  <source name="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <triangles count=”1” material=”#Bricks”>
    <input semantic="VERTEX" source="#verts" idx=”0” />
    <input semantic="NORMAL" source="#normal" idx=”1” />
    <p>0 0  1 3  2 1  0 0  2 1  3 2</p>
  </triangles>
</mesh>
```

trifans

Introduction

The **`<trifans>`** element declares the binding of geometric primitives and vertex attributes for a **`<mesh>`** element.

Concepts

The **`<trifans>`** element provides the information needed to bind vertex attributes together and then organize those vertices into connected triangles.

The vertex array information is supplied in distinct attribute arrays of the **`<mesh>`** element that are then indexed by the **`<trifans>`** element.

Each triangle described by the mesh has three vertices. The first triangle is formed from first, second, and third vertices. Each subsequent triangle is formed from the current vertex, reusing the *first* and *previous* vertices.

Attributes

The **`<trifans>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`material`	**`xs:anyURL`**

The **`count`** attribute indicates the number of triangle-fan primitives. Required attribute.

The **`material`** attribute refers to the name of a **`<material>`** element, using a URL expression, bound to the triangles. Optional attribute.

If the **`material`** attribute is not specified then the lighting and shading results are application defined.

Elements

The **`<trifans>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`mesh`**
Child elements	**`input, p, param`**
Other	

Remarks

A **`<trifans>`** element contains a sequence of **p** elements, where “p” stands for primitive. Each **p** element describes the vertex attributes for an arbitrary number of connected triangles.

Each **p** element contains indices that reference into the **`<source>`** elements. These indices are position dependent and reference the contents of the **`<source>`** elements according to the order of the **`<input>`** elements. The first index references the first *unique* **`<input>`** element; the second index references the second *unique* **`<input>`** element, and so on. This is a simple form of compression that reduces the number of indices required in each **p** element. The **`<input>`** elements are uniquely identified by their *idx* attribute values.

A complete sampling of a single vertex is completed by gathering one value from each input using the associated index in the **p** element.

The winding order of vertices produced is counter-clockwise and describe the front side of each triangle.

If the primitives are assembled without vertex normals then the application may generate per-primitive normals to enable lighting.

The **`<input>`** element may occur zero or more times.

The **p** element may occur zero or more times.

The **`<param>`** element may occur zero or more times.

The sequence of child elements must be in the order: **`<param>`, `<input>`, `p`**.

Example

Here is an example of a **`<trifans>`** element that describes two triangles. There are two **`<source>`** elements that contain the position and normal data, according to the **`<input>`** element semantics. The **p** element index values indicate the order that the input values are used.

```
<mesh>
  <source name="position" />
  <source name="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <trifans count=”1” material=”#Bricks”>
    <input semantic=”VERTEX” source=”#verts” idx=”0” />
    <input semantic="NORMAL" source="#normal" idx=”1” />
    <p>0 0  1 3  2 1  3 2</p>
  </trifans>
</mesh>
```

tristrips

Introduction

The **`<tristrips>`** element declares the binding of geometric primitives and vertex attributes for a **`<mesh>`** element.

Concepts

The **`<tristrips>`** element provides the information needed to bind vertex attributes together and then organize those vertices into connected triangles.

The vertex array information is supplied in distinct attribute arrays of the **`<mesh>`** element that are then indexed by the **`<tristrips>`** element.

Each triangle described by the mesh has three vertices. The first triangle is formed from first, second, and third vertices. Each subsequent triangle is formed from the current vertex, reusing the *previous two* vertices.

Attributes

The **`<tristrips>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`material`	**`xs:anyURL`**

The **`count`** attribute indicates the number of triangle-fan primitives. Required attribute.

The **`material`** attribute refers to the name of a **`<material>`** element, using a URL expression, bound to the triangles. Optional attribute.

If the **`material`** attribute is not specified then the lighting and shading results are application defined.

Elements

The **`<tristrips>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`mesh`**
Child elements	**`input, p, param`**
Other	None

Remarks

A **`<tristrips>`** element contains a sequence of **p** elements, where “p” stands for primitive. Each **p** element describes the vertex attributes for an arbitrary number of connected triangles.

Each **p** element contains indices that reference into the **`<source>`** elements. These indices are position dependent and reference the contents of the **`<source>`** elements according to the order of the **`<input>`** elements. The first index references the first *unique* **`<input>`** element; the second index references the second *unique* **`<input>`** element, and so on. This is a simple form of compression that reduces the number of indices required in each **p** element. The **`<input>`** elements are uniquely identified by their *idx* attribute values.

A complete sampling of a single vertex is completed by gathering one value from each input using the associated index in the **p** element.

The winding order of vertices produced is counter-clockwise and describe the front side of each triangle.

If the primitives are assembled without vertex normals then the application may generate per-primitive normals to enable lighting.

The **`<input>`** element may occur zero or more times.

The **p** element may occur zero or more times.

The **`<param>`** element may occur zero or more times.

The sequence of child elements must be in the order: **`<param>`, `<input>`, `p`**.

Example

Here is an example of a **`<tristrips>`** element that describes two triangles. There are two **`<source>`** elements that contain the position and normal data, according to the **`<input>`** element semantics. The **p** element index values indicate the order that the input values are used.

```
<mesh>
  <source name=”position” />
  <source name=”normals” />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <tristrips count=”1” material=”#Bricks”>
    <input semantic=”VERTEX” source=”#verts” idx=”0” />
    <input semantic=”NORMAL” source=”#normals” idx=”1” />
    <p>0 0  1 3  2 1  3 2</p>
  </tristrips>
</mesh>
```

vertices

Introduction

The **`<vertices>`** element declares the attributes and identity of mesh-vertices.

Concepts

The **`<vertices>`** element describes mesh-vertices in a mesh geometry or skin controller. The mesh-vertices represent the position (identity) of the vertices comprising the mesh and other vertex attributes that are invariant to tessellation.

Attributes

The **`<vertices>`** element has the following attributes.

`id`	**`xs:ID`**
`name`	**`xs:NCName`**
`count`	**`xs:nonNegativeInteger`**

The **`id`** attribute is a text string containing the unique identifier of the **`<vertices>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`count`** attribute indicates the number of value elements. Optional attribute.

Related Elements

The **`<vertices>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`mesh, skin`**
Child elements	**`input`**
Other	None

Remarks

The **`<input>`** element must occur one or more times. One input must have the semantic attribute value of “POSITION” to establish the topological identity of each vertex in the mesh.

An **`<input>`** element may have a **`semantic`** attribute whose value is COLOR. These color inputs are RGB vectors (float3).

The **`<input>`** element must *not* have the **`idx`** attribute when it is the child of a **`<vertices>`** element.

Example

Here is an example of a **`<vertices>`** element that describes the vertices of a mesh.

```
<mesh>
  <source name="position" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
</mesh>
```

source

Introduction

The **`<source>`** element declares a data repository that provides values according to the semantics of an **`<input>`** element that refers to it.

Concepts

A data source is a well-known source of information that can be accessed through an established communication channel.

The data source provides access methods to the information. These access methods implement various techniques according to the representation of the information. The information may be stored locally as an array of data or a program that generates the data.

Attributes

The **`<source>`** element has the following attributes.

id	**xs:ID**
name	**xs:NCName**

The **`id`** attribute is a text string containing the unique identifier of the **`<source>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

Related Elements

The **`<source>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**animation, mesh, skin**
Child elements	**array, technique**
Other	None

Remarks

One of the array elements (**`<array>`**, **`<bool_array>`**, **`<float_array>`**, **`<int_array>`**, or **`<Name_array>`**) may occur 0 or more times. There may be multiple array types in a single source. They are not mutually exclusive.

Within the **`<technique>`** element, the common profile's **`<accessor>`** elements describe the output of the **`<source>`** element. The output is assembled from the **`<accessor>`** elements in the order they are specified.

The sequence of child elements must be in the order: array elements, **`<technique>`**.

Example

Here is an example of a **<source>** element that contains an array of floating-point values that comprise a single RGB color.

```
<source name=”Colors”>
  <array name=”values” type=”float” count=”3”>
    0.8 0.8 0.8
  </array>
  <technique profile=”COMMON”>
    <accessor source=”#values” count=”1” stride=”3”>
      <param name=”R” type=”float” />
      <param name=”G” type=”float” />
      <param name=”B” type=”float” />
    </accessor>
  </technique>
</source>
```

accessor

Introduction

The **`<accessor>`** element declares an access pattern to one of the array elements: **`<float_array>`**, **`<int_array>`**, **`<Name_array>`**, **`<bool_array>`**, and the generic **`<array>`** element.

The **`<accessor>`** element describes access to arrays that are organized in either an interleaved or non-interleaved manner according to the permutations of the **`offset`** and **`stride`** attributes.

Concepts

The **`<accessor>`** element describes a stream of values from an array data source. The array values are fetched according to the indices supplied by the parent element's collation elements. The output of the accessor is described by its child **`<param>`** elements.

Attributes

The **`<accessor>`** element has the following attributes.

`id`	**`xs:ID`**
`count`	**`xs:nonNegativeInteger`**
`offset`	**`xs:nonNegativeInteger`**
`source`	**`xs:anyURL`**
`stride`	**`xs:nonNegativeInteger`**

The **`id`** attribute is a text string containing the unique identifier of the **`<accessor>`** element. This value must be unique within the instance document. Optional attribute.

The **`count`** attribute indicates the number of times the array is accessed. Required attribute.

The **`offset`** attribute indicates the index of the first value in the array that is accessed. The default value is 0. Optional attribute.

The **`source`** attribute indicates the location of the array to access using a URL expression. Required attribute.

The **`stride`** attribute indicates the number of values to access per *count* iteration according to the type of data in the source array. The default value is 1, indicating that a single value is accessed. Optional attribute.

Related Elements

The **`<accessor>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`technique`**
Child elements	**`param`**
Other	None

The **`<param>`** element describes the output of the **`<accessor>`** element sufficiently to bind that output to consumers of the data stream.

Remarks

The **`<param>`** element may occur zero or more times.

The number and order of **`<param>`** elements define the output of the **`<accessor>`** element. Parameters are bound to values in the order they are specified. Parameter ordering is irrelevant to the array data source of the **`<accessor>`** element. This means that no swizzling of the data occurs.

The type attribute of the **`<param>`** element, when it is a child of the **`<accessor>`** element, is restricted to the set of array types: **`int`**, **`float`**, **`Name`**, **`bool`**, and **`token`**.

A **`<param>`** element without a *name* attribute is unbound. This is an indication that the value is not passed to the consumer.

The **`source`** attribute of the **`<accessor>`** element may refer to an array data source outside the scope of the instance document.

The **`stride`** attribute must have a value equal to or greater then the number of **`<param>`** elements. If there are fewer **`<param>`** elements then indicated by the stride value, the unbound array data source values are skipped.

Example

Here is an example of an **`<accessor>`** element that describes a stream of 3 pairs of integer values, while skipping every second value in the array because the second **`<param>`** element has no `name` attribute.

