COLLADA – Digital Asset Schema Release 1.4.1

Specification (2nd Edition)

March 2008

Editors: Mark Barnes and Ellen Levy Finch, Sony Computer Entertainment Inc.

Publication date: March 2008

Khronos Group
P.O. Box 1019
Clearlake Park, CA 95424, U.S.A.

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

This document describes the COLLADA schema. COLLADA is a COLLAborative Design Activity that defines an XML-based schema to enable 3-D authoring applications to freely exchange digital assets without loss of information, enabling multiple software packages to be combined into extremely powerful tool chains.

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

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 of a COLLADA instance document is presented in "Appendix A".

Audience

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

Readers of this document should:

- Have knowledge of XML and 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 APIs such as OpenGL®.

Content of this Document

This document consists of the following chapters:

Chapter/Section	Description
Chapter 1: Design Considerations	Issues concerning the COLLADA design.
Chapter 2: Tool Requirements and Options	COLLADA tool requirements for implementors.
Chapter 3: Design Considerations	A general description of the schema and its design, and introduction of key concepts necessary for understanding and using COLLADA.
Chapter 4: Programming Guide	Detailed instructions for some aspects of programming using COLLADA.
Chapter 5: COLLADA Core Elements Reference	Detailed reference descriptions of the core elements in the COLLADA schema.
Chapter 6: COLLADA Physics Reference	Detailed reference descriptions of COLLADA Physics elements.
Chapter 7: Getting Started with COLLADA FX	Concepts and usage notes for COLLADA FX elements.
Chapter 8: COLLADA FX Reference	Detailed reference descriptions of COLLADA FX elements.
Chapter 9: COLLADA Types	Definitions of some simple COLLADA types.
Appendix A: COLLADA Example	An example COLLADA instance document.
Appendix B: Profile GLSL Example	A detailed example of the COLLADA FX **<profile_GLSL>** element.

Chapter/Section	Description
Glossary	Definitions of terms used in this document, including XML terminology.
General Index	Index of concepts and key terms.
Index of COLLADA Elements	Index to all COLLADA elements, including minor elements that do not have their own reference pages.

Typographic Conventions and Notation

Certain typographic conventions are used throughout this manual to clarify the meaning of the text:

Conventions	Description
Regular text	Descriptive text
<blue text>	XML elements
Courier-type font	Attribute names
Courier bold	File names
blue	Hyperlinks
Italic text	New terms or emphasis
Italic Courier	Placeholders for values in commands or code
element1 / element2	*element1* is the parent, *element2* is the child; for further information, refer to "Xpath Syntax" at http://www.w3schools.com/xpath/xpath_syntax.asp

Notation and Organization in the Reference Chapters

The schema reference chapters describe each feature of the COLLADA schema syntax. 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
Related Elements	Lists of parent elements and of other related elements
Child Elements	Lists of valid child elements and descriptions of each
Details	Information concerning the usage of the element
Example	Example usage of the element

Child Element Conventions

The Child Elements table lists all child elements for the specified element. For each child:

- "See main entry" means that one of the Reference chapters has a main entry for the child element, so refer to it for details about the child's usage, attributes, and children.
- If there is not a main entry in the Reference chapters, or if the local child element's properties vary from the main entry, information about the child element is given either in the Child Elements table or in an additional element-specific subsection.

For example:

Name/example	Description	Default	Occurrences
<camera>	*Brief_description*. See main entry. *(This means that there is a main Reference entry for camera, so look there for details.)*		1 or more
<technique_common>	*Brief_description.* See the following subsection. *(This means that details are given here but in a separate table.)*	N/A *(means not applicable)*	

Name/example	Description	Default	Occurrences
<yfov sid="...">	Description, including discussion of attributes, content, and relevant child elements. *(This means that there is no main Reference entry for* **yfov***. Details are given here.)*	*none (italic lowercase means none assigned)* NONE *(means the value NONE)*	

Child Element Order

XML allows a schema definition to include notation that requires elements to occur in a certain order within their parent element. When this reference states that child elements must appear in the following order, it refers to a declaration similar to the following, in which the XML **<sequence>** element states that **<extra>** must follow **<asset>**:

```
<xs:sequence>
   <xs:element ref="asset" minOccurs="0"/>
   <xs:element ref="extra" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
```

XML also provides notation indicating that two or more child elements can occur in any order. When this reference states that two child elements can appear in any order, it refers to the XML **<choice>** element with an unbounded maximum. For example, in the following, **<image>** and **<newparam>** must appear before **<extra>** and after **<asset>**, but in that position, they can occur in any order, and the unbounded attribute specifies that you can include as many of them as needed in any combination:

```
<xs:sequence>
   <xs:element ref="asset" minOccurs="0"/>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="image"/>
          <xs:element name="newparam" type="common_newparam_type"/>
   </xs:choice>
   <xs:element ref="extra" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
```

Other Sources of Information

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

- Be familiar with shading languages such as NVIDIA® Cg or Pixar RenderMan®.
- Have a general knowledge and understanding of computer graphics and graphics APIs such as OpenGL®.
- *Collada: Sailing the Gulf of 3d Digital Content Creation* by Remi Arnaud and Mark C. Barnes; AK Peters, Ltd., August 30, 2006; ISBN-13: 978-1568812878
- 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®

For more information on COLLADA, visit:

- www.khronos.org/collada
- collada.org/

Chapter 1:
Design Considerations

Introduction

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

Assumptions and Dependencies

During the first phase of the design of the COLLADA Asset Schema, the contributors discussed and agreed on the following assumptions:

- This is not a game engine format. We assume that COLLADA will be beneficial to users of authoring tools and to content-creation pipelines for interactive applications. 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.
- 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).

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 to 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 among 3-D applications.
- To be a catalyst for digital-asset schema design among developers and DCC, hardware, and middleware vendors.

The following subsections explain the goals and discuss their consequences and rationales.

Liberate Digital Assets from Proprietary Binary Formats

Goal: To liberate digital assets from proprietary binary formats into a well-specified, XML-based, open-source format.

Digital assets are the largest part of most 3-D applications.

Developers have enormous investment in assets that are stored in opaque proprietary formats. Exporting the data from the tools requires considerable investment to develop software for proprietary, complex software development kits. Even after this investment has been made, it is still impossible to modify the data outside of the tool and import it again later. It is necessary to permanently update the exporters with the ever-evolving tools, with 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 to developers using advanced features, and a problem for hardware vendors in promoting 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 as it is difficult to use several DCC tools in the same project.

This goal led to several decisions, including:

- COLLADA will use XML.

  XML provides a well-defined framework. Issues such as 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 only a sample instance document 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 application.
- COLLADA will not use binary data inside XML.

  Some discussion often occurs about 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, because many languages do not easily support binary data inside XML files nor do they support manipulation of binary data in general. Keeping COLLADA completely text based supports the most options. COLLADA does provide mechanisms to store external binary data and to reference it from a COLLADA asset.
- The COLLADA common profile will expand over time to include as much common data as possible.

Provide a Standard Common Format

Goal: To provide a standard common format so that COLLADA assets can be used directly in existing content tool-chains, and to facilitate this integration.

This goal led to the COMMON profile. The intent is that, if a user's tools can read a COLLADA asset and use the data presented by the common profile, the user should be able to use any DCC tool for content creation.

To facilitate the integration of COLLADA assets into tool chains, it appears that COLLADA must 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 with specific data structures used by existing content tool chains.

COLLADA can enable the 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.

Be Adopted by Many Digital-Content Users

Goal: To be adopted by as many digital-content users as possible.

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 the developer with the right information and enable the measurement of the quality of COLLADA tools. This includes:

- 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 the “Tool Requirements and Options” chapter.)
- 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 importers so that those tasks can be incorporated into an automated build process.

Provide an Easy Integration Mechanism

Goal: 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. This leads to the following goals:

- Design the COLLADA API and future enhancements to 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 in the 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 metadata.
- Enforce the asset metadata usage in exporters and importers.

Serve as Basis for Common Data Exchange

Goal: To be a basis for common data exchange among 3-D packages.

The biggest consequence of this goal is that the COLLADA common profile will be an ongoing exercise. Currently, it covers polygon-based models, materials and shaders, and some animations and DAG-based scene graphs. In the future it will cover NURBS, subdivision surfaces, and other, more complex data types in a common way that makes exchanging that information among tools a possibility.

Catalyze Digital Asset Schema Design

Goal: To be a catalyst for digital-asset schema design among developers and DCC, hardware, and middleware vendors.

There is a fierce competition among and within market segments: the DCC vendors, the hardware vendors, the middleware vendors, and game developers. But all 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 among the tool chains.
- DCC vendors suffer from the amount of support and specific development required to make developers happy.
- Developers suffer 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 that everybody accept 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. Note that:

- 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 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 and XMLSPY® from ALTOVA GmbH. are being used to iteratively develop and validate the schema for the format. In addition, XMLSPY® from Altova GmbH. is being used to validate files against the COLLADA schema and to create drawings for documentation During the design phase, the Microsoft Visual Studio® XML Designer and XMLSPY® from ALTOVA GmbH. are being used to iteratively develop and validate the schema for the format. In addition, XMLSPY® from Altova GmbH. is being used to validate files against the COLLADA schema and to create drawings for documentation.

Chapter 2:
Tool Requirements and Options

Introduction

Any fully compliant COLLADA tool must support the entire specification of data represented in the schema. What may not 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 that does not support layers, so it would not be required to export layer information (assuming that the export of such layer information is normally 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 that probably would be valuable for some subset of anticipated users as advanced or esoteric options.

The requirements explored in this chapter are placed on tools 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 Conformance 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

Data	Must be possible to export
Translation	Translations
Scaling	Scales
Rotation	Rotations
Parenting	Parent relationships
Static object instantiation	Instances of static objects. Such an object can have multiple transforms
Animated object instantiation	Instances of animated objects. Such an object can have multiple transforms
Skewing	Skews

Data	Must be possible to export
Transparency / reflectivity	Additional material parameters for transparency and reflectivity
Texture-mapping method	A texture-mapping method (cylindrical, spherical, etc.)
Transform with no geometry	It must be possible to transform something with no geometry (e.g., locator, NULL)

Materials and Textures

Data	Must be possible to export
RGB textures	An arbitrary number of RGB textures
RGBA textures	An arbitrary number of RGBA textures
Baked Procedural Texture Coordinates	Baked procedural texture coordinates
Common profile material	A common profile material (**`PHONG`**, **`LAMBERT`**, etc.)
Per-face material	Per-face materials

Vertex Attributes

Data	Must be possible to export
Vertex texture coordinates	An arbitrary number of Texture Coordinates per vertex
Vertex normals	Vertex normals
Vertex binormals	Vertex binormals
Vertex tangents	Vertex tangents
Vertex UV coordinates	Vertex UV coordinates (distinct from texture coordinates)
Vertex colors	Vertex colors
Custom vertex attributes	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 the tool that exported the data, 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

Scene Data

Data	Must be possible to export
Empty nodes	Empty nodes
Cameras	Cameras
Spotlights	Spotlights
Directional lights	Directional lights
Point lights	Point lights
Ambient lights	Ambient lights

Exporter User Interface Options

Data	Must be possible to export
Export triangle list	Triangle lists
Export polygon list	Polygon lists
Bake matrices	Baked matrices
Single **<matrix>** element	An instance document that contains only a single **<matrix>** element for each node. (See the following “Single <matrix> Element Option” discussion.)

Single <matrix> Element Option

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 that 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 (for example, Cg, GLSL, HLSL).

Importers

Scope

The responsibility of a COLLADA importer is to read all the specified data according to certain essential options.

In general, importers should provide perfect inverse functions of everything that a corresponding exporter does. Importers must provide the inverse function operation of every export option described in the “Exporters” section where it is possible to do so. This section describes only issues where the requirements placed on importers diverge or need clarification from the obvious inverse method of exporters.

Requirements

It must be possible to import all conforming 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.

Chapter 3: Schema Concepts

Introduction

The COLLADA schema is an eXtensible Markup Language (XML) database schema. The XML Schema language is used to describe the COLLADA feature set.

Documents that use the COLLADA schema – that is, that contain XML elements that are valid according to the COLLADA schema – are called COLLADA instance documents. The file extension chosen for these documents is ".dae" (an acronym for Digital Asset Exchange). When an Internet search was completed, no preexisting usage was found.

This chapter briefly introduces basic XML terminology, describes how COLLADA elements can refer to other COLLADA elements, and provides additional conceptual information about how COLLADA works.

XML Overview

XML provides a standard language for describing the content, structure, and semantics of files, documents, or datasets. An XML document consists primarily of *elements*, which are blocks of information surrounded by start and end *tags*. For example:

```
<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>
```

This contains four elements: **<node>**, **<translate>**, **<rotate>**, and **<matrix>**. The latter three elements are nested within the **<node>** element; elements can be nested to an arbitrary depth.

Elements can have attributes, which describe some aspect of the element. For example, the id attribute of the **<node>** element in the example has the value "here"; this might differentiate it from another **<node>** element whose id is "there". In this case, the attribute's name is id; its value is here.

For additional information about XML vocabulary, see the "Glossary."

Address Syntax

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

- URI addressing: 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.
- COLLADA target addressing: The `target` attributes of 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.

URI Address Referencing

URI Fragment Identifier

Many COLLADA elements have an `id` attribute. These elements can be addressed using the Uniform Resource Identifier (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 addresses only 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 preceded with the pound sign (#). In a `target` attribute, there is no pound sign because it is not a URI. For example, the same **`<source>`** element is addressed as follows using each notation:

```
<source id="here" />
<input source="#here" />
<bind target="here" />
```

For example, within a COLLADA instance document, a light defined with the ID “Lt-Light” can be referenced using **`<instance_light url = "#Lt-Light">`**. In the following example, the light node element refers to the light element found in the light library.

```
<library_lights>
  <light id="Lt-Light" name="light">
    <technique_common>
      <ambient
        <color>1 1 1</color>
      </ambient>
    </technique_common>
  </light>
</library>
...
<node id="Light" name="Light">
  <translate>-5.000000 10.000000 4.000000</translate>
  <rotate>0 0 1 0</rotate>
  <rotate>0 1 0 0</rotate>
  <rotate>1 0 0 0</rotate>
  <instance_light url="#Lt-Light" />
</node>
```

URI Path Syntax

The syntax of URIs is defined in the Internet Engineering Task Force (IETF) RFC 3986, “Uniform Resource Identifier (URI): Generic Syntax,” available at http://www.ietf.org/rfc/rfc3986.txt. It defines a URI as consisting of five hierarchical parts: the scheme, authority, path, query, and fragment. In BNF, the syntax is:

```
scheme ":" hierarchy-part ["?" query ] ["#" fragment ]
hierarchy-part =       "//" authority path-abempty
                / path-absolute
                / path-rootless
                / path-empty
```

The scheme and the path are required. The path, however, can be empty.

URI syntax requires that pathname levels (such as directories) be separated with slashes (/) and that other special characters within the path be escaped, for example, converting spaces to their hexadecimal representation %20. An absolute path that does not conform to IETF format must be adjusted to do so. For example, the absolute Windows path “C:\foo\bar\my file#27.dae”, by URI syntax definition, could be interpreted as a relative path (starting with “C”) to the current base URI—and, furthermore, backslashes could be treated the same as any other text character, not as valid separators. Although some applications look for Windows paths and convert them to valid URIs, not all applications do. Therefore, always use valid URI syntax, which for this example would be “C:/foo/bar/my%20file%2327.dae”.

Note: Whenever possible, it is better coding practice to use paths that are relative to the location of the file that references them rather than absolute paths.

COLLADA Target Addressing

A scoped identifier (sid) is an **xs:NCName** with the added constraint that its value is unique within the scope of its parent element, among the set of sids at the same path level, as found using a breadth-first traveral. A sid might be ambiguous across **<technique>** elements.

The target attribute syntax has several parts:

- The first part is the id attribute of an element in the instance document or a dot segment (".") indicating that this is a relative address.
- Zero or more identifiers (sid) follow. Each is preceded by a literal slash (/) as a path separator; if this part is empty, then there is no literal slash. The scoped identifiers are taken from a child of the element identified by the first part. For nested elements, multiple scoped identifiers can 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 (for example, all values of a matrix). If this part is present, it can take one of two forms:
  - The name of the member value (field) indicating symbolic access. This notation consists of:
    - A literal period (.) indicating member selection access.
    - The symbolic name of the member value (field). The "Common Glossary" subsection later in this chapter 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 left parenthesis (( ) indicating array selection access.
    - A number of the field, starting at zero for the first field.
    - A literal right parenthesis ( )) closing the expression.

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

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>
```

Three of the **<channel>** elements target 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 or sid attributes to all in-between elements.

For example, you can target the Y of a camera without adding sid attributes for **<optics>** and the **<technique>** elements. Some elements don't even allow id and sid attributes.

It is also possible to target the yfov of that camera in multiple techniques without having 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="#YFOVSampler" target="Camera01/YFOV"/>
...
<camera id="#Camera01">
   <optics>
      <technique_common>
          <perspective>
          <yfov sid="YFOV">45.0</yfov>
          <aspect_ratio>1.33333</aspect_ratio>
          <znear>1.0</znear>
          <zfar>1000.0</zfar>
          </perspective>
      </technique_common>
      <technique profile="OTHER">
          <param sid="YFOV" type="float">45.0</param>
          <otherStuff type="MySpecialCamera">DATA</otherStuff>
      </technique>
   </optics>
</camera>
```

Notice that the same sid="YFOV" attribute is used even though the name of the parameter is different in each technique. This is valid.

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.

Instantiation and External Referencing

The actual data representation of an object might be stored only once. However, the object can appear in a 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. The family of **instance_*** elements enables a COLLADA document to instantiate COLLADA objects.

Each instance inherits the local coordinate system from its parent, including the applicable **<unit>** and **<up_axis>** settings, to determine its position, orientation, and scale.

Each instance of the object can be unique or can share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A nonunique (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 instantiation. When the mechanism to achieve this effect is external to the current scene or resource, it is called external referencing.

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

COLLADA contains several **instance_*** elements, which instantiate their related elements. For example, **<instance_animation>** describes an instance of **<animation>**. The `url` attribute of an instance element points to an element of the appropriate related type.

In core COLLADA, these elements are

- **<instance_animation>**
- **<instance_camera>**
- **<instance_controller>**
- **<instance_geometry>**
- **<instance_light>**
- **<instance_node>**
- **<instance_visual_scene>**

In COLLADA Physics, these elements are

- **<instance_force_field>**
- **<instance_physics_material>**
- **<instance_physics_model>**
- **<instance_physics_scene>**
- **<instance_rigid_body>**
- **<instance_rigid_constraint>**

In COLLADA FX, these elements are

- **<instance_effect>**
- **<instance_material>**

The Common Profile

The COLLADA schema defines **<technique>** elements that establish 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.[1]

One aspect of the COLLADA design is the presence of techniques for a common profile. The **<technique_common>** and **<profile_COMMON>** elements explicitly invoke this profile. All tools that parse COLLADA content must understand this common profile. Therefore, COLLADA provides a definition for the common profile.

Naming Conventions

The COLLADA common profile uses the following naming conventions for canonical names:

- Parameter names are uppercase. For example, the values for the **<param>** element's name attribute are all uppercase letters:

```
<param name="X" type="float"/>
```

[1] 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 types are lowercase when they correspond to a primitive type in the COLLADA schema, in the XML Schema, or in the C/C++ languages. Type names are otherwise intercapitalized. For example, the values for the **<param>** element's type attribute follow this rule:

```
<param name="X" type="float"/>
```

- Input and parameter semantic names are uppercase. For example, the values for the **<input>** and **<newparam>** elements' `semantic` attribute are all uppercase letters:

```
<input semantic="POSITION" source="#grid-Position"/>
<newparam sid="blah">
  <semantic>DOUBLE_SIDED</semantic>
  <float>1.0</float>
</newparam>
```

Common Profile Elements

The COLLADA common profile is declared by the **<technique_common>** or **<profile_COMMON>** elements. For example:

```
<technique_common>
<!-- This scope is in the common profile -->
</technique_common>
```

Elements that appear outside the scope of a **<technique_common>** element are not in any profile, much less the common profile. For example, an **<input>** element that appears within the scope of the **<polygons>** element is not in the common profile; rather, it is invariant to all techniques.

Example and Discussion on Techniques

```
<channel source="#YFOVSampler" target="Camera01/YFOV"/>
...
<camera id="#Camera01">
   <optics>
      <technique_common>
          <perspective>
          <yfov sid="YFOV">45.0</yfov>
          <aspect_ratio>1.33333</aspect_ratio>
          <znear>1.0</znear>
          <zfar>1000.0</zfar>
          </perspective>
      </technique_common>
      <technique profile="OTHER">
          <param sid="YFOV" type="float">45.0</param>
          <otherStuff type="MySpecialCamera">DATA</otherStuff>
      </technique>
   </optics>
</camera>
```

Note:

- All consuming applications must recognize **<technique_common>**. Information in this technique is designed to be used as the reliable fallback when no other technique is recognized by the current runtime.
- If an exporting application uses any **<technique>** elements, it must include a **<technique_common>**.
- All techniques in a specific location represent the same concept, object, or process, but might provide entirely different information for that representation depending on the target application.
- A consuming application can choose among the **<technique>**s; if it doesn't explicitly choose, **<technique_common>** is the default.

Common Glossary

This section lists the canonical names of parameters and semantics that are within the common profile. Also listed are the member-selection symbolic names for the `target` attribute addressing scheme.

The common **<param>** (core) `name` attribute and **<newparam>** semantic values are:

Value of name or semantic attribute	Type	Typical Context	Description	Default
A	**float**	**<material>**, **<texture>**	Alpha color component	N/A
ANGLE	**float**	**<animation>**, **<light>**	Euler angle	N/A
B	**float**	**<material>**, **<texture>**	Blue color component	N/A
DOUBLE_SIDED	Boolean	**<material>**	Rendering state	N/A
G	**float**	**<material>**, **<texture>**	Green color component	N/A
P	**float**	**<geometry>**	Third texture coordinate	N/A
Q	**float**	**<geometry>**	Fourth texture coordinate	N/A
R	**float**	**<material>**, **<texture>**	Red color component	N/A
S	**float**	**<geometry>**	First texture coordinate	N/A
T	**float**	**<geometry>**	Second texture coordinate	N/A
TIME	**float**	**<animation>**	Time in seconds	N/A
U	**float**	**<geometry>**	First generic parameter	N/A
V	**float**	**<geometry>**	Second generic parameter	N/A
W	**float**	**<animation>**, **<controller>**, **<geometry>**	Fourth Cartesian coordinate	N/A
X	**float**	**<animation>**, **<controller>**, **<geometry>**	First Cartesian coordinate	N/A
Y	**float**	**<animation>**, **<controller>**, **<geometry>**	Second Cartesian coordinate	N/A
Z	**float**	**<animation>**, **<controller>**, **<geometry>**	Third Cartesian coordinate	N/A

The common **<channel>** `target` attribute member selection values are:

Value of target attribute	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**	Third texture coordinate
Q	**float**	Fourth texture coordinate
R	**float**	Red color component
S	**float**	First texture coordinate
T	**float**	Second texture coordinate
TIME	**float**	Time in seconds
U	**float**	First generic parameter
V	**float**	Second generic parameter
W	**float**	Fourth Cartesian coordinate
X	**float**	First Cartesian coordinate
Y	**float**	Second Cartesian coordinate
Z	**float**	Third Cartesian coordinate

Recall that array index notation, using left and right parentheses, can be used to target vector and matrix fields.

Chapter 4: Programming Guide

Introduction

This chapter provides some detailed explanations for COLLADA programming.

Curve Interpolation

This section provides information to describe an unambiguous implementation of **<geometry>**/**<spline>** and **<animation>**/**<sampler>** curves.

Introduction

Both **<geometry>**/**<spline>** and **<animation>**/**<sampler>** define curves. The first represents curves that can be displayed; the second represents curves that are used to create animations.

COLLADA defines a semantic attribute for the **<input>** element that identifies the data needed for interpolating curves. The values for this attribute include **POSITION**, **INTERPOLATION**, **LINEAR_STEPS**, **INPUT**, **OUTPUT**, **IN_TANGENT**, **OUT_TANGENT**, and **CONTINUITY**. In addition, the **<Name_array>** within a source allows an application to specify the type of curve to be processed; the common profile defines the values **BEZIER**, **LINEAR**, **BSPLINE**, and **HERMITE**. This section describes how COLLADA uses these semantics and curve names.

Spline Curves (<geometry>/<spline>)

The COLLADA specification of animated curves (**<animation>**/**<sampler>**) is derived from the dataflow definition of the drawing primitive for cubic polynomial curves (**<geometry>**/**<spline>**).

A curve is defined in segments. Each segment is defined by two endpoints. Each endpoint of a segment is also the beginning point of the next segment. The endpoints for segment[i] are given by **POSITION** [i] and **POSITION**[i+1]. Therefore, a curve with n segments will have n+1 positions. Points can be defined in two or in three dimensions (2-D or 3-D).

The behavior of a curve between its endpoints is given by a specific interpolation method and additional coefficients. Each segment can be interpolated with a different method. By convention, the interpolation method for a segment is attached to the first point, so the interpolation method for segment[i] is stored in **INTERPOLATION**[i] . If an n-segment curve is open, then **INTERPOLATION**[v+1] is not used, but if the curve is closed (the endpoint of the last segment is connected to the beginning point of the first segment), then **INTERPOLATION**[n+1] is the interpolation method for this extra segment. The closed attribute of the **<spline>** element indicates whether the curve is closed (true) or open (false; this is the default).

LINEAR_STEPS is an optional **<input>** semantic that indicates how precisely a curve needs to be interpolated. In general, a complex curve inside a segment is done by approximations using a subdivision on line segments. The number of subdivisions is given by **LINEAR_STEPS**.

Here's how a spline definition might look in COLLADA:

```
<spline closed="true">
  <source id= "positions" >
     <!-- contains n+1 values -->  </source>
  <source id="interpolations" >
```

```
        <!-- contains n+1 values -->  </source>
    <source … >
        <!-- n+1 values --> </source>
    <source … >
        <!-- n+1 values --> </source>
    <control_vertices>
        <input semantic="POSITION" source = "#positions"/>
        <input semantic="INTERPOLATION" source="#interpolations"/>
        <input … <!--additional inputs depending on interpolation methods -->
```

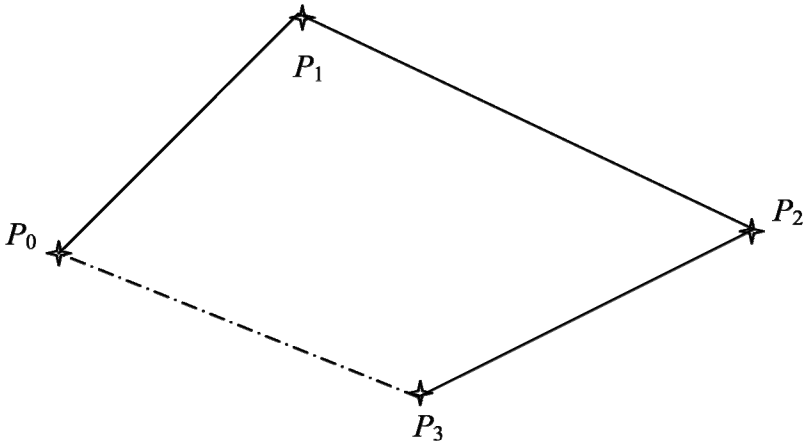
CONTINUITY is an optional **<input>** semantic that indicates how the tangents were constrained when the curve was designed. Valid **CONTINUITY** values are:

- C_0 : Point-wise continuous; curves share the same point where they join.
- C_1 : Continuous derivatives; curves share the same parametric derivatives where they join.
- G_1 (geometric continuity): Same as C_0 but also requires that tangents point in the same direction.

Linear Splines

Linear interpolation is the simplest; it means that the curve for the given segment is a straight line between the beginning and end points. It does not require any additional control points within a segment.

The following diagram illustrates a three-segment closed **<spline>** with **LINEAR** interpolation between each pair of the four positions (P_0, P_1, P_2, P_3). Because it is a closed spline, there is a final (fourth) segment between P_3 and P_0.



A linear spline equation is given by:

$$L(s) = P_0 + (P_1 - P_0)s, s \in [0,1]$$

Another way to represent this equation is to use the matrix form:

$$L(s) = SMC$$

$$S = \begin{bmatrix} s & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} P_0 & P_1 \end{bmatrix}$$

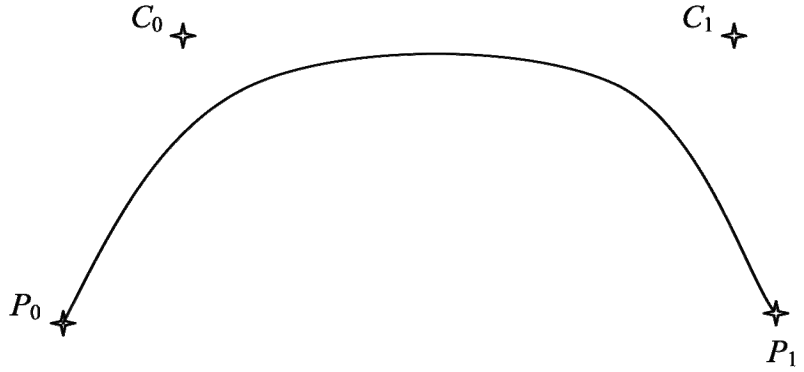
In COLLADA, a geometry vector for **LINEAR** segment[i] is defined by:

- P_0 is **POSITION**[i]
- P_1 is **POSITION**[i+1]

Bézier and Hermite Splines

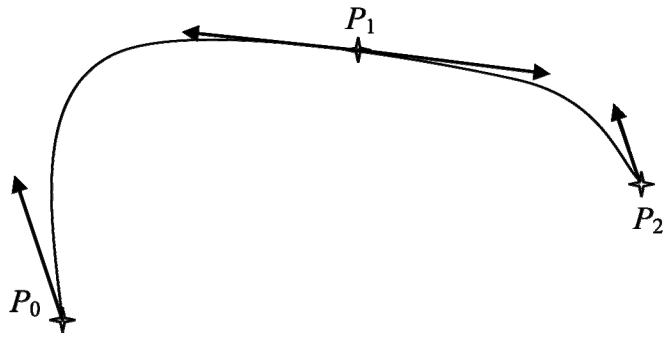
A segment can be a cubic Bézier spline, which requires two additional points to control the behavior of the curve. The following example shows one segment interpolated using **BEZIER**. It has the same beginning and end points as previously (P_0, P_1), but has two additional control points (C_0 and C_1) that provide the additional information to calculate the curve.

Note: COLLADA 1.4.1 supports only cubic Bézier curves, so there are always exactly two control points for each segment.



HERMITE is equivalent to **BEZIER**, but instead of providing the control points C_0 and C_1, tangents T_0 and T_1 are provided.

The following figure illustrates a two-segment curve with cubic Hermite interpolation:



Two tangents are attached to the point P_1. The tangent that defines the beginning of the second segment is called the **OUT_TANGENT**, because it is for the segment that begins by coming out of P_1. The tangent that defines the end of the first segment is called the **IN_TANGENT**, because this is for the segment that ends by going in to P_1.

In other words, **IN_TANGENT**[1] is the end tangent of segment[0] and **OUT_TANGENT**[1] is the beginning tangent of segment[1].

HERMITE and **BEZIER** are identical polynomial interpolations, with a variable change given by:

$$T_0 = 3(C_0 - P_0)$$

$$T_0 = 3(P_1 - C_1)$$

Equations are given as parametric equations. A parameter s that goes from 0 to 1 is used to calculate all the points on the curve. If s is 0, then the equation gives P_0; if s is 1, then the equation gives P_1. The equation is not defined outside of those values.

In COLLADA, the **<input>** semantics **IN_TANGENT** and **OUT_TANGENT** are used to store either the tangents or the control points depending on the interpolation method.

A cubic Bézier spline equation is given by:

$$B(s) = P_0(1-s)^3 + 3C_0 s(1-s)^2 + 3C_1 s^2(1-s) + P_1 s^3, s \in [0,1]$$

Another popular way of representing this equation is with the matrix form:

$$B(s) = SMC$$

$$S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} P_0 & C_0 & C_1 & P_1 \end{bmatrix}$$

In COLLADA, a geometry vector for Bézier segment[i] is defined by:

- P_0 is **POSITION**[i]
- C_0 is **OUT_TANGENT**[i]
- C_1 is **IN_TANGENT**[i+1]
- P_1 is **POSITION**[i+1]

Here's a COLLADA example of a 2-D (X,Y) Bézier curve with two segments:

```
<spline>
  <source id="positions">
       <float_array count="6" …>
       <technique_common><accessor>
        … <param name="X" offset="0" type="float" …
        … <param name="Y"  offset="1" type="float" …
        </accessor></technique_common>
  </source>
  <source id="interpolations" >
     <Name_array count="3"> BEZIER BEZIER BEZIER</Name_array>    <!-- last one
ignored for open curves -->
      <technique_common><accessor>
      … <param name="INTERPOLATION" type="Name" …
       </accessor> </technique_common> </source>
  <source id="in_tangents" >
       <float_array count="6" …> (first one ignored for open curves)
       <technique_common><accessor>
        … <param name="X" offset="0" type="float" …
        … <param name="Y"  offset="1" type="float" …
          </accessor> </technique_common> </source>
  <source id="out_tangents">
        <float_array count="6" …> <!-- last one ignored for open curves -->
        <technique_common><accessor>
         … <param name="X" offset="0" type="float" …
         … <param name="Y" offset="1" type="float" …
          </accessor></technique_common> </source>
  <control_vertices>
      <input semantic="POSITION" source = "#positions"/>
      <input semantic="INTERPOLATION" source="#interpolations"/>
      <input semantic="IN_TANGENT" source="#in_tangents"/>
      <input semantic="OUT_TANGENT" source="#out_tangents"/>
```

A cubic Hermite spline equation is given by:

$$H(s) = P_0(2s^3 - 3s^2 + 1) + T_0(s^3 - 2s^2 + s) + P_1(-2s^3 + 3s^2) + T_1(s^3 - s^2), s \in [0,1]$$

In its matrix form, this is:

$$H(s) = SMC$$

$$S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} P_0 & P_1 & T_0 & T_1 \end{bmatrix}$$

$$where:$$

$$T_0 = 3(C_0 - P_0)$$

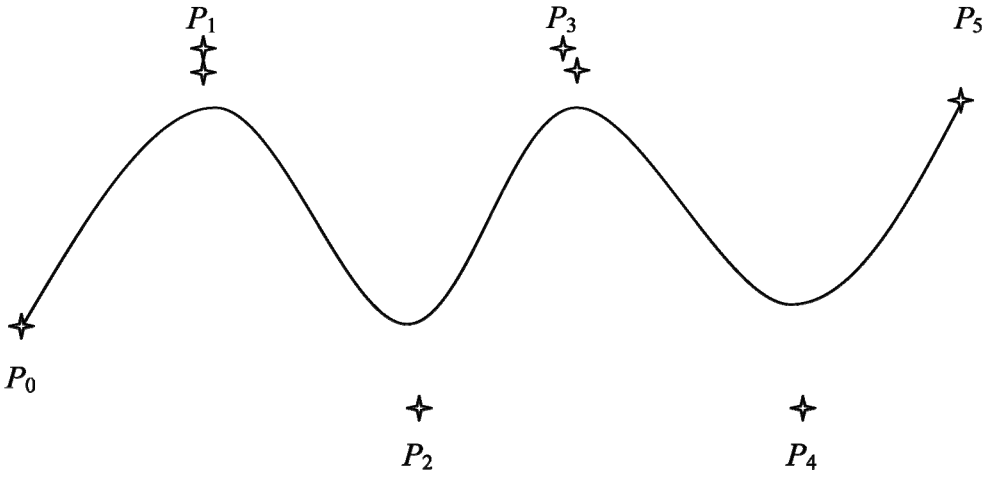
$$T_1 = 3(P_1 - C_1)$$

In COLLADA, a geometry vector for **HERMITE** segment[*i*] is defined by:

- P_0 is **POSITION**[*i*]
- P_1 is **POSITION**[*i*+1]
- T_0 is **OUT_TANGENT**[*i*]
- T_1 is **IN_TANGENT**[*i*+1]

B-Splines

Basis splines (B-splines) are defined by a series of control points, for which the curve is guaranteed only to go through the first and the last point, such as in the following figure:



COLLADA 1.4.1 defines uniform cubic B-spline interpolations. The behavior of a curve between the endpoints P_1 and P_2 is given by the following equation, using the previous (P_0) and next (P_2) points. For an open curve, P_0 does not have a previous point; therefore, the mirror of P_1 through P_0 is used. The same logic applies to the last point; the mirror of P_4 through P_5 is used for the equation.

A B-spline is described by the following matrix-form equations, for the segment going from P_i to P_{i+1}.

$$B_i(s) = SMC$$

$$S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix}$$

$$M = u * \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

$$u = 1/6$$

$$C = \begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix}$$

In COLLADA, defining this B-spline geometry vector requires using only two **<input>** elements, **POSITION** and **INTERPOLATION**, with:

- P_i = **POSITION**[i]
- **INTERPOLATION**[i]=**BSPLINE**.

Cardinal Splines

The cardinal spline is a cubic Hermite spline whose tangents are defined by the endpoints and a tension parameter. The tangents are calculated with the previous and the next point following the segment:

$T_i = ½ (1-c) (P_{i+1} - P_{i-1})$

where c is the tension parameter, which is a constant that modifies the length of the tangent. This parameter is not specified separately in COLLADA 1.4 and is instead baked into the tangents that are provided by the **OUT_TANGENT** and **IN_TANGENT** inputs to the sampler.

The cardinal spline can be put into matrix form, using the same geometry vector C as for the **BSPLINE**:

$$D_i(s) = SMC$$

$$S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix}$$

$$t = (1-c)/2$$

$$M = \begin{bmatrix} -t & 2-t & t-2 & t \\ 2t & t-3 & 3-2t & 1-t \\ -t & 0 & t & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix}$$

Skinning a Skeleton in COLLADA

Skinning is a technique for deforming geometry by linearly weighting vertices to the joints (also called bones) of a skeleton. This section provides a description of and equations for skinning a skeleton in COLLADA.

Overview

A skinning **<controller>** associates a geometry with a skeleton. The skeleton is considered to be in its resting position, or bind pose. The bind pose is the world-space position and orientation of each joint when the skeleton was bound to the geometry. This world space is also called the bind-pose space to distinguish it from other world-space coordinate systems.

A skinning **<instance_controller>** instantiates a skinning **<controller>** and associates it with a run-time skeleton. COLLADA defines skinning in object space, so the **<instance_controller>**'s placement in the **<node>** hierarchy contributes to the final vertex location. Object-space skinning provides the maximum amount of flexibility. The output of object-space skinning is vertices in the object space of the **<node>** coordinate system that contains the **<instance_controller>**.

When vertices are skinned in object space, it is easy and efficient to render the same skinned geometry in multiple locations. This is important when multiple actors are displayed simultaneously in the same pose but in different locations. Events like this happen most frequently in the animation of large crowds, parallel machines, and multiactor choreography. Each actor in the same pose shares the same skinned vertex data.

Skinning Definitions

Definitions related to skinning in COLLADA:

- Bind shape: The vertices of the mesh referred to by the source attribute of the **<skin>** element.
- Bind-shape matrix: A single matrix that represents the transform of the bind-shape at the time when the mesh was bound to a skeleton. This matrix transforms the bind-shape from object space to bind-space.
- Joints: The bones of a skeleton are defined by their joints; the base of each bone extends to the next joint. In bind space, joints are in their bind pose: the position and orientation at the time the joints of the skeleton were bound to the bind shape. In the **<visual_scene>**, the joints are oriented according to the poses and animations of the actor. The world-space location of the joints may not directly match the mesh; it is dependent on the root matrix used to convert the mesh back into object-space.
- Weights: How much a joint influences the final destination of a vertex. A vertex is typically weighted to one or more joints, where the sum of the weights equals 1. A vertex is transformed by each joint independently. The multiply transformed vertex results are linearly combined according to their weights to generate the skinned vertex.
- Inverse bind-pose matrix: The inverse of the joint's bind-space transformation matrix at the time the bind shape was bound to this joint.

Skinning Equations

The skinning calculation for each vertex v in a bind shape is

$$outv = \sum_{i=0}^{n} \left\{ \left(\left(v * \mathrm{BSM} \right) * \mathrm{IBM}i * \mathrm{JM}i \right) * \mathrm{JW} \right\}$$

where:

- n: The number of joints that influence vertex v
- BSM: Bind-shape matrix
- IBMi: Inverse bind-pose matrix of joint i
- JMi: Transformation matrix of joint i
- JW: Weight of the influence of joint i on vertex v

Note: v, BSM, IBMi, and JW are constants with regards to some skeletal animation. Depending on your application, it may be beneficial to premultiply BSM with IBMi or v with BSM.

Equation Notes

The main difference between world-space skinning and object-space skinning lies in the definition of JMi:

- For world-space skinning, $\mathrm{JM}i$ is the transformation matrix of the joint from object space to world space.
- For object-space skinning, $\mathrm{JM}i$ is a transformation matrix of the joint from object space to another object space. The first object-space transformation is the geometry's object-space transformation where the bind shape was originally defined. The second object-space transformation is the destination object space, which is selected by the **<instance_controller><skeleton>**.

It is easiest to conceptualize this transformation by considering the other spaces that may fall between these spaces to construct this final matrix. One method is to go from geometry object space to world space as you might see with world-space skinning, then transform from world space to the skeleton's object space using the inverse of the skeleton's world-space matrix.

It is important to note that the skeleton's matrix referred to here is not the bind-shape matrix. It is the **<node>** in the **<visual_scene>** referenced by **<instance_controller><skeleton>** and that might not have the same values. Using the **<node>** referenced by **<instance_controller><skeleton>** provides maximum flexibility for locating and animating skeletons in the scene. It removes all restrictions over the bind space of the skin and the object space of the skeleton in the scene. This is because the animation is always relative to what you pick as the root node.

If you were to hypothetically use the bind-shape matrix instead, then the skeleton would always have to be located and animated relative to the bind-shape matrix's location and orientation in the scene. If you are animating multiple characters at once, this can be disorienting because there is a high probability of overlap. It is worth noting that the node's world-space matrix, referenced by **<instance_controller><skeleton>**, can be equal to a skin's bind-shape matrix and that would match the behavior just mentioned; or it can be equal to an identity matrix to match the behavior of world-space skinning. Enabling these options makes object-space skinning the most flexible model.

The result of the preceding equation is a vertex in skeleton-relative object space, so it must still be multiplied by a transform from object space to world space to produce the final vertex. This last step is typically done in a vertex shader and this matrix is the world-space transformation matrix for the node that owns the **<instance_controller>**.

There is a simple trick to animating a skeleton and its **<instance_controller>** simultaneously. If you place the **<instance_controller>** inside the root of **<skeleton>** then the last two matrices cancel each other, which gives a solution much like world-space skinning. The mesh will always follow the skeleton.

Chapter 5: COLLADA Core Elements Reference

Introduction

This section covers the core elements that represent the basic functionality and infrastructure of the COLLADA schema, outside of the effects (FX) and physics frameworks. This includes elements that describe geometry, animation, skinning, assets, and scenes.

Elements by Category

This chapter lists elements in alphabetical order. The following tables list elements by category, for ease in finding related elements.

Animation

animation	Categorizes the declaration of animation information.
animation_clip	Defines a section of the animation curves to be used together as an animation clip.
channel	Declares an output channel of an animation.
instance_animation	Declares the instantiation of a COLLADA animation resource.
library_animation_clips	Declares a module of **<animation_clip>** elements.
library_animations	Declares a module of **<animation>** elements.
sampler	Declares an interpolation sampling function for an animation.

Camera

camera	Declares a view into the scene hierarchy or scene graph. The camera contains elements that describe the camera's optics and imager.
imager	Represents the image sensor of a camera (for example, film or CCD).
instance_camera	Declares the instantiation of a COLLADA camera resource.
library_cameras	Declares a module of **<camera>** elements.
optics	Represents the apparatus on a camera that projects the image onto the image sensor.
orthographic	Describes the field of view of an orthographic camera.
perspective	Describes the field of view of a perspective camera.

Controller

controller	Categorizes the declaration of generic control information.
instance_controller	Declares the instantiation of a COLLADA controller resource.
joints	Declares the association between joint nodes and attribute data.
library_controllers	Declares a module of **<controller>** elements.
morph	Describes the data required to blend between sets of static meshes.
skeleton	Indicates where a skin controller is to start to search for the joint nodes that it needs.
skin	Contains vertex and primitive information sufficient to describe blend-weight skinning.

targets	Declares morph targets, their weights, and any user-defined attributes associated with them.
vertex_weights	Describes the combination of joints and weights used by a skin.

Data Flow

accessor	Declares an access pattern to one of the array elements **<float_array>**, **<int_array>**, **<Name_array>**, **<bool_array>**, and **<IDREF_array>**.
bool_array	Declares the storage for a homogenous array of Boolean values.
float_array	Declares the storage for a homogenous array of floating-point values.
IDREF_array	Declares the storage for a homogenous array of ID reference values.
int_array	Stores a homogenous array of integer values.
Name_array	Stores a homogenous array of symbolic name values.
param (core)	Declares parametric information for its parent element.
source	Declares a data repository that provides values according to the semantics of an **<input>** element that refers to it.
input (indexed)	Declares the input semantics of a data source.
input (unindexed)	Declares the input semantics of a data source.

Extensibility

extra	Provides arbitrary additional information about or related to its parent element.
technique (core)	Declares the information used to process some portion of the content. Each technique conforms to an associated profile.
technique_common	Specifies the information for a specific element for the common profile that all COLLADA implementations must support.

Geometry

control_vertices	Describes the control vertices (CVs) of a spline.
geometry	Describes the visual shape and appearance of an object in a scene.
instance_geometry	Declares the instantiation of a COLLADA geometry resource.
library_geometries	Declares a module of **<geometry>** elements.
lines	Declares the binding of geometric primitives and vertex attributes for a **<mesh>** element.
linestrips	Declares a binding of geometric primitives and vertex attributes for a **<mesh>** element.
mesh	Describes basic geometric meshes using vertex and primitive information.
polygons	Declares the binding of geometric primitives and vertex attributes for a **<mesh>** element.
polylist	Declares the binding of geometric primitives and vertex attributes for a **<mesh>** element.
spline	Describes a multisegment spline with control vertex (CV) and segment information.
triangles	Declares the binding of geometric primitives and vertex attributes for a **<mesh>** element. (or) Provides the information needed to bind vertex attributes together and then organize those vertices into individual triangles.
trifans	Declares the binding of geometric primitives and vertex attributes for a **<mesh>** element. (or) Provides the information needed to bind vertex attributes together and then organize those vertices into connected triangles.

tristrips	Declares the binding of geometric primitives and vertex attributes for a **<mesh>** element. (or) Provides the information needed to bind vertex attributes together and then organize those vertices into connected triangles
vertices	Declares the attributes and identity of mesh-vertices.

Lighting

ambient (core)	Describes an ambient light source.
color	Describes the color of its parent light element.
directional	Describes a directional light source.
instance_light	Declares the instantiation of a COLLADA light resource.
library_lights	Declares a module of **<image>** elements.
light	Declares a light source that illuminates a scene.
point	Describes a point light source.
spot	Describes a spot light source.

Metadata

asset	Defines asset-management information regarding its parent element.
COLLADA	Declares the root of the document that contains some of the content in the COLLADA schema.
contributor	Defines authoring information for asset management.

Scene

instance_node	Declares the instantiation of a COLLADA node resource.
instance_visual_scene	Declares the instantiation of a COLLADA **visual_scene** resource.
library_nodes	Declares a module of **<node>** elements.
library_visual_scenes	Declares a module of **<visual_scene>** elements.
node	Embodies the hierarchical relationship of elements in a scene.
scene	Embodies the entire set of information that can be visualized from the contents of a COLLADA resource.
visual_scene	Embodies the entire set of information that can be visualized from the contents of a COLLADA resource.

Transform

lookat	Contains a position and orientation transformation suitable for aiming a camera.
matrix	Describes transformations that embody mathematical changes to points within a coordinate system or the coordinate system itself.
rotate	Specifies how to rotate an object around an axis.
scale	Specifies how to change an object's size.
skew	Specifies how to deform an object along one axis.
translate	Changes the position of an object in a coordinate system without any rotation.

accessor

Category: **Data Flow**

Introduction

Describes a stream of values from an array data source.

Concepts

The **<accessor>** element declares an access pattern into one of the array elements **<float_array>**, **<int_array>**, **<Name_array>**, **<bool_array>**, and **<IDREF_array>** or into an external array source. The arrays can be organized in either an interleaved or noninterleaved manner, depending on the **offset** and stride attributes.

The output of the accessor is described by its child **<param>** elements.

Attributes

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

count	**uint**	The number of times the array is accessed. Required.
offset	**uint**	The index of the first value to be read from the array. The default is 0. Optional.
source	**xs:anyURI**	The location of the array to access using a URI expression. Required. This element may refer to a COLLADA array element or to an array data source outside the scope of the instance document; the source does not need to be a COLLADA document.
stride	**uint**	The number of values that are to be considered a unit during each access to the array. The default is 1, indicating that a single value is accessed. Optional.

Related Elements

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

Parent elements	**source** / **technique_common**
Child elements	See the following subsection.
Other	**bool_array**, **float_array**, **IDREF_array**, **int_array**, **Name_array**, **mesh**, **convex_mesh**

Child Elements

Name/example	Description	Default	Occurrences
<param> (core)	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 **IDREF**. See main entry.	N/A	0 or more

Details

The number and order of **<param>** elements define the output of the **<accessor>** element. Parameters are bound to values in the order in which both are specified. No reordering of the data can occur. A **<param>** element without a name attribute indicates that the value is not part of the output, so the element is unbound.

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

Example

Here is an example of a basic **<accessor>** element:

```
<source>
  <int_array name="values" count="9">
    1 2 3 4 5 6 7 8 9
  </int_array>
  <technique_common>
              <accessor source="#values" count="9">
    <param name="A" type="int"/>
  </accessor>
  </technique_common>
</source>
```

default stride = 1										
	array offset →	0	1	2	3	4	5	6	7	8
	values →	1	2	3	4	5	6	7	8	9
accessor count	1st pass	A								
	2nd pass		A							
	3rd pass			A						
	etc.									
	9th pass									A

Here is an example of an **<accessor>** element that describes a stream of three 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 0 1 2 0 2 3 0 3
  </int_array>
  <technique_common>
    <accessor source="#values" count="3" stride="3">
      <param name="A" type="int"/>
      <param type="int"/>
      <param name="B" type="int"/>
    </accessor>
  </technique_common>
</source>
```

stride = 3										
	array offset →	0	1	2	3	4	5	6	7	
	values →	1	0	1	2	0	2	3	0	
accessor count	1st pass	A		B						
	2nd pass				A		B			
	3rd pass							A		

Here is another example showing every third value being skipped because there is no **<param>** element binding it to the output although 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_common>
    <accessor source="#values" count="3" stride="3">
      <param name="B" type="int"/>
      <param name="A" type="int"/>
    </accessor>
  </technique_common>
</source>
```

	stride = 3									
	array offset →	0	1	2	3	4	5	6	7	
	values →	1	1	0	2	2	0	3	3	
accessor count	1st pass	B	A							
	2nd pass				B	A				
	3rd pass							B	A	

In this example, every third value is skipped because there is no **<param>** element binding it to the output, although the stride attribute is still three. It also disregards the first three values (because the **offset**=3 begins at the fourth value) and the last three values (because the **count** is only 4, so only 12 values are read):

```
<source>
   <int_array name="values" count="18">
      1 1 0 2 2 0 3 3 0 4 4 0 5 5 0 6 6 0
   </int_array>
   <technique_common>
      <accessor source="#values" offset="3" count="4" stride="3">
         <param name="A" type="int"/>
         <param name="C" type="int"/>
      </accessor>
   </technique_common>
</source>
```

	stride = 3																	
	array offset →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	values →	1	1	0	2	2	0	3	3	0	4	4	0	5	5	0	6	6	0
accessor count	1st pass				A	C													
	2nd pass							A	C										
	3rd pass										A	C							
	4th pass													A	C				

Semantics in an **<input>** imply a specific data ordering in a source (such as X, Y, Z or R, G, B); the actual names of the **<param>**s in the **<source>**'s **<accessor>** are not significant. The names in **<param>**s do not imply any kind of binding, but the absence of a name or whole **<param>** (if it is at the end of the list) indicates data to be skipped.

To properly read a source through an **<accessor>**, the program has to consider the data expected by a particular semantic and compare it to stride, **offset**, and the number of params with nonnull names to decide how many values to read. Then, when reading, it has to skip over the data that corresponds to **<param>**s with no name.

Assume that your program has a vertex-map-like array that it is trying to fill with geometry:

```
struct
{
    float x_pos, y_pos, z_pos, x_norm, y_norm, z_norm, text1_U,
    tex1_V, tex2_U tex2_V;
} my_array[1000];
```

Given this source:

```
<source id=test1>
  <float_array name="values" count="9">
    1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
  </ float_array>
  <technique_common>
    <accessor source="#values" count="3" stride="3">
      <param name="A" type="float"/>
      <param name="F" type="float"/>
      <param name="X" type="float"/>
    </accessor>
  </technique_common>
</source>
```

with the following input:

```
<triangles count="1">
    <input semantic="POSITION" source="#test1" offset="0"/>
    <p>0 1 2</p>
```

If you read the data into **my_array** sequentially, because the stride of the accessor is 3 and all the **<param>**s have names, the **<source>** is assumed to contain 3D positions and **my_array** would be filled in like this:

x_pos	y_pos	z_pos	x_norm	y_norm	z_norm	tex1_U	tex1_V	tex2_U	tex2_V
1.0	2.0	3.0							
4.0	5.0	6.0							
7.0	8.0	9.0							

Change the accessor to this:

```
<accessor source="#values" count="3" stride="3">
   <param name="A" type="float"/>
   <param type="float"/>
   <param name="X" type="float"/>
</accessor>
```

Because the second **<param>** has no name, it is skipped. With only two named **<param>**s, the **<source>** is assumed to contain 2D positions and is read like this:

x_pos	y_pos	z_pos	x_norm	y_norm	z_norm	tex1_U	tex1_V	tex2_U	tex2_V
1.0	3.0								
4.0	6.0								
7.0	9.0								

Now, if you wanted to pack the equivalent of an entire vertex array into one float array:

```
<source id=positions>
   <float_array name="values" count="30">
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30
   </ float _array>
   <technique_common>
```

```
      <accessor source="#values" count="3" stride="10">
         <param name="A" type="float"/>
         <param name="F" type="float"/>
         <param name="X" type="float"/>
      </accessor>
   </technique_common>
</source>
<source id=normals>
   <technique_common>
      <accessor source="#values" count="3" stride="10">
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param name="A" type="float"/>
         <param name="F" type="float"/>
         <param name="X" type="float"/>
      </accessor>
   </technique_common>
</source>
<source id=texture1>
   <technique_common>
      <accessor source="#values" count="3" stride="10">
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param name="A" type="float"/>
         <param name="F" type="float"/>
      </accessor>
   </technique_common>
</source>
<source id=texture2>
   <technique_common>
     <accessor source="#values" count="3" stride="10">
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param type="float"/>
         <param name="F" type="float"/>
         <param name="X" type="float"/>
      </accessor>
   </technique_common>
</source>

<triangles count="1">
   <input semantic="POSITION" source="#positions" offset="0"/>
   <input semantic="NORMAL" source="#normals" offset="0"/>
   <input semantic="TEXCOORD" source="#texture1" offset="0"/>
   <input semantic="TEXCOORD" source="#texture2" offset="0"/>
   <p>1 2 3</p>
```

Based on the **<param>** count in each **<accessor>**, you would be assuming that you were reading 3D positions, 3D normals, and 2D texture coordinates.

x_pos	y_pos	z_pos	x_norm	y_norm	z_norm	tex1_U	tex1_V	tex2_U	tex2_V
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	17.0	18.0	19.0	20.0
21.0	22.0	23.0	24.0	25.0	26.0	27.0	28.0	29.0	30.0

Note that you can also use the **<accessor>** offset attribute to skip leading fields of data. For example, the **<accessor>** in the source id=texture1 in the preceding example could be written the following way and it would work the same:

```
<accessor source="#values" count="3" stride="10" offset="6">
   <param name="A" type="float"/>
   <param name="F" type="float"/>
</accessor>
```

ambient

(core)

Category: **Lighting**

Introduction

Describes an ambient light source.

For information about the **<ambient>** element in FX elements, see "**Note**."

Concepts

The **<ambient>** element declares the parameters required to describe an ambient light source. An ambient light is one that lights everything evenly, regardless of location or orientation.

Attributes

The **<ambient>** element has no attributes.

Related Elements

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

Parent elements	**light** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<color>	Contains three floating-point numbers specifying the color of the light. See main entry.	None	1

Details

Example

Here is an example of an **<ambient>** element:

```
<light id="blue">
  <technique_common>
    <ambient>
      <color>0.1 0.1 0.5</color>
    </ambient>
  </technique_common>
</light>
```

animation

Category: **Animation**

Introduction

Categorizes the declaration of animation information.

Concepts

The animation hierarchy contains elements that describe the animation's key-frame data and sampler functions, ordered in such a way as to group animations that should be executed together.

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 key-frame animation.

A *key frame* is a two-dimensional (2-D) sampling of data. The first dimension is called the input and is usually time, but can be any other real value. The second dimension is called the output and represents the value being animated. Using a set of key frames and an interpolation algorithm, intermediate values are computed for times between the key frames, producing a set of output values over the interval between the key frames. The set of key frames and the interpolation between them define a 2-D function called an *animation curve* or *function curve*, represented by an **<animation>** element.

For more information about interpolating animation curves, see "Curve Interpolation" in Chapter 4: Programming Guide.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the **<animation>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_animations**, **animation**
Child elements	See the following subsection.
Other	**instance_animation**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<animation>	Allows the formation of a hierarchy of related animations. See main entry.	N/A	0 or more (see "Details")
<source> (core)	See main entry.	N/A	0 or more (see "Details")
<sampler>	Describes the interpolation sampling function for the animation. See main entry.	N/A	0 or more (see "Details")

Name/example	Description	Default	Occurrences
<channel>	Describes an output channel for the animation. See main entry.	None	0 or more (see "Details")
<extra>	See main entry.	N/A	0 or more

Details

An **<animation>** element contains the elements that describe animation data to form an animation tree. The actual type and complexity of the data is represented in detail by the child elements.

The child elements follow these rules:

- The **<animation>** element must contain at least one of the following:
  - **<animation>**
  - **<sampler>** and **<channel>**
- **<sampler>** and **<channel>** must always be used together.

<animation>s that are not referenced by **<animation_clip>** elements can be applied to the scene at playback time; otherwise, see **<animation_clip>**s for playback details.

Example

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

```
<library_animations>
  <animation name="walk" id="Walk123">
      <source />
      <source />
      <sampler />
      <channel />
  </animation>
</library_animations>
```

This next example describes a simple animation tree defining a "jump" animation:

```
<library_animations>
  <animation name="jump" id="jump">
      <animation id="skeleton_root_translate">
          <source/><source/><sampler/><channel/>
      </animation>
      <animation id="left_hip_rotation">
          <source/><source/><sampler/><channel/>
      </animation>
      <animation id="left_knee_rotation">
          <source/><source/><sampler/><channel/>
      </animation>
      <animation id="right_hip_rotation">
          <source/><source/><sampler/><channel/>
      </animation>
      <animation id="right_knee_rotation">
          <source/><source/><sampler/><channel/>
      </animation>
  </animation>
</library_animations>
```

The next example shows a more complex animation tree, with some of the animations left undefined.

```
<library_animations>
  <animation name="elliots_animations" id="all_elliot">
      <animation name="elliot's spells" id="spells_elliot">
```

```
            <animation id="elliot_fire_blast"/>
            <animation id="elliot_freeze_down"/>
            <animation id="elliot_ferocity"/>
        </animation>
        <animation name="elliot's moves" id="moves_elliot">
            <animation id="elliot_walk"/>
            <animation id="elliot_run"/>
            <animation id="elliot_jump"/>
        </animation>
    </animation>
</library_animations>
```

animation_clip

Category: **Animation**

Introduction

Defines a section of a set of animation curves to be used together as an animation clip.

Concepts

Animation clips can be used to separate different pieces of a set of animation curves. For example, an animation might have a character walk, then run. The walking and running animations can be separated as two different clips. Clips can also be used to separate the animations of different characters in the same scene, or even different parts of the same character (such as upper and lower body).

Currently, animation clips cannot be instantiated inside a COLLADA document. They are for use by engines and other tools.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the **<animation>** element. This value must be unique within the instance document. Optional.
start	**xs:double**	The time in seconds of the beginning of the clip. This time is the same as that used in the key-frame data and is used to determine which set of key frames will be included in the clip. The start time does not specify when the clip will be played. If the time falls between two key frames of a referenced animation, an interpolated value should be used. The default is 0.0. Optional.
end	**xs:double**	The time in seconds of the end of the clip. This is used in the same way as the start time. If end is not specified, the value is taken to be the end time of the longest animation. Optional.
name	**xs:NCName**	Optional.

Related Elements

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

Parent elements	**library_animation_clips**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<instance_animation>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Animation Targets and Scene

Two or more **<animation_clip>**s can refer to **<animation>**s that have the same target; in addition, it is possible to have an **<animation>** with the same target that is not referred to by an **<animation_clip>** but is meant to be applied and used when the playback occurs.

<animation>s that are referred to and used in **<animation_clip>**s should not be applied to the scene at playback time; instead, apply only unreferenced **<animation>**s to the scene (used for playback).

Note: Plug-in implementors must support this strategy even if they do not fully support **<animation_clip>**. For example, DCC tools can store the contents of **<library_animations>** and **<library_animation_clips>** in banks or palettes. Any unreferenced **<animation>** is left to be processed according to the application run-time; these are the ones to load and play.

Example

Here is an example of two **<animation_clip>** elements with the allowed attributes:

```
<library_animation_clips>
  <animation_clip id="GuyWalking" start="0.25" end="1.25">
    <instance_animation url="#Guy1MoveAnim"/>
  </animation_clip>
  <animation_clip id="GuyRunning" start="2.5" end="4.5">
    <instance_animation url="#Guy1MoveAnim"/>
    <instance_animation url="#Guy1BreatheAnim"/>
  </animation_clip>
</library_animation_clips>
```

asset

Category: **Metadata**

Introduction

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 both by software tools and by humans. Asset information is often hierarchical, where the parts of a large asset are divided into smaller pieces that are managed as distinct assets themselves.

Attributes

The **<asset>** element has no attributes.

Related Elements

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

Parent elements	**camera**, **COLLADA**, **light**, **material**, source, **geometry**, **image**, **animation**, **animation_clip**, **controller**, **extra**, **node**, **visual_scene**, **library_***, **effect**, **force_field**, **physics_material**, **physics_scene**, **physics_model**, **profile_***, **technique** (FX) (in **profile_CG**, **profile_COMMON**, and **profile_GLES**)
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<contributor>	Defines data related to a contributor that worked on the parent element. See main entry.	N/A	0 or more
<created>	Contains date and time that the parent element was created. Represented in an ISO 8601 format as per the XML Schema dateTime primitive type. Element has no attributes.	None	1
<keywords>	Contains a list of words used as search criteria for the parent element. Element has no attributes.	None	0 or 1
<modified>	Contains date and time that the parent element was last modified. Represented in an ISO 8601 format as per the XML Schema dateTime primitive type. Element has no attributes.	None	1
<revision>	Contains revision information for the parent element. Element has no attributes.	None	0 of 1
<subject>	Contains a description of the topical subject of the parent element. Element has no attributes.	None	0 or 1
<title>	Contains title information for the parent element. Element has no attributes.	None	0 or 1

Name/example	Description	Default	Occurrences
<unit> **meter=...** **name=...** **/>**	Defines unit of distance for COLLADA elements and objects. This unit of distance applies to all spatial measurements within the scope of **<asset>**'s parent element, unless overridden by a more local **<asset>**/**<unit>**. The value of the unit is self-describing and does not have to be consistent with any real-world measurement. Its optional attributes are: • **name**: The name (**xs:NMTOKEN**) of the distance unit. For example, "meter", "centimeter", "inches", or "parsec". This can be the real name of a measurement, or an imaginary name. • **meter**: How many real-world meters in one distance unit as a floating-point number. For example, 1.0 for the name "meter"; 1000 for the name "kilometer"; 0.3048 for the name "foot". For more information, see "About Physical Units" in Chapter 6: COLLADA Physics Reference. • **name**: meter	**name**: **meter** **meter**: 1.0	0 or 1
<up_axis>	Contains descriptive information about the coordinate system of the geometric data. All coordinates are right-handed by definition. Valid values are **X_UP**, **Y_UP**, or **Z_UP**. This element specifies which axis is considered upward, which is considered to the right, and which is considered inward. See "Details." Element has no attributes.	**Y_UP**	0 or 1

Details

In the case of hierarchical **<asset>** elements, where both the parent and child assets supply a value for the same metadata, such as for **<unit>**, the child asset's value supersedes the parent's value within the scope of the child element. This applies recursively.

Up Axis Values

The **<up_axis>** element's values have the following meanings:

Value	Right Axis	Up Axis	In Axis
X-UP	Negative y	Positive x	Positive z
Y_UP	Positive x	Positive y	Positive z
Z_UP	Positive x	Positive z	Negative y

Example

Here is an example of an **<asset>** element that describes the parent **<COLLADA>** element, and hence the entire document:

```
<COLLADA>
  <asset>
    <created>2005-06-27T21:00:00Z</created>
    <keywords>COLLADA interchange</keywords>
    <modified>2005-06-27T21:00:00Z</modified>
    <unit name="nautical_league" meter="5556.0" />
    <up_axis>Z_UP</up_axis>
  </asset>
</COLLADA>
```

bool_array

Category: **Data Flow**

Introduction

Declares the storage for a homogenous array of Boolean values.

Concepts

The **<bool_array>** element stores 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	**uint**	The number of values in the array. Required.
id	**xs:ID**	A text string containing the unique identifier of this element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**source** (core)
Child elements	None
Other	**accessor**

Details

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

Example

Here is an example of a **<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>
```

camera

Category: **Camera**

Introduction

Declares a view of the visual scene hierarchy or scene graph. The camera contains elements that describe the camera's optics and imager.

Concepts

A camera embodies the eye point of the viewer looking at the visual scene. It 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 onto 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**	A text string containing the unique identifier of the **<camera>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_cameras**
Child elements	See the following subsection.
Other	**instance_camera**

Child Elements

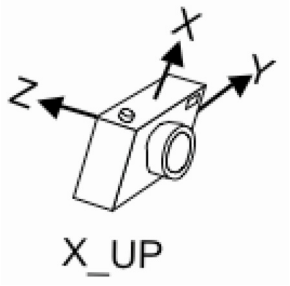
Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	Defines the directions of the axes and the units of measurement for the camera's view. Also contains information about the creation of this element. See main entry.	N/A	0 or 1
<optics>	Describes the field of view and viewing frustum using canonical parameters. See main entry.	N/A	1
<imager>	Represents the image sensor of a camera (for example, film or CCD). See main entry.	N/A	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

For simple cameras, a generic technique needs to contain only an optics element.

The camera is defined such that the local +X axis is to the right, the lens looks towards the local -Z axis, and the top of the camera is aligned with the local +Y axis (also see the **<lookat>** element). This orientation is affected by the **<asset>** element's **<up_axis>** value.



Example

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

```
<camera name="eyepoint">
  <optics>
    <technique_common>
      <perspective>
        <yfov>45</yfov>
        <aspect_ratio>1.33333</aspect_ratio>
        <znear>1.0</znear>
        <zfar>1000.0</zfar>
      </perspective>
    </technique_common>
  </optics>
</camera>
```

channel

Category: **Animation**

Introduction

Declares an output channel of an animation.

Concepts

As an animation's sampler transforms values 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:

source	**xs:URIFragmentType**	The location of the animation sampler using a URL expression. Required.
target	**xs:token**	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 the "Address Syntax" section in Chapter 3: Schema Concepts. Required.

Related Elements

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

Parent elements	**animation**
Child elements	None
Other	None

Details

This element encloses no data.

Example

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

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

COLLADA

Category: **Metadata**

Introduction

Declares the root of the document that contains some of the content in the COLLADA schema.

Concepts

The COLLADA schema is XML based; therefore, it must have exactly one document root element or document entity to be a well-formed XML document. The COLLADA element serves that purpose.

Attributes

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

version	**VersionType**	The COLLADA schema revision with which the instance document conforms. Valid values are 1.4.0 and 1.4.1. Required.
xmlns	**xs:anyURI**	This XML Schema namespace attribute applies to this element to identify the schema for an instance document.
base	**xs:anyURI**	The XML Base specification describes a facility, similar to that of HTML BASE, for defining base URIs for parts of XML documents. It defines a single attribute, xml:base, and describes in detail the procedure for its use in processing relative URI references. Refer to http://www.w3.org/XML/1998/namespace.

Related Elements

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

Parent elements	No parent elements
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	1
library_element	Any quantity and combination of any library elements can appear in any order: **<library_animations>** **<library_animation_clips>** **<library_cameras>** **<library_controllers>** **<library_effects>** **<library_force_fields>** **<library_geometries>** **<library_images>** **<library_lights>** **<library_materials>** **<library_nodes>**	N/A	0 or more

Name/example	Description	Default	Occurrences
	<library_physics_materials> **<library_physics_models>** **<library_physics_scenes>** **<library_visual_scenes>** See main entries.		
<scene>	See main entry.	N/A	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

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

Example

The following example outlines an empty COLLADA instance document whose schema version is “1.4.1”:

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema" version="1.4.1">
  <asset>
    <created/>
    <modified/>
  </asset>
  <library_geometries/>
  <library_visual_scenes/>
  <scene />
</COLLADA>
```

color

Category: **Lighting**

Introduction

Describes the color of its parent light element.

Concepts

In the context of **<light>**, the **<color>** element contains three float values describing the RGB color of its parent light element.

In the context of **<profile_COMMON>**, it contains four float values describing the RGBA color of its parent element.

Attributes

The **<color>** element has the following attribute:

sid	**xs:NCName**	For **<profile_COMMON>** parent elements only. A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	In **<light>**: **ambient** (core), **directional**, **point**, **spot** In **<profile_COMMON>**: elements of type **common_color_or_texture_type** (**ambient**, **emission**, **diffuse**, **reflective**, **specular**, **transparent**)
Child elements	None
Other	None

Details

Example

contributor

Category: **Metadata**

Introduction

Defines authoring information for asset management.

Concepts

In modern production pipelines, especially as art teams are steadily increasing in size, it is becoming more likely that a single asset may be worked on by multiple authors, possibly even using multiple tools. This information may be important for an asset management system and its content format is application-defined.

Attributes

The **<contributor>** element has no attributes.

Related Elements

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

Parent elements	**asset**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<author>	Contains a string with the author's name. This element has no attributes.	None	0 or 1
<authoring_tool>	Contains a string with the name of the authoring tool. This element has no attributes.	None	0 or 1
<comments>	Contains a string with comments from this contributor. This element has no attributes.	None	0 or 1
<copyright>	Contains a string with copyright information. This element has no attributes.	None	0 or 1
<source_data>	Contains a URI reference (**xs:anyURI**) to the source data used for this asset. This element has no attributes.	None	0 or 1

Details

Example

Here is an example of a **<contributor>** element for an asset:

```
<asset>
  <contributor>
    <author>Bob the Artist</author>
    <authoring_tool>Super3DmodelMaker3000</authoring_tool>
```

```
      <comments>This is a big Tank</comments>
      <copyright>Bob's game shack: all rights reserved</copyright>
      <source_data>c:\models\tank.s3d</source_data>
    </contributor>
  </asset>
```

controller

Category: **Controller**

Introduction

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**	A text string containing the unique identifier of the **<controller>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_controllers**
Child elements	See the following subsection.
Other	**instance_controller**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
control_element	The element that contains control data. Must be either: • **<skin>** • **<morph>** See main entries.	N/A	1
<extra>	See main entry.	N/A	0 or more

Details

A **<controller>** element is a general, generic mechanism to describe active or dynamic content, and it contains elements that describe control data. The actual type and complexity of the data is represented in detail by the child elements.

Example

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

```
<library_controllers>
  <controller name="skinner" id="skinner456">
    <skin/>
  </controller>
</library_controllers>
```

control_vertices

Category: **Geometry**

Introduction

Describes the control vertices (CVs) of a spline.