```
<source>
  <int_array name="values" count="9">
    1 1 1 2 2 2 3 3 3
  </int_array>
  <technique profile="COMMON">
    <accessor source="#values" count="3" stride="3">
      <param name="A" type="int" />
      <param type="int" />
      <param name="C" type="int" />
    </accessor>
  </technique>
</source>
```

Here is another example showing every third value being skipped because there is no **`<param>`** element binding it to the output and the **`stride`** attribute is still three.

```
<source>
  <int_array name="values" count="9">
    1 1 0 2 2 0 3 3 0
  </int_array>
  <technique profile="COMMON">
    <accessor source="#values" count="3" stride="3">
      <param name="A" type="int" />
      <param name="B" type="int" />
    </accessor>
  </technique>
</source>
```

This next example produces the same output as the previous example but using two **`<accessor>`** elements instead of one.

```
<source>
  <int_array name="values" count="9">
    1 1 0 2 2 0 3 3 0
  </int_array>
  <technique profile="COMMON">
  <accessor source="#values" count="3" stride="3" offset="0">
    <param name="A" type="int" />
  </accessor>
  <accessor source="#values" count="3" stride="3" offset="1">
    <param name="B" type="int" />
  </accessor>
  </technique>
</source>
```

array

Introduction

The **`<array>`** element declares the storage for a homogenous array of generic data values.

Concepts

The **`<array>`** element stores the data values for generic use within the COLLADA schema. The arrays themselves are generic and without semantics. They simply describe a sequence of values that are all the same type.

Attributes

The **`<array>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`id`	**`xs:ID`**
`name`	**`xs:NCName`**
`type`	One of: **`int, float, name, token`**
`minInclusive`	**`xs:integer`**
`maxInclusive`	**`xs:integer`**
`digits`	**`xs:short`**
`magnitude`	**`xs:short`**

The **`count`** attribute indicates the number of values in the array. Required attribute.

The **`id`** attribute is a text string containing the unique identifier of the **`<array>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`type`** attribute indicates the data type, such as floating-point or integer numbers of the values contained in the array. The array may also contain a list of names or tokens conforming to the XML Schema Language derived types xs:Name and xs:token, respectively. Required attribute.

The **`minInclusive`** attribute indicates the smallest integer value that can be contained in the array. The default value is –2147483648. Optional attribute.

The **`maxInclusive`** attribute indicates the largest integer value that can be contained in the array. The default value is 2147483647. Optional attribute.

The **`digits`** attribute indicates the number of significant decimal digits of the float values that can be contained in the array. The default value is 6. Optional attribute.

The **`magnitude`** attribute indicates the largest exponent of the float values that can be contained in the array. The default value is 38. Optional attribute.

Related Elements

The **`<array>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`source`**
Child elements	No child elements
Other	None

Remarks

An **`<array>`** element contains a list of values of the indicated type. These values are a repository of data to **`<source>`** elements.

The **`<array>`** element is deprecated in this version of COLLADA and may be removed in the future.

Example

Here is an example of an **`<array>`** element that describes a sequence of floating-point numbers totaling nine values.

```
<array name=”data” type=”float” count=”9”>
  1.0 0.0 0.0
  0.0 0.0 0.0
  1.0 1.0 0.0
</array>
```

bool_array

Introduction

The **<bool_array>** element declares the storage for a homogenous array of Boolean values.

Concepts

The **<bool_array>** element stores the data values for generic use within the COLLADA schema. The arrays themselves are strongly typed but without semantics. They simply describe a sequence of XML Boolean values.

Attributes

The **<bool_array>** element has the following attributes.

count	**xs:nonNegativeInteger**
id	**xs:ID**
name	**xs:NCName**

The **count** attribute indicates the number of values in the array. Required attribute.

The **id** attribute is a text string containing the unique identifier of this element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

Related Elements

The **<bool_array>** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**source**
Child elements	No child elements
Other	None

Remarks

A **<bool_array>** element contains a list of XML Boolean values. These values are a repository of data to **<source>** elements.

Example

Here is an example of an **<bool_array>** element that describes a sequence of four Boolean values.

```
<bool_array id=”flags” name=”myFlags” count=”4”>
  true true false false
</bool_array>
```

float_array

Introduction

The **`<float_array>`** element declares the storage for a homogenous array of floating point values.

Concepts

The **`<float_array>`** element stores the data values for generic use within the COLLADA schema. The arrays themselves are strongly typed but without semantics. They simply describe a sequence of floating point values.

Attributes

The **`<float_array>`** element has the following attributes.

`count`	**`xs:nonNegativeInteger`**
`id`	**`xs:ID`**
`name`	**`xs:NCName`**
`digits`	**`xs:short`**
`magnitude`	**`xs:short`**

The **`count`** attribute indicates the number of values in the array. Required attribute.

The **`id`** attribute is a text string containing the unique identifier of this element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`digits`** attribute indicates the number of significant decimal digits of the float values that can be contained in the array. The default value is 6. Optional attribute.

The **`magnitude`** attribute indicates the largest exponent of the float values that can be contained in the array. The default value is 38. Optional attribute.

Related Elements

The **`<float_array>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`source`**
Child elements	No child elements
Other	None

Remarks

A **`<float_array>`** element contains a list of floating point values. These values are a repository of data to **`<source>`** elements.

Example

Here is an example of an **<float_array>** element that describes a sequence of nine floating-point values.

```
<float_array id=”floats” name=”myFloats” count=”9”>
  1.0 0.0 0.0
  0.0 0.0 0.0
  1.0 1.0 0.0
</float_array>
```

int_array

Introduction

The **`<int_array>`** element declares the storage for a homogenous array of integer values.

Concepts

The **`<int_array>`** element stores the data values for generic use within the COLLADA schema. The arrays themselves are strongly typed but without semantics. They simply describe a sequence of integer values.

Attributes

The **`<int_array>`** element has the following attributes.

count	**xs:nonNegativeInteger**
id	**xs:ID**
name	**xs:NCName**
minInclusive	**xs:integer**
maxInclusive	**xs:integer**

The **count** attribute indicates the number of values in the array. Required attribute.

The **id** attribute is a text string containing the unique identifier of this element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

The **minInclusive** attribute indicates the smallest integer value that can be contained in the array. The default value is –2147483648. Optional attribute.

The **maxInclusive** attribute indicates the largest integer value that can be contained in the array. The default value is 2147483647. Optional attribute.

Related Elements

The **`<int_array>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**source**
Child elements	No child elements
Other	None

Remarks

An **`<int_array>`** element contains a list of integer values. These values are a repository of data to **`<source>`** elements.

Example

Here is an example of an **`<int_array>`** element that describes a sequence of five integer numbers.

```
<int_array id="integers" name="myInts" count="5">
  1 2 3 4 5
</int_array>
```

Name_array

Introduction

The **<Name_array>** element declares the storage for a homogenous array of symbolic name values.

Concepts

The **<Name_array>** element stores the data values for generic use within the COLLADA schema. The arrays themselves are strongly typed but without semantics. They simply describe a sequence of XML name values.

Attributes

The **<Name_array>** element has the following attributes.

count	**xs:nonNegativeInteger**
id	**xs:ID**
name	**xs:NCName**

The **count** attribute indicates the number of values in the array. Required attribute.

The **id** attribute is a text string containing the unique identifier of this element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

Related Elements

The **<Name_array>** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**source**
Child elements	No child elements
Other	None

Remarks

An **<Name_array>** element contains a list of XML name values. These values are a repository of data to **<source>** elements.

Example

Here is an example of an **<Name_array>** element that describes a sequence of four name values.

```
<Name_array id=”names” name=”myNames” count=”4”>
  Node1 Node2 Joint3 WristJoint
</Name_array>
```

input

Introduction

The **`<input>`** element declares the input semantics of a data source.

Concepts

The **`<input>`** element declares the input connections that a consumer requires.

Attributes

The **`<input>`** element has the following attributes.

`idx`	**`xs:nonNegativeInteger`**
`semantic`	**`xs:NMTOKEN`**
`source`	**`xs:anyURL`**

The **`idx`** attribute uniquely identifies the input among the set of **`<input>`** elements in scope. Required attribute if **`<input>`** is a child of **`<combiner>`**, **`<lines>`**, **`<linestrips>`**, **`<polygons>`**, **`<triangles>`**, **`<trifans>`**, or **`<tristrips>`**.

The **`semantic`** attribute is the user-defined meaning of the input connection. Required attribute.

The **`source`** attribute indicates the location of the data source. Required attribute.

Related Elements

The **`<input>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`combiner, joints, lines, linestrips, pass, polygons, sampler, technique, triangles, trifans, tristrips, vertices`**
Child elements	No child elements
Other	None

Remarks

Each input can be uniquely identified by its **`idx`** attribute within the scope of its parent element.

An **`<input>`** element may have a **`semantic`** attribute whose value is COLOR. These color inputs are RGB (float3).

Example

Here is an example of two **`<input>`** elements that describe the sources of vertex positions and normals for a **`polygon`** element.

```
<mesh>
  <source name="grid-Position" />
  <source name="grid-0-Normal" />
  <vertices id=”grid-Verts”>
    <input semantic="POSITION" source="#grid-Position"/>
  <vertices>
  <polygons count=”1” material=”#Bricks”>
    <input semantic="VERTEX" source="#grid-Verts" idx=”0” />
    <input semantic="NORMAL" source="#grid-Normal" idx=”1” />
    <p>0 0 2 1 3 2 1 3</p>
  </polygons>
</mesh>
```

material

Materials describe the visual appearance of a geometric object.

Introduction

The **`<material>`** element categorizes the declaration of rendering appearance information. The **`<material>`** element contains declarations of shaders, parameters, techniques, and vertex and pixel programs.

Concepts

In computer graphics, geometric objects can have many parameters that describe their material properties. These material properties are the parameters for the rendering computations that produce the visual appearance of the object in the final output.

The specific set of material parameters depend upon the graphics rendering system employed. Fixed, function, graphics pipelines require parameters to solve a predefined illumination model, such as Phong illumination. These parameters include terms for ambient, diffuse and specular reflectance, for example.

In programmable graphics pipelines, the programmer defines the set of material parameters. These parameters satisfy the rendering algorithm defined in the vertex and pixel programs.

Attributes

The **`<material>`** element has the following attributes.

`id`	**`xs:ID`**
`name`	**`xs:NCName`**

The **`id`** attribute is a text string containing the unique identifier of the **`<material>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

Related Elements

The **`<material>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`library`**
Child elements	**`asset, param, shader`**
Other	None

Remarks

A **`<material>`** element is the base container for all rendering and shading information. The visual appearance of a geometric object is described by its material properties.

The **`<material>`** element may contain zero or more **`<param>`** elements.

The **`<material>`** element must contain one or more **`<shader>`** elements.

The sequence of child elements must be in the order: **`<asset>`, `<param>`, `<shader>`**.

Material transparency is specifiied using two named **`<param>`** elements: TRANSPARENCY and TRANSPARENT. The **`<param>`** named TRANSPARENCY is a scale factor on the TRANSPARENT color. The scale factor ranges from 0.0 (opaque) to 1.0 (transparent). A value greater than 0.0 indicates that the material is somewhat transparent. The **`<param>`** named TRANSPARENT is an RGB color that represents the distinct amount of red, green, and blue transparency. Conceptually, the transparent color is modulated by the transparency scale factor to produce the actual transparency per color component. The resulting transparent color represents the weighting factor between reflective and transmissive light. For the Phong lighting model this modulates the diffuse color only. The color blend calculation is as follows:

```
Output.Red = DIFFUSE.Red x (1.0 - TRANSPARENT.Red x TRANSPARENCY) + Output.Red x
TRANSPARENT.R x TRANSPARENCY
Output.Green = DIFFUSE.Green x (1.0 - TRANSPARENT.Green x TRANSPARENCY) +
Output.Green x TRANSPARENT.Green x TRANSPARENCY
Output.Blue = DIFFUSE.Blue x (1.0 - TRANSPARENT.Blue x TRANSPARENCY) +
Output.Blue x TRANSPARENT.Blue x TRANSPARENCY
```

Material reflectivity is specified using two named **`<param>`** elements: REFLECTIVITY and REFLECTIVE. The **`<param>`** named REFLECTIVY is a scale factor on the REFLECTIVE color. The scale factor ranges from 0.0 (matte) to 1.0 (mirror). A value greater than 0.0 indicates that the material is somewhat reflective. The **`<param>`** named REFLECTIVE is an RGB color that represents the distinct amount of red, green, and blue reflectivity. Conceptually, the reflective color is modulated by the reflectivity scale factor to produce the actual reflectivity per color component. The resulting reflective color is additive and applies after the lighting calculation.

Example

Here is an example of a **`<material>`** element that describes a simple Phong lighting model. The material is contained in a material **`<library>`** element.

```
<library type=”MATERIAL”>
  <material name="Blue" id="Blue">
    <shader>
      <technique profile="COMMON">
        <pass>
          <program url="PHONG">
            <param name="AMBIENT" type="float3" flow="IN">0.000000 0.000000
0.000000</param>
            <param name="DIFFUSE" type="float3" flow="IN">0.137255 0.403922
0.870588</param>
            <param name="SPECULAR" type="float3" flow="IN">0.500000 0.500000
0.500000</param>
            <param name="SHININESS" type="float" flow="IN">16.000000</param>
            <param name="TRANSPARENT" type="float3">1.000000 0.000000
0.000000</param>
            <param name="TRANSPARENCY" type="float">0.500000</param>
          </program>
        </pass>
      </technique>
    </shader>
  </material>
</library>
```

shader

Shaders describe the process of rendering the appearance of a geometric object.

Introduction

The **`<shader>`** element declares the rendering techniques used to produce the appearance of a material. The **`<shader>`** element contains declarations of parameters and techniques.

Concepts

In computer graphics, the process of rendering the appearance of geometric objects involves many computations. The execution of these computations is called shading or rendering in the generic sense. The collection of program code and parameter information is generically called a shader.

A shader executes within the run-time environment provided by the graphics or rendering engine on the target platform. Since a shader is meaningful across multiple target platforms, its description includes one or more techniques.

Attributes

The **`<shader>`** element has the following attributes.

id	**xs:ID**
name	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the **`<shader>`** element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

Related Elements

The **`<shader>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**material**
Child elements	**param, technique**
Other	None

Remarks

The **`<param>`** element may occur zero or more times.

The **`<technique>`** element must occur one or more times.

A **`<param>`** element specified as a child of the **`<shader>`** element is in the scope of all the shader's techniques. This implies that the parameter is invariant across all the techniques as well.

Each **`<technique>`** element specified corresponds to a (platform) profile. This implies that the set of techniques are mutually exclusive. However, an application may choose to apply information from multiple techniques.

The sequence of child elements must be in the order: **`<param>`, `<technique>`**.

Example

Here is an example of a **`<shader>`** element that describes a simple Phong lighting model. The shader is contained in a **`<material>`** element.

```
<material id=”default_material”>
  <shader name=”Phong” >
    <technique profile="COMMON">
      <pass>
        <program url="PHONG">
          <param name="DIFFUSE" type="float3">0.8 0.8 0.8</param>
        </program>
      </pass>
    </technique>
  </shader>
</material>
```

pass

A pass describes an algorithm that is executed during a single iteration of the rendering engine.

Introduction

The **`<pass>`** element declares the information required to execute a rendering pass.

Concepts

Computer graphics rendering can involve arbitrarily complex algorithms. Conversely, graphics hardware may be resource limited or otherwise constrained so that it cannot execute the entire algorithm at once. Therefore, the algorithm is often broken down in simpler steps or passes.

The rendering engine executes each pass in the specified order. The results of each pass are combined together to produce the final result. This methodology is called multi-pass rendering.

Attributes

The **`<pass>`** element does not allow attributes.

Related Elements

The **`<pass>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`technique`**
Child elements	**`param, input, program`**
Other	None

Remarks

The **`<param>`** element may occur zero or more times. A **`<param>`** element specified as a child of the **`<pass>`** element is in the scope of only that pass.

The **`<input>`** element may occur zero or more times.

The **`<program>`** element may occur zero or one time.

The sequence of child elements must be in the order: **`<param>, <input>, <program>`**.

The **`<input>`** element must *not* have the **`idx`** attribute when it is the child of a **`<pass>`** element.

Example

Here is an example of a **`<shader>`** element that describes a two-pass technique. Each pass is executed in the order that the **`<pass>`** elements are specified.

```
<shader>
  <technique profile="example">
    <pass>
      <param name=”AMBIENT” type=”float3”>
        0.2 0.2 0.2
      </param>
      <program url=“PHONG” />
    </pass>
    <pass>
      <program name=”bump_map” />
    </pass>
  </technique>
</shader>
```

technique

Introduction

The **`<technique>`** element declares the information used to process some portion of the content. Each technique conforms to an associated profile.

Concepts

A technique embodies a method or procedure used to process information to produce an effect, such as the visual appearance of a geometric object. The information described in the technique is context sensitive. A technique for a shader may be very different than the one for a data source.

Two things define the context for a technique: its profile and its parent element in the instance document.

A technique conforms to a profile of information that describes the platform environment in which the technique is understood. The set of available profiles is outside the scope of this document, except for the COMMON Profile.

Techniques act as a "switch". If more than one is present for a particular portion of content, on import, one or the other is picked, but not both. Selection should be based on which profile the importing application can support.

Techniques contain application data and programs, making them assets that can be managed as a unit.

Attributes

The **`<technique>`** element has the following attributes.

profile	**xs:string**

The **`profile`** attribute indicates the type of profile. This is a vendor defined character string that indicates the platform or capability target for the technique. Required attribute.

Related Elements

The **`<technique>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**camera, extra, shader, source, texture**
Child elements	**accessor, combiner, imager, input, joints, optics, param, pass, program**
Other	None

Remarks

The set of permissible child elements is constrained by the parent element of the **`<technique>`** element.

The **`<accessor>`** element may occur zero or more times, when the technique is the child of a **`<source>`** element.

The **`<asset>`** element may occur zero or one time.

The **`<input>`** element may occur zero or more times, when the technique is the child of a **`<texture>`** element.

The **`<param>`** element may occur zero or more times, except when the technique is the child of a **`<camera>`** element.

The **`<program>`** element may occur zero or one time, when the technique is the child of a **`<shader>`**, **`<source>`**, or **`<texture>`** element.

The **`<pass>`** element may occur zero or more times, when the technique is the child of a **`<shader>`** element.

A **`<technique>`** element, when it is a child of the **`<camera>`** element, must contain an **`optics`** element and may also contain zero or more **`imager`** elements, in that order. This is discussed in the camera section.

A **`<technique>`** element, when it is a child of the **`<source>`** element, may contain one **`<combiner>`** element. It may also contain one **`<joints>`** element.

The sequence of child elements, when a <technique> is a child of the **`<source>`** element, must be in the order: **`<asset>`**, **`<param>`**, **`<accessor>`**, **`<combiner>`**, **`<joints>`**, **`<program>`**.

The **`<input>`** element must *not* have the **`idx`** attribute when it is the child of a **`<technique>`** element.

Example

Here is an example of a **`<shader>`** element that has two **`<technique>`** elements. Each technique pertains to a specific profile.