Concepts

Information about both a control vertex and its related segment are stored on the control verticess. Segment data applies to the spline segment that starts at the given control vertex.

Each control vertex must provide a position. It is strongly suggested that you provide a source of interpolation methods to be used on the segment that starts at this control vertex. Otherwise, the linear interpolation is assumed. For Bézier and Hermite interpolations, input and output tangents must be provided for each control vertex.

Additionally, each control vertex may have an arbitrary amount user-specific information, through custom data sources. The custom data sources must provide enough data to have one value – however large – for each control vertex.

Attributes

The **<control_vertices>** element has no attributes.

Related Elements

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

Parent elements	**spline**
Child elements	See the following subsection.
Other	source

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (unshared)	At least one **<input>** (unshared) element must have a semantic attribute whose value is **POSITION**. See main entry.	None	1 or more
<extra>	See main entry.	N/A	0 or more

Details

COLLADA recognizes the following polynomial interpolation types for **<control_vertices>**: **LINEAR**, **BEZIER**, **CARDINAL**, **HERMITE**, and **BSPLINE**. These symbolic names are used in a **<Name_array>** within a **<source>** element. These values are fed into the control vertices by an **<input>** element that includes a semantic attribute with a value of **POSITION**.

For more information, “Curve Interpolation” in Chapter 4: Programming Guide.

The COMMON profile defines the following **<input>** semantics for **<control_vertices>**:

<input> semantic value	Type	Description	Default
POSITION	any multidimensional float	The position of the control vertex.	N/A
INTERPOLATION	Name	The type of polynomial interpolation to represent the segment starting at the CV. Common-profile types are: **LINEAR**, **BEZIER**, **HERMITE**, **CARDINAL**, and **BSPLINE**.	**LINEAR**
IN_TANGENT	any multidimensional float	The tangent that controls the shape of the segment preceding the CV (**BEZIER** and **HERMITE**). The number of dimensions to the values of this source must match the number of dimensions in the **POSITION** source.	N/A
OUT_TANGENT	any multidimensional float	The tangent that controls the shape of the segment following the CV (**BEZIER** and **HERMITE**). The number of dimensions to the values of this source must match the number of dimensions in the **POSITION** source.	N/A
CONTINUITY	Name	(Optional.) Defines the continuity constraint at the CV. The common-profile types are C0, C1, G1.	N/A
LINEAR_STEPS	int	(Optional.) The number of piece-wise linear approximation steps to use for the spline segment that follows this CV.	N/A

Details

- For mixed interpolation splines, if at least one segment has the **BEZIER** or **HERMITE** interpolation type, then one **IN_TANGENT** value and one **OUT_TANGENT** value must be provided for every control vertex.
- The data types of all child elements must be the same. For example, when the interpolation type is **BEZIER** or **HERMITE**:
- Valid: **POSITION**, **IN_TANGENT**, and **OUT_TANGENT** all defined as **float2**.
- Invalid: **POSITION** as **float2** and **IN_TANGENT** or **OUT_TANGENT** as **float3**.
- There are constraints among child elements. For example, the quantity of **POSITION** child elements must equal the quantity of **INTERPOLATION** elements. For details, see **<spline>**.

Example

See **<spline>**.

directional

Category: **Lighting**

Introduction

Describes a directional light source.

Concepts

The **<directional>** element declares the parameters required to describe a directional light source. A directional light is one that lights everything from the same direction, regardless of location.

The light's default direction vector in local coordinates is [0,0,-1], pointing down the negative z axis. The actual direction of the light is defined by the transform of the node where the light is instantiated.

Attributes

The **<directional>** element has no attributes.

Related Elements

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

Parent elements	**light** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<color>	Contains three floating-point numbers specifying the color of the light. See main entry.	None	1

Details

Example

Here is an example of a **<directional>** element:

```
<light id="blue">
  <technique_common>
    <directional>
      <color>0.1 0.1 0.5</color>
    </directional>
  </technique_common>
</light>
```

extra

Category: **Extensibility**

Introduction

Provides arbitrary additional information about or related to 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 arbitrary XML elements and data.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the **<extra>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
type	**xs:NMTOKEN**	A hint as to the type of information that the particular **<extra>** element represents. This text string must be understood by the application. Optional.

Related Elements

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

Parent elements	**animation**, **animation_clip**, **attachment**, **bind_material**, **box**, **camera**, **capsule**, **COLLADA**, **controller**, **control_vertices**, **convex_mesh**, **cylinder**, **effect**, **force_field**, **format_hint**, **geometry**, **image**, **imager**, **instance_***, **joints**, **library_***, **light**, **lines**, **linestrips**, **material**, **mesh**, **morph**, **node**, **optics**, **pass**, **physics_material**, **physics_model**, **physics_scene**, **plane**, **polygons**, **polylist**, **profile_CG**, **profile_COMMON**, **profile_GLES**, **profile_GLSL**, **ref_attachment**, **rigid_body**, **rigid_constraint**, **sampler_state**, **sampler_***, **scene**, **shape**, **skin**, **sphere**, **spline**, **surface**, **tapered_capsule**, **tapered_cylinder**, **targets**, **technique** (FX) (in **profile_CG**, **profile_COMMON**, **profile_GLES**, and **profile_GLSL**), **texture**, **texture_pipeline**, **texture_unit**, **triangles**, **trifans**, tristrips, **vertex_weights**, **vertices**, **visual_scene**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<technique> (core)	See main entry.	N/A	1 or more

Details

Example

Here is an example of an **<extra>** element that outlines both structured and unstructured additional content:

```
<geometry>
  <extra>
    <technique profile="Max" xmlns:max="some/max/schema">
      <param name="wow" sid="animated" type="string">a validated string
parameter from the COLLADA schema.</param>
      <max:someElement>defined in the Max schema and
validated.</max:someElement>
      <uhoh>something well-formed and legal, but that can't be validated because
there is no schema for it!</uhoh>
    </technique>
  </extra>
<geometry>
```

The following example shows how **<extra>** and **<technique>** can work together:

```
<light>
   <!-- Application chooses one of the following three techniques -->
   <technique_common> ... </technique_common>
   <technique profile="ProductA"> ... </technique>
   <technique profile="ProductB"> ... </technique>
   <!-- Application chooses zero or more of the following two extras -->
   <!-- and one technique within each extra. -->
   <extra type="basic">
      <technique profile="ProductA"> ... </technique>
      <technique profile="ProductB"> ... </technique>
   </extra>
   <extra type="bonus">
      <technique profile="ProductB"> ... </technique>
   </extra>
```

Examples of two choices are:

- **technique_common**, extra "basic" (product B), and extra "bonus" (product B)
- **technique** (product B), extra "basic" (product B), and extra "bonus" (product B)

float_array

Category: **Data Flow**

Introduction

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	**uint**	The number of values in the array. Required.
id	**xs:ID**	A text string containing the unique identifier of this element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
digits	**xs:short**	The number of significant decimal digits of the float values that can be contained in the array. The default is 6. Optional.
magnitude	**xs:short**	The largest exponent of the float values that can be contained in the array. The default is 38. Optional.

Related Elements

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

Parent elements	**source** (core)
Child elements	None
Other	**accessor**

Details

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

Example

Here is an example of a **<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>
```

geometry

Category: **Geometry**

Introduction

Describes the visual shape and appearance of an object in a scene.

Concepts

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, convex mesh, or spline.

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:

- B-Spline
- Bézier
- Mesh
- NURBS
- Patch

This is by no means an exhaustive list. Currently, COLLADA supports only polygonal meshes and splines.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the **<geometry>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	A text string containing the name of the **<geometry>** element. Optional.

Related Elements

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

Parent elements	**library_geometries**
Child elements	See the following subsection.
Other	**instance_geometry**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
geometric_element	The element that describes geometric data. Must be exactly one of: • **<convex_mesh>** • **<mesh>** • **<spline>** See main entries.	N/A	1

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<geometry>** element. See main entry.	N/A	0 or more

Details

A **<geometry>** element contains elements that describe geometric data. The actual type and complexity of the data is represented in detail by the child elements.

Example

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

```
<library_geometries>
  <geometry name="cube" id="cube123">
    <mesh>
      <source id="box-Pos"/>
      <vertices id="box-Vtx">
        <input semantic="POSITION" source="#box-Pos"/>
      </vertices>
    </mesh>
  </geometry>
</library_geometries>
```

IDREF_array

Category: **Data Flow**

Introduction

Declares the storage for a homogenous array of ID reference values.

Concepts

The **<IDREF_array>** element stores string values that reference IDs within the instance document.

Attributes

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

count	**uint**	The number of values in the array. Required.
id	**xs:ID**	A text string containing the unique identifier of this element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**source** (core)
Child elements	None
Other	**accessor**

Details

An **<IDREF_array>** element contains a list of XML IDREF values. These values are a repository of data for **<source>** elements.

Example

Here is an example of an **<IDREF_array>** element that refers to **<node>** elements in the document:

```
<library_nodes>
   <node id="Node1"/>
   <node id="Node2"/>
   <node id="Joint3"/>
   <node id="WristJoint"/>
</library_nodes>

<IDREF_array id="refs" name="myRefs" count="4">
  Node1 Node2 Joint3 WristJoint
</IDREF_array>
```

imager

Category: **Camera**

Introduction

Represents the image sensor of a camera (for example, film or CCD).

Concepts

The optics of a camera projects an 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 consider only three wavelengths within the visible range: red, green, and blue (primary colors). This is usually represented as three 8-bit values.

An image sensor therefore performs two tasks:

- Spectral sampling
- Dynamic range remapping

The combination of these is called *tone mapping*, which 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 store the actual pixel colors as **float3**s, or even as arrays of floats (multispectral renderers), then perform tone mapping to create a 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.

Attributes

The **<imager>** element has no attributes.

Related Elements

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

Parent elements	**camera**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<technique> (core)	See main entry.	N/A	1 or more

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<imager>** element. See main entry.	N/A	0 or more

Details

The **<imager>** element is optional. The COMMON profile omits it (there is no **<technique_common>** for this element) 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

Multispectral 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">
  <optics>
    <technique_common>....</technique_common>
    <technique profile="MyFancyGIRenderer">
      <param name="FocalLength" type="float">180.0</param>
      <param name="Aperture" type="float">5.6</param>
    </technique>
  </optics>
  <imager>
    <technique profile="MyFancyGIRenderer">
      <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>
    </technique>
  </imager>
</camera>
```

input

(shared)

Category: **Data Flow**

Introduction

Declares the input semantics of a data source and connects a consumer to that source.

Note: There are two **<input>** variants; see also "**<input>** (unshared)."

Concepts

The **<input>** element declares the input connections to a data source that a consumer requires. A data source is a container of raw data that lacks semantic meaning so that the data can be reused within the document. To use the data, a consumer declares a connection to it with the desired semantic information.

The **<source>** and **<input>** elements are part of the COLLADA dataflow model. This model is also known as stream processing, pipe, or producer-consumer. An input connection is the dataflow path from a **<source>** to a sink (the dataflow consumers, which are **<input>**'s parents, such as **<polylist>**).

In COLLADA, all inputs are driven by index values. A consumer samples an input by supplying an index value to an input. Some consumers have multiple inputs that can share the same index values. Inputs that have the same **offset** attribute value are driven by the same index value from the consumer. This is an optimization that reduces the total number of indexes that the consumer must store. These inputs are described in this section as shared inputs but otherwise operate in the same manner as unshared inputs.

Attributes

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

offset	**uint**	The offset into the list of indices defined by the parent element's **<p>** or **<v>** subelement. If two **<input>** elements share the same offset, they are indexed the same. This is a simple form of compression for the list of indices and also defines the order in which the inputs are used. Required.
semantic	**xs:NMTOKEN**	The user-defined meaning of the input connection. Required. See "Details" for the list of common **<input>** semantic attribute values enumerated in the COLLADA schema (type **Common_profile_input**).
source	**xs:URIFragmentType**	The location of the data source. Required.
set	**uint**	Which inputs to group as a single set. This is helpful when multiple inputs share the same semantics. Optional.

Related Elements

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

Parent elements	**lines**, **linestrips**, **polygons**, **polylist**, **triangles**, **trifans**, tristrips, **vertex_weights**
Child elements	None
Other	**p** (in **lines**, **linestrips**, **polygons**, **polylist**, **triangles**, **trifans**, tristrips); **v** (in **vertex_weights**)

Details

Each input connection can be uniquely identified by its offset attribute within the scope of its parent element.

The common **<input>** semantic attribute values are:

Value of semantic attribute	Description
BINORMAL	Geometric binormal (bitangent) vector
COLOR	Color coordinate vector. Color inputs are RGB (**float3**)
CONTINUITY	Continuity constraint at the control vertex (CV). See also "Curve Interpolation" in Chapter 4: Programming Guide.
IMAGE	Raster or MIP-level input.
INPUT	Sampler input. See also "Curve Interpolation" in Chapter 4: Programming Guide.
IN_TANGENT	Tangent vector for preceding control point. See also "Curve Interpolation" in Chapter 4: Programming Guide.
INTERPOLATION	Sampler interpolation type. See also "Curve Interpolation" in Chapter 4: Programming Guide.
INV_BIND_MATRIX	Inverse of local-to-world matrix.
JOINT	Skin influence identifier
LINEAR_STEPS	Number of piece-wise linear approximation steps to use for the spline segment that follows this CV. See also "Curve Interpolation" in Chapter 4: Programming Guide.
MORPH_TARGET	Morph targets for mesh morphing
MORPH_WEIGHT	Weights for mesh morphing
NORMAL	Normal vector
OUTPUT	Sampler output. See also "Curve Interpolation" in Chapter 4: Programming Guide.
OUT_TANGENT	Tangent vector for succeeding control point. See also "Curve Interpolation" in Chapter 4: Programming Guide.
POSITION	Geometric coordinate vector. See also "Curve Interpolation" in Chapter 4: Programming Guide.
TANGENT	Geometric tangent vector
TEXBINORMAL	Texture binormal (bitangent) vector
TEXCOORD	Texture coordinate vector
TEXTANGENT	Texture tangent vector
UV	Generic parameter vector
VERTEX	Mesh vertex
WEIGHT	Skin influence weighting value

Example

Here is an example of six **<input>** elements that describe the sources of vertex positions, normals, and two sets of texture coordinates along with their texture space tangents for a **<polygons>** element. The offset attribute indicates the index from the **<p>** element that the input will use to sample the source data. When two or more **<input>** elements have the same offset value, it means that they share the same index in the **<p>** element. This is a simple form of index compression that saves space in the document.

The set attribute indicates the logical organization of **<input>** elements that belong in the same logical set of information. In this example, there are two sets of **TEXCOORD** and **TEXTANGENT** pairs:

```
<mesh>
  <source name="grid-Position"/>
  <source name="grid-0-Normal"/>
  <source name="texCoords1"/>
```

```
  <source name="grid-texTangents1"/>
  <source name="texCoords2"/>
  <source name="grid-texTangents2"/>
  <vertices id="grid-Verts">
    <input semantic="POSITION" source="#grid-Position"/>
  <vertices>
  <polygons count="1" material="Bricks">
    <input semantic="VERTEX" source="#grid-Verts" offset="0"/>
    <input semantic="NORMAL" source="#grid-Normal" offset="1"/>
    <input semantic="TEXCOORD" source="#texCoords1" offset="2" set="0"/>
    <input semantic="TEXCOORD" source="#texCoords2" offset="2" set="1"/>
    <input semantic="TEXTANGENT" source="#texTangents1" offset="2" set="0"/>
    <input semantic="TEXTANGENT" source="#texTangents2" offset="2" set="1"/>
    <p>0 0 0 2 1 1 3 2 2 1 3 3</p>
  </polygons>
</mesh>
```

input

(unshared)

Category: **Data Flow**

Introduction

Declares the input semantics of a data source and connects a consumer to that source.

Note: There are two **<input>** variants; see also "**<input>** (shared)."

Concepts

The **<input>** element declares the input connections that a consumer requires. A data source is a container of raw data that lacks semantic meaning so that the data can be reused within the document. To use the data, a consumer declares a connection to it with the desired semantic information.

The **<source>** and **<input>** elements are part of the COLLADA dataflow model. This model is also known as stream processing, pipe, or producer-consumer. An input connection is the dataflow path from a **<source>** to a sink (the dataflow consumers, which are **<input>**'s parents, such as **<vertices>**).

In COLLADA, all inputs are driven by index values. A consumer samples an input by supplying an index value to an input. Some consumers have simple inputs that are driven by unique index values. These inputs are described in this section as unshared inputs but otherwise operate in the same manner as shared inputs.

Attributes

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

semantic	**xs:NMTOKEN**	The user-defined meaning of the input connection. Required. See the list of common **<input>** semantic attribute values in the "Common Glossary" in Chapter 3: Schema Concepts. See "<**input**> (shared)" for the list of common **<input>** semantic attribute values enumerated in the COLLADA schema (type **Common_profile_input**).
source	**xs:URIFragmentType**	The location of the data source. Required.

Related Elements

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

Parent elements	**joints**, **sampler**, **targets**, **vertices**, **control_vertices**
Child elements	None
Other	None

Details

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

See "Curve Interpolation" in Chapter 4: Programming Guide for a description of the semantic values that are useful in in **<sampler>** animation curves.

Example

Here is an example for **<sampler><input>**:

```
<animation>
```

```
  <source id="translate_X-input">
    ...
  </source>
  <source id="translate_X-output">
    ...
  </source>
  <source id="translate_X-intangents">
    ...
  </source>
 <source id="translate_X-outtangents">
    ...
  </source>
  <source id="translate_X-interpolations">
    ...
  </source>
  <sampler id="translate_X-sampler">
    <input semantic="INPUT" source="#translate_X-input"/>
    <input semantic="OUTPUT" source="#translate_X-output"/>
    <input semantic="IN_TANGENT" source="#translate_X-intangents"/>
    <input semantic="OUT_TANGENT" source="#translate_X-outtangents"/>
    <input semantic="INTERPOLATION" source="#translate_X-interpolations"/>
  </sampler>
  <channel source="#translate_X-sampler" target="pCube1/translate.X"/>
</animation>
```

instance_animation

Category: **Animation**

Introduction

Instantiates a COLLADA animation resource.

Concepts

An **<instance_animation>** element instantiates an object described by an **<animation>** element. Multiple **<instance_animation>** elements are grouped together to create animation clips in the **<animation_clip>** element.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<animation>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**animation_clip**
Child elements	See the following subsection.
Other	**animation**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<animation>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of an **<instance_animation>** element that refers to a locally defined **<animation>** element identified by the ID "anim". The instance is translated some distance from the original:

```
<library_animations>
  <animation id="anim"/>
</library_animations>
<library_animation_clips>
  <animation_clip start="1.0" end="5.0"/>
    <instance_animation url="#anim"/>
  </animation_clip>
</library_animation_clips>
```

instance_camera

Category: **Camera**

Introduction

Instantiates a COLLADA camera resource.

Concepts

The **<instance_camera>** element instantiates an object described by a **<camera>** element to activate it in the visual scene. A camera object is instantiated within the local coordinate system of its parent **<node>** and that determines its position, orientation, and scale.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<camera>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**node**
Child elements	See the following subsection.
Other	**camera**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<instance_camera>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of an **<instance_camera>** element that refers to a locally defined **<camera>** element identified by the ID cam. The instance is translated some distance from the original:

```
<library_cameras>
```

```
  <camera id="cam"/>
</library_cameras>
<node>
  <node>
    <translate>11.0 12.0 13.0</translate>
    <instance_camera url="#cam"/>
  </node>
</node>
```

instance_controller

Category: **Controller**

Introduction

Instantiates a COLLADA controller resource.

Concepts

The **<instance_controller>** element instantiates an object described by a **<controller>** element. A controller object is instantiated within the local coordinate system of its parent **<node>** and that determines its position, orientation, and scale. The controller allows deformations of meshes based on skinning animations or morphing animations.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URI of the location of the **<controller>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**node**
Child elements	See the following subsection.
Other	**controller**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<skeleton>	Indicates where a skin controller is to start to search for the joint nodes it needs. This element is meaningless for morph controllers. See main entry.	None	0 or more
<bind_material>	See main entry.	N/A	0 or 1
<extra>	Provides arbitrary additional information about or related to the **<instance_controller>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of an **<instance_controller>** element that refers to a locally defined **<controller>** element identified as “skin”. The instance is translated some distance from the original:

```
<library_controllers>
  <controller id="skin"/>
</library_controllers>
<node id="skel"/>
   ...
</node>
<node>
  <translate>11.0 12.0 13.0</translate>
  <instance_controller url="#skin"/>
    <skeleton>#skel</skeleton>
  </instance_controller>
</node>
```

The following is an Example of two **<instance_controller>** elements that refer to the same locally defined **<controller>** element identified as “skin”. The two skin instances are bound to different instances of a skeleton using the **<skeleton>** element:

```
<library_controllers>
  <controller id="skin">
    <skin source="#base_mesh">
      <source id="Joints">
        <Name_array count="4"> Root Spine1 Spine2 Head </Name_array>
        ...
      </source>
      <source id="Weights"/>
      <source id="Inv_bind_mats"/>
      <joints>
        <input source="#Joints" semantic="JOINT"/>
      </joints>
      <vertex_weights/>
    </skin>
  </controller>
</library_controllers>
<library_nodes>
  <node id="Skeleton1" sid="Root">
    <node sid="Spine1">
      <node sid="Spine2">
        <node sid="Head"/>
      </node>
    </node>
  </node>
</library_nodes>
<node id="skel01">
   <instance_node url="#Skeleton1"/>
</node>
<node id="skel02">
  <instance_node url="#Skeleton1"/>
</node>
<node>
  <instance_controller url="#skin"/>
    <skeleton>#skel01</skeleton>
  </instance_controller>
</node>
<node>
  <instance_controller url="#skin"/>
    <skeleton>#skel02</skeleton>
```

```
      </instance_controller>
    </node>
```

instance_geometry

Category: **Geometry**

Introduction

Instantiates a COLLADA geometry resource.

Concepts

The **<instance_geometry>** element instantiates an object described by a **<geometry>** element. A geometry object is instantiated within the local coordinate system of its parent **<node>** or **<shape>** and that determines its position, orientation, and scale. COLLADA supports convex mesh, mesh, and spline primitives.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<geometry>** element to instantiate. Required. Can refer to a local instance or external reference.

For a a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**node**, **shape**
Child elements	See the following subsection.
Other	**geometry**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<bind_material>	Binds material symbols to material instances. This allows a single geometry to be instantiated into a scene multiple times each with a different appearance. See main entry.	None	0 or 1
<extra>	Provides arbitrary additional information about or related to the **<instance_geometry>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of an **<instance_geometry>** element that refers to a locally defined **<geometry>** element identified by the ID “cube”. The instance is translated some distance from the original:

```
<library_geometries>
  <geometry id="cube"/>
</library_geometries>
<node>
  <node>
    <translate>11.0 12.0 13.0</translate>
    <instance_geometry url="#cube"/>
  </node>
</node>
```

instance_light

Category: **Lighting**

Introduction

Instantiates a COLLADA light resource.

Concepts

The **<instance_light>** element instantiates an object described by a **<light>** element to activate it in the visual scene. Directional, point, and spot light objects are instantiated within the local coordinate system of their parent **<node>** and that determines their position, orientation, and scale. The exception is ambient light; because ambient light radiates in all directions equally, it is not affected by these spatial transformations.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<light>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**node**
Child elements	See the following subsection.
Other	**light**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<instance_light>** element. See main entry.	N/A	0 or more

Details

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_light>** element that refers to a locally defined **<light>** element identified by the id “light”. The instance is translated some distance from the original:

```
<library_lights>
  <light id="light"/>
</library_lights>
<node>
  <node>
    <translate>11.0 12.0 13.0</translate>
    <instance_light url="#light"/>
  </node>
</node>
```

instance_node

Category: **Scene**

Introduction

Instantiates a COLLADA node resource.

Concepts

An **<instance_node>** creates an instance of an object described by a **<node>** element. Each instance of a **<node>** element refers to an element in the **<node>** hierarchy that has its own local coordinate system defined for placing objects in the scene.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<node>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**node**
Child elements	See the following subsection.
Other	**node**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<instance_node>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of an **<instance_node>** element that refers to a locally defined **<node>** element identified by the ID "myNode". The instance is translated some distance from the original:

```
<library_nodes>
  <node id="myNode"/>
```

```
</library_nodes>
<node>
  <node>
    <translate>11.0 12.0 13.0</translate>
    <instance_node url="#myNode"/>
  </node>
</node>
```

instance_visual_scene

Category: **Scene**

Introduction

Instantiates a COLLADA **visual_scene** resource.

Concepts

An **<instance_visual_scene>** instantiates the visual aspects of a scene. The **<scene>** element can contain, at most, one **<instance_visual_scene>** element. This constraint creates a one-to-one relationship between the document, the top-level scene, and its visual description. This provides applications and tools, especially those that support only one scene, an indication of the primary scene to load.

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<visual_scene>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**scene**
Child elements	See the following subsection.
Other	**visual_scene**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	Provides arbitrary additional information about or related to the **<instance_visual_scene>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of an **<instance_visual_scene>** element that refers to a locally defined **<visual_scene>** element identified by the ID "**vis_scene**":

```
<library_visual_scenes>
  <visual_scene id="vis_scene"/>
</library_visual_scenes>
<scene>
  <instance_visual_scene url="#vis_scene"/>
</scene>
```

int_array

Category: **Data Flow**

Introduction

Stores 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	**uint**	The number of values in the array. Required.
id	**xs:ID**	A text string containing the unique identifier of this element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
minInclusive	**xs:integer**	The smallest integer value that can be contained in the array. The default is -2147483648. Optional.
maxInclusive	**xs:integer**	The largest integer value that can be contained in the array. The default is 2147483647. Optional.

Related Elements

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

Parent elements	**source** (core)
Child elements	None
Other	**accessor**

Details

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>
```

joints

Category: **Controller**

Introduction

Declares the association between joint nodes and attribute data.

Concepts

Associates joint, or skeleton, nodes with attribute data. In COLLADA, this is specified by the inverse bind matrix of each joint (influence) in the skeleton.

Attributes

The **<joints>** element has no attributes.

Related Elements

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

Parent elements	**skin**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (unshared)	At least one **<input>** element must have the semantic **JOINT**. The **<source>** referenced by the input with the **JOINT** semantic should contain a **<Name_array>** that contains **sid**s to identify the joint nodes. **sid**s are used instead of IDREFs to allow a skin controller to be instantiated multiple times, where each instance can be animated independently. See main entry.	None	2 or more
<extra>	Provides arbitrary additional information about or related to the **<joints>** element. See main entry.	N/A	0 or more

Details

Example

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

```
<skin source="#geometry_mesh">
  <joints>
    <input semantic="JOINT" source="#joints"/>
    <input semantic="INV_BIND_MATRIX" source="#inv-bind-matrices"/>
  </joints>
</skin>
```

library_animation_clips

Category: **Animation**

Introduction

Declares a module of **<animation_clip>** elements.

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_animation_clips>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_animation_clips>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	None	0 or more
<animation_clip>	See main entry.	N/A	1 or more
<extra>	Provides arbitrary additional information about or related to the **<library_animation_clips>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_animation_clips>** element:

```
<library_animation_clips>
  <animation_clip>
      <instance_animation url="#animation1" />
  </animation_clip>
  <animation_clip>
      <instance_animation url="#animation2" />
      <instance_animation url="#animation3" />
  </animation_clip>
</library_animation_clips>
```

library_animations

Category: **Animation**

Introduction

Declares a module of **<animation>** elements.

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_animations>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_animations>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<animation>	See main entry.	N/A	1 or more
<extra>	Provides arbitrary additional information about or related to the **<library_animations>** element. See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_animations>** element:

```
<library_animations>
  <animation name="animation1" />
  <animation name="animation2" />
  <animation name="animation3" />
</library_animations>
```

library_cameras

Category: **Camera**

Introduction

Declares a module of **<camera>** elements.

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_cameras>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_cameras>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<camera>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_cameras>** element:

```
<library_cameras>
  <camera name="eyepoint">
      ...
  </camera>
  <camera name="overhead">
      ...
  </camera>
</library_cameras>
```

library_controllers

Category: **Controller**

Introduction

Declares a module of **<controller>** elements.

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_controllers>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_controllers>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<controller>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_controllers>** element:

```
<library_controllers>
  <controller>
      <skin source="#geometry_mesh">
         ...
      </skin>
   </controller>
</library_controllers>
```

library_geometries

Category: **Geometry**

Introduction

Declares a module of **<geometry>** elements.

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_geometries>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_geometries>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<geometry>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_geometries>** element:

```
<library_geometries>
  <geometry name="cube" id="cube123">
    <mesh>
      <source id="box-Pos"/>
      <vertices id="box-Vtx">
        <input semantic="POSITION" source="#box-Pos"/>
      </vertices>
    </mesh>
  </geometry>
</library_geometries>
```

library_lights

Category: **Lighting**

Introduction

Declares a module of **<light>** elements.

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_lights>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<light>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_lights>** element:

```
<library_lights>
  <light id="light1">
    ...
  </light>

  <light id="light2">
    ...
  </light>
</library_lights>
```

library_nodes

Category: **Scene**

Introduction

Declares a module of **<node>** elements.

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_nodes>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<node>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_nodes>** element:

```
<library_nodes>
  <node id="node1">
    ...
  </node>

  <node id="node2">
    ...
  </node>
</library_nodes>
```

library_visual_scenes

Category: **Scene**

Introduction

Declares a module of **<visual_scene>** elements.

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_visual_scenes>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<visual_scene>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_visual_scenes>** element:

```
<library_visual_scenes>
  <visual_scene id="vis_sce1">
    ...
  </visual_scene>

  <visual_scene id="vis_sce2">
    ...
  </visual_scene>

</library_visual_scenes>
```

light

Category: **Lighting**

Introduction

Declares a light source that illuminates a 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. COLLADA supports:

- 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**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_lights**
Child elements	See the following subsection.
Other	**instance_light**, **ambient** (core), **directional**, **point**, **spot**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<technique_common>	Specifies light information for the common profile that all COLLADA implementations must support. See “The Common Profile” section for usage information. Must contain exactly one **<ambient>** (core), **<directional>**, **<point>**, or **<spot>** element; see their main entries.	N/A	1

Name/example	Description	Default	Occurrences
<technique> (core)	Each **<technique>** specifies light information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_lights>** element that contains a directional **<light>** element that is instantiated in a visual scene, rotated to portray a sunset:

```
<library_lights>
  <light id="sun" name="the-sun">
    <technique_common>
      <directional>
        <color>1.0 1.0 1.0</color>
      </directional>
    </technique_common>
  </light>
</library_lights>
<library_visual_scenes>
  <visual_scene>
    <node>
      <rotate>1 0 0 -10</rotate>
      <instance_light url="#sun"/>
    </node>
  </visual_scene>
</library_visual_scenes>
```

lines

Category: **Geometry**

Introduction

Provides the information needed for a mesh to bind vertex attributes together and then organize those vertices into individual lines.

Concepts

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

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 the 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:

name	**xs:NCName**	The text string name of this element. Optional.
count	**uint**	The number of line primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<p>	Contains indices that describe the vertex attributes for an arbitrary number of individual lines. The indices in a **<p>** (“primitives”) element refer to different inputs depending on their order. The first index in a **<p>** element refers to all inputs with an offset of 0. The second index refers to all inputs with an offset of 1. Each vertex of the line is made up of one index into each input. After each input is used, the next index again refers to the inputs with offset of 0 and begins a new vertex.	None	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of the **<lines>** element: collating three **<input>** elements into two separate lines, where the last two inputs use the same offset:

```
<mesh>
  <source id="position"/>
  <source id="texcoord0"/>
  <source id="texcoord1"/>
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <lines count="2">
    <input semantic="VERTEX" source="#verts" offset="0"/>
    <input semantic="TEXCOORD" source="#texcoord0" offset="1"/>
    <input semantic="TEXCOORD" source="#texcoord1" offset="1"/>
    <p>10 10 11 11 21 21 22 22</p>
  </lines>
</mesh>
```

linestrips

Category: **Geometry**

Introduction

Provides the information needed to bind vertex attributes together and then organize those vertices into connected line-strips.

Concepts

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

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:

name	**xs:NCName**	The text string name of this element. Optional.
count	**uint**	The number of line-strip primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<p>	Contains indices that describe the vertex attributes for an arbitrary number of connected line segments. The indices in a **<p>** (“primitive”) element refer to different inputs depending on their order. The first index in a **<p>** element refers to all inputs with an offset of 0. The second index refers to all inputs with an offset of 1. Each vertex of the line is made up of one index into each input. After each input is used, the next index again refers to the inputs with offset of 0 and begins a new vertex.	None	0 or more
<extra>	See main entry.	N/A	0 or more

Details

A **<linestrips>** element contains a sequence of **<p>** elements.

Example

Here is an example of the **<linestrips>** element that describes two line segments with three vertex attributes, where all three inputs use the same offset:

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

lookat

Category: **Transform**

Introduction

Contains a position and orientation transformation suitable for aiming a camera.

Concepts

The **<lookat>** element contains a **float3x3**, which is three mathematical vectors that describe:

1. The position of the object.
2. The position of the interest point.
3. The direction that points up.

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**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	**node**
Child elements	None
Other	None

Details

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:

1. Eye position is given as **Px**, **Py**, **Pz**.
2. Interest point is given as **Ix**, **Iy**, **Iz**.
3. 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.

For more information about how transformation elements are applied, see **<node>**.

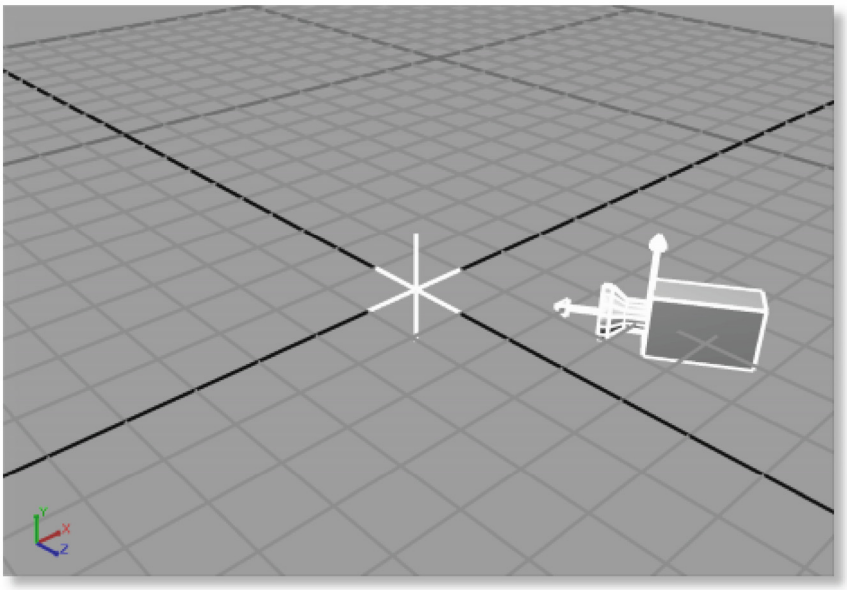
Example

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 id="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>
<instance_camera url="#camera1"/>
...
```

Figure 5-1: **<lookat>** element; the 3-D “cross-hair” represents the interest-point position



matrix

Category: **Transform**

Introduction

Describes transformations that embody mathematical changes to points within a coordinate system or the coordinate system itself.

Concepts

The **<matrix>** element contains a float4x4, which is a 4-by-4 matrix of floating-point values.

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.

Attributes

The **<matrix>** element has the following attribute:

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	**node**
Child elements	None
Other	None

Details

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.

For more information about how transformation elements are applied, see **<node>**.