```
<shader>
  <technique profile="COMMON">
    <pass />
  </technique>
  <technique profile="Cg">
    <pass />
  </technique>
</shader>
```

image

Images embody the graphical representation of a scene or object.

Introduction

The **`<image>`** element declares the storage for the graphical representation of an object. The **`<image>`** element best describes raster image data, but can conceivably handle other forms of imagery.

Concepts

Digital imagery comes in three main forms of data: raster, vector and hybrid. Raster imagery is comprised of a sequence of brightness or color values, called picture elements (pixels) that together form the complete picture. Vector imagery uses mathematical formulae for curves, lines, and shapes to describe a picture or drawing. Hybrid imagery combines both raster and vector information, leveraging their respective strengths, to describe the picture.

Raster imagery data is organized in N-dimensional arrays. This array organization can be leveraged by texture lookup functions to access non-color values such as displacement, normal, or height field values.

Attributes

id	**xs:ID**
name	**xs:NCName**
source	**xs:anyURL**
format	**xs:string**
height	**xs:nonNegativeInteger**
width	**xs:nonNegativeInteger**
depth	**xs:nonNegativeInteger**

The **`id`** attribute is a text string containing the unique identifier of the **`<image>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`source`** attribute is a valid URL of the image source data. If this attribute is present, then it refers to an image asset that may contain information about the dimensions and format of the image. Optional attribute.

The **`format`** attribute is a text string value that indicates the image format. Optional attribute.

The **`height`** attribute is an integer value that indicates the height of the image in pixel units. Optional attribute.

The **`width`** attribute is an integer value that indicates the width of the image in pixel units. Optional attribute.

The **`depth`** attribute is an integer value that indicates the depth of the image in pixel units. A 2-D image has a depth of 1, which is also the default value. Optional attribute.

Related Elements

The **<image>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**library, texture**
Child elements	None
Other	None

Remarks

The **<image>** element may contain a sequence of hexadecimal encoded binary octets representing the embedded image data. These values are interpreted as pixel values according to the **format** attribute.

Example

Here is an example of an **<image>** element that refers to an external PNG asset.

```
<library type=”IMAGE”>
  <image name="WoodFloor" source="Textures/WoodFloor-01.png"/>
</library>
```

light

Introduction

The **`<light>`** element declares a light source that illuminates the scene.

Concepts

A light embodies a source of illumination shining on the visual scene.

A light source can be located within the scene or infinitely far away.

Light sources have many different properties and radiate light in many different patterns and frequencies.

An ambient light source radiates light from all directions at once. The intensity of an ambient light source is not attenuated.

A point light source radiates light in all directions from a known location in space. The intensity of a point light source is attenuated as the distance to the light source increases.

A directional light source radiates light in one direction from a known direction in space that is infinitely far away. The intensity of a directional light source is not attenuated.

A spot light source radiates light in one direction from a known location in space. The light radiates from the spot light source in a cone shape. The intensity of the light is attenuated as the radiation angle increases away from the direction of the light source. The intensity of a spot light source is also attenuated as the distance to the light source increases.

Attributes

The **`<light>`** element has the following attributes:

`id`	**`xs:ID`**
`name`	**`xs:NCName`**
`type`	`One of:` **`AMBIENT, DIRECTIONAL, POINT, SPOT`**

The **`id`** attribute is a text string containing the unique identifier of the **`<light>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`type`** attribute indicates the basic type of light source. The default value is “POINT”. Optional attribute.

Related Elements

The **`<light>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`library`**
Child elements	**`asset, param`**
Other	None

Remarks

The **`<asset>`** element may occur zero or one time.

The **`<param>`** element must occur one or more times.

The **`<asset>`** element must occur before any **`<param>`** elements if it is present.

The child **`<param>`** elements describe the values that the light contributes to a material's computation. The name attribute of one of the **`<param>`** elements must have the value "COLOR" as it is defined in the COMMON Profile for all light types. The "COLOR" value of the **`<light>`** element modulates the COMMON Profile parameters named "AMBIENT", "DIFFUSE", and "SPECULAR" of a **`<material>`** element, according to the type attribute value of the <light> element. If the type attribute value is "AMBIENT" then the material's "AMBIENT" parameter is modulated. If the type attribute value is "DIRECTIONAL", "POINT", or "SPOT" then the material's "DIFFUSE" and "SPECULAR" parmeters are modulated.

If the type attribute value is "DIRECTIONAL" or "SPOT" then the light's default direction vector in local coordinates is [0,0,-1], pointing down the -Z axis.

If the type attribute value is "POINT" or "SPOT" then two additional **`<param>`** elements are required: one named "ATTENUATION", and one named "ATTENUATION_SCALE", as defined in the COMMON Profile.

If the type attribute value is "SPOT" then all the **`<param>`** elements required by "POINT" are required. In addition three more **`<param>`** elements are required: one named "ANGLE" for the cone shape, one named "FALLOFF", and one named "FALLOFF_SCALE", as defined in the COMMON Profile.

The **`<light>`** element may be referenced by an **`<instance>`** element to position and orient the light in the scene. A light that is not instantiated in the scene is not active.

Example

Here is an example of a **`<library>`** element that contains a directional **`<light>`** element that is instantiated in the scene, rotated to portray a sunset.

```
<library type="LIGHT">
  <light id="sun" name="the-sun" type="DIRECTIONAL">
    <param name="COLOR" type="float3">1.0 1.0 1.0</param>
  </light>
</library>
<scene>
  <node>
    <instance url="#sun" />
    <rotate>1 0 0 -10</rotate>
</scene>
```

Here is an example of a spot <light> element that is that is instantiated in the scene above the origin:

```
<library type="LIGHT">
  <light id="spot1" name="spotty" type="SPOT">
    <param name="COLOR" type="float4" flow="IN">1.0 1.0 1.0 1.0</param>
    <param name="ATTENUATION" type="token">CONSTANT</param>
    <param name="ATTENUATION_SCALE" type="float">1.0</param>
    <param name="ANGLE" type="float">90.0</param>
    <param name="FALLOFF" type="token">CONSTANT</param>
    <param name="FALLOFF_SCALE" type="float">1.0</param>
  </light>
</library>
<scene>
  <node>
    <instance url="#sun" />
    <rotate>1 0 0 90</rotate>
  <translate>0.0 150.0 0.0</translate>
</scene>
```

texture

Textures embody the visual detail of the surface of a geometric object.

Introduction

The **<texture>** element embodies the sampling aspects of texturing.

Concepts

Textures need three types of information:

1. Texel (**<texture>** element, after pixel) data that can contain values such as color, intensity, elevation (bump- or displacement-map), normals etc. This can be pre-authored as an image, or it can be procedurally generated (for example, a marble or wood texture).
2. Mapping information that determines how the image is placed on a 3-D object. This information is specific to the geometry that references a texture; therefore, it is stored on the geometry. This allows for sharing textures between models while using different mapping on each. The types of mapping information are:
   - Explicit texture coordinates (S and T) on vertices of primitives
   - Method: planar (X-Y, Y-Z etc.), cylindrical, parametric (U-V), etc.
   - Tiling (also called wrapping)
3. Blending information determines how the texel data will be interpreted. For example: the alpha channel of the image might be elevation used for bump mapping and the color (R, G, B) determines how reflective the surface point should be. Blending information is specific to the material and shader that uses a texture and is stored with the shader.

Attributes

The **<texture>** element has the following attributes:

id	**xs:ID**
name	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the **<texture>** element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

Related Elements

The **<texture>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**library**
Child elements	**asset, param, technique**
Other	None

Remarks

The **`<asset>`** element may occur zero or one time.

The **`<param>`** element may occur zero or more times. The **`<param>`** elements describe the texture's dataflow with the associated material.

The **`<texture>`** element may contain zero or more **`<technique>`** elements. The **`<technique>`** elements embody its functionality such as image lookup or procedural generation.

The sequence of child elements must be in the order: **`<asset>`, `<param>`, `<technique>`**.

Example

Here is an example of a **`<texture>`** element that declares one generic technique whose input source is an **`<image>`** element. The texture binds that input to a single output parameter named "DIFFUSE".

```
<library type="TEXTURE">
  <texture name="Tx-Textures-Aluminium">
    <param name="DIFFUSE" type="float3" />
  <technique profile="COMMON">
    <input semantic="IMAGE" source="#Img-Aluminium"/>
  </technique>
  </texture>
</library>
```

program

Introduction

The **`<program>`** element declares parameters and program code that executes within the application run-time environment or the graphics pipeline.

Concepts

A computer graphics rendering engine has a pipelined architecture. The graphics pipeline accepts geometric data and processes it into picture elements (pixels) suitable for the display device.

The geometric data is composed of vertex information that is transformed by programs that process one vertex at a time. These are vertex programs.

The output of a vertex program passes through one or more stages of the graphics pipeline where it becomes input to a fragment or pixel program. A pixel program produces the final pixel value that is written to the output buffer.

Attributes

The **`<program>`** element has the following attributes:

`id`	**`xs:ID`**
`name`	**`xs:Name`**
`url`	**`xs:anyURL`**

The **`id`** attribute is a text string containing the unique identifier of the **`<program>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`url`** attribute is a valid URL to the location of the program resource. This resource contains information understood by the application run-time according to the technique profile in scope. Optional attribute.

Related Elements

The **`<program>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`imager, library, optics, pass, technique`**
Child elements	**`asset, code, entry, param`**
Other	None

Remarks

The **`<code>`** element may occur zero or one time.

The **`<param>`** element may occur zero or more times.

A **`<param>`** element, specified as a child of the **`<program>`** element, indicates the interface to the program. The value of such a parameter may vary each time the program executes.

The sequence of child elements must be in the order: **`<asset>`**, **`<param>`**, **`<entry>`**, **`<code>`**.

Programs use the same id/url linking (define/reference) facility as other elements. To define a program, specify an **`id`** attribute value:

```
<program id="ShaderPrgInThisFile">
```

To refer to a program (i.e. a "function call"), reference that **`id`** value in a **`url`** attribute:

```
<program url="#ShaderPrgInThisFile">
<program url="http://www.shaders.com/shiny/glass.xml">
```

Another way to refer to a program is by using a name as the URL value such as:

```
<program url="PHONG">
```

In this case[1], the name refers to a well-known or built-in program. It's not a URI fragment (#PHONG) because it's not in the same instance document. It's implicitly defined in the COMMON profile.

Example

Here is an example of a **`<program>`** element that describes a bump-mapping algorithm.

```
<pass>
  <program name=”Bump”>
    <param name=”position” type=”float3” />
    <param name=”normal” type=”float3” />
    <code />
  </program>
</pass>
```

[1] Developer Note: This is non-standard usage for a URL. The closest conforming usage would be to make the name a URN (see RFC#2141) and that would require the syntax to be “urn:PHONG:blah” or “urn:blah:PHONG” (where “blah” is TBD).

code

Introduction

The **`<code>`** element declares the executable code for a **`<program>`** element.

Concepts

Computer processing units execute instructions sequentially. A stream of instructions that defines an algorithm is called a program. A program is divided into instructions and data values. The instructions are commonly called program code or code.

Code that is human readable is called source code as it is the source of the instructions that the computer executes. The computer executes source code that has been transformed into binary encoded data that conforms to the instruction set of the processor. Therefore, the computer executes binary code.

Attributes

The **`<code>`** element has the following attributes:

`id`	**`xs:ID`**
`lang`	**`xs:NMTOKEN`**
`profile`	**`xs:string`**
`semantic`	**`xs:NMTOKEN`**
`url`	**`xs:anyURL`**

The **`id`** attribute is a text string containing the unique identifier of the **`<code>`** element. This value must be unique within the instance document. Optional attribute.

The **`lang`** attribute indicates the programming language that the code is stored in. Required attribute.

The **`profile`** attribute indicates the type of profile. This is a vendor defined character string that indicates the platform or capability target for the code. Optional attribute.

The **`semantic`** attribute is the user-defined meaning of the code. Optional attribute.

The **`url`** attribute is a valid URL to the location of the code resource. Optional attribute.

Related Elements

The **`<code>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`library, program`**
Child elements	No child elements
Other	None

Remarks

The **`<code>`** element may contain a text string value. This string value is interpreted as source code.

Example

Here is an example of a `<program>` element that describes a Phong algorithm. The `<code>` element refers to program code using a **`url`** attribute.

```
<program id=”PHONG”>
  <param name=”IN.Position” type=”float4” />
  <param name=”IN.Normal” type=”float4” />
  <code semantic=”VERTEX_PROGRAM” profile=”CgVP20” lang=”Cg”
url=”file://somewhere/bump_vp” />
</program>
```

entry

Introduction

The **`<entry>`** element declares an entry point into the executable code for a **`<program>`** element.

Concepts

Computer processing units execute instructions sequentially. A well-known point in the instruction stream where execution can begin is called an entry point.

Attributes

The **`<entry>`** element has the following attributes:

function	**xs:NMTOKEN**
semantic	**xs:NMTOKEN**

The **`function`** attribute indicates the name of the entry point. Required attribute.

The **`semantic`** attribute is the user-defined meaning of the code. Optional attribute.

Elements

The **`<entry>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**program**
Child elements	**param**
Other	None

Remarks

The **`<entry>`** element may contain zero or more **`<param>`** elements.

In the scope of the **`<entry>`** element, the **`<param>`** element must have the **`qualifier`** attribute. The value of the **`qualifier`** attribute depends on the technique profile in scope.

In the scope of the **`<entry>`** element, the **`<param>`** element may *not* have the **`flow`** attribute.

Example

Here is an example of a **`<program>`** element that describes a Phong algorithm. The **`<entry>`** element declares the Cg function entry point that is somewhere in the **`<code>`** element.

```
<program name="CgPhong">
  <entry semantic="VERTEX_PROGRAM" function="VertexToFragment">
    <param name="IN.Position" type="float4" semantic="POSITION" />
    <param name="IN.Normal" type="float4" semantic="NORMAL" />
    <param name="LightPosition0" type="float4" semantic="LIGHT[0].LocalPosition" />
    <param name="ModelViewProjMatrix" type="float4x4"
semantic="MODELVIEW_PROJECTION" />
  </entry>
  <code semantic="VERTEX_PROGRAM" profile="CgVP20 CgVP30" />
</program>
```

param

Introduction

The `<param>` element declares parametric information regarding its parent element.

Concepts

A functional or programmatical format requires a means for users to specify parametric information. This information represents function parameter (argument) data.

Material shader programs may contain code representing vertex or pixel programs. These programs require parameters as part of their state information.

The basic declaration of a parameter describes the name, data type, and value data of the parameter. That parameter name identifies it to the function or program. The parameter type indicates the encoding of its value. The parameter value is the actual data.

Attributes

The `<param>` element has the following attributes.

id	**xs:ID**
name	**xs:NCName**
flow	One of: **IN, OUT, INOUT**
semantic	**xs:token**
type	**xs:NMTOKEN**
sid	**xs:NCName**

The **id** attribute is a text string containing the unique identifier of the `<param>` element. This value must be unique within the instance document. Optional attribute.

The **name** attribute is the text string name of this element. Optional attribute.

The **flow** attribute indicates the direction of data flow. Optional attribute.

The **semantic** attribute is the user-defined meaning of the parameter. Optional attribute.

The **type** attribute indicates the type of the value data. This text string must be understood by the application. Required attribute.

The **sid** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The `<param>` element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**accessor, combiner, entry, light, material, pass, program, shader, technique, texture, lines, linestrips, polygons, triangles, trifans, tristrips**
Child elements	No child elements
Other	None

Remarks

The **`<param>`** element describes parameters for generic data flow, program data, and entry points.

Example

Here is an example of two **`<param>`** elements that describe the arguments to a shader program.

```
<program url=“PHONG”>
  <param name=”diffuse” type=”float3”>
    0.8 0.8 0.8
  </param>
  <param name=”shininess” type=”float”>
    3.14159
  </param>
</program>
```

matrix

Matrix transformations embody mathematical changes to points within a coordinate systems or the coordinate system itself.

Introduction

The **`<matrix>`** element contains a 4-by-4 matrix of floating-point values.

Concepts

Computer graphics employ linear algebraic techniques to transform data. The general form of a 3-D coordinate system is represented as a 4-by-4 matrix. These matrices can be organized hierarchically, via the scene graph, to form a concatenation of coordinated frames of reference.

Matrices in COLLADA are column matrices in the mathematical sense. These matrices are written in row-major order to aid the human reader. See the example below.

Attributes

The **`<matrix>`** element has the following attributes:

sid	**xs:NCName**

The **`sid`** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The **`<matrix>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**node, scene**
Child elements	No child elements
Other	None

Remarks

The **`<matrix>`** element contains a list of 16 floating-point values. These values are organized into a 4-by-4 column-order matrix suitable for matrix composition.

Example

Here is an example of a **`<matrix>`** element forming a translation matrix that translates 2 units along the X-axis, 3 units along the Y-axis, and 4 units along the Z-axis.

```
<matrix>
  1.0 0.0 0.0 2.0
  0.0 1.0 0.0 3.0
  0.0 0.0 1.0 4.0
  0.0 0.0 0.0 1.0
</matrix>
```

lookat

Introduction

The **`<lookat>`** element contains a position and orientation transformation suitable for aiming a camera.

The **`<lookat>`** element contains three mathematical vectors within it that describe:

1. The position of the object;
2. The position of the interest point;
3. The direction that points up.

Concepts

Positioning and orienting a camera or object in the scene is often complicated when using a matrix. A lookat transform is an intuitive way to specify an eye position, interest point, and orientation.

Attributes

The **`<lookat>`** element has the following attributes:

`sid`	**`xs:NCName`**

The **`sid`** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The **`<lookat>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`node, scene`**
Child elements	No child elements
Other	None

Remarks

The **`<lookat>`** element contains a list of 9 floating-point values. As in the OpenGL® Utilities (GLU) implementation, these values are organized into three vectors as follows:

- Eye position is given as Px, Py, Pz.
- Interest point is given as Ix, Iy, Iz.
- Up-axis direction is given as UPx, UPy, UPz.

When computing the equivalent (viewing) matrix the interest point is mapped to the negative Z-axis and the eye position to the origin. The up-axis is mapped to the positive Y-axis of the viewing plane.

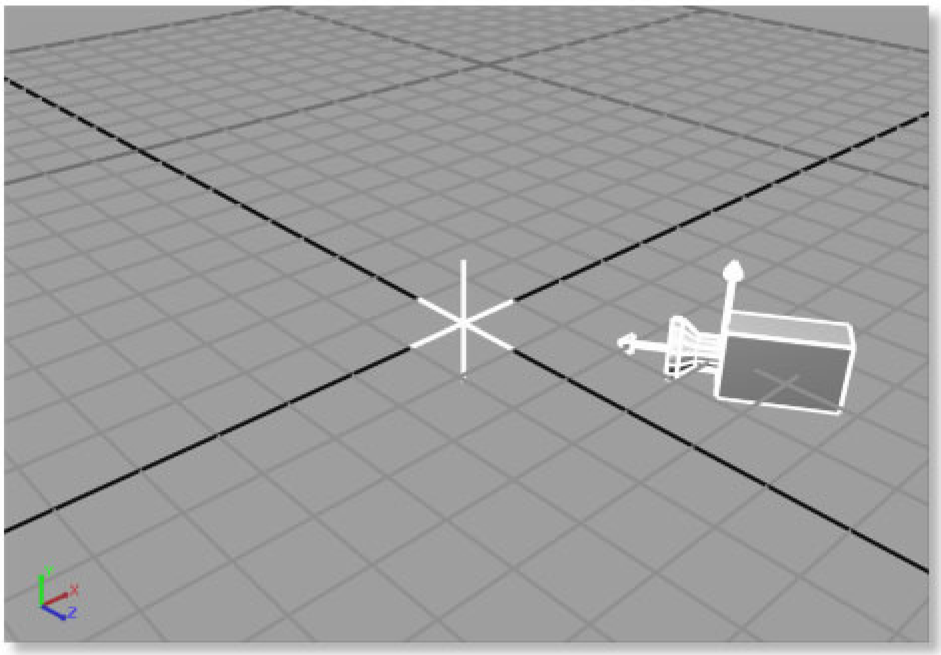
The values are specified in local, object coordinates.

Example

Here is an example of a **<lookat>** element indicating a position of [10,20,30], centered on the local origin, with the Y-axis rotated up.