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>
```

mesh

Category: **Geometry**

Introduction

Describes basic geometric meshes using vertex and primitive information.

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 no attributes.

Related Elements

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

Parent elements	**geometry**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source>	Provides the bulk of the mesh's vertex data. See main entry.	N/A	1 or more
<vertices>	Describes the mesh-vertex attributes and establishes their topological identity. See main entry.	N/A	1

Name/example	Description		Default	Occurrences
primitive_elements	Geometric primitives, which assemble values from the inputs into vertex attribute data. Can be any combination of the following in any order:			
	<lines>	Contains line primitives. See main entry.	N/A	0 or more
	<linestrips>	Contains line-strip primitives. See main entry.	N/A	0 or more
	<polygons>	Contains polygon primitives which may contain holes. See main entry.	N/A	0 or more
	<polylist>	Contains polygon primitives that cannot contain holes. See main entry.	N/A	0 or more
	<triangles>	Contains triangle primitives. See main entry.	N/A	0 or more
	<trifans>	Contains triangle-fan primitives. See main entry.	N/A	0 or more
	<tristrips>	Contains triangle-strip primitives. See main entry.	N/A	0 or more
<extra>	See main entry.		N/A	0 or more

Details

To describe geometric primitives that are formed from the vertex data, the **<mesh>** element may contain zero or more of the primitive elements **<lines>**, **<linestrips>**, **<polygons>**, **<polylist>**, **<triangles>**, **<trifans>**, and <tristrips>.

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 one **<input>** (unshared) element with a semantic attribute whose value is **POSITION**.

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

Example

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

```
<mesh>
  <source id="box-Pos"/>
  <vertices id="box-Vtx">
     <input semantic="POSITION" source="#box-Pos">
  </vertices>
</mesh>
```

In a situation where you want to share index data, that is, to optimize the index data, and still have distinct set attributes, you can move the **<input>** element from the **<vertices>** element into the primitive element(s) and reuse the **offset** attribute value of the input with **VERTEX** semantic:

```
<vertices>
  <input semantic="POSITION"/>
  <input semantic="TEXCOORD"/>
  <input semantic="NORMAL"/>
</vertices>
<polygons>
```

```
    <input semantic="VERTEX" offset="0"/>
  ...
```

use the following:

```
  <vertices>
    <input semantic="POSITION"/>
  </vertices>
  <polygons>
    <input semantic="VERTEX" offset="0"/>
    <input semantic="TEXCOORD" offset="0" set="1"/>
    <input semantic="NORMAL" offset="0" set="4"/>
    ...
```

morph

Category: **Controller**

Introduction

Describes the data required to blend between sets of static meshes.

Concepts

Each possible mesh that can be blended (a morph target) must be specified. The method attribute can be used to specify how to combine these meshes. In addition, there is a "base mesh" which is used by certain methods as a reference or baseline for the blending operation (see below).

The result of a morph mesh is usually a linear combination of other meshes (whether they are static, skinned, or something else). These input meshes are called the morph targets. A major constraint is that the morph targets must all have the same set of vertices (even if they are in different positions). The combination of the morph targets only interpolates the data in their **<vertices>** elements. Therefore, all of the morph targets' **<vertices>** elements must have the same structure. For any vertex attributes not in the **<vertices>** element, those of the base mesh are used as is and those in the other morph targets are ignored.

A **<morph>** element is specified as a base mesh, a set of other meshes, a set of weights, and a method for combining them. There are different methods available to combine morph targets; the method attribute specifies which is used. The two common methods are:

- **NORMALIZED**
  - `(Target1, Target2, ...)*(w1, w2, ...) = (1-w1-w2-...)*BaseMesh + w1*Target1 + w2*Target2 + ...`
- **RELATIVE**
  - `(Target1, Target2, ...) + (w1, w2, ...) = BaseMesh + w1*Target1 + w2*Target2 + ...`

Attributes

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

sid	**xs:anyURI**	Refers to the **<geometry>** that describes the base mesh. Required.
method	**MorphMethodType**	Which blending technique to use. Valid values are **NORMALIZED** and **RELATIVE**. The default is **NORMALIZED**. Optional.

Related Elements

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

Parent elements	**controller**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source>	Data for morph weights and for morph targets. See main entry.	N/A	2 or more
<targets>	Input meshes (morph targets) to be blended. This must contain at least one child **<input>** element with a semantic of **MORPH_WEIGHT** and one with a semantic of **MORPH_TARGET**. See main entry.	N/A	1
<extra>	See main entry.	N/A	0 or more

Details

See the annotated example at http://collada.org/mediawiki/index.php/Skin_and_morph.

Example

Here is an example of an empty **<morph>** element:

```
<morph source="#the-base-mesh" method="RELATIVE">
  <source id="morph-targets"/>
  <source id="morph-weights"/>
  <targets>
    <input semantic="MORPH_TARGET" source="#morph-targets"/>
    <input semantic="MORPH_WEIGHT" source="#morph-weights"/>
  </targets>
  <extra/>
</morph>
```

Name_array

Category: **Data Flow**

Introduction

Stores a homogenous array of symbolic name values.

Concepts

The **<Name_array>** element stores name values as data for generic use within the COLLADA schema. The array itself is strongly typed but without semantics. It simply stores a sequence of XML name values.

Attributes

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

count	**uint**	The number of values in the array. Required.
id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**source** (core)
Child elements	See the following subsection.
Other	**accessor**

Details

An **<Name_array>** element contains a list of XML name values (**xs:Name**). These values are a repository of data to **<source>** elements. An application can specify any application-defined name values.

For example, **<Name_array>**, when used as a source for curve-interpolation descriptions, allows an application to specify the type of curve to be processed; the common profile defines the values **BEZIER**, **LINEAR**, **BSPLINE**, and **HERMITE** for curves.

Example

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

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

Here is an example that supplies interpolation types to a sampler:

```
<source id="translate_X-interpolations">
  <Name_array id="translate_X-interpolations-array" count="2">
    BEZIER BEZIER
  </Name_array>
  <technique_common>
    <accessor source="#translate_X-interpolations-array" count="2" stride="1">
      <param name="INTERPOLATION" type="Name"/>
    </accessor>
```

```
    </technique_common>
  </source>
  <sampler id="translate_X-sampler">
    <input semantic="INTERPOLATION" source="#translate_X-interpolations"/>
  </sampler>
```

node

Category: **Scene**

Embodies the hierarchical relationship of elements in a scene.

Introduction

The **<node>** element declares a point of interest in a scene. A node denotes one point on a branch of the scene graph. The **<node>** element is essentially the root of a subgraph 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. COLLADA uses the term node to denote interior nodes. Arcs are also called paths.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
type	**NodeType**	The type of the **<node>** element. Valid values are **JOINT** or **NODE**. The default is **NODE**. Optional.
layer	**ListOfNames**	The names of the layers to which this node belongs. For example, a value of foreground glowing" indicates that this node belongs to both the layer named foregroundand the layer named glowing. The default is empty, indicating that the node doesn't belong to any layer. Optional.

Related Elements

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

Parent elements	**library_nodes**, **node**, **visual_scene**
Child elements	See the following subsection.
Other	**instance_node**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	Allows the node to express asset management information. See main entry.	N/A	0 or 1
transformation_elements	Any combination of the following transformation elements: • **<lookat>** • **<matrix>** • **<rotate>**	None	0 or more

Name/example	Description	Default	Occurrences
	• **<scale>** • **<skew>** • **<translate>** See main entries.		
<instance_camera>	Allows the node to instantiate a camera object. See main entry.	N/A	0 or more
<instance_controller>	Allows the node to instantiate a controller object. See main entry.	N/A	0 or more
<instance_geometry>	Allows the node to instantiate a geometry object. See main entry.	N/A	0 or more
<instance_light>	Allows the node to instantiate a light object. See main entry.	N/A	0 or more
<instance_node>	Allows the node to instantiate a hierarchy of other nodes. See main entry.	N/A	0 or more
<node>	Allows the node to recursively define hierarchy. See main entry.	N/A	0 or more
<extra>	Allows the node to recursively define hierarchy. See main entry.	N/A	0 or more

Details

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.

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

The transformation elements transform the coordinate system of the **<node>** element. Mathematically, this means that the transformation elements are converted to matrices and postmultiplied in the order in which they are specified within the **<node>** to compose the coordinate system.

Example

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

```
<visual_scene>
  <node name="earth">
  </node>
  <node name="sky">
  </node>
</visual_scene>
```

optics

Category: **Camera**

Introduction

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 (for example, 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 eyepiece.

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).

Attributes

The **<optics>** element has no attributes.

Related Elements

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

Parent elements	**camera**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<technique_common>	Specifies optics information for the common profile that all COLLADA implementations must support. See "The Common Profile" section for usage information and the following subsection for child element details. Also see main entry.	N/A	1
<technique> (core)	Each **<technique>** specifies optics information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Child Elements for optics / technique_common

Name/example	Description	Default	Occurrences
<orthographic> or **<perspective>**	The projection type. See main entries.	N/A	1

Details

The COMMON profile defines the optics types **<perspective>** and **<orthographic>**. All other **<optics>** types must be specified within a profile-specific **<technique>**.

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">
  <optics>
    <technique_common>
      <perspective>
        <yfov>45.0</yfov>
        <aspect_ratio>1.33333</aspect_ratio>
        <znear>0.1</znear>
        <zfar>32767.0</zfar>
      </perspective>
    </technique_common>
  </optics>
</camera>
```

orthographic

Category: **Camera**

Introduction

Describes the field of view of an orthographic camera.

Concepts

Orthographic projection describes a way of drawing a 3-D scene on a 2-D surface. In an orthographic projection, the apparent size of an object does not depend on its distance from the camera.

Compare to **<perspective>**.

Attributes

The **<orthographic>** element has no attributes.

Related Elements

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

Parent elements	**optics** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Note: The **<orthographic>** element must contain one of:

- A single **<xmag>** element
- A single **<ymag>** element
- Both an **<xmag>** and a **<ymag>** element
- The **<aspect_ratio>** element and either **<xmag>** or **<ymag>**

These describe the field of view of the camera. If the **<aspect_ratio>** element is not present, the aspect ratio is to be calculated from the **<xmag>** or **<ymag>** elements and the current viewport.

Name/example	Description	Default	Occurrences
<xmag sid="...">	Contains a floating-point number describing the horizontal (X) magnification of the view. The sid is optional.	None	See “Note”
<ymag sid="...">	Contains a floating-point number describing the vertical (Y) magnification of the view. The sid is optional.	None	See “Note”
<extra>	See main entry.	N/A	0 or more
<aspect_ratio sid="...">	Contains a floating-point number describing the aspect ratio of the field of view. The sid is optional.	None	See “Note”
<znear sid="...">	Contains a floating-point number that describes the distance to the near clipping plane. The sid is optional.	None	1
<zfar sid="...">	Contains a floating-point number that describes the distance to the far clipping plane. The sid is optional.	None	1

Details

The X and Y magnifications are simple scale factors, applied to the X and Y components of the orthographic viewport. As such, if your default orthographic viewport is **`[[-1,1],[-1,1]]`** as in OpenGL and DirectX, your COLLADA orthographic viewport becomes **`[[-xmag,xmag],[-ymag,ymag]]`**. This gives an orthographic width of **`xmag/2`** and an orthographic height of **`ymag/2`**.

The center screen pixel is assumed to be (0,0) in screen coordinates.

Example

Here is an example of an **`<orthographic>`** element specifying a standard view (no magnification and a standard aspect ratio):

```
<orthographic>
  <xmag sid="animated_zoom">1.0</xmag>
  <aspect_ratio>0.1</aspect_ratio>
  <znear>0.1</znear>
  <zfar>1000.0</zfar>
</orthographic>
```

param

(core)

Category: **Data Flow**

Introduction

Declares parametric information for 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 **<param>** element contains information of type **xs:string**, which is the parameter's actual value.

Attributes

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

name	**xs:NCName**	The text string name of this element. Optional.
sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
type	**xs:NMTOKEN**	The type of the value data. This text string must be understood by the application. Required.
semantic	**xs:NMTOKEN**	The user-defined meaning of the parameter. Optional.

Related Elements

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

Parent elements	**accessor**, **bind_material**
Child elements	None
Other	None

Details

The **<param>** element describes parameters for generic data flow.

Example

Here is an example of two **<param>** elements that describe the output of an **<accessor>**:

```
<accessor source="#values" count="3" stride="3">
      <param name="A" type="int" />
      <param name="B" type="int" />
</accessor>
```

perspective

Category: **Camera**

Introduction

Describes the field of view of a perspective 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.

Compare to **<orthographic>**.

Attributes

The **<perspective>** element has no attributes.

Related Elements

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

Parent elements	**optics** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present.

Note: The **<perspective>** element must contain one of:

- A single **<xfov>** element
- A single **<yfov>** element
- Both an **<xfov>** and a **<yfov>** element
- The **<aspect_ratio>** element and either **<xfov>** or **<yfov>**

These describe the field of view of the camera. In the first two cases, the application can calculate the camera aspect ratio based on the viewport aspect ratio.

Name/example	Description	Default	Occurrences
<xfov sid="...">	Contains a floating-point number describing the horizontal field of view in degrees. The sid is optional.	None	See “Note”
<yfov sid="...">	Contains a floating-point number describing the vertical field of view in degrees. The sid is optional.	None	See “Note”
<aspect_ratio sid="...">	Contains a floating-point number describing the aspect ratio of the field of view. The sid is optional.	None	See “Note”
<znear sid="...">	Contains a floating-point number that describes the distance to the near clipping plane. The sid is optional.	None	1
<zfar sid="...">	Contains a floating-point number that describes the distance to the far clipping plane. The sid is optional.	None	1

Details

If the **<aspect_ratio>** element is not specified, it is calculated from the **<xfov>** or **<yfov>** elements and the current viewport. The aspect ratio is defined as the ratio of the field of view's width over its height; therefore, the aspect ratio can be derived from, or be used to derive, the field of view parameters: **aspect_ratio = xfov / yfov**.

The center screen pixel is assumed to be (0,0) in screen coordinates.

The distances to the clipping planes are specified in the current units as defined by **<asset>**/**<unit>** in the scope for this element.

Example

Here is an example of a **<perspective>** element specifying a horizontal field-of-view of 90 degrees that also may be targeted by an animation:

```
<perspective>
  <xfov sid="animated_zoom">90.0</xfov>
  <aspect_ratio>1.333</aspect_ratio>
  <znear>0.1</znear>
  <zfar>1000.0</zfar>
</perspective>
```

point

Category: **Lighting**

Introduction

Describes a point light source.

Concepts

The **<point>** element declares the parameters required to describe a point light source. 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.

The position of the light is defined by the transform of the node in which it is instantiated.

Attributes

The **<point>** element has no attributes.

Related Elements

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

Parent elements	**light** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<color>	Contains three floating-point numbers specifying the color of the light. See main entry.	None	1
<constant_attenuation sid="...">	See “Details.” The sid is optional.	1.0	0 or 1
<linear_attenuation sid="...">	See “Details.” The sid is optional.	0.0	0 or 1
<quadratic_attenuation sid="...">	Contains a floating-point number that describes the distance to the near clipping plane. The sid is optional.	None	1
<zfar sid="...">	See “Details.” The sid is optional.	0.0	0 or 1

Details

The **<constant_attenuation>**, **<linear_attenuation>**, and **<quadratic_attenuation>** are used to calculate the total attenuation of this light given a distance. The equation used is

```
A = constant_attenuation + ( Dist * linear_attenuation ) + (( Dist^2 ) *
quadratic_attenuation ).
```

Example

Here is an example of a **<point>** element:

```
<light id="blue">
  <technique_common>
    <point>
      <color>0.1 0.1 0.5</color>
      <linear_attenuation>0.3</linear_attenuation>
    </point>
  </technique_common>
</light>
```

polygons

Category: **Geometry**

Introduction

Provides the information needed for a mesh to bind vertex attributes together and then organize those vertices into individual polygons.

Concepts

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

Note: Polygons are not the preferred way of storing data. Use **<triangles>** or **<polylist>** for the most efficient representation of geometry. Use **<polygons>** only if holes are needed, and even then, only for the specific portions with holes.

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 also may be concave or even self-intersecting. The polygons may also contain holes. Polygon primitives that contain four or more vertices may be non-planar as well.

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 ambiguous. 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 (for example, 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 binormal 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: specified with the TEXTANGENT and TEXBINORMAL semantics and the set attribute on the **<input>** (shared) elements
- standard (geometric) tangents: specified with the TANGENT and BINORMAL semantics on the **<input>** (shared) elements

Attributes

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

count	**uint**	The number of polygon primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.
name	**xs:NCName**	Optional.

Related Elements

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

Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<p>	Contains a list of UInts that specifies the vertex attributes (indices) for an individual polygon. See “Details.”	None	0 or more
<ph>	Describes a polygon that contains one or more holes. See the following subsection.	0	0 or more
<extra>	See main entry.	N/A	0 or more

polygons / ph child element

The **<ph>** element has no attributes.

Child elements of **<ph>** must appear in the following order if present:

Name/example	Description	Default	Occurrences
<p>	Contains a list of UInts that specifies the vertex attributes (indices) for an individual polygon. See “Details.”	None	1
<h>	Contains a list of UInts that specifies the indices of a hole in the polygon specified by **<p>**. See “Details.”	None	1 or more

Details

The indices in a **<p>** (“primitive”) (or **<h>**) element refer to different inputs depending on their order. The first index in a **<p>** element refers to all inputs with an offset of 0. The second index refers to all inputs with an offset of 1. Each vertex of the polygon is made up of one index into each input. After each input is used, the next index again refers to the inputs with offset of 0 and begins a new vertex.

The winding order of vertices produced is counter-clockwise and describes 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.

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>** (shared) element semantics. The **<p>** element index values indicate the order in which the input values are used:

```
<mesh>
  <source id="position" />
  <source id="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <polygons count="1" material="Bricks">
```

```
      <input semantic="VERTEX" source="#verts" offset="0"/>
      <input semantic="NORMAL" source="#normal" offset="1"/>
      <p>0 0  2 1  3 2  1 3</p>
    </polygons>
  </mesh>
```

Here's a simple example of how to specify geometric tangents. (Note that, because the normal and tangent inputs both have an **offset** of 1, they share an entry in the **<p>** element.)

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

Here's a simple example of how to specify texture space tangents. (Note that the texture space tangents are associated with the specific set of texture coordinates by the set attribute and not the offset or the order of the inputs.)

```
<mesh>
  <source id="position"/>
  <source id="normal"/>
  <source id="tex-coord"/>
  <source id="tex-tangent"/>
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <polygons count="1" material="Bricks">
    <input semantic="VERTEX" source="#verts" offset="0"/>
    <input semantic="NORMAL" source="#normal" offset="1"/>
    <input semantic="TEXCOORD" source="#tex-coord" offset="2" set="0"/>
    <input semantic="TEXTANGENT" source="#tex-tangent" offset="3" set="0"/>
    <p>0 0 0 1  2 1 2 0  3 2 1 2  1 3 3 3</p>
  </polygons>
</mesh>
```

polylist

Category: **Geometry**

Introduction

Provides the information needed for a mesh to bind vertex attributes together and then organize those vertices into individual polygons.

Concepts

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

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

The polygons described in **<polylist>** can contain an arbitrary numbers of vertices. Polylist primitives that contain four or more vertices may be nonplanar as well.

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 ambiguous. 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 (for example, 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 binormal 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: specified with the **TEXTANGENT** and **TEXBINORMAL** semantics and the set attribute on the **<input>** (shared) elements
- standard (geometric) tangents: specified with the **TANGENT** and **BINORMAL** semantics on the **<input>** (shared) elements.

Attributes

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

name	**xs:NCName**	The text string name of this element. Optional.
count	**uint**	The number of polygon primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Parent elements	**mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<vcount>	Contains a list of integers, each specifying the number of vertices for one polygon described by the **<polylist>** element.	None	0 or 1
<p>	Contains a list of integers that specify the vertex attributes (indices) for an individual polylist. (“p” stands for “primitive”.)	None	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

The winding order of vertices produced is counter-clockwise and describes 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.

Example

Here is an example of a **<polylist>** element that describes two quadrilaterals and a triangle. The **<polylist>** element contains two **<source>** elements that contain the position and normal data, according to the **<input>** (shared) element semantics. The **<p>** element index values indicate the order in which the input values are used:

```
<mesh>
  <source id="position" />
  <source id="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <polylist count="3" material="Bricks">
    <input semantic="VERTEX" source="#verts" offset="0" />
    <input semantic="NORMAL" source="#normal" offset="1" />
    <vcount>4 4 3</vcount>
    <p>0 0  2 1  3 2  1 3  4 4  6 5  7 6  5 7  8 8  10 9  9 10</p>
  </polylist>
</mesh>
```

rotate

Category: **Transform**

Introduction

Specifies how to rotate an object around an axis.

Concepts

Rotations change the orientation of objects in a coordinate system without any translation. Computer graphics techniques apply a rotational transformation in order to orient or otherwise move values with respect to a coordinate system. Conversely, rotation can mean the translation of the coordinate axes about the local origin.

This element contains an angle and a mathematical vector that represents the axis of rotation.

Attributes

The **<rotate>** element has the following attribute:

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	**node**, **technique_common** / **mass_frame** in **rigid_body** and **instance_rigid_body**, **shape**, **ref_attachment**, **attachment**
Child elements	None
Other	None

Details

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 followed by an angle in degrees.

For more information about how transformation elements are applied, see **<node>**.

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>
```

sampler

Category: **Animation**

Introduction

Declares an interpolation sampling function for an animation.

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, as are interpolation type symbolic names. Animation channels direct the output data values of the sampler to their targets.

Animation Curves (<animation>/<sampler>)

Animations use curves to define how animated parameters evolve over time. The definition of the curves is similar to the definitions for the **<geometry>**/**<spline>**, except that there is a special one-dimensional axis that contains the keys for the animation. The keys define how a given parameter, or a set of parameters, evolves with time throughout the animation.

Keys are often TIME values, but they can be any other variable. For example, it is possible to associate the rotation of a wheel of a train with the position of the train on the track, so, by moving the train forward or backward, the wheel and other mechanisms can automatically move.

Animations are limited to monotonic curves in the key axis. In other words, animation keys need to be sorted in increasing order of **INPUT** and cannot be duplicated. This implies that animation curves cannot be closed.

The keys are stored in the **<source>** array, and they replace the first axis of all the **POSITION** inputs of the **<geometry>**/**<spline>**. Several parameters can be animated with different curves with the same key values. Those parameters are given by the **OUTPUT** array.

In short:

POSITION[*i*].X = **INPUT**[*i*]

POSITION[*i*].Y = **OUTPUT**[*i*]

And for n curves, the point i of the curve j is:

POSITION[*j*][*i*] = **INPUT**[*j*][*i*]

POSITION[*j*][*i*+1] = **OUTPUT**[*j*][*i*]

Attributes

The **<sampler>** element has the following attribute:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.

Related Elements

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

Parent elements	**animation**
Child elements	See the following subsection.
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<input> (unshared)	At least one **<input>** (unshared) element must have a semantic attribute whose value is **INTERPOLATION**. See main entry.	None	1 or more

Details

Sampling points are described by the **<input>** elements, which 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**, or **OUTPUT**.

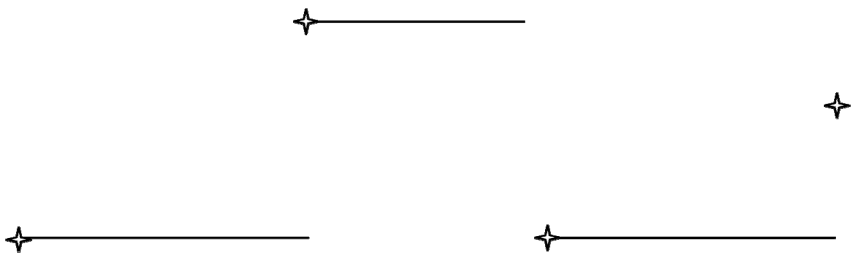
COLLADA recognizes the following interpolation types: **LINEAR**, **BEZIER**, **CARDINAL**, **HERMITE**, **BSPLINE**, and **STEP**. These symbolic names are held in a **<source>** element that contains a **<Name_array>** that stores them. These values are fed into the sampler by the **INTERPOLATION** **<input>** element.

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

For more information, see “Curve Interpolation” in Chapter 4: Programming Guide.

STEP Interpolation

Animation curves allow an additional type of interpolation: **STEP**. This says that the value remains constant to the value of the first point of the segment, until the next segment, as in the following curve:



The COLLADA code for this would be:

```
<animation>
  <source id="time_axis" >
      <float_array count="4"… >
       … <technique_common><accessor>
            <param name="TIME">
         …</accessor></technique_common>
   …</source>
  <source id="positions" >
       <float_array count="4" …>
        <technique_common>… <accessor>
            <param name="name_of_parameter_animated" type="float" …
         …</accessor></technique_common>
```

```
    …</source>
   <source id="interpolations" >
      <Name_array count="4"> STEP STEP STEP STEP </Name_array>    <!-- last one
ignored -->
       <technique_common>… … <accessor>
            <param name="INTERPOLATION" type="Name" …
        …</accessor></technique_common>
    …</source>
   <sampler>
       <input semantic="INPUT" source = "#time_axis" />
       <input semantic="OUTPUT" source="#positions" />
       <input semantic="INTERPOLATION" source="#interpolations" />
```

Linear Animation Curves

The **LINEAR** interpolation is similar to **STEP**, but the parameter's value is interpolated linearly between the key values.

Bézier and Hermite Animation Curves

BEZIER and **HERMITE** interpolations are similar to the description given for **<spline>** except that there is no **POSITION** **<input>** semantic, but rather **INPUT** and **OUTPUT** semantics. The **INPUT** and **OUTPUT** semantics are always 1-D parameters. As explained already, if **OUTPUT** has more than one dimension, then several parameters are interpolated independently using the same key values. The **IN_TANGENT** and **OUT_TANGENT** semantics have one key value, and then one value for each parameter.

The same equations for cubic Bézier and Hermite interpolation already defined for **<spline>** are to be used, with the following geometry vector, for parameter *j*, segment[*i*]:

For Bézier:

- P_0 is (**INPUT**[*i*] , **OUTPUT**[*j*][*i*])
- C_0 (or T_0) is (**OUT_TANGENT**[0][*i*] , **OUT_TANGENT**[*j*][*i*])
- C_1 (or T_1) is (**IN_TANGENT**[0][*i*+1], **IN_TANGENT**[j][i+1])
- P_1 is (**INPUT**[i+1], **OUTPUT**[*j*][*i*+1])

Special Case: 1-D Tangent Values

Some exporters have been exporting curves with a degenerate form of tangent. This is not supported by the COLLADA specification, and the degenerate cases should disappear with updates to the affected exporters. The following is provided for informational purposes only.

In this special case of 1-D tangent data, the **OUT_TANGENT** and **IN_TANGENT** do not include the key values, and therefore have the same dimension as the **OUTPUT** array.

The missing key values are provided as a linear interpolation of the keys provided by the **INPUT** segment. The geometry vector values are provided the same way as for a regular animation curve:

- P_0 is (**INPUT**[*i*] , **OUTPUT**[*j*][*i*]
- C_0 is (**INPUT**[*i*]/3 + **INPUT**[*i*+1] *2/3 , **OUT_TANGENT**[*j*][*i*])
- C_1 is (**INPUT**[*i*]*2/3 + **INPUT**[*i*+1]/3, **IN_TANGENT**[*j*][*i*+1])
- C_1 is (**INPUT**[*i*+1], **OUTPUT**[*j*][*i*+1])

B-Spline and Cardinal Animation Curves

The same principles discussed previously apply to **BSPLINE** and **CARDINAL** curves. The **POSITION** is given by combining the **INPUT** and **OUTPUT**. The same equations defined previously apply to these animation curves.

Example

Here is an example of a **<sampler>** element that evaluates the y-axis values of a key-frame source element whose id is "**group1_translate-anim-outputY**". The **INTERPOLATION** inputs are shown in their **<source>** element for added clarity:

```
<animation id="group1_translate-anim">
  <source id="group1_translate-anim-inputY">
   ...
  </source>
  <source id="group1_translate-anim-outputY">
   ...
  </source>
  <source id="group1_translate-anim-interpY">
    <Name_array count="3" id="group1_translate-anim-interpY-array">
      BEZIER BEZIER BEZIER
    </Name_array>
    <technique_common>
      <accessor count="3" source="#group1_translate-anim-interpY-array">
        <param name="Y" type="Name"/>
      </accessor>
    </technique_common>
  </source>
  <sampler id="group1_translate-anim-samplerY">
    <input semantic="INPUT" source="#group1_translate-anim-inputY"/>
    <input semantic="OUTPUT" source="#group1_translate-anim-outputY"/>
    <input semantic="IN_TANGENT" source="#group1_translate-anim-intanY"/>
    <input semantic="OUT_TANGENT" source="#group1_translate-anim-outtanY"/>
    <input semantic="INTERPOLATION" source="#group1_translate-anim-interpY"/>
  </sampler>
  <channel source="#group1_translate-anim-samplerY"
           target="group1/translate.Y"/>
</animation>
```

scale

Category: **Transform**

Introduction

Specifies how to change an object's size.

Concepts

Scaling changes the size of objects in a coordinate system without any rotation or translation. Computer graphics techniques apply a scale transformation to change the size or proportions of values with respect to a coordinate system axis.

This element contains a mathematical vector that represents the relative proportions of the x, y, and z axes of a coordinate system.

Attributes

The **<scale>** element has the following attribute:

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	**node**
Child elements	None
Other	None

Details

The **<scale>** element contains a list of three floating-point values. These values are organized into a column vector suitable for matrix composition.

For more information about how transformation elements are applied, see **<node>**.

Example

Here is an example of a **<scale>** element that describes a uniform increase in size of an object (or coordinate system) by a factor of two:

```
<scale>
  2.0 2.0 2.0
</scale>
```

scene

Category: **Scene**

Introduction

Embodies the entire set of information that can be visualized from the contents of a COLLADA resource.

Concepts

Each COLLADA document can contain, at most, one **<scene>** element.

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

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 no attributes.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<instance_physics_scene>	See main entry.	N/A	0 or more
<instance_visual_scene>	See main entry.	N/A	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

There is at most one **<scene>** element declared under the **<COLLADA>** document (root) element. The scene graph is built from the **<visual_scene>** elements instantiated under **<scene>**. The instantiated **<physics_scene>** elements describe any physics being applied to the scene.

Example

The following example shows a simple **<scene>** element that instantiates a visual scene with the id 'world':

```
<COLLADA>
  <scene>
    <instance_visual_scene url="#world"/>
```

```
    </scene>
</COLLADA>
```

skeleton

Category: **Controller**

Introduction

Indicates where a skin controller is to start to search for the joint nodes that it needs.

Concepts

As a scene graph increases in complexity, the same object might have to appear in the scene more than once. To save space, the actual data representation of an object can be stored once and referenced in multiple places. However, the scene might require that the object be transformed in various ways each time it appears. In the case of a skin controller, the object's transformation is derived from a set of external nodes.

There may be occasions where multiple instances of the same skin controller need to reference separate instances of a set of nodes. This is the case when each controller needs to be animated independently because, to animate a skin controller, you must animate the nodes that influence it.

There may also be occasions where instances of different skin controllers might need to reference the same set of nodes, for example when attaching clothing or armor to a character. This allows the transformation of both controllers from the manipulation of a single set of nodes.

Attributes

The **<skeleton>** element has no attributes.

Related Elements

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

Parent elements	**instance_controller**
Child elements	None
Other	None

Details

This element contains a URI of type **xs:anyURI**.

Example

The following example shows how the **<skeleton>** element is used to bind two controller instances that refer to the same locally defined **<controller>** element, identified as "skin", to different instances of a skeleton:

```
<library_controllers>
  <controller id="skin">
    <skin source="#base_mesh">
      <source id="Joints">
        <Name_array count="4"> Root Spine1 Spine2 Head </Name_array>
        ...
      </source>
      <source id="Weights"/>
      <source id="Inv_bind_mats"/>
      <joints>
        <input source="#Joints" semantic="JOINT"/>
```

```
      </joints>
      <vertex_weights/>
    </skin>
  </controller>
</library_controllers>
<library_nodes>
  <node id="Skeleton1" sid="Root">
    <node sid="Spine1">
      <node sid="Spine2">
        <node sid="Head"/>
      </node>
    </node>
  </node>
</library_nodes>
<node id="skel01">
   <instance_node url="#Skeleton1"/>
</node>
<node id="skel02">
  <instance_node url="#Skeleton1"/>
</node>
<node>
  <instance_controller url="#skin">
    <skeleton>#skel01</skeleton>
  </instance_controller>
</node>
<node>
  <instance_controller url="#skin">
    <skeleton>#skel02</skeleton>
  </instance_controller>
</node>
```

skew

Category: **Transform**

Introduction

Specifies how to deform an object along one axis.

Concepts

Skew (shear) deforms an object along one axis of a coordinate 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 to deform objects or to correct distortion in images.

This element contains an angle and two mathematical vectors that represent the axis of rotation and the axis of translation.

Attributes

The **<skew>** element has the following attribute:

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	**node**
Child elements	None
Other	None

Details

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 followed by two column vectors specifying the axes of rotation and translation.

For more information about how transformation elements are applied, see **<node>**.

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>
```

skin

Category: **Controller**

Introduction

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:

- Preprocessing, 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 attribute:

source	**xs:anyURI**	A URI reference to the base mesh (a static mesh or a morphed mesh). This also provides the bind-shape of the skinned mesh. Required.

Related Elements

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

Parent elements	**controller**
Child elements	None
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<bind_shape_matrix>	Provides extra information about the position and orientation of the base mesh before binding. Contains sixteen floating-point numbers representing a four-by-four matrix in column-major order; it is written in row-major order in the COLLADA document for human readability. If **<bind_shape_matrix>** is not specified then an identity matrix may be used as the **<bind_shape_matrix>**. This element has no attributes.	None	0 or 1
<source>	Provides most of the data required for skinning the given base mesh. See main entry.	N/A	3 or more
<joints>	Aggregates the per-joint information needed for this skin. See main entry.	N/A	1
<vertex_weights>	Describes a per-vertex combination of joints and weights used in this skin. An index of –1 into the array of joints refers to the bind shape. Weights should be normalized before use. See main entry.	N/A	1
<extra>	See main entry.	N/A	0 or more

Details

The skinning calculation for each vertex v in a bind shape is

$$outv = \sum_{i=0}^{n} \{((v * \mathrm{BSM}) * \mathrm{IBM}i * \mathrm{JM}i) * \mathrm{JW}\}$$

where:

- n: number of joints that influence vertex v
- BSM: bind shape matrix
- IBMi: inverse bind matrix of joint i
- JMi: joint matrix of joint i
- JW: joint weight/influence of joint i on vertex v

Common optimizations include:

- $(v * \mathrm{BSM})$ is calculated and stored at load time.

Definitions related to skinning in COLLADA:

- Bind shape (or base mesh): The vertices of the mesh referred to by the source attribute of the **<skin>** element.
- Joints: Nodes specified by sid in the **<source>** referred to by the **<input>** (unshared) element with semantic="**JOINT**". The sid s are typically stored in a **<Name_array>** where one name represents one sid (node). Upon instantiation of a skin controller, the **<skeleton>** elements define where to start the sid lookup. The joint matrices can be obtained at runtime from these nodes.
- Weights: Values in the **<source>** referred to by the **<input>** (unshared) element with semantic="**WEIGHT**". Typically stored in a **<float_array>** and taken one floating-point number at a time. The **<vertex_weights>** element describes the combination of joints and weights used by the skin.
- Inverse bind matrix: Values in the **<source>** element referred to by the **<input>** (unshared) element with semantic="**INV_BIND_MATRIX**". Typically stored in a **<float_array>** taken 16 floating-point numbers at a time. The **<joints>** element associates the joints to their inverse bind matrices.
- Bind shape matrix: A single matrix that represents the transform of the bind shape before skinning.

Example

Here is an example of a **<skin>** element with the allowed attributes:

```
<controller id="skin">
    <skin source="#base_mesh">
      <source id="Joints">
        <Name_array count="4"> Root Spine1 Spine2 Head </Name_array>
        ...
      </source>
      <source id="Weights">
        <float_array count="4"> 0.0 0.33 0.66 1.0 </float_array>
        ...
      </source>
      <source id="Inv_bind_mats">
        <float_array count="64"> ... </float_array>
        ...
      </source>
      <joints>
        <input semantic="JOINT" source="#Joints"/>
        <input semantic="INV_BIND_MATRIX" source="#Inv_bind_mats"/>
      </joints>
      <vertex_weights count="4">
        <input semantic="JOINT" source="#Joints"/>
        <input semantic="WEIGHT" source="#Weights"/>
        <vcount>3 2 2 3</vcount>
        <v>
        -1 0  0 1  1 2
        -1 3  1 4
        -1 3  2 4
        -1 0  3 1  2 2
        </v>
      </vertex_weights>
    </skin>
  </controller>
```

source

(core)

Category: **Data Flow**

Introduction

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

Note: For **<source>** in **<sampler*>** elements, see those elements.