```
<node name="Camera" id="Camera">
  <instance url="#camera"/>
  <lookat>
    2.0  0.0  3.0  <!-- eye position (X,Y,Z)       -->
    0.0  0.0  0.0  <!-- interest position (X,Y,Z)  -->
    0.0  1.0  0.0  <!-- up-vector position (X,Y,Z) -->
  </lookat>
  ...
```

Figure 3-1: <lookat> element; the 3D “cross-hair” represents the interest-point position



perspective

Introduction

The `<perspective>` element contains the horizontal field of view of the viewer or "camera".

Concepts

Perspective embodies the appearance of objects relative to each other as determined by their distance from a viewer. Computer graphics techniques apply a perspective projection in order to render 3-D objects onto 2-D surfaces to create properly proportioned images on display monitors.

Attributes

The `<perspective>` element has the following attributes:

sid	**xs:NCName**

The **sid** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The `<perspective>` element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**node, scene**
Child elements	No child elements
Other	None

Remarks

As in the RenderMan® specification, the focal point of the operation is at the local origin and the direction is along the z-axis. The focal length of the implied "camera" is 1.0.

Example

Here is an example of a `<perspective>` element specifying a field-of-view of 90 degrees.

```
<perspective>
  90.0
</perspective>
```

rotate

Introduction

The **`<rotate>`** element contains an angle and a mathematical vector that represents the axis of rotation.

Concepts

Rotations change the orientation of objects in a coordinated system without any translation. Computer graphics techniques apply a rotational transformation in order to orient or otherwise move values with respect to a coordinated system. Conversely, rotation can mean the translation of the coordinated axes about the local origin.

Attributes

The **`<rotate>`** element has the following attributes:

`sid`	**`xs:NCName`**

The **`sid`** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The **`<rotate>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`node, scene`**
Child elements	No child elements
Other	None

Remarks

The **`<rotate>`** element contains a list of four floating-point values, similar to rotations in the OpenGL® and RenderMan® specification. These values are organized into a column vector [X, Y, Z] specifying the axis of rotation and an angle in degrees.

Example

Here is an example of a **`<rotate>`** element forming a rotation of 90 degrees about the y-axis.

```
<rotate>
  0.0 1.0 0.0 90.0
</rotate>
```

scale

Introduction

The **`<scale>`** element contains a mathematical vector that represents the relative proportions of the X, Y and Z axes of a coordinated system.

Concepts

Scaling changes the size of objects in a coordinated system without any rotation or translation. Computer graphics techniques apply a scale transformation in order to change the size or proportions of values with respect to a coordinate system axis.

Attributes

The **`<scale>`** element has the following attributes:

sid	**xs:NCName**

The **`sid`** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The **`<scale>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**node, scene**
Child elements	No child elements
Other	None

Remarks

The **`<scale>`** element contains a list of three floating-point values. These values are organized into a column vector suitable for matrix composition.

Example

Here is an example of a **`<scale>`** element that describes a uniform increase in size of an object (or coordinated system) by a factor of two.

```
<scale>
  2.0 2.0 2.0
</scale>
```

skew

Introduction

The **<skew>** element contains an angle and two mathematical vectors that represent the axis of rotation and the axis of translation.

Concepts

Skew (shear) deforms an object along one axis of a coordinated system. It translates values along the affected axis in a direction that is parallel to that axis. Computer graphics techniques apply a skew or shear transformation in order to deform objects or to correct distortion in images.

Attributes

The **<skew>** element has the following attributes:

sid	**xs:NCName**

The **sid** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The **<skew>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**node, scene**
Child elements	No child elements
Other	None

Remarks

As in the RenderMan® specification, the **<skew>** element contains a list of seven floating-point values. These values are organized into an angle in degrees with two column vectors specifying the axes of rotation and translation.

Example

Here is an example of a **<skew>** element forming a displacement of points along the x-axis due to a rotation of 45 degrees around the y-axis.

```
<skew>
  45.0 0.0 1.0 0.0 1.0 0.0 0.0
</skew>
```

translate

Introduction

The **`<translate>`** element contains a mathematical vector that represents the distance along the X, Y and Z-axes.

Concepts

Translations change the position of objects in a coordinate system without any rotation. Computer graphics techniques apply a translation transformation in order to position or, move values with respect to a coordinate system. Conversely, translation means to move the origin of the local coordinate system.

Attributes

The **`<translate>`** element has the following attributes:

`sid`	**`xs:NCName`**

The **`sid`** attribute is a text string value containing the sub-identifier of this element. This value must be unique within the scope of the parent element. Optional attribute.

Related Elements

The **`<translate>`** element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**`node, scene`**
Child elements	No child elements
Other	None

Remarks

The **`<translate>`** element contains a list of three floating-point values. These values are organized into a column vector suitable for a matrix composition.

Example

Here is an example of a **`<translate>`** element forming a displacement of 10 units along the x-axis.

```
<translate>
  10.0 0.0 0.0
</translate>
```

asset

Introduction

The **`<asset>`** element defines asset management information regarding its parent element.

Concepts

Computers store vast amounts of information. An asset is a set of information that is organized into a distinct collection and managed as a unit. A wide range of attributes describes assets so that the information can be maintained and understood by software tools and humans.

Attributes

The **`<asset>`** element has no attributes:

Related Elements

The **`<asset>`** element relates to the following elements.

Occurrences	Number of elements defined in the schema
Parent elements	**`camera, COLLADA, light, material, program, technique, texture`**
Child elements	**`author, authoring_tool, created, modified, revision, source_data, copyright, title, subject, keywords, comments, unit, up_axis`**
Other	None

The **`author`** element contains the name of an author of the parent element. The **`author`** element may appear zero or more times.

The **`authoring_tool`** element contains the name of an application or tools used to create the parent element. The **`authoring_tool`** element may appear zero or one time.

The **`created`** element contains the date and time that the parent element was created and is represented in an ISO 8601 format. The **`created`** element may appear zero or one time.

The **`modified`** element contains the date and time that the parent element was last modified and represented in an ISO 8601 format. The **`modified`** element may appear zero or one time.

The **`revision`** element contains the revision information for the parent element. The **`revision`** element may appear zero or one time.

The **`source_data`** element contains the URI to the source data from which the parent element was created. The **`source_data`** element may appear zero or one time.

The **`copyright`** element contains the author’s copyright information applicable to the parent element. The **`copyright`** element may appear zero or more times.

The **`title`** element contains the title information for the parent element. The **`title`** element may appear zero or one time.

The **`subject`** element contains a description of the topical subject of the parent element. The **`subject`** element may appear zero or one time.

The **keywords** element contains a list of words used as search criteria for the parent element. The **keywords** element may appear zero or more times.

The **comments** element contains descriptive information about the parent element. The **comments** element may appear zero or more times.

The **unit** element contains descriptive information about unit of measure. It has attributes for the name of the unit and the measurement with respect to the meter. The **unit** element may appear zero or one time. The default value for the name attribute is "meter". The default value for the meter attribute is "1.0".

The **up_axis** element contains descriptive information about coordinate system of the geometric data. All coordinates are right-handed by definition. This element specifies which axis is considered up. The default is the Y-axis. The **up_axis** element may appear zero or one time.

Remarks

The **created** element accepts dates and times in an ISO 8601 format as per the XML Schema dataTime primitive type.

The **modified** element accepts dates and times in an ISO 8601 format as per the XML Schema dataTime primitive type.

The **unit** element contains no text data.

The **up_axis** element contains one of: **X_UP**, **Y_UP**, or **Z_UP**.

The remaining child elements all contain character string values.

Example

Here is an example of an **<asset>** element that describes the parent **<COLLADA>** element, and hence the entire document.

```
<COLLADA>
  <asset>
    <author>Mark Barnes</author>
    <keywords>COLLADA interchange</keywords>
    <comments>example asset</comments>
    <unit name="nautical_league" meter="5556.0" />
    <up_axis>Z_UP</up_axis>
  </asset>
</COLLADA>
```

extra

Introduction

The **`<extra>`** element declares additional information regarding its parent element.

Concepts

An extensible schema requires a means for users to specify arbitrary information. This extra information can represent additional real data or semantic (meta) data to the application.

COLLADA represents extra information as techniques containing an arbitrary number of **`<param>`** elements.

Attributes

The **`<extra>`** element has the following attributes:

`id`	**`xs:ID`**
`name`	**`xs:NCName`**
`type`	**`xs:NMTOKEN`**

The **`id`** attribute is a text string containing the unique identifier of the **`<extra>`** element. This value must be unique within the instance document. Optional attribute.

The **`name`** attribute is the text string name of this element. Optional attribute.

The **`type`** attribute indicates the type of the value data. This text string must be understood by the application. Optional attribute.

Related Elements

Occurrences	Number of elements defined in the schema
Parent elements	**`geometry, node, scene`**
Child elements	**`technique`**
Other	None

Remarks

The **`<technique>`** element may occur zero or more times.

The **`<extra>`** element restricts the permissible child elements of the **`<technique>`** element to the **`<asset>`** and **`<param>`** elements.

Example

Here is an example of an **`<extra>`** element that outlines both structured and unstructured additional content.