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**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Required.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**morph**, **animation**, **mesh**, **convex_mesh**, **skin**, **spline**
Child elements	See the following subsection.
Other	**accessor**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
array_element	A data array element. Can be one of: • **<IDREF_array>** • **<Name_array>** • **<bool_array>** • **<float_array>** • **<int_array>** See main entries.	None	0 or 1
<technique_common>	Specifies source information for the common profile that all COLLADA implementations must support. See “The Common Profile” section for usage information and the following subsection for child element details.	N/A	0 or 1

Name/example	Description	Default	Occurrences
<technique> (core)	Each **<technique>** specifies source information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more

Child Elements of source / technique_common

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	1

Details

Example

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

```
<source id="color_source" name="Colors">
  <float_array id="values" count="3">
    0.8 0.8 0.8
  </float_array>
  <technique_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_common>
</source>
```

spline

Category: **Geometry**

Introduction

Describes a multisegment spline with control vertex (CV) and segment information.

Concepts

The organization of **<spline>** is very similar to that of **<mesh>**. A **<spline>** contains **<source>** elements that provide the attributes and a **<control_vertices>** element to "assemble" the attribute streams. Information about each segment is stored with the information about its preceding control vertex.

Attributes

The **<spline>** element has the following attribute:

closed	**bool**	Whether there is a segment connecting the first and last control vertices. The default is "false", indicating that the spline is open. Optional.

Related Elements

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

Parent elements	**geometry**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source>	Provides the values for the CVs and segments of the spline. See main entry.	N/A	1 or more
<control_vertices>	Describes the CVs of the spline. See main entry.	N/A	1
<extra>	See main entry.	N/A	0 or more

Details

For more information, see:

- **<control_vertices>**
- "Curve Interpolation" in Chapter 4: Programming Guide.

Example

Here is an example of an empty **<spline>** element with the allowed attributes:

```
<spline closed="true">
  <source id="CVs-Pos" />
  <source id="CVs-Interp" />
  <source id="CVs-LinSteps" />
  <control_vertices>
    <input semantic="POSITION" source="#CVs-Pos"/>
```

```
      <input semantic="INTERPOLATION" source="#CVs-Interp"/>
      <input semantic="LINEAR_STEPS" source="#CVs-LinSteps"/>
    </control_vertices>
  </spline>
```

spot

Category: **Lighting**

Introduction

Describes a spot light source.

Concepts

A spot light source radiates light in one direction in a cone shape from a known location in space. 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.

The light's default direction vector in local coordinates is [0,0,-1], pointing down the negative z axis. The actual direction of the light is defined by the transform of the node in which the light is instantiated.

Attributes

The **<spot>** element has no attributes.

Related Elements

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

Parent elements	**light** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<color>	Contains three floating-point spot numbers specifying the color of the light. See main entry.	None	1
<constant_attenuation sid="...">	The sid is optional.	1.0	0 or 1
<linear_attenuation sid="...">	The sid is optional.	0.0	0 or 1
<quadratic_attenuation sid="...">	The sid is optional.	0.0	0 or 1
<falloff_angle sid="...">	The sid is optional.	180.0	0 or 1
<falloff_exponent sid="...">	The sid is optional.	0.0	0 or 1

Details

The **<constant_attenuation>**, **<linear_attenuation>**, and **<quadratic_attenuation>** are used to calculate the total attenuation of this light given a distance. The equation used is

```
A = constant_attenuation + ( Dist * linear_attenuation ) + (( Dist^2 ) *
quadratic_attenuation )
```

The **<falloff_angle>** and **<falloff_exponent>** are used to specify the amount of attenuation based on the direction of the light.

Example

Here is an example of a **<spot>** element:

```
<light id="blue">
  <technique_common>
    <spot>
      <color>0.1 0.1 0.5</color>
      <linear_attenuation>0.3</linear_attenuation>
    </spot>
  </technique_common>
</light>
```

targets

Category: **Controller**

Introduction

Declares morph targets, their weights, and any user-defined attributes associated with them.

Concepts

The **<targets>** element declares the morph targets and the morph weights. The **<input>** elements define the set of meshes to be blended, and the array of weights used to blend between them. They can also be used to specify additional information to be associated with the morph targets.

Attributes

The **<targets>** element has no attributes.

Related Elements

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

Parent elements	**morph**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (unshared)	Must occur once with **semantic="MORPH_TARGET"** and once with **semantic="MORPH_WEIGHT"**. See main entry.	None	2 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a complete **<targets>** element:

```
<targets>
  <input source="#morph-targets" semantic="MORPH_TARGET">
  <input source="#morph-weights" semantic="MORPH_WEIGHT">
</targets>
```

technique

(core)

Category: **Extensibility**

Introduction

Declares the information used to process some portion of the content. Each technique conforms to an associated profile.

For **<technique>** in **<profile_*>** elements, see "**<technique>** (FX)."

Concepts

A technique describes information needed by a specific platform or program. The platform or program is specified with the profile attribute. Two things define the context for a technique: its profile and its parent element in the instance document.

Techniques generally 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 usually 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 attribute:

profile	**xs:NMTOKEN**	The type of profile. This is a vendor-defined character string that indicates the platform or capability target for the technique. Required.
xmlns	**xs:anyURI**	This XML Schema namespace attribute identifies an additional schema to use for validating the content of this instance document. Optional.

Related Elements

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

Parent elements	**extra**, **source** (core), **light**, **optics**, **imager**, **force_field**, **physics_material**, **physics_scene**, **rigid_body**, **rigid_constraint**, **instance_rigid_body**, **bind_material**
Child elements	See "Details"
Other	None

Details

The **<technique>** element can contain any well-formed XML data. Any data that can be, will be validated against the COLLADA schema. It is also possible to specify another schema to use for validating the data. Anything else will also be considered legal, but cannot actually be validated.

Example

Here is an example of the different things that can be done in a **<technique>**:

```
<technique profile="Max" xmlns:max="some/max/schema">
  <param name="wow" sid="animated" type="string">a validated string parameter
from the COLLADA schema.</param>
  <max:someElement>defined in the Max schema and validated.</max:someElement>
```

```
  <uhoh>something well-formed and legal, but that can't be validated because
there is no schema for it!</uhoh>
</technique>
```

The following example shows roughly equivalent operations for the platform or profile named "OTHER" and for all other platforms (the **technique_common** information):

```
<channel source="#YFOVSampler" target="Camera01/YFOV"/>
...
<camera id="#Camera01">
   <optics>
      <technique_common>
          <perspective>
          <yfov sid="YFOV">45.0</yfov>
          <aspect_ratio>1.33333</aspect_ratio>
          <znear>1.0</znear>
          <zfar>1000.0</zfar>
          </perspective>
      </technique_common>
      <technique profile="OTHER">
          <param sid="YFOV" type="float">45.0</param>
          <otherStuff type="MySpecialCamera">DATA</otherStuff>
      </technique>
   </optics>
</camera>
```

technique_common

Category: **Extensibility**

Introduction

Specifies information for a specific element for the common profile that all COLLADA implementations must support.

Concepts

Specifies technique information that consuming applications can use if no technique specific to the application is provided in the COLLADA document.

In other words, if an element has **<technique>** child elements for one or more specific profiles, applications reading the COLLADA document should use the technique most appropriate for the application. If none of the specific **<technique>**s is appropriate, the application must use the element's **<technique_common>** instead, if one is specified.

Each element's **<technique_common>** attributes and children are unique. Refer to each parent element for details.

Attributes

See main entries for each parent element.

Related Elements

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

Parent elements	**bind_material**, **instance_rigid_body**, **light**, **optics**, **physics_material**, **physics_scene**, **rigid_body**, **rigid_constraint**, **source** (core)
Child elements	See main entries for each parent element.
Other	**technique**

Remarks

For additional information about the common profile and customized profiles, see "The Common Profile" section.

Example

See parent elements.

translate

Category: **Transform**

Introduction

Changes the position of an object in a local coordinate system.

Concepts

This element contains a mathematical vector that represents the distance along the x, y, and z axes.

Computer graphics techniques apply a translation transformation 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 attribute:

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.

Related Elements

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

Parent elements	**node**, **shape**, **technique_common** / **mass_frame** in **rigid_body** and **instance_rigid_body**, **ref_attachment**, **attachment**
Child elements	None
Other	None

Details

The **<translate>** element contains a list of three floating-point values. These values are organized into a column vector suitable for a matrix composition.

For more information about how transformation elements are applied, see **<node>**.

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>
```

triangles

Category: **Geometry**

Introduction

Provides the information needed to for a mesh to bind vertex attributes together and then organize those vertices into individual triangles.

Concepts

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

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:

name	**xs:NCName**	The text string name of this element. Optional.
count	**uint**	The number of triangle primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<p>	(“p” stands for primitive.) Contains indices that describe the vertex attributes for a number of triangles. The indices reference into the parent’s **<source>** elements that are referenced by the **<input>** elements. This element has no attributes. See “Details.”	None	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

The indices in a **<p>** element refer to different inputs depending on their order. The first index in a **<p>** element refers to all inputs with an **offset** of 0. The second index refers to all inputs with an **offset** of 1.

Each vertex of the triangle is made up of one index into each input. After each input is used, the next index again refers to the inputs with **offset** of 0 and begins a new vertex.

The winding order of vertices produced is counterclockwise and describes 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.

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 in which the input values are used:

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

trifans

Category: **Geometry**

Introduction

Provides the information needed for a mesh to bind vertex attributes together and then organize those vertices into connected triangles.

Concepts

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

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 the first, second, and third vertices. Each subsequent triangle is formed from the current vertex, reusing the first and the previous vertices.

Attributes

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

name	**xs:NCName**	The text string name of this element. Optional.
count	**uint**	The number of triangle-fan primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<p>	(“p” stands for primitive.) Contains indices that describe the vertex attributes for an arbitrary number of connected triangles. The indices reference into the parent’s **<source>** elements that are referenced by the **<input>** elements. This element has no attributes. See “Details.”	None	0 or more
<extra>	See main entry.	N/A	0 or more

Details

A **<trifans>** element can contain a sequence of **<p>** elements.

The indices in a **<p>** element refer to different inputs depending on their order. The first index in a **<p>** element refers to all inputs with an **offset** of 0. The second index refers to all inputs with an **offset** of 1. Each vertex of the triangle is made up of one index into each input. After each input is used, the next index again refers to the inputs with **offset** of 0 and begins a new vertex.

The winding order of vertices produced is counterclockwise and describes 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.

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 in which the input values are used:

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

tristrips

Category: **Geometry**

Introduction

Provides the information needed for a mesh to bind vertex attributes together and then organize those vertices into connected triangles.

Concepts

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

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 the 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:

name	**xs:NCName**	The text string name of this element. Optional.
count	**uint**	The number of triangle-strip primitives. Required.
material	**xs:NCName**	Declares a symbol for a material. This symbol is bound to a material at the time of instantiation; see **<instance_geometry>** and **<bind_material>**. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

The <tristrips> element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	See main entry.	None	0 or more
<p>	(“p” stands for primitive.) Contains indices that describe the vertex attributes for an arbitrary number of connected triangles. The indices reference into the parent’s **<source>** elements that are referenced by the **<input>** elements. This element has no attributes. See “Details.”	None	0 or more
<extra>	See main entry.	N/A	0 or more

Details

A <tristrips> element can contain a sequence of **<p>** elements.

The indices in a **<p>** element refer to different inputs depending on their order. The first index in a **<p>** element refers to all inputs with an **offset** of 0. The second index refers to all inputs with an **offset** of 1. Each vertex of the triangle is made up of one index into each input. After each input is used, the next index again refers to the inputs with **offset** of 0 and begins a new vertex.

The winding order of vertices produced is counterclockwise for the first, (third, fifth, etc.) triangle and clockwise for the second (fourth, sixth, etc.) and describes 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.

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 in which the input values are used:

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

vertex_weights

Category: **Controller**

Introduction

Describes the combination of joints and weights used by a skin.

Concepts

The **<vertex_weights>** element associates a set of joint-weight pairs with each vertex in the base mesh.

Attributes

The **<vertex_weights>** element has the following attribute:

count	uint	The number of vertices in the base mesh. Required.

Related Elements

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

Parent elements	**skin**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (shared)	One of the **<input>** elements, as a child of **<vertex_weights>**, must specify **semantic="JOINT"**. The **<input>** elements describe the joints and the attributes to be associated with them. See main entry.	None	2 or more
<vcount>	Contains a list of integers, each specifying the number of bones associated with one of the influences defined by **<vertex_weights>**. This element has no attributes.	None	0 or 1
<v>	Contains a list of indices that describe which bones and attributes are associated with each vertex. An index of -1 into the array of joints refers to the bind shape. Weights should be normalized before use. This element has no attributes.	None	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of an empty **<vertex_weights>** element:

```
<skin>
  <vertex_weights count="">
    <input semantic="JOINT"/>
    <input/>
    <vcount/>
    <v/>
    <extra/>
  </vertex_weights>
</skin>
```

Here is an example of a more complete **<vertex_weights>** element. Note that the **<vcount>** element says that the first vertex has 3 bones, the second has 2, etc. Also, the **<v>** element says that the first vertex is weighted with `weights[0]` towards the bind shape, `weights[1]` towards bone 0, and `weights[2]` towards bone 1:

```
<skin>
  <source id="joints"/>
  <source id="weights"/>
  <vertex_weights count="4">
      <input semantic="JOINT" source="#joints"/>
      <input semantic="WEIGHT" source="#weights"/>
    <vcount>3 2 2 3</vcount>
    <v>
        -1 0  0 1  1 2
        -1 3  1 4
        -1 3  2 4
        -1 0  3 1  2 2
    </v>
  </vertex_weights>
</skin>
```

vertices

Category: **Geometry**

Introduction

Declares the attributes and identity of mesh-vertices.

Concepts

The **<vertices>** element describes mesh-vertices in a mesh. 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**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Required.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**mesh**, **convex_mesh**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<input> (unshared)	One input must specify **semantic**="**POSITION**" to establish the topological identity of each vertex in the mesh. See main entry.	None	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

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

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

visual_scene

Category: **Scene**

Introduction

Embodies the entire set of information that can be visualized from the contents of a COLLADA resource.

Concepts

The hierarchical structure of the **visual_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 **<visual_scene>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_visual_scenes**
Child elements	See the following subsection.
Other	**instance_visual_scene**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<node>	See main entry.	N/A	1 or more
<evaluate_scene name="..."> **<render.../>** **</evaluate_scene>**	The **<evaluate_scene>** element declares information specifying how to evaluate this **visual_scene**. Its optional name attribute (**xs:NCName**) is the text string name of this element. There must be at least one **<render>** element; see main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Details

The **<visual_scene>** element forms the root of the scene graph topology.

There might be multiple **<visual_scene>** elements declared within a **<library_visual_scenes>** element. The **<instance_visual_scene>** element in the **<scene>** element, which is declared under the **<COLLADA>** document (root) element, declares which **<visual_scene>** element is to be used for the document.

Example

The following example shows a simple outline of a COLLADA resource containing a **<visual_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/2005/11/COLLADASchema.xsd"
version="1.4.1">
  <library_visual_scenes>
    <visual_scene id="world">
      <node id="root"/>
      </visual_scene>
  </library_visual_scenes>
  <scene>
    <instance_visual_scene url="#world"/>
  </scene>
</COLLADA>
```

COLLADA supports layering and visibility. Each node has a layer attribute that is a list of **xs:NCName**. The node belongs to each layer that it lists there. Then, in the visual scene, there is an **<evaluate_scene>** that describes how a scene is to be rendered. This is also where one would use full-screen effects.

The following document fragment shows how this works. This solution works for layers. It might not be ideal for "visibility," but you can achieve the same results with it:

```
<visual_scene>
  <node id="Node1" layer="visible"/>
  <node id="Node2" layer="visible"/>
  <node id="Node3" layer="notvisible"/>
  <node id="camera"><instance_camera url="#cam01"/></node>
  <evaluate_scene>
    <render camera_node="#camera">
      <layer>visible</layer>
    </render>
  </evaluate_scene>
</visual_scene>
```

Chapter 6: COLLADA Physics Reference

Introduction

This section covers the elements that compose COLLADA Physics.

Elements by Category

This chapter lists elements in alphabetical order. The following tables list elements by category, for ease in finding related elements.

Analytical Shape

box	Declares an axis-aligned, centered box primitive.
capsule	Declares a capsule primitive that is centered on, and aligned with, the local y axis.
convex_mesh	Contains or refers to information that describes basic geometric meshes.
cylinder	Declares a cylinder primitive that is centered on, and aligned with, the local y axis.
plane	Defines an infinite plane primitive.
sphere	Describes a centered sphere primitive.
tapered_capsule	Describes a tapered capsule primitive that is centered on, and aligned with, the local y axis.
tapered_cylinder	Describes a tapered cylinder primitive that is centered on, and aligned with, the local y axis.

Physics Material

instance_physics_material	Declares the instantiation of a **<physics_material>**.
library_physics_materials	Declares a module of **<physics_material>** elements.
physics_material	Defines the physical properties of an object using a technique/profile with parameters.

Physics Model

instance_physics_model	Allows the instantiation of a physics model within another physics model, or in a physics scene.
instance_rigid_body	Allows the instantiation of a **<rigid_body>** within an **<instance_physics_model>**.
instance_rigid_constraint	Allows the instantiation of a **<rigid_constraint>** within an **<instance_physics_model>**.
library_physics_models	Declares a module of **<physics_model>** elements.
physics_model	Allows for building complex combinations of rigid bodies and constraints that may be instantiated multiple times.
rigid_body	Describes simulated bodies that do not deform.
rigid_constraint	Connects components, such as **<rigid_body>**, into complex physics models with moveable parts.

Physics Scene

force_field	Provides a general container for force fields.
instance_force_field	Declares the instantiation of a **<force_field>**.
instance_physics_scene	Declares the instantiation of a **<physics_scene>**.
library_force_fields	Declares a module of **<force_field>** elements.
library_physics_scenes	Declares a module of **<physics_scene>** elements.
physics_scene	Specifies an environment in which physical objects are instantiated and simulated.
attachment	Defines an attachment to a rigid body or a node.
ref_attachment	Defines an attachment to a rigid body or a node to be used as a reference frame.

Introduction

COLLADA Physics enables content creators to attach physical properties to objects in a visual scene.

Nodes in a visual scene can be controlled by a physical simulation. This is done by instantiating rigid bodies and constraints in a physics scene. The rigid bodies then target the appropriate nodes in the visual scene. Rigid bodies have physical properties, such as mass or inertia, that influence the simulation. Rigid bodies can collide with each other by specifying one or more collision shapes, such as geometry meshes, capsules, or boxes.

A rigid body can be dynamic or kinematic. A dynamic rigid body controls the transforms of a single visual node, making that visual node completely driven by the physical simulation. A kinematic (nondynamic) rigid body is controlled by an external animation. For example, the kinematic rigid body corresponding to the foot of a soccer player could be controlled through key-framed animations, while a soccer ball would have a dynamic rigid body.

Pairs of rigid bodies can be connected through a rigid constraint. A rigid constraint has many parameters that allow specifying angular and linear degrees of freedom. For example, a car wheel can be constrained to a chassis, so that it rotates only along the x axis, and doesn't translate or rotate along other axes.

Rigid bodies and constraints are grouped in a physics model, to define complex physical objects such as a car or a rag doll. The latter is commonly used to simulate a falling or dying character.

In version 1.4, COLLADA Physics is capable of expressing rigid dynamic systems only and does not support features such as cloth or fluids.

About Physical Units

As long as values correspond properly, Newtonian simulations (discounting quantum and relativistic effects) can be run correctly. For example, if distances and lengths are specified in meters and time is in seconds, then forces are in newtons. For this reason, many physics engines are unitless and COLLADA physics does not itself require the specification of units for each component.

If needed, units should be taken from the "base" of the COLLADA document. COLLADA uses the following units of measurement:

Measurement	Unit
time	seconds (standard units)
angle	degrees (standard units)
mass	kilograms (standard units)
distance	meters (default units). The **<asset>** element includes the **<unit>** element, by which the measure of distance can be redefined for the corresponding asset.

About Inertia

The quantity that describes how much force is needed to accelerate a body is called inertia and is indicated with the letter "I". Its rotational equivalent is called moment of inertia.

For a single, infinitely small mass "m" at a distance "r" (also called "arm") from the center of rotation:

$$I = m r^2$$

The angular momentum of such a mass is the product of its inertia and its angular velocity.

For a rigid-body with an arbitrary shape and mass distribution, we can think of the body as a set of discrete particles with varying mass.

The body's moment of inertia will be the sum of the products of the masses of the particles and the square of their distances from the rotation axis.

The inertia may be derived at any point and with any orientation relative to the rigid body. However, it is simpler to express it at the center of mass (it's also more intuitive, because a freely rotating body will always rotate around its center of mass).

The inertia is usually expressed as a tensor of the second rank, and is written in the form of a "3x3 matrix". It is computed as:

$$I_{CM} \equiv \begin{bmatrix} \sum_i m_i (y_i^2 + z_i^2) & -\sum_i m_i x_i y_i & -\sum_i m_i x_i z_i \\ -\sum_i m_i y_i x_i & \sum_i m_i (z_i^2 + x_i^2) & -\sum_i m_i y_i z_i \\ -\sum_i m_i z_i x_i & -\sum_i m_i z_i y_i & \sum_i m_i (x_i^2 + y_i^2) \end{bmatrix}$$

Where:

- m_i are the masses of the particles
- (x_i, y_i, z_i) are the particle positions

The diagonal elements of the inertia tensor are called the moments of inertia, while the off-diagonal elements are the products of inertia.

Note that the inertia tensor (at the CM) is symmetric.

Also, if the reference frame is aligned with the (local) principal axes of the rigid-body, the off-diagonal elements will become 0.

Such a purely diagonal 3x3 tensor may be expressed with 3 values (`float3`).

Then, we could for example look at the top-left element in "isolation" and interpret it as:

- for a rotation about the local x axis, the inertia is the sum of the mass of each particle times its "arm on the Y-Z plane", squared. Which is of course the definition of moment of inertia ($I = m r^2$).

If we were to sample at an off-center (of mass) point, there is an additional moment of inertia:

$$I_i \equiv \begin{bmatrix} m (y_0^2 + z_0^2) & -m x_0 y_0 & -m x_0 z_0 \\ -m y_0 x_0 & m (z_i^2 + x_i^2) & -m y_0 z_0 \\ -m z_0 x_0 & -m z_0 y_0 & m (x_0^2 + y_0^2) \end{bmatrix}$$

Where:

- m is the mass of the body
- (x_0, y_0, z_0) is the point at which the inertia is computed

The inertia of the body at an arbitrary point is the sum of the inertia tensor (at CM) and this "offset".

New Geometry Types

Physics engines run more efficiently with analytical shapes (primitives) and convex hulls as collision shapes than they do with arbitrary shapes. This is why COLLADA includes some primitive geometry types, such as **<box>** and **<sphere>**, as well as **<convex_mesh>**.

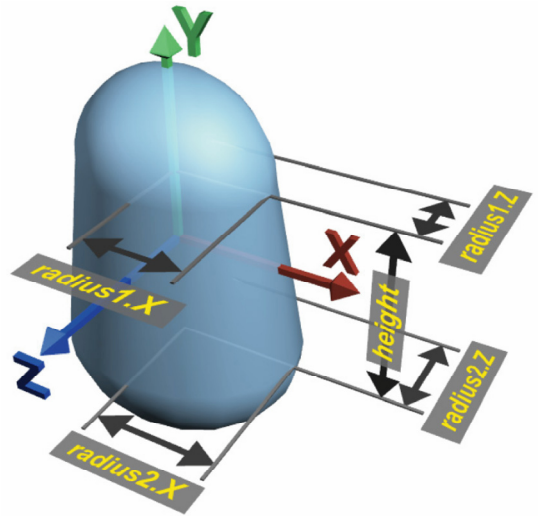
Other than **<convex_mesh>**, these shapes are not meant for rendering because meshes, subdivision surfaces, and so on are better suited for that purpose.

This system – of analytical shapes plus the convex mesh – can describe any shape.

Coordinate System Conventions for Geometric Primitives

For **<cylinder>**, **<tapered_cylinder>**, **<capsule>**, and **<tapered_capsule>**, the main axis is the y axis (right-handed, positive-y up) and the radii are given along the x and z axes, as shown in the following example:

```
<tapered_cylinder>
  <height> 2.0 </height>
  <radius1> 1.0 2.0 </radius1>   <!- radius1.X and radius1.Z -->
  <radius2> 1.5 1.8 </radius2>   <!- radius2.X and radius2.Z -->
</tapered_cylinder>
```



attachment

Category: **Physics Model**

Introduction

Defines an attachment frame, to a rigid body or a node, within a rigid constraint.

Concepts

A **<rigid_constraint>** attaches (and limits the motion between) two rigid bodies together. **<attachment>** refers to the second rigid body, and **<ref_attachment>** to the first. For example, in the case of a hinge constraint between a door and a wall, one of them is the reference attachment (in this case, the wall), and the other is the attachment (the door).

The **<attachment>** also defines the local coordinate frame for that end of the connection, relative to the rigid body (or node), using **<translate>** and **<rotate>** elements. For example, you attach the hinge (rigid constraint) to the middle of the edge of the door (rigid body), relative to the door's local origin.

Attributes

The **<attachment>** element has the following attribute:

rigid_body	**xs:anyURI**	A URI reference to a **<rigid_body>** or **<node>**. This must refer to a **<rigid_body>** either in **<attachment>** or in **<ref_attachment>**; they cannot both be **<node>**s. Required.

Related Elements

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

Parent elements	**rigid_constraint**
Child elements	See the following subsection.
Other	**ref_attachment**

Child Elements

Child elements can appear in any order if present:

Name/example	Description	Default	Occurrences
<translate>	Changes the position of the attachment point. See main entry.	N/A	0 or more
<rotate>	Changes the position of the attachment point. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Example

```
<attachment rigid_body="./SomeRigidBody">
   <translate/>
   <rotate/>
   <extra/>
</attachment>
```

For a more complete example, see **<rigid_constraint>**.

box

Category: **Analytical Shape**

Introduction

Declares an axis-aligned, centered box primitive.

Concepts

Geometric primitives, or analytical shapes, are mostly useful for collision shapes for physics. See the “New Geometry Types” section earlier in this chapter.

Attributes

The **<box>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<half_extents>	Contains 3 floating-point values that represent the extents of the box. This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<box>
  <half_extents> 2.5  1.0  1.0 </half_extents>
</box>
```

capsule

Category: **Analytical Shape**

Introduction

Declares a capsule primitive that is centered on and aligned with the local y axis.

Concepts

Geometric primitives, or analytical shapes, are mostly useful for collision shapes for physics. See the "New Geometry Types" section earlier in this chapter.

Attributes

The **<capsule>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<height>	Contains a floating-point value that represents the length of the line segment connecting the centers of the capping hemispheres. This element has no attributes.	None	1
<radius>	Contains two floating-point values that represent the radii of the capsule (it may be elliptical). This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<capsule>
  <height> 2.0 </height>
  <radius> 1.0 1.0 </radius>
</capsule>
```

convex_mesh

Category: **Analytical Shape**

Introduction

Contains or refers to information sufficient to describe basic geometric meshes.

Concepts

The definition of **<convex_mesh>** is identical to **<mesh>** except that, instead of a complete description (**<source>**, **<vertices>**, **<polygons>**, and so on), it may simply point to another **<geometry>** to derive its shape. The latter case means that the convex hull of that **<geometry>** should be computed and is indicated by the optional **convex_hull_of attribute**.

This is very useful because it allows for reusing a **<mesh>** (that is used for rendering) for physics to minimize the document size and to maintain a link to the original **<mesh>**.

The minimal way to describe a **<convex_mesh>** is to specify its vertices (via a **<vertices>** element and its corresponding source) and let the importer compute the convex hull of that point cloud.

Attributes

The **<convex_mesh>** element has the following attribute:

convex_hull_of	**xs:anyURI**	A URI string of a **<geometry>**. If specified, compute the convex hull of the specified mesh; in this case, your application should ignore **<source>** and **<vertices>**. Optional.

Related Elements

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

Parent elements	**geometry**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description		Default	Occurrences
<source>	Provides the bulk of the mesh's vertex data. See main entry.		N/A	0 or more
<vertices>	Describes the mesh-vertex attributes and establishes their topological identity. See main entry.		N/A	0 or 1
primitive_elements	Primitive elements can be any combination of the following:			
	<lines>	Contains line primitives. See main entry.	N/A	0 or more
	<linestrips>	Contains line-strip primitives. See main entry.	N/A	0 or more
	<polygons>	Contains polygon primitives which may contain holes. See main entry.	N/A	0 or more

Name/example	Description		Default	Occurrences
	<polylist>	Contains polygon primitives that cannot contain holes. See main entry.	N/A	0 or more
	<triangles>	Contains triangle primitives. See main entry.	N/A	0 or more
	<trifans>	Contains triangle-fan primitives. See main entry.	N/A	0 or more
	<tristrips>	Contains triangle-strip primitives. See main entry.	N/A	0 or more
<extra>	See main entry.		N/A	0 or more

Details

If the attribute **convex_hull_of** is not used, specify child elements **<source>** and **<vertices>** to define a valid **<convex_mesh>**.

Example

```
<geometry id="myConvexMesh">
  <convex_mesh>
    <source>...</source>
    <vertices>...</vertices>
    <polygons>...</polygons>
  </convex_mesh>
</geometry>
```

or:

```
<geometry id="myArbitraryMesh">
  <mesh>
...
  </mesh>
</geometry>
<geometry id="myConvexMesh">
  <convex_mesh convex_hull_of="#myArbitraryMesh"/>
</geometry>
```

cylinder

Category: **Analytical Shape**

Introduction

Declares a cylinder primitive that is centered on, and aligned with, the local y axis.

Concepts

Geometric primitives, or analytical shapes are mostly useful for collision shapes for physics. See the “New Geometry Types” section earlier in this chapter.

Attribute

The **<cylinder>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<height>	Contains a floating-point value that represents the length of the cylinder along the y axis. This element has no attributes.	None	1
<radius>	Contains two floating-point values that represent the radii of the cylinder (it may be elliptical). This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<cylinder>
  <height> 2.0 </height>
  <radius> 1.0 1.0 </radius>
</cylinder>
```

force_field

Category: **Physics Scene**

Introduction

Provides a general container for force fields. At the moment, it contains only techniques and extra elements.

Concepts

Force fields affect physical objects, such as rigid bodies, and may be instantiated under a **physics_scene** or an instance of **physics_model**.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	Optional.

Related Elements

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

Parent elements	**library_force_fields**
Child elements	See the following subsection.
Other	**instance_force_field**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<technique> (core)	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Currently there is no COMMON technique/profile for **<force_field>**. The **<technique>** element can contain any well-formed XML data.

Example

```
<library_force_fields>
  <force_field>
    <technique profile="SomePhysicsProfile">
      <program url="#SomeWayToDescribeAForceField">
        <param> ... </param>
        <param> ... </param>
      </program>
    </technique>
  </force_field>
</library_force_fields>
```

instance_force_field

Category: **Physics Scene**

Introduction

Instantiates an object described by a **<force_field>** element.

Concepts

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<force_field>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**instance_physics_model**, **physics_scene**
Child elements	See the following subsection.
Other	**force_field**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	See main entry.	N/A	0 or more

Details

Example

instance_physics_material

Category: **Physics Material**

Introduction

Instantiates an object described by a **<physics_material>** element.

Concepts

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<physics_material>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**rigid_body** / **technique_common**, **instance_rigid_body** / **technique_common**, **shape**
Child elements	See the following subsection.
Other	**physics_material**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	See main entry.	N/A	0 or more

Details

Example

instance_physics_model

Category: **Physics Model**

Introduction

Instantiates a physics model within another physics model, or in a physics scene.

Concepts

This element is used for two purposes: to hierarchically embed a physics model inside another physics model during its definition, and to instantiate a complete physics model within a physics scene. It is possible to override parameters of the contained rigid bodies and constraints in both usages.

When instantiating a physics model inside a physics scene, at a minimum, the rigid bodies that are included in the physics model should each be linked with the associated visual transform node they will influence. Only the instantiated rigid bodies and constraints within the **<instance_physics_model>** will be considered for the simulation. Constraints that are connected to uninstantiated rigid bodies are discarded or ignored by the simulation.

Additionally, it is possible to specify a parent attribute for the instantiated physics model. This parent will dictate the initial position and orientation of the physics models (and correspondingly, of its rigid bodies). The parent (or grandparent, etc.) can also be targeted by some animation controller, to combine key-frame kinematics of nondynamic rigid bodies with physical simulation.

For details about instance elements in COLLADA, see “Instantiation and External Referencing” in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. This allows for targeting elements of the **<instance_physics_model>** instance for **animation**. Optional.
name	**xs:NCName**	Optional.
url	**xs:anyURI**	Which **<physics_model>** to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

parent	**xs:anyURI**	Points to the id of a node in the visual scene. This allows a physics model to be instantiated under a specific transform node, which will dictate the initial position and orientation, and could be animated to influence kinematic rigid bodies. Optional.

By default, the physics model is instantiated under the world, rather than a specific transform node. This parameter is only meaningful when the parent element of the current **<physics_model>** is a **<physics_scene>**.

Related Elements

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

Parent elements	**physics_scene**, **physics_model**
Child elements	See the following subsection.
Other	**physics_model**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<instance_force_field>	Instantiates a **<force_field>** element to influence this physics model. See main entry.	N/A	0 or more
<instance_rigid_body target="#SomeNode">	Instantiates a **<rigid_body>** element and allows for overriding some or all of its properties. The target attribute defines the **<node>** element that has its transforms overwritten by this rigid-body instance. See main entry.	N/A	0 or more
<instance_rigid_constraint>	Instantiates a **<rigid_constraint>** element to override some of its properties. This element does not have a target attribute because its **<rigid_constraint>** children define which **<node>** elements are targeted. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

instance_physics_scene

Category: **Physics Scene**

Introduction

Instantiates an object described by a **<physics_scene>** element.

Concepts

For details about instance elements in COLLADA, see “Instantiation and External Referencing” in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URL of the location of the **<physics_scene>** element to instantiate. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**scene**
Child elements	See the following subsection.
Other	**physics_scene**

Child Elements

Name/example	Description	Default	Occurrences
<extra>	See main entry.	N/A	0 or more

Details

Example

instance_rigid_body

Category: **Physics Model**

Introduction

Instantiates an object described by a **<rigid_body>** within an **<instance_physics_model>**.

Concepts

Rigid bodies ultimately set the transforms of a **<node>** in the **<scene>**, whether they are directly under a **<physics_model>** or under a **<rigid_constraint>**.

When instantiating a **<physics_model>**, at a minimum, the rigid bodies that are included in that **<physics_model>** must be linked with their associated **<node>** elements.

The **<instance_rigid_body>** element is used for three purposes:

- To specify the linkage to a **<node>** element
- To optionally override parameters of a **<rigid_body>** in a specific instance
- To specify the initial state (linear and angular velocity) of a **<rigid_body>** instance

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. This allows for targeting elements of the **<rigid_body>** instance for **animation**. Optional.
name	**xs:NCName**	Optional.
body	**xs:NCName**	Which **<rigid_body>** to instantiate. Required.
target	**xs:anyURI**	Which **<node>** is influenced by this **<rigid_body>** instance. Required. Can refer to a local instance or external reference.

For a local instance, this is a relative URI 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.

For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**instance_physics_model**
Child elements	See the following subsection.
Other	**rigid_body**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<technique_common>	Specifies the rigid-body information for the common profile that all COLLADA implementations must support. See "The Common Profile" section for usage information and the following subsection for child element details.	N/A	1
<technique> (core)	Each **<technique>** specifies rigid-body information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more
<extra>	User-defined, multirepresentable data that adds information to the **<instance_rigid_body>** (as opposed to switching base data, like the **<technique>** element does). See main entry.	N/A	0 or more

Child Elements for instance_rigid_body / technique_common

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<angular_velocity>	Contains three floating-point values that specify the initial angular velocity of the **rigid_body** instance around each axis, in the form of an x-y-z Euler rotation. The measurement is in degrees per second; see "About Physical Units."	0 0 0	0 or 1
<velocity>	Contains three floating-point values that specify the initial linear velocity of the **rigid_body** instance.	0 0 0	0 or 1
<dynamic sid="..."> **false</dynamic>**	Contains a Boolean that specifies whether the **rigid_body** is movable. The sid attribute is optional.	None	0 or 1
<mass sid="..."> **0.5</mass>**	Contains a floating-point value that specifies the total mass of the **rigid_body**. The **sid** attribute is optional.	Derived from density x total shape volume	0 or 1
<mass_frame> **<translate>** ... **</translate>** **<rotate>** ... **</rotate>** **</mass_frame>**	Defines the center and orientation of mass of the rigid-body relative to the local origin of the "root" shape. This makes the off-diagonal elements of the inertia tensor (products of inertia) all 0 and allows us to just store the diagonal elements (moments of inertia). The translate and rotate child elements can each appear 0 or more times, although at least one of the two must be present. See main entries.	"identity" (center of mass is at the local origin and the principal axes are the local axes).	0 or 1
<inertia sid="..."> 1 1 1 **</inertia>**	Contains three floating-point numbers, which are the diagonal elements of the inertia tensor (moments of inertia), represented in the local frame of the center of mass. See preceding. The sid attribute is optional.	Derived from mass, shape volume and center of mass.	0 or 1

Name/example	Description	Default	Occurrences
<physics_material> or **<instance_physics_material>**	Defines or references a **physics_material** for the **rigid_body**. See main entries.	N/A	0 or 1
<shape>	See main entry.	N/A	0 or more

Example