```
<geometry>
  <extra>
    <technique profile=”COMMON”>
      <asset />
      <param />
      <param />
    </technique>
  </extra>
<geometry>
```

This page intentionally left blank.

Chapter 4:
Common Profile

This page intentionally left blank.

Introduction

The COLLADA schema defines a **`<technique>`** element that establishes a context for the representation of information that conforms to a configuration profile. This profile information is currently outside the scope of the COLLADA schema but there is a way to bring it into scope.[2]

One aspect of the COLLADA design is the presence of techniques for a common profile. All tools that parse COLLADA content must understand this common profile. Therefore, we need to provide a definition for the common profile as the schema evolves.

Naming Conventions

We have adopted the following naming conventions for canonical names within the COLLADA common profile.

- Parameter names are uppercased. This means that the values for the **`<param>`** element's *name* attribute are all uppercase letters, for example:
  ```
  param name="AMBIENT" type="float3" />
  ```
- Parameter types are lowercase when they correspond to a primitive type in the COLLADA schema, or in the XML Schema or C/C++ languages. The type names are otherwise inter-capitalized. This means that the values for the **`<param>`** element's *type* attribute follow this rule, for example:
  ```
  <param name="SHININESS" type="float" />
  <param name="DIFFUSE" type="Name" />
  ```
- Program ids are uppercased. This means that the values for the **`<program>`** element's *id* attribute are all uppercase letters, for example:
  ```
  <program id="PHONG" />
  ```
- Input semantic names are uppercased. This means that the values for the **`<input>`** element's *semantic* attribute are all uppercase letters, for example:
  ```
  <input semantic="POSITION" source="#grid-Position" />
  ```

Common Profiles

The COLLADA common profile is declared by the **`<technique>`** element with a *profile* attribute value of "COMMON". For example:

```
<technique profile="COMMON">
<!-- This scope is in the common profile -->
</technique>
```

Elements that appear outside the scope of a **`<technique>`** element are not in any profile, much less the common profile. For example, an **`<input>`** element that appears within the scope of the **`<polygon>`** element is not in the common profile; rather it is invariant to all techniques.

[2] The XML Schema Language defines the elements <xs:key> and <xs:keyref> that can define a set of constrained values. This set of constraints can then validate values bound to the indicated elements and attributes within a specified scope using a subset of the XPath 1.0 language.

Parameter as an Interface

In COLLADA, a `<param>` element declares a symbol whose *name* and *semantics* combine to form a bindable parameter within the given scope. Therefore, the parameter's name as well as its semantics defines a canonical parameter. That is to say… parameters within the common profile are canonical and well known.

The *type* of a parameter can be overloaded much like in the C/C++ language. This means that the parameter's type does not have to strictly match to be successfully bound. The types must be compatible however through simple (and sensible as defined by the application) conversion or promotion such as integer to float, or float3 to foat4, or bool to int.

Common Glossary

The canonical names of parameters and semantics, and the ids of programs, that are within the common profile are listed in the glossary section of this document. Also listed are the member selection symbolic names for the *target* attribute addressing scheme.

The common `<param>` *name* attribute values are:

Name	Type	Typical Context	Description	Default value
A	float	`<material>`, `<texture>`	Alpha color component	N/A
AMBIENT	float3	`<material>`, `<texture>`	Ambient color in Phong lighting	0.2 0.2 0.2
ANGLE	float	`<animation>,` `<light>`	Euler angle	N/A
ATTENUATION	token	`<light>`	One of: CONSTANT, LINEAR, QUADRATIC	CONSTANT
ATTENUATION_ SCALE	float	`<light>`	Attenuation scale factor	1.0
B	float	`<material>`, `<texture>`	Blue color component	N/A
BOTTOM	float	`<camera>`	Camera optics bottom edge of orthographic projection	N/A
COLOR	float3	`<geometry>`, `<light>`, `<material>`, `<texture>`	Generic color as for a light source	1.0 1.0 1.0
DIFFUSE	float3	`<material>`, `<texture>`	Diffuse color in Phong lighting	0.8 0.8 0.8
EMISSION	float3	`<material>`, `<texture>`	Emissive color in Phong lighting	0.0 0.0 0.0
FALLOFF	token	`<light>`	One of: CONSTANT, LINEAR, QUADRATIC	CONSTANT
FALLOFF_SCALE	float	`<light>`	Falloff scale factor	1.0
G	float	`<material>`, `<texture>`	Green color component	N/A
LEFT	float	`<camera>`	Camera optics left edge of orthographic projection	N/A
P	float	`<geometry>`	3rd texture coordinate	N/A

Name	Type	Typical Context	Description	Default value
Q	float	`<geometry>`	4th texture coordinate	N/A
R	float	`<material>`, `<texture>`	Red color component	N/A
REFLECTIVE	float3	`<material>`	Reflective color as for lighting	0.0 0.0 0.0
REFLECTIVITY	float	`<material>`	Reflective color scale factor	0.0
RIGHT	float	`<camera>`	Camera optics right edge of orthographic projection	N/A
S	float	`<geometry>`	1st texture coordinate	N/A
SHININESS	float	`<material>`, `<texture>`	Specular exponent in Phong lighting	0.0
SPECULAR	float3	`<material>`, `<texture>`	Specular color in Phong lighting	0.0 0.0 0.0
T	float	`<geometry>`	2nd texture coordinate	N/A
TANGENT.X TANGENT.Y TANGENT.Z	float	`<accessor>` for TEXCOORD `<input>`	Texture-space tangent	N/A
TIME	float	`<animation>`	Time in seconds	N/A
TOP	float	`<camera>`	Camera optics top edge of orthographic projection	N/A
TRANSPARENCY	float	`<material>`	Transparent color scale factor	0.0
TRANSPARENT	float3	`<material>`	Transparent color as for lighting	0.0 0.0 0.0
U	float	`<geometry>`	1st generic parameter	N/A
V	float	`<geometry>`	2nd generic parameter	N/A
W	float	`<animation>`, `<controller>`, `<geometry>`	4th Cartesian coordinate	N/A
X	float	`<animation>`, `<controller>`, `<geometry>`	1st Cartesian coordinate	N/A
XFOV	float	`<camera>`	Camera optics horizontal field of view	N/A
Y	float	`<animation>`, `<controller>`, `<geometry>`	2nd Cartesian coordinate	N/A
YFOV	float	`<camera>`	Camera optics vertical field of view	N/A
Z	float	`<animation>`, `<controller>`, `<geometry>`	3rd Cartesian coordinate	N/A
ZFAR	float	`<camera>`	Camera optics far clip plane	N/A
ZNEAR	float	`<camera>`	Camera optics near clip plane	N/A

The common **`<program>`** *id* and *url* attribute values are:

Name	Description
ANGLE_MAP	Camera optics
BEZIER	Animation sampler program
BSPLINE	Animation sampler program
CARDINAL	Animation sampler program
CONSTANT	Constant material shader

Name	Description
CUBE_MAP	Camera optics
FISH_EYE	Camera optics
HERMITE	Animation sampler program
LAMBERT	Lambert material shader
LINEAR	Animation sampler program
ORTHOGRAPHIC	Camera optics
PANORAMA	Camera optics
PERSPECTIVE	Camera optics
PHONG	Phong material shader
REAR_FISH_EYE	Camera optics
SPHERICAL	Camera optics

The common **`<code>`** and **`<entry>`** *semantic* attribute values are:

Semantic	Description
FRAGMENT_PROGRAM	Fragment or pixel shader
VERTEX_PROGRAM	Vertex shader

The common **`<input>`** *semantic* attribute values are:

Semantic	Description
BIND_SHAPE_NORMAL	Vertex normal when bound
BIND_SHAPE_POSITION	Vertex position when bound
BINORMAL	Vertex binormal when bound
COLOR	Color coordinate vector
IMAGE	Raster image
INPUT	Sampler input
IN_TANGENT	Geometric tangent vector
INTERPOLATION	Sampler interpolation type
INV_BIND_MATRIX	Inverse of local-to-world matrix
JOINT	Skin influence id
JOINTS_AND_WEIGHTS	Weighted influences bound to a vertex
NORMAL	Normal vector
OUTPUT	Sampler output
OUT_TANGENT	Sampler of out_tangent
POSITION	Geometric coordinate vector
TANGENT	Tangent vector
TEXCOORD	Texture coordinate vector
TEXTURE	Texture object
UV	Generic parameter vector
VERTEX	Mesh vertex
WEIGHT	Skin influence weighting value

The common `<channel>` and `<controller>` *target* attribute member selection values are:

Name	Type	Description
'(' # ')'['(' # ')']	float	Matrix or vector field
A	float	Alpha color component
ANGLE	float	Euler angle
B	float	Blue color component
G	float	Green color component
P	float	3rd texture coordinate
Q	float	4th texture coordinate
R	float	Red color component
S	float	1st texture coordinate
T	float	2nd texture coordinate
TIME	float	Time in seconds
U	float	1st generic parameter
V	float	2nd generic parameter
W	float	4th Cartesian coordinate
X	float	1st Cartesian coordinate
Y	float	2nd Cartesian coordinate
Z	float	3rd Cartesian coordinate

Recall that array index notation, using left and right parentheses, can be used to target vector and matrix fields.

This page intentionally left blank.

Chapter 5:
Tool Requirements and Options

This page intentionally left blank.

Introduction

It should be obvious that any fully compliant COLLADA tool must support the entire specification of data represented in the schema. What may be so obvious is the need to require more than just adherence to the schema specification. Some such additional needs are the uniform interpretation of values, the necessity of offering crucial user-configurable options, and details on how to incorporate additional discretionary features into tools. The goal of this chapter is to prioritize those issues.

Each 'Requirements' section details options that must be implemented completely by every compliant tool. One exception to this rule is when the specified information is not available within a particular application. An example is a tool which does not support layers so it would not be required to export layer information (assuming the export of such layer information was required) however every tool that did support layers would be required to export them properly.

The 'Optional' section describes options and mechanisms for things that are not necessary to implement (but probably would be valuable for some subset of anticipated users as advanced or esoteric options).

The requirements explored below are placed on tools in order to ensure quality and conformance to the purpose of COLLADA. These critical data interpretations and options aim to satisfy interoperability and configurability needs of cross-platform game-development pipelines. Ambiguity in interpretation or omission of essential options could greatly limit the benefit and utility to be gained by using COLLADA. This section has been written to minimize such shortcomings.

Each feature required in this section is tested by one or more test cases in the COLLADA Conformance Test Suite. The COLLADA Confonance Test Suite is a set of tools that automate the testing of exporters and importers for Maya®, XSI, and 3DS Max. Each test case compares the native content against that content after it has gone through the tool's COLLADA import/export plug-in. The results are captured in both an HTML page and a spreadsheet

Exporters

Scope

The responsibility of a COLLADA exporter is to write all the specified data according to certain essential options.

Requirements

Hierarchy and Transforms

Translation

It must be possible to export translations.

Scaling

It must be possible to export scales.

Rotation

It must be possible to export rotations.

Parenting

It must be possible to export parent-relationships.

Static Object Instancing

It must be possible to export instances of static objects. Such an object can have multiple transforms.

Animated Object Instancing

It must be possible to export instances of animated objects. Such an object can have multiple transforms.

Skewing

It must be possible to export skews.

Transparency / Reflectivity

It must be possible to export additional material parameters for transparency and reflectivity.

Texture-mapping Method

It must be possible to export a texture-mapping method (e.g. cylindrical, spherical, etc.)

Transform With No Geometry

It must be possible to transform something with no geometry (e.g. locator, NULL).

Materials and Textures

RGB Textures

It must be possible to export an arbitrary number of RGB textures.

RGBA Textures

It must be possible to export an arbitrary number of RGBA textures.

Baked Procedural Texture Coordinates

It must be possible to export baked procedural texture coordinates.

Common Profile Material

It must be possible to export a common profile material (e.g. PHONG, LAMBERT, etc.)

Multi-texturing

It must be possible to export multiple textures per material.

Per-face Material

It must be possible to export per-face materials.

Vertex Attributes

Vertex Texture Coordinates

It must be possible to export an arbitrary number of Texture Coordinates per vertex.

Vertex Normals

It must be possible to export vertex normals.

Vertex Binormals

It must be possible to export vertex binormals.

Vertex Tangents

It must be possible to export vertex tangets.

Vertex UV Coordinates

It must be possible to export vertex UV coordinates (distinct from texture coordinates).

Vertex Colors

It must be possible to export vertex colors.

Custom Vertex Attributes

It must be possible to export custom vertex attributes.

Animation

All of the following kinds of animations (that don't specifically state otherwise) must be able to be exported using samples or key-frames (according to a user-specified option).

Animations are usually represented in an application by the use of sparse key-frames and complex controls and constraints. These are combined by the application when the animation is played providing final output. When parsing animation data it is possible that an application will not be able to implement the full set of constraints or controllers used by that which exported it, and thus the resulting animation will not be preserved. Therefore it is necessary to provide an option to export fully resolved transformation data at regularly defined intervals. The sample rate must be specifiable by the user when samples are preferred to key-frames.

Exporting all available animated parameters is necessary. This includes:

- Material parameters
- Texture parameters
- UV placement parameters
- Light parameters
- Camera parameters
- Shader parameters
- Global environment parameters
- Mesh-construction parameters
- Node parameters
- User parameters

Variable Sampling Rate

It must be possible to export using a variable sampling rate for animations. This allows a user to specify different sampling rates for different portions of the animation to be exported.

Bind-pose Normals

It must be possible to export bind-pose normals.

Bones

It must be possible to export boned animations.

Skeletal Animation

It must be possible to export skeletal animations.

Skeletal Animation With Smooth Binding

It must be possible to export skeletal animations with smooth binding.

Animation of Light Parameters

It must be possible to export animated light parameters.

Camera Animation

It must be possible to export animated cameras.

Key-Frame Animation of Transforms

It must be possible to export animated transforms with key-frames.

Animation Function Curves

It must be possible to export animation function curves.

Scene Data

Empty Nodes

It must be possible to export empty nodes.

Cameras

It must be possible to export cameras.

Spotlights

It must be possible to export spotlights.

Directional Lights

It must be possible to export directional lights.

Point Lights

It must be possible to export point lights.

Area Lights

It must be possible to export area lights.

Ambient Lights

It must be possible to export ambient lights.

Bounding Boxes for Static Objects

It must be possible to export bounding boxes for static objects.

Bounding Boxes for Animated Objects

It must be possible to export bounding boxes for animated objects.

Exporter User Interface Options

Export Triangle List Option

It must be possible to export triangle lists.

Export Polygon List Option

It must be possible to export polygon lists.

Bake Matrices Option

It must be possible to export baked matrices.

Single `<matrix>` *Element Option*

It must be possible to export an instance document that contains only a single `<matrix>` element for each node.

Collada allows transforms to be represented by a stack of different transformation element types, which must be composed in the specified order. This representation is useful for accurate storage and/or interchange of transformations in the case where an application internally uses separate transformation stages. However, if this is implemented by an application, it should be provided as a user-option, retaining the ability to store only a single baked `<matrix>`.

A side effect of this requirement is that any other data which target specific elements inside a transformation stack (such as animation) must target the matrix instead.

Command-Line Operation

It must be possible to run the full-featured exporter entirely from a command-line interface. This requirement's purpose is to preclude exporters that demand user interaction. Of course a helpful interactive user-interface is still desirable, but interactivity must be optional (as opposed to necessary).

Optional

An exporter may add any new data.

Shader Export

An exporter may export shaders (e.g. Cg, GLSL, HLSL).

Importers

Scope

The responsibility of a COLLADA importer is to read all the specified data in according to certain essential options.

In general, importers should provide perfect inverse functions of everything a corresponding exporter does. To that end, please refer to the Exporters section (5.2) of this document to see what requirements apply to importers. Importers must provide the inverse function operation of every export option where it is possible to do so. The only things described in this section are issues where the needs placed on importers diverge or need clarification from the obvious inverse method of exporters.

Requirements

It must be possible to import all conformant COLLADA data, even if some data is not understood by the tool and retained for later export. The `<asset>` element will be used by external tools to recognize that some exported data may require synchronization.

Optional

There are no unique options for importers.

This page intentionally left blank.

Appendix A:
Cube Example

This page intentionally left blank.

Example: Cube

In this appendix, a simple example of a COLLADA instance document is shown that describes a simple white cube.

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2005/COLLADASchema" version="1.3.0">
  <library type="GEOMETRY">
    <geometry id="box" name="box">
      <mesh>
        <source id="box-Pos">
          <float_array id="box-Pos-array" count="24">
           -0.5  0.5  0.5
            0.5  0.5  0.5
           -0.5 -0.5  0.5
            0.5 -0.5  0.5
           -0.5  0.5 -0.5
            0.5  0.5 -0.5
           -0.5 -0.5 -0.5
            0.5 -0.5 -0.5
          </float_array>
          <technique profile="COMMON">
            <accessor source="#box-Pos-array" count="8" stride="3">
              <param name="X" type="float" />
              <param name="Y" type="float" />
              <param name="Z" type="float" />
            </accessor>
          </technique>
        </source>
        <vertices id="box-Vtx">
          <input semantic="POSITION" source="#box-Pos"/>
        </vertices>
        <polygons count="6">
          <input semantic="VERTEX" source="#box-Vtx" idx="0"/>
          <p>0 2 3 1</p>
          <p>0 1 5 4</p>
          <p>6 7 3 2</p>
          <p>0 4 6 2</p>
          <p>3 7 5 1</p>
          <p>5 7 6 4</p>
        </polygons>
      </mesh>
    </geometry>
  </library>
  <scene id="DefaultScene">
    <node id="Box" name="Box">
      <instance url="#box"/>
    </node>
  </scene>
</COLLADA>
```

This page intentionally left blank.

Glossary

This page intentionally left blank.

Term	Definition
XML	XML is the Extensible Markup Language. XML provides a standard language to describe the structure and semantics of documents, files or data sets. XML itself is a structural language consisting of elements, attributes, comments and text data.
Elements	An XML document consists primarily of elements. An element is a block of information that is bounded by tags at the beginning and end of the block. Elements can be nested producing a hierarchical data set if so desired.
Tags	Each XML element begins with a start tag. The syntax of a start tag includes a name surrounded by angle brackets as follows: `<tagName>` Each XML element ends with an end tag. The syntax of an end tag is as follows: `</tagName>` Between the start and end tags is an arbitrary block of information.
Attributes	An XML element can have zero or more attributes. Attributes are given within the start tag and follow the tag name itself. Each attribute is a name-value pair. The value portion of an attribute is always surrounded by quotes (“”). Attributes provide semantic information about the element they are bound on. Attributes are given within the start tag and follow the tag name itself. For example: `<tagName attribute=”value”>`
Name	The name of an attribute generally has some semantic meaning in relation to the element that it belongs to.
Value	The value of an attribute is always textual data during parsing.
Comments	XML files can contain comment text. Comments are identified by special markup of the form show below: `<!-- This is a XML comment -->`
XML Schema	The XML Schema language provides the means to describe the structure of a family of XML documents that follow the same rules for syntax, structure and semantics. XML Schema is itself written in XML, making it simpler to use when designing other XML base formats.
Validation	XML by itself does not describe any one document structure or schema. XML provides a mechanism by which an XML document can be validated. The target or instance document provides a link to schema document. Using the schema document, an XML parser can validate the instance document’s syntax and semantics by the rules given in the schema document. This process is called validation.

This page intentionally left blank.

Index

This page intentionally left blank.