```
<physics_scene id="ColladaPhysicsScene">
  <instance_physics_model sid="firstCatapultAndRockInstance"
                        url="#catapultAndRockModel" parent="#catapult1">
<!—Override attributes of a rigid_body within this physics_model      -->
<!—and specify the initial velocity of the rigid_body                 -->
    <instance_rigid_body body="./rock/rock" target="#rockNode">
      <technique_common>
        <velocity>0 -1 0</velocity>  <!—optional overrides -->
        <mass>10</mass>                          <!—heavier            -->
      </technique_common>
    </instance_rigid_body>
<!—This instance only assigns the rigid_body to its node. It does no overriding
-->
    <instance_rigid_body body="./catapult/base" target="#baseNode"/>
  </instance_physics_model>
</physics_scene>
```

instance_rigid_constraint

Category: **Physics Model**

Introduction

Instantiates an object described by a **<rigid_constraint>** within an **<instance_physics_model>**.

Concepts

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. This allows for targeting elements of the **<rigid_constraint>** instance for animation. Optional.
name	**xs:NCName**	Optional.
contraint	**xs:NCName**	Which **<rigid_constraint>** to instantiate. Required.

Related Elements

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

Parent elements	**instance_physics_model**
Child elements	See the following subsection.
Other	**rigid_constraint**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<extra>	User-defined, multirepresentable data that adds information to the **<instance_rigid_constraint>**. See main entry.	N/A	0 or more

Details

The **<instance_rigid_constraint>** instantiates a **<rigid_constraint>**, which refers to two **<rigid_body>**s (reference and target). Note that the **<instance_rigid_constraint>** only refers to the **<rigid_body>**s and does not actually instantiate them. If a **<rigid_body>** is not instantiated inside an **<instance_physics_model>**, this **<rigid_body>** and any **<rigid_constraint>** that refer to it should be thrown away. This allows just parts, not all, of a physics model to be instantiated if desired, for example, half of an army, or half of a ragdoll.

Example

library_force_fields

Category: **Physics Scene**

Introduction

Declares a module of **<force_field>** elements.

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_force_fields>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_force_fields>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<force_field>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_force_fields>** element:

```
<library_force_fields>
    <force_field>
      <technique profile="AGEIA"/>
  </force_field>
</library_force_fields>
```

library_physics_materials

Category: **Physics Material**

Introduction

Declares a module of **<physics_material>** elements.

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_physics_materials>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_physics_materials>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<physics_material>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_physics_materials>** element:

```
<library_physics_materials>
  <physics_material id="phymat1">
    ...
  </physics_material>

  <physics_material id="phymat2">
    ...
  </physics_material>
</library_physics_materials>
```

library_physics_models

Category: **Physics Model**

Introduction

Declares a module of **<physics_model>** elements.

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_physics_models>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<physics_model>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_physics_models>** element:

```
<library_physics_models>
  <physics_model id="phymod1">
    ...
  </physics_model>

  <physics_model id="phymod2">
    ...
  </physics_model>
</library_physics_models>
```

library_physics_scenes

Category: **Physics Scene**

Introduction

Declares a module of **<physics_scene>** elements.

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_physics_scenes>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<physics_scene>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_physics_scenes>** element:

```
<library_physics_scenes>
  <physics_scene id="physce1">
    ...
  </physics_scene>

  <physics_scene id="physce2">
    ...
  </physics_scene>

</library_physics_scenes>
```

physics_material

Category: **Physics Material**

Introduction

Defines the physical properties of an object. It contains a technique/profile with parameters. The COMMON profile defines the built-in names, such as **static_friction**.

Concepts

Physics materials are stored under a **<library_physics_materials>** element and may be instantiated.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	Optional.

Related Elements

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

Parent elements	**library_physics_materials**, **shape**, **instance_rigid_body** / **technique_common**, **rigid_body** / **technique_common**
Child elements	See the following subsection.
Other	**instance_physics_material**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<technique_common>	Specifies physics-material information for the common profile that all COLLADA implementations must support. See the following subsection.	N/A	1
<technique> (core)	Each **<technique>** specifies physics-material information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Child Elements for physics_material / technique_common

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<dynamic_friction sid="..."> 0.23 </dynamic_friction>	Contains a floating-point number that specifies the dynamic friction coefficient. The sid attribute is optional.	0	0 or 1

Name/example	Description	Default	Occurrences
<restitution sid="..."> 0.2 </restitution>	Contains a floating-point number that is the proportion of the kinetic energy preserved in the impact (typically ranges from 0.0 to 1.0). Also known as “bounciness” or “elasticity.” The sid attribute is optional.	0	0 or 1
<static_friction sid="..."> 0.23 </static_friction>	Contains a floating-point number that specifies the static friction coefficient. The sid attribute is optional.	0	0 or 1

Example

```
<physics_material id="WoodPhysMtl">
  <technique_common>
    <dynamic_friction> 0.12 </dynamic_friction>
    <restitution> 0.05 </restitution>
    <static_friction> 0.23 </static_friction>
  </technique_common>
</physics_material>
```

physics_model

Category: **Physics Model**

Introduction

Allows for building complex combinations of rigid bodies and constraints that may be instantiated multiple times.

Concepts

This element is used to define and group physical objects that are instantiated under **<physics_scene>**. Physics models might be as simple as a single rigid-body, or as complex as a biomechanically described human character with bones and other body parts (i.e. rigid bodies), and muscles linking them (i.e. rigid constraints). It is also possible for a physics model to contain other previously-defined physics models. For example, a house physics model could contain a number of instantiated physics models, such as walls made from bricks.

This element defines the structure of such a model and the **<instance_physics_model>** element instantiates a **<physics_model>** and it can override many of its parameters.

Each child element defined inside a physics model has an sid attribute instead of an id. The sid is used to access and override components of a physics-model at the point of instantiation.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_physics_models**
Child elements	See the following subsection.
Other	**instance_physics_model**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<rigid_body>	Defines a **<rigid_body>** element and sets its nondefault properties. See main entry.	N/A	0 or more
<rigid_constraint>	Defines a **<rigid_constraint>** element and allows for overriding some or all of its properties. See main entry.	N/A	0 or more
<instance_physics_model>	Instantiates a physics model from the given url, and assigns an sid to it, to distinguish it from other child elements. See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
<extra>	See main entry.	N/A	0 or more

Example

```
<library_physics_models>
    <!-- Defines a catapult physics model that can be reused and/or -->
    <!-- modified in other physics models or in a physics_scene. -->
      <physics_model id="catapultModel">
        <!-- This is the base of the catapult, defined inline. -->
        <rigid_body sid="base">
          <technique_common>
            <dynamic>false</dynamic>
            <instance_physics_material url="#catapultBasePhysicsMaterial"/>
            <shape>
              <instance_geometry url="#catapultBaseConvexMesh"/>

              <!-- Local position of base relative to the catapult model. -->
              <translate> 0 -1 0 </translate>
            </shape>

          </technique_common>
        </rigid_body>

        <!-- The top (or arm) of the catapult is defined similarly. -->
        <rigid_body sid="top">
          <technique_common>
            <dynamic>true</dynamic>
            <shape>
              <instance_geometry url="#catapultTopConvexMesh"/>
              <translate> 0 3 0 </translate>
            </shape>
          </technique_common>
        </rigid_body>

        <!-- Define the angular spring that drives the catapult movement.
             Optionally, a url could have been provided to copy a rigid
             constraint from some other physics model. -->
        <rigid_constraint sid="spring_constraint">
        <ref_attachment rigid_body="./base">
          <translate sid="translate">-2. 1. 0</translate>
        </ref_attachment>
        <attachment rigid_body="./top">
          <translate sid="translate">1.23205 -1.86603 0</translate>
          <rotate sid="rotateZ">0 0 1 -30.</rotate>
        </attachment>
        <technique_common>
          <limits>
            <swing_cone_and_twist>
              <min> -180.0 0.0 0.0 </min>
              <max> 180.0 0.0 0.0 </max>
            </swing_cone_and_twist>
          </limits>
          <spring>
            <angular>
              <stiffness>500</stiffness>
              <damping>0.3</damping>
              <target_value>90</target_value>
            </angular>
```

```
        </spring>
      </technique_common>
    </rigid_constraint>
  </physics_model>

  <!-- This physics model combines the two previously defined models. -->
  <physics_model id="catapultAndRockModel">
    <!-- This rock is taken from a library of predefined physics models. -->
    <instance_physics_model sid="rock"
        url="http://feelingsoftware.com/models/rocks.dae#rockModels/bigRock">
      <!-- Placement of rock on catapult in catapultAndRockModel space -->
      <translate> 0 4 0 </translate>
    </instance_physics_model>
    <instance_physics_model sid="catapult" url="#catapultModel"/>
  </physics_model>
</library_physics_models>
```

physics_scene

Category: **Physics Scene**

Introduction

Specifies an environment in which physical objects are instantiated and simulated.

Concepts

COLLADA allows for multiple simulations to run independently for the following main reasons:

- Multiple simulations may need different global settings and they might even run on different physics engines or on different hardware.
- By providing such a high-level grouping mechanism, we can minimize interactions to improve performance. For example, rigid bodies in one physics scene are known not collide with rigid bodies of other physics scenes, so no collision tests need to be done between them.
- It allows for supporting multiple levels of detail (LOD)

The **<physics_scene>** element may contain techniques, extra elements, and a list of **<instance_physics_model>** elements.

The "active" **<physics_scene>**s (ones that are simulated) are indicated by instantiating them under the main **<scene>**.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_physics_scenes**
Child elements	See the following subsection.
Other	**instance_physics_scene**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<instance_force_field>	Instantiates a **<force_field>** element to influence this physics scene. See main entry.	N/A	0 or more
<instance_physics_model>	Instantiates a **<physics_model>** element and allows for overriding some or all of its children. See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
<technique_common>	Specifies physics-scene information for the common profile that all COLLADA implementations must support. See "The Common Profile" section for usage information and the following subsection for child element details.	N/A	1
<technique> (core)	Each **<technique>** specifies physics-scene information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Child Elements for physics_scene / technique_common

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<gravity>	A vector representation of the scene's gravity force field. It is given as a denormalized direction vector of three floating-point values that indicate both the magnitude and direction of acceleration caused by the field.	N/A	0 or 1
<time_step>	The integration time step, measured in seconds, of the physics scene. This value is engine-specific. If omitted, the physics engine's default is used. A floating-point number.	N/A	0 or 1

Example

```
<library_physics_scenes>
<!-- regular physics scene. -->
  <physics_scene id="ColladaPhysicsScene">
     <instance_physics_model sid="firstCatapultAndRockInstance">
                    url="#catapultAndRockModel" parent"#catapult1">
<!-- Instance of physics model, with overrides.
     The current transform matrix will dictate the initial position and
     orientation of the physics model in world space. -->
      <instance_rigid_body body="./rock/rock" target="#rockNode">
        <technique_common>
          <velocity>0 -1 0</velocity> <!-- optional overrides -->
          <mass>10</mass>                          <!-- heavier            -->
        </technique_common>
      </instance_rigid_body>
      <instance_rigid_body body="./catapult/top" target="#catapultTopNode"/>
      <instance_rigid_body body="./catapult/base" target="#baseNode"/>
    </instance_physics_model>
    <technique_common>
      <gravity>0 -9.8 0</gravity>
      <time_step>3.e-002</time_step>
    </technique_common>
  </physics_scene>
</library_physics_scenes>
<!-- A scene where an "army" of two physically simulated catapults is
instantiated -->
<library_visual_scenes>
  <visual_scene id="battlefield">
    <node id="catapult1">
      <translate sid="translate">0 -0.9 0</translate>
```

```
      <node id="rockNode">
        <instance_geometry url="#someRockVisualGeometry"/>
      </node>
      <node id="catapultTopNode">
        <instance_geometry url="#someVisualCatapultTopGeometry"/>
      </node>
      <node id="catapultBaseNode">
        <instance_geometry url="#someVisualCatapultBaseGeometry"/>
      </node>
    </node>
<!-- Can replicate a physics model by instantiating one of its parent nodes -->
    <node id="catapult2">
      <translate/>             <!-- Position the second catapult somewhere else
-->
      <rotate/>
      <instance_node url="#catapultNode1"/> <!-- replicate physics model &
visuals -->
    </node>
  </visual_scene>
</library_visual_scenes>
<scene>
<!-- Indicates that the physics scene is applicable to this visual scene    -->
  <instance_physics_scene url="#ColladaPhysicsScene"/>
  <instance_visual_scene url="#battlefield"/>
</scene>
```

plane

Category: **Analytical Shape**

Introduction

Defines an infinite plane primitive.

Concepts

Geometric primitives, or analytical shapes, are mostly useful for collision shapes for physics. See the "New Geometry Types" section earlier in this chapter.

Attributes

The **<plane>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<equation>	Contains four floating-point values that represent the coefficients for the plane's equation: **Ax + By + Cz + D = 0** This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<plane>
<!--  Plane equation: Ax + By + Cz + D = 0 -->
<!-- A, B, C, D coefficients (normal & D) -->
  <equation> 0.0  1.0  0.0  0.0 </equation>    <!-- The X-Z plane (ground) -->
</plane>
```

ref_attachment

Category: **Physics model**

Introduction

Defines an attachment frame of reference, to a rigid body or a node, within a rigid constraint.

Concepts

A **<rigid_constraint>** attaches (and limits the motion between) two rigid bodies together. **<ref_attachment>** refers to the first rigid body, and **<attachment>** to the second. For example, in the case of a hinge constraint between a door and a wall, one of them is the reference attachment (in this case, the wall), and the other is the attachment (the door).

The **<ref_attachment>** also defines the local coordinate frame for that end of the connection, relative to the rigid body (or node), using **<translate>** and **<rotate>** elements. For example, you attach the hinge (rigid constraint) to the middle of the edge of the wall (rigid body), relative to the wall's local origin.

Attributes

The **<ref_attachment>** element has the following attribute:

rigid_body	**xs:anyURI**	A URI reference to a **<rigid_body>** or **<node>**. This must refer to a **<rigid_body>** either in **<attachment>** or in **<ref_attachment>**; they cannot both be **<node>**s. Required.

Related Elements

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

Parent elements	**rigid_constraint**
Child elements	See the following subsection.
Other	**attachment**

Child Elements

Child elements can appear in any order if present:

Name/example	Description	Default	Occurrences
<translate>	Changes the position of the attachment point. See main entry.	N/A	0 or more
<rotate>	Changes the position of the attachment point. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Example

```
<ref_attachment rigid_body="./SomeRigidBody">
   <translate/>
   <rotate/>
   <extra/>
</ref_attachment>
```

For a more complete example, see **<rigid_constraint>**.

rigid_body

Category: **Physics Model**

Introduction

Describes simulated bodies that do not deform. These bodies may or may not be connected by constraints (hinge, ball-joint, and so on).

Concepts

Rigid bodies, constraints, and so on are encapsulated in **<physics_model>** elements to allow the instantiation of complex models.

Rigid bodies consist of parameters and a collection of shapes for collision detection. Each shape may be rotated and/or translated to allow for building complex collision shapes ("bounding shape"). These shapes are described by one or more **<shape>** elements.

Attributes

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

sid	**xs:NCName**	A text string containing the scoped identifier of the **<rigid_body>** element. This value must be unique among its sibling elements. Associates each rigid body with a visual **<node>** when a **<physics_model>** is instantiated. Required.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**physics_model**
Child elements	See the following subsection.
Other	**instance_rigid_body**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<technique_common>	Specifies rigid-body information for the common profile that every COLLADA implmentation must support. See "The Common Profile" section for usage information and the following subsection for child element details.	N/A	1
<technique> (core)	Each **<technique>** specifies rigid-body information for a specific profile as designated by the **<technique>**'s profile attribute. See main entry.	N/A	0 or more
<extra>	User-defined, multirepresentable data that adds information to the **<rigid_body>** (as opposed to switching base-data, like the **<technique>** element does). See main entry.	N/A	0 or more

Child Elements for rigid_body / technique_common

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
`<dynamic sid="...">false</dynamic>`	Contains a Boolean that specifies whether the `rigid_body` is movable. The sid attribute is optional.	N/A	0 or 1
`<mass sid="...">0.5</mass>`	Contains a floating-point value that specifies the total mass of the `rigid_body`. The sid attribute is optional.	Derived from density x total shape volume.	0 or 1
`<mass_frame>` `<translate>` ... `</translate>` `<rotate>` ... `</rotate>` `</mass_frame>`	Defines the center and orientation of mass of the `rigid_body` relative to the local origin of the "root" shape. This makes the off-diagonal elements of the inertia tensor (products of inertia) all 0 and allows us to just store the diagonal elements (moments of inertia). The `<translate>` and `<rotate>` child elements can each appear 0 or more times, although at least one of the two must be present. See their main entries.	"identity" (center of mass is at the local origin and the principal axes are the local axes).	0 or 1
`<inertia sid="...">1 1 1</inertia>`	Contains three floating-point numbers, which are the diagonal elements of the inertia tensor (moments of inertia), which is represented in the local frame of the center of mass. See preceding. The sid attribute is optional.	Derived from mass, shape volume and center of mass.	0 or 1
`<physics_material>` or `<instance_physics_material>`	Defines or references a `physics_material` for the `rigid_body`. See main entries.	N/A	0 or 1
`<shape>`	See main entry.	N/A	1 or more

Density, Mass, and Inertia (Tensor) Definition Rules

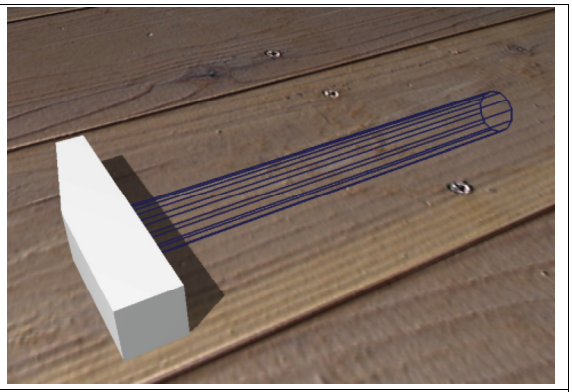
- Both the rigid-body and its shapes may specify either mass or density. If neither is defined, density will default to 1.0 and mass will be computed using the total volume of the shapes.
- If mass is defined, density will be ignored.
- Volume and total mass are computed as the sum of the volumes and masses of the shapes, even if they intersect. No Boolean operations (CSG), like "union" or "difference" are expected of the tools.
- The rigid-body mass, inertia etc. is the "trump all" definition. If the sum of shape masses don't add up to that value, they will be "normalized" to add up to the mass of the rigid-body. For example a body with a total mass of 6 and 2 shapes: mass=1 and mass = 2 will be interpreted as: total mass (6) = 2+4.

Example

Here is a compound rigid-body. Note the difference between the shapes meant for physics (cylinder primitive and simple convex hull) and the ones for rendering (textured, tapered handle and beveled head):

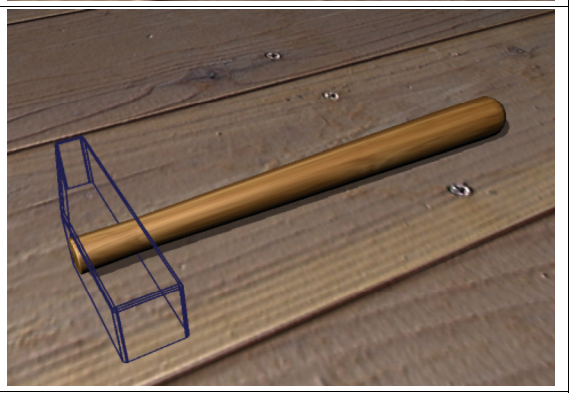
```
<library_geometries>
  <geometry id="hammerHeadForPhysics">

      <mesh>
...
      </mesh>
  <geometry>
```



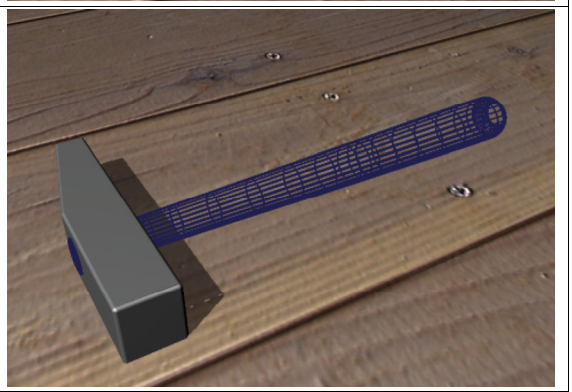
```
  <geometry id="hammerHandleToRender">

    <mesh>
...
    </mesh>
  </geometry>
```



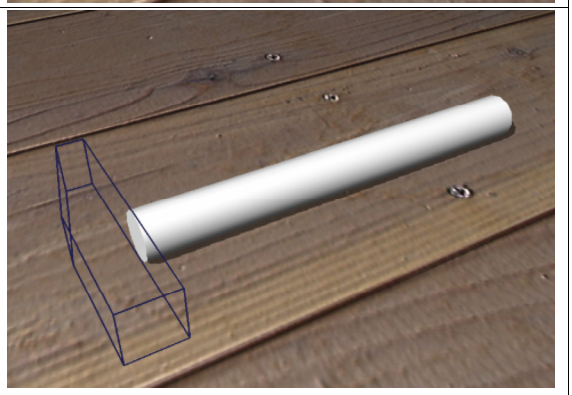
```
 <geometry id="hammerHeadToRender">

    <mesh>
...
    </mesh>
  </geometry>
<library_geometries>
```



```
<library_physics_models>
  <physics_model id="HammerPhysicsModel">
    <rigid_body sid="HammerHandleRigidBody">
      <technique_common>

        <mass> 0.25 </mass>
        <mass_frame> ... </mass_frame>
        <inertia> ... </inertia>
        <shape>
          <instance_physics_material
                url="#WoodPhysMtl"/>
<!- This geometry is small and not used
elsewhere, so it is inlined -->
          <cylinder>
            <height> 8.0 </height>
```



```
            <radius> 0.5 0.5 </radius>
          </cylinder>

        </shape>
        <shape>
          <mass> 1.0 </mass>
          <!- This geometry is referenced
rather than inlined -->
          <instance_physics_material
                 url="#SteelPhysMtl"/>
          <instance_geometry

url="#hammerHeadForPhysics"/>
          <translate> 0.0  4.0  0.0
</translate>
        </shape>
      </technique_common>
    </rigid_body>
  </physics model>
</library_rigid_bodies>
```

rigid_constraint

Category: **Physics Model**

Introduction

Connects components, such as **<rigid_body>**, into complex physics models with moveable parts.

Concepts

Building interesting physical models generally means attaching some of the rigid bodies together, using springs, ball joints, or other types of rigid constraints.

COLLADA supports constraints that link two rigid bodies or a rigid body and a coordinate frame in the scene hierarchy (for example, world space). Instead of defining a large combination of constraint primitive elements, COLLADA offers one very flexible element, the general six-degrees-of-freedom (DOF) constraint. Simpler constraints (for example, linear or angular spring, ball joint, hinge) may be expressed in terms of this general constraint.

A constraint is specified by:

- Two attachment frames, defined using a translation and orientation relative to a rigid body's local space or to a coordinate frame in the scene hierarchy. To remain consistent with the rest of COLLADA, this is expressed using standard **<translate>** and **<rotate>** elements.
- Its degrees-of-freedom (DOF). A DOF specifies the variability along a given axis of translation or axis of rotation, expressed in the space of the attachment frame. For example, a door hinge typically has one degree of freedom, along a given axis of rotation. In contrast, a slider joint has one degree of freedom along a single axis of translation.

Degrees-of-freedom and limits are specified by the very flexible **<limits>** element.

Attributes

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

sid	**xs:NCName**	A text string containing the scoped identifier of the **<rigid_constraint>** element. This value must be unique within the scope of the parent element. Required.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**physics_model**
Child elements	See the following subsection.
Other	**instance_rigid_constraint**

Child Elements

Child elements must appear in the following order if present:

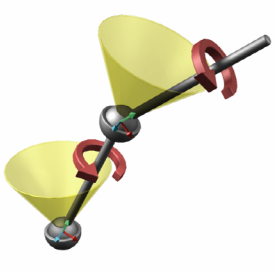
Name/example	Description	Default	Occurrences
<ref_attachment>	Defines the attachment frame of reference (to a **rigid_body** or a node) within a rigid constraint. See main entry.	N/A	1

Name/example	Description	Default	Occurrences
<attachment>	Defines an attachment frame (to a rigid body or a node) within a rigid constraint. See main entry.	N/A	1
<technique_common>	Specifies rigid-constraint information for the common profile that all COLLADA implementations must support. See “The Common Profile” section for usage information and the following subsection for child element details.	N/A	1
<technique> (core)	Each **<technique>** specifies rigid-constraint information for a specific profile as designated by the **<technique>**’s profile attribute. See main entry.	N/A	0 or more
<extra>	User-defined, multirepresentable data. See main entry.	N/A	0 or more

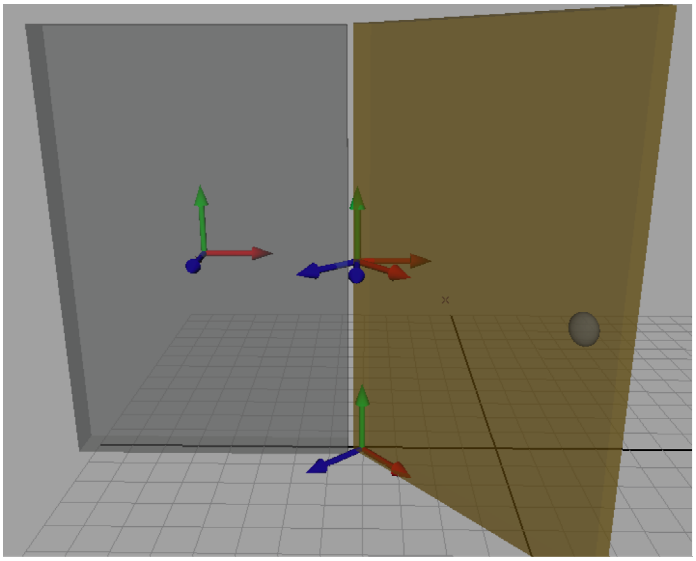
Child Elements for rigid_constraint / technique_common

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<enabled sid=”...”> true</enabled>	Contains a Boolean. If false, the **<constraint>** doesn’t exert any force or influence on the rigid bodies. The sid attribute is optional.	True	0 or 1
<interpenetrate sid=”...”>true </interpenetrate>	Contains a Boolean. If true, the attached rigid bodies may interpenetrate. The sid attribute is optional.	False	0 or 1
Two constraints with “swing-cone and twist”-type angular limits: **<limits> <swing_cone_and_twist> <min sid="..."> -15.0 -15.0 -INF </min> <max sid="..."> 15.0 15.0 INF </max> </swing_cone_and_twist> <linear> <min sid="..."> 0 0 0 </min> <max sid="..."> 0 0 0 </max> </linear> </limits>**	The **<limits>** element provides a flexible way to specify the constraint limits (degrees of freedom and ranges). This element has no attributes. This element may contain the optional child elements **<swing_cone_and_twist>**, **<linear>**, or both, which must appear in the order shown if both are used. If these limit descriptions are not sufficient, use a custom **<technique>**. The **<linear>** element describes linear (translational) limits along each axis. The **<swing_cone_and_twist>** element describes the angular limits along each rotation axis in degrees. The **<min>** and **<max>** elements are optional but must appear in the order shown if used. Their sid attributes are optional. They each contain three floating-point values representing x, y, and z limits. The values INF and –INF,	**linear**: **min: 0.0 0.0 0 .0** **max: 0.0 0.0 0.0** **swing_cone_and_ twist:** **min: 0.0 0.0 0 .0** **max: 0.0 0.0 0.0** This corresponds to a completely fixed rigid constraint, that is, the two rigid bodies do not move relative to each other. (No rotation or translation allowed.)	0 or 1

Name/example	Description	Default	Occurrences
	corresponding to +/- infinity, can also be used to indicate that there is no limit along that axis. Limits are expressed in the space of **ref_attachment**. In **<swing_code_and_twist>**, the x and y limits describe a "swing cone" and the z limits describe the "twist angle" range (see diagram on the left).		
Example 1: **<spring>** **<linear>** **<stiffness** **sid="...">5.4544** **</stiffness>** **<damping** **sid="...">0.4132** **</damping>** **<target_value** **sid="...">3** **</target_value>** **</linear>** **</spring>** Example 2: **<spring>** **<angular>** **<stiffness>5.4544** **</stiffness>** **<damping>0.4132** **</damping>** **<target_value>90** **</target_value>** **</angular>** **</spring>**	Spring is based on either distance (**<linear>**) or angle (**<angular>**), or both; if both are specified, **<angular>** must appear first. Each can have three optional child elements, which must appear in the order shown if used. They each contain a single floating-point value. Their sid attributes are optional. The **<stiffness>** (also called spring coefficient) has units of force/distance for **<linear>** or force/angle in degrees for **<angular>**. Spring is expressed in the space of **ref_attachment**.	**stiffness**: 1.0 **damping**: 0.0 **target_value**: 0.0 This corresponds to an "infinitely rigid" constraint, that is, no spring.	0 or 1

Examples



This example demonstrates a door with a hinge. The wall rigid body (in gray, on the right) has its local space frame in its center. The door has its local space on the floor and rotated 45 degrees on the y axis. The hinge constraint is limited to rotate +/- 90 degrees on its y axis. Each attachment frame has its translate/rotate transforms defined in terms of the rigid body's local space.

```
<library_physics_models>
  <physics_model>
    <rigid_body sid="doorRigidBody">
      <technique_common>...</technique_common>
    </rigid_body>
    <rigid_body sid="wallRigidBody">
      <technique_common>...</technique_common>
    </rigid_body>

    <rigid_constraint sid="rigidHingeConstraint">
      <ref_attachment rigid_body="#wallRigidBody">
          <translate sid="translate">5 0 0</translate>
      </ref_attachment>
      <attachment rigid_body="#doorRigidBody">
          <translate sid="translate">0 8 0</translate>
          <rotate sid="rotateX">0 1 0 -45.0</rotate>
      </attachment>
<!—Adding sid attributes here allows us to target the limits from animations -->
      <technique_common>
        <limits>
          <swing_cone_and_twist>
            <min sid="swing_min">0  90 0</min>
            <max sid="swing_max">0 -90 0</max>
          </swing_cone_and_twist>
        </limits>
      </technique_common>
    </rigid_constraint>
  </physics_model>
</library_physics_models>
```

shape

Category: **Physics Model**

Introduction

Describes components of a **<rigid_body>**.

Concepts

Rigid-bodies may contain a single shape or a collection of shapes for collision detection. Each shape may be rotated and/or translated to allow for building complex collision shapes (“bounding shape”).

These shapes are described by **<shape>** elements, each of which may contain:

- a **<physics_material>** definition or instance
- physical properties (mass, inertia, etc.)
- transforms (**<rotate>**, **<translate>**)
- an instance or an inlined definition of a **<geometry>**

Shapes may be “hollow” (for example, a chocolate bunny), meaning that the mass is not distributed through the whole volume, but close to the surface. The mass, inertia, density, and center of mass attributes should be set accordingly.

Attributes

The **<shape>** element has no attributes.

Related Elements

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

Parent elements	**rigid_body** / **technique_common**, **instance_rigid_body** / **technique_common**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<hollow sid=”...”> true</hollow>	Contains a Boolean. If true, the mass is distributed along the surface of the shape. The sid is optional.	None	0 or 1
<mass sid=”...”> 0.5 </mass>	Contains a floating-point number specifying the mass of the shape. The sid is optional.	Derived from density x shape volume	0 or 1
<density sid=”...”> 0.5 </density>	Contains a floating-point number specifying the density of the shape. The sid is optional.	Derived from mass/shape volume	0 or 1

Name/example	Description	Default	Occurrences
inline definition or instance: **<physics_material>** or **<instance_physics_material>**	The **<physics_material>** used for this shape.	From the geometry that is instantiated or defined by the **<shape>**.	0 or 1
geometry of the shape	This can be either of the following: • An inline definition using one of the following elements: **<plane>**, **<box>**, **<sphere>**, **<cylinder>**, **<tapered_cylinder>**, **<capsule>**, or **<tapered_capsule>** • A geometry instance using the **<instance_geometry>** element, which references other geometry types (**<mesh>**, **<convex_mesh>**, **<spline>**, and so on) .	N/A	1
<rotate>, **<translate>**	Transformation for the shape. Any combination of these elements in any order. See **<node>** for additional information.	No transforms	0 or more
<extra>	See main entry.	N/A	0 or more

Details

See also the **<rigid_body>** element.

Example

```
<library_rigid_bodies>
  <rigid_body sid="HammerHandleRigidBody">
    <technique_common>
      <shape>
        <mass> 0.25 </mass>
        <instance_physics_material url="#WoodPhysMtl"/>
        <instance_geometry url="#hammerHandleForPhysics"/>

      </shape>
    </technique_common>
  </rigid_body>
<library_rigid_bodies>
```

sphere

Category: **Analytical Shape**

Introduction

Describes a centered sphere primitive.

Concepts

Geometric primitives, or analytical shapes are mostly useful for collision shapes for physics. See the “New Geometry Types” section earlier in this chapter.

Attributes

The **<sphere>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<radius>	Contains a floating-point value that represents the radius of the sphere. This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<sphere>
  <radius> 1.0 </radius>
</sphere>
```

tapered_capsule

Category: **Analytical Shape**

Introduction

Describes a tapered capsule primitive that is centered on, and aligned with, the local y axis.

Concepts

Geometric primitives, or analytical shapes, are mostly useful for collision shapes for physics. See the “New Geometry Types” section earlier in this chapter.

Attributes

The **<tapered_capsule>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<height>	Contains a floating-point value that represents the length of the line segment connecting the centers of the capping hemispheres. This element has no attributes.	None	1
<radius1>	Contains two floating-point values that represent the radii of the tapered capsule at the positive (height/2) Y value. Both ends of the tapered capsule may be elliptical. This element has no attributes.	None	1
<radius2>	Contains two floating-point values that represent the radii of the tapered capsule at the negative (height/2) Y value. Both ends of the tapered capsule may be elliptical. This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<tapered_capsule>
  <height> 2.0 </height>
  <radius1> 1.0 1.0 </radius1>
  <radius2> 1.0 0.5 </radius2>
</tapered_capsule>
```

tapered_cylinder

Category: **Analytical Shape**

Introduction

Describes a tapered cylinder primitive that is centered on and aligned with the local y axis.

Concepts

Geometric primitives, or analytical shapes, are mostly useful for collision shapes for physics. See the "New Geometry Types" section earlier in this chapter.

Attributes

The **<tapered_cylinder>** element has no attributes.

Related Elements

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

Parent elements	**shape**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<height>	Contains a floating-point value that represents the length of the cylinder along the y axis. This element has no attributes.	None	1
<radius1>	Contains two floating-point values that represent the radii of the tapered cylinder at the positive (height/2) Y value. Both ends of the tapered cylinder may be elliptical. This element has no attributes.	None	1
<radius2>	Contains two floating-point values that represent the radii of the tapered cylinder at the negative (height/2) Y value. Both ends of the tapered cylinder may be elliptical. This element has no attributes.	None	1
<extra>	See main entry.	N/A	0 or more

Example

```
<tapered_cylinder>
  <height> 2.0 </height>
  <radius1> 1.0 2.0 </radius1>
  <radius2> 1.5 1.8 </radius2>
</tapered_cylinder>
```

Chapter 7:
Getting Started with COLLADA FX

Introduction

COLLADA FX enables authors to describe how to apply color to a visual scene. It is a flexible abstraction for describing material properties across many platforms and application programming interfaces (APIs).

The FX elements of the COLLADA schema allow the description of:

- Single and multipass effects, which are abstract material definitions (for example, plastic)
- Effect parameterizations (using **<newparam>**)
- Effect metadata
- Binding to the scene graph
- Multiple techniques
- Inline and external source code or binary

Multiple application programming interfaces (APIs) are supported through **<profile_*>** elements, allowing each effect to be described for multiple platforms. Each platform can be fully described, with platform-specific data types, render states, and capabilities.

Within each platform, each effect can be described using many techniques. A technique is a user-labeled description of a style of rendering (for example, “daytime,” “nighttime,” “magic,” “superhero_mode”), a different level of detail, or a method of calculation (for example, “approximate,” “accurate,” “high_LOD,” “low_LOD”).

At a higher level, a material system allows predefined effects to be specialized into a specific instance by providing values to parameters other than the default values in the effect definition. This allows a single effect to be used as a basis for many different materials.

Finally, materials are put to use when they are bound to one or more points in the scene graph. The **<bind_material>** element in the scene graph’s geometry may instantiate one or more materials and connect them to segments of the geometry. Within the material instance, further specialization of effects may occur by binding to resources in the scene graph, such as lights and cameras, or pairing texture coordinates.

Using Profiles for Platform-Specific Effects

The **<profile_*>** elements allow each effect to be described for multiple platforms.

About Profiles

The **<profile_*>** elements encapsulate all platform-specific values and declarations for effects in a particular profile. They define the clear interface between concrete, platform-specific data types and the abstract COLLADA data types used in the rest of the document.

COLLADA FX supports the following profiles:

- **profile_COMMON**: Encapsulate all the values and declarations for a platform-independent fixed-function shader. All platforms are required to support **<profile_COMMON>**. Effects in this profile are designed to be used as the reliable fallback when no other profile is recognized by the current effects runtime.

- **profile_CG**: Encapsulates effects for use with the Cg high-level language.
- **profile_GLES**: Encapsulates effects for use with OpenGL ES.
- **profile_GLSL**: Encapsulates effects for use with OpenGL Shading Language.

Notes:

- Unlike techniques, you do NOT have to specify a **profile_COMMON** if you're specifying other profiles.
- **Profile_COMMON** is for different effects; **technique_common** is for extensibility.

FX Element Attributes and Structures In Profiles

The specific attributes and child elements of each FX element may vary depending on the profile scope in which it is used. The profile scopes can be grouped as follows:

- External to FX
- Effect (elements are valid in effect scope outside specific profiles)
- Common profile
- Cg profile
- Open GL ES (GLES) profile
- GLSL profile

The following table shows which elements are valid in which scope:

Element	External	Effect	Common	Cg	GLES	GLSL
<alpha>	–	–	–	–	YES	–
<ambient> (FX)	–	–	YES	–	–	–
<annotate>	YES	YES	YES	YES	YES	YES
<argument>	–	–	–	–	YES	–
<array>	–	–	–	YES	–	–
<bind> (material)	YES	–	–	–	–	–
<bind> (shader)	–	–	–	YES	–	YES
<bind_material>	YES	–	–	–	–	–
<bind_vertex_input>	YES	–	–	–	–	–
<blinn>	–	–	YES	–	–	–
<code>	–	–	–	YES	–	YES
<color> (FX)	–	–	YES	–	–	–
<color_clear>	–	–	–	YES	YES	YES
<color_target>	–	–	–	YES	YES	YES
<compiler_options>	–	–	–	YES	–	YES
<compiler_target>	–	–	–	YES	–	YES
<connect_param>	–	–	–	YES	–	–
<constant> (FX)	–	–	YES	–	–	–
<constant> (combiner)	–	–	–	–	YES	–
<depth_clear>	–	–	–	YES	YES	YES
<depth_target>	–	–	–	YES	YES	YES
<diffuse>	–	–	YES	–	–	–
<draw>	–	–	–	YES	YES	YES
<effect>	–		–	–	–	–
<emission>	–	–	YES	–	–	–

Element	External	Effect	Common	Cg	GLES	GLSL
<generator>	–	–	–	YES	–	YES
<include>	–	–	–	YES	–	YES
<index_of_refraction>	–	–	YES	–	–	–
<instance_effect>	YES	–	–	–	–	–
<instance_material>	YES	–	–	–	–	–
<lambert>	–	–	YES	–	–	–
<library_effects>	YES	–	–	–	–	–
<library_materials>	YES	–	–	–	–	–
<material>	YES	–	–	–	–	–
<modifier>	YES	YES	YES	YES	YES	YES
<name>	–	–	–	YES	–	YES
<newparam>	–	YES	YES	YES	YES	YES
<param> (FX)	–	–	YES	YES	–	YES
<pass>	–	–	–	YES	YES	YES
<phong>	–	–	YES	–	–	–
<profile_CG>	–	–	–	YES	–	–
<profile_COMMON>	–	–	YES	–	–	–
<profile_GLES>	–	–	–	–	YES	–
<profile_GLSL>	–	–	–	–	–	YES
<reflective>	–	–	YES	–	–	–
<reflectivity>	–	–	YES	–	–	–
<RGB>	–	–	–	–	YES	–
<sampler_state>	–	–	–	–	YES	–
<sampler1D>	–	–	YES	YES	–	YES
<sampler2D>	YES	YES	YES	YES	–	YES
<sampler3D>	YES	YES	YES	YES	–	YES
<samplerCUBE>	YES	YES	YES	YES	–	YES
<samplerDEPTH>	YES	YES	YES	YES	–	YES
<samplerRECT>	YES	YES	YES	YES	–	YES
<semantic>	YES	YES	YES	YES	YES	YES
<setparam>	YES	YES	YES	YES	YES	YES
<shader>	–	–	–	YES	–	YES
<shininess>	–	–	YES	–	–	–
<specular>	–	–	YES	–	–	–
<stencil_clear>	–	–	–	YES	YES	YES
<stencil_target>	–	–	–	YES	YES	YES
<surface>	YES	YES	YES	YES	YES	YES
<technique> (FX)	–	–	YES	YES	YES	YES
<technique_hint>	YES	–	–	–	–	–
<texcombiner>	–	–	–	–	YES	–
<texenv>	–	–	–	–	YES	–
<texture>	–	–	YES	–	–	–
<texture_pipeline>	–	–	–	–	YES	–
<texture_unit>	–	–	–	–	YES	–

Element	External	Effect	Common	Cg	GLES	GLSL
<transparency>	–	–		–	–	–
<transparent>	–	–	YES	–	–	–
<usertype>	–	–	–	YES	–	–

About Parameters

In COLLADA, a **<param>** (core) or **<newparam>** element declares a symbol whose name or semantics declares a bindable parameter within the given scope. Therefore, the parameter's name as well as its semantics define 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 as 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 **float4**, or bool to int.

In **<effect>** scope, parameters are available to all platforms, but parameters declared inside a specific **<profile_*>** block are available only to shaders that are also inside that profile. Parameters declared outside of this barrier may require casting when used inside a **<profile_*>** block.

COLLADA provides the following element for working with general parameters:

- **<param>** (core): Defines a parameter and sets its type and value for immediate use.

COLLADA provides the following elements for working with parameters in effects:

- **<newparam>**: Creates a parameter.
- **<setparam>**: Changes or sets the type and value of a parameter.
- **<modifier>**: Specifies the volatility or linkage of parameters, such as UNIFORM or SHARED, among others.
- **<array>**: In **<newparam>** or **<setparam>**, defines the parameter to be an array.
- **<usertype>**: In **<newparam>** or **<setparam>**, defines the parameter to be a structure.
- **<connect_param>**: Creates a symbolic connection between two existing parameters.
- **<annotate>**: Represents an object of the form symbol=value for use in parameters and in other places within FX.
- **<param>** (FX): Refers to an existing parameter created by **<newparam>**.

Locating a Parameter in <bind> and <bind_vertex_input>

The **<bind>** and **<bind_vertex_input>** elements bind the target to a parameter in an **<effect>**. The search string that identifies the parameter in the **<effect>** is specified by the `semantic` attribute. When locating the parameter in the **<effect>**, search in the following order:

- Find a COLLADA FX parameter by semantic.
- If the profile contains shading language code, find a parameter within the shader by semantic.
- Find a COLLADA FX parameter by `sid`.
- If the profile contains shading language code, find a parameter within the shader by name.

Shaders

COLLADA provides several elements that describe shaders:

- **<blinn>**
- **<constant>**
- **<lambert>**
- **<phong>**

and several elements that describe aspects of shaders:

- **<bind>** (shader)
- **<code>**
- **<compiler_options>**
- **<compiler_target>**
- **<include>**
- **<name>**
- **<shader>**

Rendering

Determining Transparency (Opacity)

If either **<transparent>** or **<transparency>** exists then transparency rendering is activated, the renderer needs to turn on alpha blending mode, and the following equations define how to combine the two values. Use these equations to get the correct results based on the opaque setting of **<transparent>**, where `fb` is the frame buffer (that is, the image behind what is being rendered) and mat is the material color before the transparency calculation.

- In `A_ONE` opaque mode:

```
result.r = fb.r * (1.0f - transparent.a * transparency) + mat.r *
(transparent.a * transparency)
result.g = fb.g * (1.0f - transparent.a * transparency) + mat.g *
(transparent.a * transparency)
result.b = fb.b * (1.0f - transparent.a * transparency) + mat.b *
(transparent.a * transparency)
result.a = fb.a * (1.0f - transparent.a * transparency) + mat.a *
(transparent.a * transparency)
```

- In **RGB_ZERO** opaque mode:

```
result.r = fb.r * (transparent.r * transparency) + mat.r *
(1.0f -transparent.r * transparency)
result.g = fb.g * (transparent.g * transparency) + mat.g *
(1.0f -transparent.g * transparency)
result.b = fb.b * (transparent.b * transparency) + mat.b *
(1.0f -transparent.b * transparency)
result.a = fb.a * (luminance(transparent.rgb) * transparency) + mat.a *
(1.0f - luminance(transparent.rgb) * transparency)
```

where luminance is the function, based on the ISO/CIE color standards (see ITU-R Recommendation BT.709-4), that averages the color channels into one value:

```
luminance = (color.r * 0.212671) +
            (color.g * 0.715160) +
            (color.b * 0.072169)
```

The interaction between **<transparent>** and **<transparency>** is as follows:

- If **<transparent>** does not exist then it has no effect on the equation's result, and the opaque mode is the default opaque mode. This is equivalent to:

  ```
  transparent = <color> 1.0 1.0 1.0 1.0 </color>
  ```

- If **<transparency>** does not exist then it has no effect on the equation's result. This is equivalent to a factor that is 1.0:

  ```
  transparency = <float> 1.0 <float>
  ```

- If both **<transparent>** and **<transparency>** exist then both are honored.

In the following example, the colors are used as specified but the RGB values are ignored for transparency calculations because `A_ONE` specifies that the transparency information comes from the alpha channel, not the RGB channels:

```
<transparent opaque=A_ONE><color>1 0 0.5 0</color></transparent>
```

Texturing

Texture Mapping in <profile_COMMON>

This section provides an introduction to textures, samplers, surfaces, and images.

To use an image as a texture, use the element relationships as follows:

```
texture->sampler->surface->image
```

From the smallest part to the largest:

- An **<image>** is embedded or referenced file data. It might be a format of traditional 2D planes, such as BMP, or it might be a complicated 3D format, such as DDS or OpenEXR, consisting of multiple image planes. An image is not, by itself, a surface although it might contain the complete set of information to form a surface.
- A **<surface>** element collects one or more images to form a single cohesive structure designed for 3D hardware concepts, such as MIP mapping, cubes, and volumes.
- A **<sampler*>** contains instructions on how to read data at a specific 1D, 2D, or 3D coordinate from a **<surface>**. It references the **<surface>** and specifies what operations to perform to sample the data at a given coordinate. A sampler's instructions include information on how to map the coordinate onto the image, such as wrap or mirror. The instructions also include filtering modes to instruct how one or more texels near by coordinate are combined to produce the final output color.
- A profile_COMMON's **<texture>**'s responsibility is to bind geometry's texture coordinate set (array) to a **<sampler*>** so that the sampler can fetch the correct colors. The texcoord attribute on the **<texture>** is actually a semantic name. It is expected that the **<instance_geometry>**'s **<instance_material><bind_vertex_input>** makes the connection between the **<texture>**'s texcoord attribute and the mesh's texture coordinate array.

Some DCC applications also specify **<extra>** information that modifies the **TEXCOORD**s before they are plugged into the sampler, such as `offsetU`, offsetV, `rotateUV`, or noise.

The following is an example of texturing using **<instance_material>** and related elements to instantiate a material with an **<image>** supplied through a **<sampler2D>** parameter:

```
...
<image id="image_id">
  <init_from>image_file.dds</init_from>
</image>
...
<effect id="effect_id">
 ...
  <profile_COMMON>
    <technique sid="technique_sid">
      <newparam sid="surface_param_id">
        <surface>
          <init_from>image_id</init_from>
          ...
        </surface>
      </newparam>
      <newparam sid="sampler2D_param_id">
        <sampler2D>
          <source>surface_param_id</source>
          ...
        </sampler2D>
      </newparam>
      <lambert>
        <diffuse>
          <texture texture="sampler2D_param_id" texcoord="myUVs"/>
        </diffuse>
      </lambert>
      ...
</effect>
...
<material id="material_id">
  <instance_effect url="#effect_id" />
</material>
...
<geometry id="geometry_id">
  ...
  <input semantic="TEXCOORD" source="#..." offset=".." />
  <triangles material="material_symbol" count"...">
  ...
</geometry>
...
<scene>
  ...
  <instance_geometry url="#geometry_id">
    <bind_material>
        <technique_common>
          <instance_material symbol="material_symbol" target="#material_id">
              <bind_vertex_input semantic="myUVs" input_semantic="TEXCOORD" />
          </instance_material>
        </technique_common>
    </bind_material>
  </instance_geometry>
  ...
</scene>
```

Chapter 8: COLLADA FX Reference

Introduction

This section covers the elements that compose COLLADA FX.

Elements by Category

This chapter lists elements in alphabetical order. The following tables list elements by category, for ease in finding related elements.

Effects

bind_vertex_input	Binds geometry vertex inputs to effect vertex inputs upon instantiation.
effect	Provides a self-contained description of a COLLADA effect.
instance_effect	Declares the instantiation of a COLLADA material resource.
library_effects	Declares a module of **<effect>** elements.
semantic	Provides metainformation that describes the purpose of a parameter declaration.
technique (FX)	Holds a description of the textures, samplers, shaders, parameters, and passes necessary for rendering this effect using one method.
technique_hint	Adds a hint for a platform of which technique to use in this effect.

Materials

bind (material)	Materials Binds values to uniform inputs of a shader or binds values to effect parameters upon instantiation.
bind_material	Binds a specific material to a piece of geometry, binding varying and uniform parameters at the same time.
instance_material	Declares the instantiation of a COLLADA material resource.
library_materials	Declares a module of **<material>** elements.
material	Describes the visual appearance of a geometric object.

Parameters

annotate	Adds a strongly typed annotation remark to the parent object.
array	Creates a parameter of a one-dimensional array type.
connect_param	Creates a symbolic connection between two previously defined parameters.
modifier	Provides additional information about the volatility or linkage of a **<newparam>** declaration.
newparam	Creates a new, named **<param>** object in the FX Runtime, and assigns it a type, an initial value, and additional attributes at declaration time.
param (FX)	References a predefined parameter in shader binding declarations.
setparam	Assigns a new value to a previously defined parameter.
usertype	Creates an instance of a structured class for a parameter.

Profiles

profile_CG	Declares platform-specific data types and **<technique>**s for the Cg language.
profile_COMMON	Opens a block of platform-independent declarations for the common, fixed-function shader.
profile_GLES	Declares platform-specific data types and **<technique>**s for OpenGL ES.
profile_GLSL	Declares platform-specific data types and **<technique>**s for OpenGL Shading Language.

Rendering

render	Describes one effect pass to evaluate a scene.
blinn	Produces a specularly shaded surface where the specular reflection is shaded according to the Blinn BRDF approximation.
color_clear	Specifies whether a render target surface is to be cleared, and which value to use.
color_target	Specifies which **<surface>** will receive the color information from the output of this pass.
common_color_or_texture_type	A type that describes color attributes of fixed-function shader elements inside **<profile_COMMON>** effects.
common_float_or_param_type contains (**index_of_refraction**, **reflectivity**, **shininess**, **transparency**)	A type that describes the scalar attributes of fixed-function shader elements inside **<profile_COMMON>** effects.
constant	Produces a constantly shaded surface that is independent of lighting.
depth_clear	Specifies whether a render target surface is to be cleared, and which value to use.
depth_target	Specifies which **<surface>** will receive the depth information from the output of this pass.
draw	Specifies a user-defined string instructing the FX Runtime what kind of geometry to submit.
lambert	Produces a diffuse shaded surface that is independent of lighting.
pass	Provides a static declaration of all the render states, shaders, and settings for one rendering pipeline.
phong	Produces a specularly shaded surface where the specular reflection is shaded according the Phong BRDF approximation.
stencil_clear	Specifies whether a render target surface is to be cleared, and which value to use.
stencil_target	Specifies which **<surface>** will receive the stencil information from the output of this pass.

Shaders

bind (shader)	Binds values to uniform inputs of a shader or binds values to effect parameters upon instantiation.
code	Provides an inline block of source code.
compiler_options	Contains command-line options for the shader compiler.
compiler_target	Declares which profile or platform the compiler is targeting this shader for.
include	Imports source code or precompiled binary shaders into the FX Runtime by referencing an external resource.
name	Provides the entry symbol for the shader function.
shader	Declares and prepares a shader for execution in the rendering pipeline of a **<pass>**.

Texturing

alpha	Defines the alpha portion of a **<texture_pipeline>** command for combiner-mode texturing.
argument	Defines an argument of the RGB or alpha component of a texture-unit combiner-style texturing command.
generator	Describes a procedural surface generator.
image	Declares the storage for the graphical representation of an object.
library_images	Declares a module of **<image>** elements.
RGB	Defines the RGB portion of a **<texture_pipeline>** command for combiner-mode texturing.
sampler_state	Provides a two-dimensional texture sampler state for **<profile_GLES>**.
sampler1D	Declares a one-dimensional texture sampler.
sampler2D	Declares a two-dimensional texture sampler.
sampler3D	Declares a three-dimensional texture sampler.
samplerCUBE	Declares a texture sampler for cube maps.
samplerDEPTH	Declares a texture sampler for depth maps.
samplerRECT	Declares a RECT texture sampler.
surface	Declares a resource that can be used both as the source for texture samples and as the target of a rendering pass.
texcombiner	Defines a **<texture_pipeline>** command for combiner-mode texturing.
texenv	Defines a **<texture_pipeline>** command for simple, noncombiner-mode texturing.
texture_pipeline	Defines a set of texturing commands that will be converted into multitexturing operations using **glTexEnv** in regular and combiner mode.
texture_unit	Defines a texture unit that will be mapped to hardware texture units based on its usage in **<texture_pipeline>** commands.

Introduction

See Chapter 7: Getting Started with COLLADA FX.

alpha

Category: **Texturing**

Profile: **GLES**

Introduction

Defines the alpha portion of a **<texture_pipeline>** command. This is a combiner-mode texturing operation.

Concepts

See **<texcombiner>** for details about assignments and overall concepts.

Attributes

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

operator	**REPLACE | MODULATE | ADD | ADD_SIGNED | INTERPOLATE | SUBTRACT**	Infers the use of **glTexEnv(TEXTURE_ENV, COMBINE_ALPHA, operator)**. Optional. See **<texcombiner>** for details.
scale	**float**	Infers the use of **glTexEnv(TEXTURE_ENV, ALPHA_SCALE, scale)**. Optional. . See **<texcombiner>** for details.

Related Elements

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

Parent elements	**texcombiner**
Child elements	See the following subsection.
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<argument>	Sets up the arguments required for the given operator to be executed. See main entry.	N/A	1 to 3

Details

See **<texcombiner>** for details.

Example

See **<texture_pipeline>**.

annotate

Category: **Parameters**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Adds a strongly typed annotation remark to the parent object.

Concepts

Annotations represent objects of the form **SYMBOL** = **VALUE**, where **SYMBOL** is a user-defined identifier and **VALUE** is a strongly typed value. Annotations communicate metainformation from the Effect Runtime to the application only and are not interpreted by the COLLADA document.

Attributes

The **<annotate>** element has the following attribute:

name	**xs:NCName**	The text string name of this element that represents the **SYMBOL** in an object of the form **SYMBOL** = **VALUE**. Required.

Related Elements

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

Parent elements	**effect**, **technique** (FX) (in **profile_CG**, **profile_GLES**, **profile_GLSL**), **pass**, **newparam** (in **profile_CG**, **profile_GLES**, **profile_GLSL**, and **effect**), **technique**/**newparam** (in **profile_CG**, **profile_GLES**, **profile_GLSL**), **technique**/**setparam** (in **profile_GLSL** and **profile_GLES**), **generator**, **generator**/**setparam**, **shader**
Child elements	See the following subsection.
Other	None

Child Elements

Name/example	Description	Default	Occurrences
VALUE_TYPE	A strongly typed value that represents the **VALUE** in an object of the form **SYMBOL** = **VALUE**. Consists of a COLLADA type element that contains a value of that type. Valid type elements are: **bool**, **bool2**, **bool3**, **bool4**, **int**, **int2**, **int3**, **int4**, **float**, **float2**, **float3**, **float4**, **float2x2**, **float3x3**, **float4x4**, **string** See Chapter 9: COLLADA Types.	N/A	1

Details

There is currently no standard set of annotations.

Example

```
<annotate name="UIWidget"> <string> slider </string> </annotate>
<annotate name="UIMinValue"> <float> 0.0 </float> </annotate>
<annotate name="UIMaxValue"> <float> 255.0 </float> </annotate>
```

argument

Category: **Texturing**

Profile: **GLES**

Introduction

Defines an argument of the RGB or alpha component of a texture-unit combiner-style texturing command.

Concepts

See **<texture_pipeline>** for more details about assignments and bigger picture.

This element is context-sensitive based on its parent element.

Attributes

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

Note: In the following table, "##" means concatenate, ***idx*** represents the index in which the argument appeared inside its parent command (**<texenv>** or **<texcombiner>**), and ***source*** is a placeholder for a value.

source	Enumeration	Optional. Identifies where the source data for the argument will come from: When the parent is **<RGB>**, this infers a call to **glTexEnv(TEXTURE_ENV, SRC##*idx*##_RGB, *source*)**. When the parent is **<alpha>**, this infers a call to **glTexEnv(TEXTURE_ENV, SRC##*idx*##_ALPHA, *source*)**. Valid values are **TEXTURE** | **CONSTANT** | **PRIMARY** | **PREVIOUS**. There is no default.
operand	Enumeration	Optional. provides details about how the value should be read from the source: When the parent is **<RGB>**, this infers a call to **glTexEnv(TEXTURE_ENV, OPERAND##*idx*##_RGB, *source*)** and valid values are: **SRC_COLOR** | **ONE_MINUS_SRC_COLOR** | **SRC_ALPHA** | **ONE_MINUS_SRC_ALPHA**; the default is **SRC_COLOR**. When the parent is **<alpha>**, this infers a call to **glTexEnv(TEXTURE_ENV, OPERAND##*idx*##_ALPHA, *source*)** and valid values are: **SRC_ALPHA** | **ONE_MINUS_SRC_ALPHA**; the default is **SRC_ALPHA**.
unit	**xs:NCName**	Optional. The name of a texture unit from which the source is to be read. Used only when ***source*="TEXTURE"**. Acceptable values depend upon which version of OpenGL ES the shader is designed for: • GLES 1.0, all arguments within a **<texenv>** element must refer to the same texture unit because there is no combiner crossbar. • GLES 1.1, the texture combiner crossbar is available, so the unit attribute can refer to any texture-unit name.

Related Elements

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

Parent elements	**RGB**, **alpha**
Child elements	None
Other	None

Details

<argument> sets up the arguments required for the given operator to be executed.

Example

See **<texture_pipeline>**.

array

Category: **Parameters**

Profile: **CG**

Introduction

Creates a parameter of a one-dimensional array type.

Concepts

Array type parameters pass sequences of elements to shaders. Array types are sequences of a single data type, To create a multidimensional array declare it as an array of array types.

Arrays can be either unsized or sized declarations, with an unsized array requiring a concrete size (and data) to be set using **<setparam>** before it can be used as a parameter for a shader.

Attributes

The **<array>** element has the following attribute:

length	**xs:positiveInteger**	The number of elements in the array. Required in **<newparam>**; optional in **<setparam>**.

Related Elements

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

Parent elements	**newparam** (in some cases), **setparam** (in some cases), **usertype**
Child elements	See the following subsections.
Other	None

Child Elements in CG Scope

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
cg_value_type	See “Value Types” at the end of the chapter for value types valid in CG scope.	N/A	0 or more
<connect_param>	Valid only when **<array>** is a child of **<newparam>**. See main entry.	N/A	0 or more
<usertype>	See main entry.	N/A	0 or more
<array>	Declares another dimension in the array.	N/A	0 or more

Child Elements in GLSL Scope

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
glsl_value_type	See “Value Types” at the end of the chapter for value types valid in GLSL scope.	N/A	0 or more
<array>	Declares another dimension in the array.	N/A	0 or more

Details

After creation, array elements can be addressed directly in **<setparam>** declarations using the normal CG syntax for array indexing and structure dereferencing, for example, "`array[3].element`".

Note: Although the schema does not prevent using members of different types, the convention that must be followed for an array to be valid is the use of a consistent value type.

Example

```
<newparam sid="numbers">
  <array length="4">
    <float>1.0</float>
    <float>2.0</float>
    <float>3.0</float>
    <float>4.0</float>
  </array>
</newparam>
<setparam ref="numbers[2]">
  <float>2.5</float>
</setparam>
```

bind

(material)

Category: **Materials**

Profile: **External**

Introduction

Binds values to uniform inputs of a shader or binds values to effect parameters upon instantiation.

Concepts

Shaders with uniform parameters can have values bound to their inputs at compile time, and need values assigned to the uniform parameters at execution time. These values can be literal values, constant parameters or uniform parameters. In the case of constant values, these declarations of parameters for the shader can be used by the compiler to produce optimized shaders for that specific declaration.

<bind> is also used to map predefined parameters to uniform inputs at run time, allowing the FX Runtime to automatically assign values to a shader from its pool of predefined parameters.

Attributes

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

semantic	**xs:NCName**	Which effect parameter to bind. Optional
target	**xs:token**	The location of the value to bind to the specified semantic. This text string is a path-name following a simple syntax described in the “Addressing Syntax” section. Required.

Related Elements

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

Parent elements	**instance_material**
Child elements	None
Other	None

Details

Some FX Runtime compilers require that every uniform input is bound before compilation can happen, while other FX Runtimes can “semicompile” shaders into nonexecutable object code that can be inspected for unbound inputs.

The **<bind>** and **<bind_vertex_input>** elements bind the target to a parameter in an **<effect>**. The search string that identifies the parameter in the **<effect>** is specified by the semantic attribute. When locating the parameter in the **<effect>**, search in the following order:

- Find a COLLADA FX parameter by semantic
- If the profile contains shading language code, find a parameter within the shader by semantic.
- Find a COLLADA FX parameter by sid.
- If the profile contains shading language code, find a parameter within the shader by name.

Example

```
<instance_material symbol="RedMat" target="#RedCGEffect">
  <bind semantic="LIGHTPOS0" target="LightNode/translate"/>
</instance_material>
```

bind

(shader)

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Binds values to uniform inputs of a shader or binds values to effect parameters upon instantiation.

Concepts

Shaders with uniform parameters can have values bound to their inputs at compile time, and need values assigned to the uniform parameters at execution time. These values can be literal values, constant parameters or uniform parameters. In the case of constant values, these declarations of parameters for the shader can be used by the compiler to produce optimized shaders for that specific declaration.

<bind> is also used to map predefined parameters to uniform inputs at run time, allowing the FX Runtime to automatically assign values to a shader from its pool of predefined parameters.

Attributes

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

symbol	**xs:NCName**	The identifier for a uniform input parameter to the shader (a formal function parameter or in-scope global) that will be bound to an external resource. Required.

Related Elements

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

Parent elements	**shader**
Child elements	See the following subsection.
Other	None

Child Elements

Note: Exactly one of the child elements **<param>** or a value type must appear. They are mutually exclusive.

Name/example	Description	Default	Occurrences
<param> (FX)	See main entry.	None	See “Note”
cg_value_type or *glsl_value_type*	See “Value Types” at the end of the chapter for value types valid in CG or GLSL scope, depending on context.	N/A	See “Note”

Details

Some FX Runtime compilers require that every uniform input is bound before compilation can happen, while other FX Runtimes can “semicompile” shaders into nonexecutable object code that can be inspected for unbound inputs.

Example

```
<shader stage="VERTEXPROGRAM">
  <name source="fooShader-code-1">main</name>
  <bind symbol="diffusecol">
    <float3> 0.30 .52 0.05 </float3>
  </bind>
  <bind symbol="lightpos">
    <param ref="OverheadLightPos_03">
  </bind>
</shader>
```

bind_material

Category: **Materials**

Profile: **External**

Introduction

Binds a specific material to a piece of geometry, binding varying and uniform parameters at the same time.

Concepts

When a piece of geometry is declared, it can request that the geometry have a particular material, for example,

```
<polygons name="leftarm" count="2445" material="bluePaint">
```

This abstract symbol needs to be bound to a particular material instance. The application does the instantiation when processing the **<instance_geometry>** elements within the **<bind_material>** elements. The application scans the geometry for material attributes and binds actual material objects to them as indicated by the **<instance_material>** symbol attributes. See "Example" below.

While a material is bound, shader parameters might also need to be resolved. For example, if an effect requires two light source positions as inputs but the scene contains eight unique light sources, which two light sources will be used on the material? If an effect requires one set of texture coordinates on an object, but the geometry defined two sets of texcoords, which set will be used for this effect? **<bind_material>** is the mechanism for disambiguating inputs in the scene graph.

Inputs are bound to the scene graph by naming the semantic attached to the parameters and connecting them by COLLADA URL syntax to individual elements of nodes in the scene graph, right down to the individual elements of vectors.

Attributes

The **<bind_material>** element has no attributes.

Related Elements

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

Parent elements	**instance_geometry**, **instance_controller**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<param> (core)	In **<bind_material>** these are added to be targets for animation. These objects can then be bound to input parameters in the normal manner without requiring the animation targeting system to parse the internal layout of an **<effect>**. See main entry.	None	0 or more

Name/example	Description	Default	Occurrences
<technique_common>	Specifies material binding information for the common profile that all COLLADA implementations must support. See “The Common Profile” section for usage information and the following subsection for child element details.	N/A	1
<technique> (core)	Each **<technique>** specifies material binding information for a specific profile as designated by the **<technique>**’s profile attribute. See main entry.	None	0 or more
<extra>	See main entry.	N/A	0 or more

Child Elements for bind_material / technique_common

Name/example	Description	Default	Occurrences
<instance_material>	See main entry.	N/A	1 or more

Details

Example

```
<instance_geometry url="#BeechTree">
  <bind_material>
    <param sid="windAmount" semantic="WINDSPEED" type="float3"/>
    <technique_common>
      <instance_material symbol="leaf" target="MidsummerLeaf01"/>
      <instance_material symbol=" RedMat "target="`">
        <bind semantic="LIGHTPOS0" target="LightNode/translate"/>
        <bind semantic="TEXCOORD0" target="BeechTree/texcoord2"/>
      </instance_material>
    </technique_common>
  </bind_material>
</instance_geometry>
```

The following example shows **<bind_material>** binding a material with a geometry. The connection between the **<material>** id attribute and the material attribute of a **<polygons>** element is established by the **<instance_material>** element:

```
...
<material id="MyMaterial">  ... </material>
...
<geometry>
   ...
   <polygons name="leftarm" count="2445" material="bluePaint">
   ...
</geometry>
...
<scene>
  ...
  <instance_geometry ...>
    <bind_material>
      <technique_common>
        <instance_material symbol="bluePaint" target="MyMaterial">
          ...
        </instance_material>
      </technique_common>
    </bind_material>
  </instance_geometry>
  ...
</scene>
```

bind_vertex_input

Category:

Profile: **External**

Introduction

Binds geometry vertex inputs to effect vertex inputs upon instantiation.

Concepts

This element is useful, for example, in binding a vertex-program parameter to a **<source>**. The vertex program needs data already gathered from sources. This data comes from the **<input>** elements under the collation elements such as **<polygons>** or **<triangles>**. Inputs access the data in **<source>**s and guarantee that it corresponds with the polygon vertex "fetch". To reference the **<input>**s for binding, use **<bind_vertex_input>**.

Attributes

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

semantic	**xs:NCName**	Which effect parameter to bind. Required.
input_semantic	**xs:NCName**	Which input semantic to bind. Required.
input_set	**uint**	Which input set to bind. Optional.

Related Elements

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

Parent elements	**instance_material**
Child elements	None
Other	input

Details

The **<bind_vertex_input>** element binds geometry vertex streams (identified as **<input>** elements within geometry elements) to material effect vertex stream semantics. Although applications commonly perform automatic binding of vertex streams with identical semantic identifiers, there are frequently mismatches in a semantic identifier's meaning. Use **<bind_vertex_input>** to remove these ambiguities, which are most commonly caused by:

- Generalizations; for example, **TEXCOORD0** vs. **DIFFUSE-TEXCOORD**
- Spelling differences; for example, **COLOR** vs. **COLOUR**
- Abbreviations
- Verbosity
- Synonyms

The **<bind>** and **<bind_vertex_input>** elements bind the target to a parameter in an **<effect>**. The search string that identifies the parameter in the **<effect>** is specified by the semantic attribute. When locating the parameter in the **<effect>**, search in the following order:

- Find a COLLADA FX parameter by semantic
- If the profile contains shading language code, find a parameter within the shader by semantic.

- Find a COLLADA FX parameter by sid.
- If the profile contains shading language code, find a parameter within the shader by name.

Example

The following example applies a wet-feathers material to a duck model. The duck model may have normal map texture coordinates, which it calls **TEXCOORD0** (semantic=**TEXCOORD** and set=0), and base color texture coordinates, which it calls **TEXCOORD1**.

There are circumstances where semantic names for texture coordinates (or other goemetry streams) do not match up. For example, the wet-feathers material may have normal map texture coordinates called **TEXCOORD1** and base color texture coordinates called **TEXCOORD0**. In this case, the meanings of these identical names have been swapped so, to bind these mismatched objects, swap them using **<bind_vertex_input>**.

Note that the semantic attribute refers to the semantic in the material effect while the attributes prefixed with **input_** refer to the geometry vertex **<bind_vertex_input>** streams, which are identified by the combination of a semantic name and a set number.

```
<instance_geometry url="#duck">
  <bind_material>
    <technique_common>
      <instance_material symbol="region1" target="#wet-feathers">
          <bind_vertex_input semantic="TEXCOORD1"
                 input_semantic="TEXCOORD" input_set="0"/>
          <bind_vertex_input semantic="TEXCOORD0"
                 input_semantic="TEXCOORD" input_set="1"/>
      </instance_material>
    </technique_common>
  </bind_material>
</instance_geometry>
```

blinn

Category: **Shaders**

Profile: **COMMON**

Introduction

Produces a specularly shaded surface with a Blinn BRDF approximation.

Concepts

Used inside a **<profile_COMMON>** effect, **<blinn>** declares a fixed-function pipeline that produces a shaded surface according to the Blinn-Torrance-Sparrow lighting model or a close approximation.

This equation is complex and detailed via the ACM, so it is not detailed here. Refer to "Models of Light Reflection for Computer Synthesized Pictures," SIGGRAPH 77, pp 192-198 (http://portal.acm.org/citation.cfm?id=563893).

Maximizing Compatibility:

To maximize application compatibility, it is suggested that developers use the Blinn-Torrance-Sparrow for **<shininess>** range of 0 to 1. For **<shininess>** greater than 1.0, the COLLADA author was probably using an application that followed the Blinn-Phong lighting model, so it is recommended to support both Blinn equations according to whichever range the shininess resides in.

The Blinn-Phong equation

The Blinn-Phong equation is:

$$color = <emission> + <ambient> * al + <diffuse> * \max(N \bullet L, 0) + <specular> * \max(H \bullet N, 0)^{<shininess>}$$

where:

- al — A constant amount of ambient light contribution coming from the scene. In the COMMON profile, this is the sum of all the **<light><technique_common><ambient>** values in the **<visual_scene>**.
- N — Normal vector (normalized)
- L — Light vector (normalized)
- I — Eye vector (normalized)
- H — Half-angle vector,calculated as halfway between the unit Eye and Light vectors, using the equation H= normalize(I+L)

Attributes

The **<blinn>** element has no attributes.

Related Elements

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

Parent elements	**technique** (FX) in **profile_COMMON**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<emission>	Declares the amount of light emitted from the surface of this object. See **common_color_or_texture_type**.	N/A	0 or 1
<ambient> (FX)	Declares the amount of ambient light emitted from the surface of this object. See **common_color_or_texture_type**.	N/A	0 or 1
<diffuse>	Declares the amount of light diffusely reflected from the surface of this object. See **common_color_or_texture_type**.	N/A	0 or 1
<specular>	Declares the color of light specularly reflected from the surface of this object. See **common_color_or_texture_type**.	N/A	0 or 1
<shininess>	Declares the specularity or roughness of the specular reflection lobe. See **common_float_or_param_type**.	N/A	0 or 1
<reflective>	Declares the color of a perfect mirror reflection. See **common_color_or_texture_type**.	N/A	0 or 1
<reflectivity>	Declares the amount of perfect mirror reflection to be added to the reflected light as a value between 0.0 and 1.0. See **common_float_or_param_type**.	N/A	0 or 1
<transparent>	Declares the color of perfectly refracted light. See **common_color_or_texture_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<transparency>	Declares the amount of perfectly refracted light added to the reflected color as a scalar value between 0.0 and 1.0. See **common_float_or_param_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<index_of_refraction>	Declares the index of refraction for perfectly refracted light as a single scalar index. See **common_float_or_param_type**.	N/A	0 or 1

Details

Example

This is an example of a dark red effect with a shiny, pinpoint specular highlight.

```
<library_effects>
    <effect id="blinn1-fx">
      <profile_COMMON>
        <technique sid="common">
          <blinn>
            <emission>
              <color>0 0 0 1.0</color>
            </emission>
            <ambient>
```

```
          <color>0 0 0 1.0</color>
        </ambient>
        <diffuse>
          <color>0.500000 0.002000 0 1.0</color>
        </diffuse>
        <specular>
          <color>0.500000 0.500000 0.500000 1.0</color>
        </specular>
        <shininess>
          <float>0.107420</float>
        </shininess>
        <reflective>
          <color>0 0 0 1.0</color>
        </reflective>
        <reflectivity>
          <float>0</float>
        </reflectivity>
        <transparent>
          <color>0 0 0 1.0</color>
        </transparent>
        <transparency>
          <float>1.000000</float>
        </transparency>
        <index_of_refraction>
          <float>0</float>
        </index_of_refraction>
      </blinn>
    </technique>
  </profile_COMMON>
</effect>
```

code

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Provides an inline block of source code.

Concepts

Source code can be inlined into the **<effect>** declaration to be used to compile shaders.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. An identifier for the source code to allow the block to be locally referenced by other elements. Optional.

Related Elements

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

Parent elements	**technique** (FX) (in **profile_CG**, **profile_GLSL**), **generator**, **profile_CG**, **profile_GLSL**
Child elements	None
Other	None

Details

Inlined source code must escape all XML identifier characters, for example, converting “<” to “<”.

Example

```
<code sid="lighting_code">
matrix4x4 mat : MODELVIEWMATRIX;
float4 lighting_fn( varying float3 pos : POSITION,
   ...
</code>
```

color_clear

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Specifies whether a render target surface is to be cleared, and which value to use.

Concepts

Before drawing, render target surfaces may need to be reset to a blank canvas or default. The **<color_clear>** declarations specify which value to use. If no clearing statement is included, the target surface is unchanged as rendering begins.

Attributes

The **<color_clear>** element has no attributes in GLES scope.

It has the following attribute in CG and GLSL scope:

index	**xs:nonNegativeInteger**	Which of the multiple render targets is being set. The default is 0. Optional.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	None

Details

This element contains four float values representing the red, green, blue, and alpha channels.

When this element exists inside a pass, it a cue to the runtime that a particular backbuffer or render-target resource should be cleared. This means that all existing image data in the resource should be replaced with the color provided. This element puts the resource into a fresh and known state so that other operations that use this resource execute as expected.

The index attribute identifies the resource that you want to clear. An index of 0 identifies the primary resource. The primary resource may be the backbuffer or the override provided with an appropriate **<*_target>** element (**<color_target>**, **<depth_target>**, or **<stencil_target>**).

Current platforms have fairly restrictive rules for setting up multiple render targets (MRTs). For example, MRTs can have only four color buffers, which must all be the same size and pixel format, one depth buffer, and one stencil buffer active for all color buffers. The COLLADA FX declaration is designed to be looser in its restrictions, so an FX runtime must validate that a particular MRT declaration in a **<pass>** is possible before attempting to apply it, and flag it as an error if it fails.

Example

```
<color_clear index="0">0.0 0.0 0.0 0.0</color_clear>
```

color_target

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Specifies which **<surface>** will receive the color information from the output of this pass.

Concepts

Multiple Render Targets (MRTs) allow fragment shaders to output more than one value per pass, or to redirect the standard depth and stencil units to read from and write to arbitrary offscreen buffers. These elements tell the FX Runtime which previously defined surfaces to use.

Attributes

The **<color_target>** element has no attributes in GLES scope.

It has the following attributes in CG and GLSL scope:

index	**xs:nonNegativeInteger**	Indexes one of the Multiple Render Targets. The default is 0. Optional.
slice	**xs:nonNegativeInteger**	Indexes a subimage inside a target **<surface>**, including a single MIP-map level, a unique cube face, or a layer of a 3-D texture. The default is 0. Optional.
mip	**xs:nonNegativeInteger**	The default is 0. Optional.
face	**xs:nonNegativeInteger**	Valid values are **POSITIVE_X**, **NEGATIVE_X**, **POSITIVE_Y**, **NEGATIVE_Y**, **POSITIVE_Z**, and **NEGATIVE_Z**. The default is **POSITIVE_X**. Optional.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	**newparam**, **surface**

Details

Current platforms have fairly restrictive rules for setting up MRTs; for example, only four color buffers, which must be all of the same size and pixel format, and only one depth buffer and one stencil buffer active for all color buffers. The COLLADA FX declaration is designed to be looser in its restrictions, so an FX runtime must validate that a particular MRT declaration in a **<pass>** is possible before attempting to apply it, and flag it as an error if it fails.

This element contains an **xs:NCName** value that references a **<newparam>** containing a **<surface>**. If no **<color_target>** is specified, the FX runtime will use the default backbuffer set for its platform.

Example

```
<newparam sid="surfaceTex">
  <surface type="2D"/>
</newparam>
<pass>
  <color_target>surfaceTex</color_target>
</pass>
```

common_color_or_texture_type

Category: **Rendering**

Profile: **COMMON**

Introduction

A type that describes color attributes of fixed-function shader elements inside **<profile_COMMON>** effects.

Concepts

This type describes the attributes and related elements of the following elements:

- **<ambient>** (FX)
- **<diffuse>**
- **<emission>**
- **<reflective>**
- **<specular>**
- **<transparent>**

Attributes

Only **<transparent>** has an attribute; other elements of type **common_color_or_texture_type** have no attributes.

opaque	**fx_opaque_enum**	Specifies from which channel to take transparency information. Optional. Valid values are: • **A_ONE** (the default): Takes the transparency information from the color's alpha channel, where the value 1.0 is opaque. • **RGB_ZERO**: Takes the transparency information from the color's red, green, and blue channels, where the value 0.0 is opaque, with each channel modulated independently.

Related Elements

Elements of type **common_color_or_texture_type** relate to the following elements:

Parent elements	**constant** (FX), **lambert**, **phong**, **blinn**
Child elements	See the following subsection.
Other	None

Child Elements

Note: Exactly one of the child elements **<color>**, **<param>**, or **<texture>** must appear. They are mutually exclusive.

Name/example	Description	Default	Occurrences
<color>	The value is a literal color, specified by four floating-point numbers in RGBA order. See main entry.	N/A	See "Note"
<param> (FX)	The value is specified by a reference to a previously defined parameter in the current scope that can be cast directly to a **<float4>**. See main entry.	N/A	See "Note"

Name/example	Description	Default	Occurrences
<texture **texture="myParam"** **texcoord="myUVs">** **<extra…/>** **</texture>**	The value is specified by a reference to a previously defined **<sampler2D>** object. The texcoord attribute provides a semantic token, which will be referenced within **<bind_material>** to bind an array of texcoords from a **<geometry>** instance to the **<texture_unit>**. Both attributes are required and are of type **xs:NCName**. The **<extra>** child element can appear 0 or 1 time.	N/A	See “Note”

Details

The schema does not specify default colors for **<ambient>**, **<diffuse>** and other child elements of the shaders **<blinn>**, **<constant>**, **<lambert>**, and **<phong>**. If any child element is unspecified, apply the specified shader equation without that portion. This provides equivalent results to explicitly specifying black for that child element. For example, the equation for **<phong>** without **<diffuse>** would be:

$$color = < emission > + < ambient > * al + < specular > * \max\left(R \bullet I, 0\right)^{<shininess>}$$

Example

common_float_or_param_type

Category: **Rendering**

Profile: **COMMON**

Introduction

A type that describes the scalar attributes of fixed-function shader elements inside **<profile_COMMON>** effects.

Concepts

This type describes the attributes and related elements of the following elements:

- **<index_of_refraction>**
- **<reflectivity>**
- **<shininess>**
- **<transparency>**

Attributes

Elements of type **common_float_or_param_type** have no attributes.

Related Elements

Elements of type **common_float_or_param_type** relate to the following elements:

Parent elements	**constant** (FX), **lambert**, **phong**, **blinn**
Child elements	See the following subsection.
Other	None

Child Elements

Note: Exactly one of the child elements **<float>** or **<param>** must appear. They are mutually exclusive.

Name/example	Description	Default	Occurrences
<float sid="mySID">	The value is represented by a literal floating-point scalar, for example: **<float> 3.14 </float>** The sid attribute is optional.	None	See “Note”
<param> (FX)	The value is represented by a reference to a previously defined parameter that can be directly cast to a floating-point scalar. See main entry.	None	See “Note”

Details

Behavior of **<transparent>** and **<transparency>**.

See discussion in **common_color_or_texture_type**.

Example

compiler_options

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Contains command-line options for a shader compiler.

Concepts

The shader compiler is an external tool that accepts shader program source code and input and compiles it into machine executable object code. The shader compiler accepts command-line options that configure it to perform specific operations.

Attributes

The **<compiler_options>** element has no attributes.

Related Elements

The **<compiler_options>** relates to the following elements:

Parent elements	**shader**
Child elements	None
Other	None

Details

This element contains text that is the command-line options given to the tool as a text string.

Example

```
<compiler_options> -o3 –finline_level 6 </compiler_options>
```

compiler_target

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Declares the profile or platform for which the compiler is targeting this shader.

Concepts

Some FX Runtime compilers can generate object code for several platforms or generations of hardware. This declaration specifies the compiler target profile as a string.

Attributes

The **<compiler_target>** element has no attributes.

Related Elements

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

Parent elements	**shader**
Child elements	None
Other	None

Details

Contains an **xs:NMTOKEN**. The **<compiler_target>** is a member of **<shader>** for **<profile_CG>** and identifies the combination of hardware sophistication and API for which the shader is intended to be used. For CG, depending on your target, different keywords may be acceptable in your shader code. Common strings are **arbvp1**, **arbfp1**, **glslv**, and **glslf**. These values are not explicitly enumerated to enable future expandability. Refer to the appropriate shader and compiler reference manuals for more extensive lists..

Example

For OpenGL ARB vertex shader:

```
<compiler_target>arbvp1<compiler_target>
```

For OpenGL ARB fragment shader:

```
<compiler_target>arbfp1<compiler_target>
```

For OpenGL GLSL vertex shader:

```
<compiler_target>glslv<compiler_target>
```

For OpenGL GLSL fragment shader:

```
<compiler_target>glslf<compiler_target>
```

connect_param

Category: **Parameters**

Profile: **CG**

Introduction

Creates a symbolic connection between two previously defined parameters.

Concepts

Connecting parameters allows a single parameter to be connected to inputs in many shaders. By setting this parent value all child references are automatically updated.

This connection mechanism allows common parameter values to be set once and reused many times, and is also the mechanism that allows concrete classes to be attached to abstract interfaces. For example, a shader may have an abstract interface of type “Light” as a uniform input parameter, and this declaration can be fully resolved by connecting an instance of a concrete **<usertype>** structure to that parameter.

Attributes

The **<connect_param>** element has the following attribute:

ref	**xs:token**	References the target parameter to be connected to the current parameter. Required.

Related Elements

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

Parent elements	**setparam**, **array**, **usertype**
Child elements	None
Other	**newparam**, param

Details

Example

```
<setparam ref="scene.light[2]">
  <connect_param ref="OverheadSpotlight_B"/>
</setparam>
```

constant

(FX)

Category: **Shaders**

Profile: **COMMON**

Introduction

Produces a constantly shaded surface that is independent of lighting.

Note: For the **<constant>** related to texture combiners, see **<texenv>** and **<texcombiner>**.

Concepts

Used inside a **<profile_COMMON>** effect, declares a fixed-function pipeline that produces a constantly shaded surface that is independent of lighting.

The reflected color is calculated as:

$$color = < emission > + < ambient > * al$$

where:

- al — A constant amount of ambient light contribution coming from the scene. In the COMMON profile, this is the sum of all the **<light><technique_common><ambient><color>** values in the **<visual_scene>**.

Attributes

The **<constant>** element has no attributes.

Related Elements

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

Parent elements	**technique** (FX) in **profile_COMMON**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<emission>	Declares the amount of light emitted from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1
<reflective>	Declares the color of a perfect mirror reflection. See **common_color_or_texture_type**	N/A	0 or 1
<reflectivity>	Declares the amount of perfect mirror reflection to be added to the reflected light as a value between 0.0 and 1.0. See **common_float_or_param_type**	N/A	0 or 1

Name/example	Description	Default	Occurrences
<transparent>	Declares the color of perfectly refracted light. See **common_color_or_texture_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<transparency>	Declares the amount of perfectly refracted light added to the reflected color as a scalar value between 0.0 and 1.0. See **common_float_or_param_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<index_of_refraction>	Declares the index of refraction for perfectly refracted light as a single scalar index. See **common_float_or_param_type**	N/A	0 or 1

Details

Example

The following example shows a cube that is fully transparent (invisible):

```
<effect id="surfaceShader1-fx">
      <profile_COMMON>
        <technique sid="common">
          <constant>
            <emission>
              <color>0 0 0 1.0</color>
            </emission>
            <reflective>
              <color>1.000000 1.000000 1.000000 1.0</color>
            </reflective>
            <reflectivity>
              <float>0.000000</float>
            </reflectivity>
            <transparent opaque="RGB_ZERO">
              <color>1.000000 1.000000 1.000000 1.0</color>
            </transparent>
            <transparency>
              <float>1.000000</float>
            </transparency>
            <index_of_refraction>
              <float>0</float>
            </index_of_refraction>
          </constant>
        </technique>
      </profile_COMMON>
    </effect>
```

In the preceding example, to change the cube to opaque black from transparent, change this:

```
<transparent opaque="RGB_ZERO"
```

to this:

```
<transparent opaque="A_ONE"
```

The following example simply sets the constant to red:

```
<profile_COMMON>
   <technique sid="T1">
      <constant>
         <emission><color>1.0 0.0 0.0 1.0</color></emission>
      </constant>
   </technique>
</profile_COMMON>
```

This example takes the color from a parameter:

```
<profile_COMMON>
   <newparam sid="myColor">
      <float4> 0.2 0.56 0.35 1</float4>
   </newparam>
   <technique sid="T1">
      <constant>
         <emission><param ref="myColor"/></emission>
      </constant>
   </technique>
</profile_COMMON>
```

Move-rasterizer-based viewers will not support reflection or transparent properties.

depth_clear

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Specifies whether a render target surface is to be cleared, and which value to use.

Concepts

Before drawing, render target surfaces may need resetting to a blank canvas or to a default. These **<depth_clear>** declarations specify which value to use. If no clearing statement is included, the target surface is unchanged as rendering begins.

Attributes

The **<depth_clear>** element has no attributes in GLES scope.

It has the following attribute in CG and GLSL scope:

index	**xs:nonNegativeInteger**	Which of the multiple render targets (MRTs) is being set. The default is 0. Optional.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	None

Details

This element contains a single float value that is used to clear a resource.

When this element exists inside a pass, it a cue to the runtime that a particular backbuffer or render-target resource should be cleared. This means that all existing image data in the resource should be replaced with the float value provided. This puts the resource into a fresh and known state so that other operations with this resource execute as expected.

The index attribute identifies the resource that you want to clear. An index of 0 identifies the primary resource. The primary resource may be the backbuffer or the override provided with an appropriate **<*_target>** element (**<color_target>**, **<depth_target>**, or **<stencil_target>**)

Current platforms have fairly restrictive rules for setting up MRTs; for example, MRTs can have only four color buffers, which must be all of the same size and pixel format, and only one depth buffer and one stencil buffer active for all color buffers. The COLLADA FX declaration is designed to be looser in its restrictions, so an FX runtime must validate that a particular MRT declaration in a **<pass>** is possible before attempting to apply it, and flag it as an error if it fails.

Example

```
<depth_clear index="0">0.0</depth_clear>
```

depth_target

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Specifies which **<surface>** will receive the depth information from the output of this pass.

Concepts

Multiple Render Targets (MRTs) allow fragment shaders to output more than one value per pass, or to redirect the standard depth and stencil units to read from and write to arbitrary offscreen buffers. These elements tell the FX Runtime which previously defined surfaces to use.

Attributes

The **<depth_target>** element has no attributes in GLES scope.

It has the following attributes in CG and GLSL scope:

index	**xs:nonNegativeInteger**	Indexes one of the Multiple Render Targets (MRTs). The default is 0. Optional.
slice	**xs:nonNegativeInteger**	Indexes a subimage inside a target **<surface>**, including a single MIP-map level, a unique cube face, or a layer of a 3-D texture. The default is 0. Optional.
mip	**xs:nonNegativeInteger**	The default is 0. Optional.
face	Enumeration	Valid values are **POSITIVE_X**, **NEGATIVE_X**, **POSITIVE_Y**, **NEGATIVE_Y**, **POSITIVE_Z**, and **NEGATIVE_Z**. The default is **POSITIVE_X**. Optional.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	None

Details

Current platforms have fairly restrictive rules for setting up MRTs; for example, only four color buffers, which must be all of the same size and pixel format, and only one depth buffer and one stencil buffer active for all color buffers. The COLLADA FX declaration is designed to be looser in its restrictions, so an FX runtime must validate that a particular MRT declaration in a **<pass>** is possible before attempting to apply it, and flag it as an error if it fails.

If no **<depth_target>** is specified, the FX Runtime will use the default depthbuffer set for its platform.

This element contains an **xs:NCName** that identifies the surface to receive the depth output.

Example

```
<newparam sid="depthSurface">
  <surface type="2D"/>
</newparam>
```

```
<pass>
  <depth_target>depthSurface</depth_target>
  <depth_clear>0.0</depth_clear>
</pass>
```

draw

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Instructs the FX Runtime what kind of geometry to submit.

Concepts

When executing multipass techniques, each pass may require different types of geometry to be submitted. One pass may require a model to be submitted, another pass may need a full screen quad to exercise a fragment shader over each pixel in an offscreen buffer, while another pass may need only front-facing polygons. **<draw>** declares a user-defined string that can be used as a semantic describing to the FX Runtime what geometry is expected for this pass.

Attributes

The **<draw>** element has no attributes.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	None

Details

The **<draw>** element contains an **xs:string**. The following list includes common strings to use in **<draw>**, although you are not limited to only these strings:

- **GEOMETRY**: The geometry associated with this **<instance_geometry>** or **<instance_material>**.
- **SCENE_GEOMETRY**: Draw the entire scene's geometry but with this effect, not the effects or materials already associated with the geometry. This is for techniques such as shadow-buffer generation, where you might be interested only in extracting the Z value from the light. This is without regard to ordering on the assumption that ZBuffer handles order.
- **SCENE_IMAGE**: Draw the entire scene into my targets. Use the appropriate effects or materials for each object. This is for effects that need an accurate image of the scene to work on for effects such as postprocessing blurs. This is without regard to ordering on the assumption that ZBuffer handles order.
- **FULL_SCREEN_QUAD**: Positions are 0,0 to 1,1 and the UVs match.
- **FULL_SCREEN_QUAD_PLUS_HALF_PIXEL**: Positions are 0,0 to 1,1 and the UVs are off by plus ½ of a pixel's UV size.

Example

```
<!-- Draw the scene to the MyRenderTarget surface -->
<pass>
   <color_target>MyRenderTarget</color_target>
```

```
      <draw>SCENE_IMAGE</draw>
</pass>

<!-- The engine should submit a screen-aligned quad despite whatever geometry
the material might be attached to... draw it to the screen-->
<pass>
      <draw>FULL_SCREEN_QUAD</draw>
</pass>

<!-- Draw the geoemtry that the material is attached to ... draw it to the
screen -->
<pass>
      <draw>GEOMETRY</draw>
</pass>
```

effect

Category: **Effects**

Profile: **Effect**

Introduction

Provides a self-contained description of a COLLADA effect.

Concepts

Programmable pipelines allow stages of the 3-D pipeline to be programmed using high-level languages. These shaders often require very specific data to be passed to them and require the rest of the 3-D pipeline to be set up in a particular way in order to function. Shader Effects is a way of describing not only shaders, but also the environment in which they will execute. The environment requires description of images, samplers, shaders, input and output parameters, uniform parameters, and render-state settings.

Additionally, some algorithms require several passes to render the effect. This is supported by breaking pipeline descriptions into an ordered collection of **<pass>** objects. These are grouped into **<technique>**s that describe one of several ways of generating an effect.

Elements inside the **<effect>** declaration assume the use of an underlying library of code that handles the creation, use, and management of shaders, source code, parameters, etc. We shall refer to this underlying library as the "FX Runtime".

Parameters declared inside the **<effect>** element but outside of any **<profile_*>** element are said to be in "**<effect>** scope". Parameters inside **<effect>** scope can be drawn only from a constrained list of basic data types and, after declaration, are available to **<shader>**s and declarations across all profiles. **<effect>** scope provides a handy way to parameterize many profiles and techniques with a single parameter.

Attributes

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

id	**xs:ID**	Global identifier for this object. Required.
name	**xs:NCName**	Pretty-print name for this effect. Optional.

Related Elements

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

Parent elements	**library_effects**
Child elements	See the following subsection.
Other	**instance_effect**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<annotate>	See main entry.	N/A	0 or more
<image>	See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
<newparam>	See main entry.	N/A	0 or more
profile	At least one profile must appear, but any number of any of the following profiles can be included: • **<profile_CG>** • **<profile_GLES>** • **<profile_GLSL>** • **<profile_COMMON>** See main entries.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

```
<effect id="fx-simple-uid71243231231" name="simple_diffuse_with_one_light">
    <profile_CG>
      <!-- see profile_CG example -->
    </profile_CG>
</effect>
```

generator

Category: **Texturing**

Profile: **CG**, **GLSL**

Introduction

Describes a procedural surface generator.

Concepts

Attributes

The **<generator>** element has no attributes.

Related Elements

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

Parent elements	**surface** (only CG and GLSL scope)
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
source_code	At least one of the following must appear. This specifies the generator code or its external location: • **<code>** • **<include>** See main entries.	N/A	1 or more
<name>	See main entry.	None	1
<setparam>	See main entry.	N/A	0 or more

Details

<generator> specifies source code that initializes each pixel in a **<surface>**.

Example

image

Category: **Texturing**

Introduction

Declares the storage for the graphical representation of an object.

Concepts

Digital imagery comes in three main forms of data: raster, vector, and hybrid. Raster imagery comprises 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.

The **<image>** element best describes raster image data, but can conceivably handle other forms of imagery. Raster imagery data is organized in n-dimensional arrays. This array organization can be leveraged by texture look-up functions to access noncolor values such as displacement, normal, or height field values.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the **<image>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
format	**xs:token**	A text string value that indicates the image format. It describes the encoding of the image in **<data>** or the format of the image referenced by **<init_from>** if it is in a nonstandard format that cannot be identified by its file extension. For example, if **<data>** in a COLLADA document contains the digital contents of a JPEG file, then set this attribute to "JPG". Optional.
height	**uint**	An integer value that indicates the height of the image in pixels. Optional.
width	**uint**	An integer value that indicates the width of the image in pixels. Optional.
depth	**uint**	An integer value that indicates the depth of the image in pixels. A 2-D image has a depth of 1, which is the default. Optional.

Related Elements

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

Parent elements	**library_images**, **effect**, **profile_CG**, **profile_GLSL**, **profile_COMMON**, **profile_GLES**, **technique** (FX) (in **profile_CG**, **profile_COMMON**, **profile_GLES**, **profile_GLSL**)
Child elements	See the following subsection.
Other	**instance_image**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
image_source	This specifies either embedded image data or an external image file. One of the following must appear: • **<data>**: Contains a sequence of hexadecimal encoded binary octets composing the embedded image data. This element has no attributes or child elements. • **<init_from>**: Contains a string of type **xs:anyURI** that specifies an external image file. This element has no attributes or child elements.	None	1
<extra>	Contains embedded image data. See main entry.	N/A	0 or more

Details

Example

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

```
<library_images>
  <image name="WoodFloor">
    <init_from>Textures/WoodFloor-01.png</init_from>
  </image>
</library_images>
```

include

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Imports source code or precompiled binary shaders into the FX Runtime by referencing an external resource.

Concepts

Attributes

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

sid	**xs:NCName**	Identifier for this source code block or binary shader. Required.
url	**xs:anyURI**	Location where the resource can be found. Required.

Related Elements

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

Parent elements	**technique** (FX) (in **profile_CG**, **profile_GLSL**), **generator**, **profile_CG**, **profile_GLSL**
Child elements	None
Other	None

Details

The **<include>** element itself contains no data.

Example

```
<include sid="ShinyShader" url="file://assets/source/shader.glsl"/>
```

instance_effect

Category: **Effects**

Profile: **External**

Introduction

Instantiates a COLLADA material resource.

Concepts

For details about instance elements in COLLADA, see “Instantiation and External Referencing” in Chapter 3: Schema Concepts.

<instance_effect> instantiates an effect definition from the **<library_effects>** and customize its parameters.

The url attribute references the effect.

<setparam>s assign values to specific effect and profile parameters that will be unique to the instance.

<technique_hint>s indicate the desired or last-used technique inside an effect profile. This allows the user to maintain the same look-and-feel of the effect instance as the last time that the user used it. Some runtime render engines may choose new techniques on the fly, but it is important for some effects and for DCC consistency to maintain the same visual appearance during authoring.

Attributes

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

sid	**xs:NCName**	Identifier for this source code block or binary shader. Required.
name	**xs:NCName**	The text string name of this element. Optional.
url	**xs:anyURI**	The URI of the location of the **<effect>** element to instantiate. Required. Can refer to a local instance or external reference. For a local instance, this is a relative URI 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. For an external reference, this is an absolute or relative URL.

Related Elements

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

Parent elements	**material**, **render**
Child elements	See the following subsection.
Other	**effect**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<technique_hint>	See main entry.	N/A	0 or more
<setparam>	See main entry.	N/A	0 or more
<extra>	Contains embedded image data. See main entry.	N/A	0 or more

Details

Example

```
<material id="BlueCarPaint" name="Light blue car paint">
  <instance_effect url="CarPaint">
    <technique_hint profile="CG" platform="PS3" ref="precalc_texture"/>
    <setparam ref="diffuse_color">
      <float3> 0.3 0.25 0.85 </float3>
      </setparam>
  </instance_effect>
</material>
```

instance_material

Category: **Materials**

Profile: **External**

Introduction

Instantiates a COLLADA material resource.

Concepts

For details about instance elements in COLLADA, see "Instantiation and External Referencing" in Chapter 3: Schema Concepts.

To use a material, it is instantiated and attached to the geometry. The symbol attribute of **<instance_material>** indicates what geometry the material is attached to and the target attribute references the material that it is instantiating.

In addition to identifying the section of the geometry to attach to (symbol), this element also defines how the vertex stream will be remapped and how scene objects will be bound to material effect parameters. These are the connections that can be done only very late and that depend on the scene geometry to which it is being connected.

<bind> connects a parameter in the material's effect by semantic to a target in the scene.

<bind_vertex_input> connects a vertex shader's vertex stream semantics (for example, TEXCOORD2) to the geometry's vertex input stream specified by the **input_semantic** and **input_set** attributes.

Attributes

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

sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Optional.
name	**xs:NCName**	The text string name of this element. Optional.
target	**xs:anyURI**	The URI of the location of the **<material>** element to instantiate. Required. Can refer to a local instance or external reference. For a local instance, this is a relative URI 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. For an external reference, this is an absolute or relative URL.
symbol	**xs:NCName**	Which symbol defined from within the geometry this material binds to. Required.

Related Elements

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

Parent elements	**technique_common** in **bind_material**
Child elements	See the following subsection.
Other	**material**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<bind> (material)	See main entry.	N/A	0 or more
<bind_vertex_input>	Binds vertex inputs to effect parameters upon instantiation. See main entry.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

```
<instance_geometry url="#BeechTree">
  <bind_material>
    <param sid="windAmount" semantic="WINDSPEED" type="float3"/>
    <technique_common>
      <instance_material symbol="leaf" target="#MidsummerLeaf01"/>
      <instance_material symbol="bark" target="#MidsummerBark03">
        <bind semantic="LIGHTPOS1" target="/scene/light01/pos"/>
        <bind semantic="TEXCOORD0" target="BeechTree/texcoord2"/>
      </instance_material>
    </technique_common>
  </bind_material>
</instance_geometry>
```

lambert

Category: **Shaders**

Profile: **COMMON**

Introduction

Produces a diffuse shaded surface that is independent of lighting.

Concepts

Used inside a **<profile_COMMON>** effect, declares a fixed-function pipeline that produces a diffuse shaded surface that is independent of lighting.

The result is based on Lambert's Law, which states that when light hits a rough surface, the light is reflected in all directions equally. The reflected color is calculated simply as:

$$color = < emission > + < ambient > * al + < diffuse > * \max(N \bullet L, 0)$$

where:

- al — A constant amount of ambient light contribution coming from the scene. In the COMMON profile, this is the sum of all the **<light><technique_common><ambient><color>** values in the **<visual_scene>**.
- N — Normal vector
- L — Light vector

Attributes

The **<lambert>** element has no attributes.

Related Elements

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

Parent elements	**technique** (FX) in **profile_COMMON**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<emission>	Declares the amount of light emitted from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1
<ambient> (FX)	Declares the amount of ambient light emitted from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1
<diffuse>	Declares the amount of light diffusely reflected from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1

Name/example	Description	Default	Occurrences
<reflective>	Declares the color of a perfect mirror reflection. See **common_color_or_texture_type**	N/A	0 or 1
<reflectivity>	Declares the amount of perfect mirror reflection to be added to the reflected light as a value between 0.0 and 1.0. See **common_float_or_param_type**	N/A	0 or 1
<transparent>	Declares the color of perfectly refracted light. See **common_color_or_texture_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<transparency>	Declares the amount of perfectly refracted light added to the reflected color as a scalar value between 0.0 and 1.0. See **common_float_or_param_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<index_of_refraction>	Declares the index of refraction for perfectly refracted light as a single scalar index. See **common_float_or_param_type**	N/A	0 or 1

Details

Example

```
<profile_COMMON>
   <newparam sid="myDiffuseColor">
      <float3> 0.2 0.56 0.35 </float3>
   </newparam>
   <technique sid="T1">
      <lambert>
         <emission><color>1.0 0.0 0.0 1.0</color></emission>
         <ambient><color>1.0 0.0 0.0 1.0</color></ambient>
         <diffuse><param ref="myDiffuseColor"/></diffuse>
         <reflective><color>1.0 1.0 1.0 1.0</color></reflective>
         <reflectivity><float>0.5</float></reflectivity>
         <transparent><color>0.0 0.0 1.0 1.0</color></transparent>
         <transparency><float>1.0</float></transparency>
      </lambert>
   </technique>
</profile_COMMON>
```

library_effects

Category: **Effects**

Profile: **External**

Introduction

Declares a module of **<effect>** elements.

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_effects>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_effects>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<effect>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_effects>** element:

```
<library_effects>
  <effect id="fullscreen_effect1">
     ...
  </effect>
</library_effects>
```

library_images

Category: **Texturing**

Introduction

Declares a module of **<image>** elements.

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_images>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the **<library_images>** element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<effect>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Example

Here is an example of a **<library_images>** element:

```
<library_images>
   <image name="Rose">
      <init_from>../flowers/rose01.jpg</init_from>
   </image>
</library_images>
```

library_materials

Category: **Materials**

Profile: **External**

Introduction

Declares a module of **<material>** elements.

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_materials>** element has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**COLLADA**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<material>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a **<library_materials>** element:

```
<library_materials>
  <material id="mat1">
    ...
  </material >

  <material id="mat2">
    ...
  </material>

</library_materials>
```

material

Category: **Materials**

Profile: **External**

Introduction

Describes the visual appearance of a geometric object.

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**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
name	**xs:NCName**	The text string name of this element. Optional.

Related Elements

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

Parent elements	**library_materials**
Child elements	See the following subsection.
Other	**instance_material**

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<instance_effect>	See main entry.	N/A	1
<extra>	See main entry.	N/A	0 or more

Details

Example

Here is an example of a simple **<material>** element. The material is contained in a material **<library_materials>** element:

```
<library_materials>
  <material id="Blue">
    <instance_effect url="#phongEffect">
      <setparam ref="AMBIENT">
        <float3>0.0 0.0 0.1</float3>
      </setparam>
      <setparam ref="DIFFUSE">
        <float3>0.15 0.15 0.1</float3>
      </setparam>
      <setparam ref="SPECULAR">
        <float3>0.5 0.5 0.5</float3>
      </setparam>
      <setparam ref="SHININESS">
        <float>16.0</float>
      </setparam>
      </instance_effect>
  </material>
</library_materials>
```

modifier

Category: **Parameters**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Provides additional information about the volatility or linkage of a **<newparam>** declaration.

Concepts

Allows COLLADA FX parameter declarations to specify constant, external, or uniform parameters.

Attributes

The **<modifier>** element has no attributes.

Related Elements

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

Parent elements	**newparam**
Child elements	None
Other	None

Details

Contains a linkage modifier. Not every linkage modifier is supported by every FX runtime. Valid modifiers are:

- **CONST**
- **UNIFORM**
- **VARYING**
- **STATIC**
- **VOLATILE**
- **EXTERN**
- **SHARED**

Example

```
<newparam sid="diffuseColor">
  <annotate name="UIWidget"><string>none</string></annotate>
  <semantic>DIFFUSE</semantic>
  <modifier>EXTERN</modifier>
  <float3> 0.30 0.56 0.12 </float>
</newparam>
```

name

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Provides the entry symbol for the shader function.

Concepts

Shader compilers require the name of a function to compile into shader object or binary code. FX Runtimes that use the translation unit paradigm can optionally specify the translation unit or symbol table to search for the symbol inside.

Attributes

The **<name>** element has the following attribute:

source	**xs:NCName**	The sid of the **<code>** or **<include>** block where this symbol will be found. Optional.

Related Elements

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

Parent elements	**shader**, **generator**
Child elements	None
Other	None

Details

The **<name>** element contains an **xs:NCName** that is the name (symbol) of the entry-point function for the shader function; source specifies where the entry point exists.

Example

```
<profile_CG>
   <include sid="foo">
    float4 fragment_shader(float4 tex:TEXCOORD)
    {
     return float4(0,0,1,1);
    }
   </include>
   <technique>
     <pass>
       <shader>
         <name source="foo">fragment_shader</name>
       </shader>
     </pass>
   </technique>
</profile_CG>
```

newparam

Category: **Parameters**

Profile: **Effect**, **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Creates a new, named **<param>** (FX) object in the FX Runtime, and assigns it a type, an initial value, and additional attributes at declaration time.

Concepts

Parameters are typed data objects that are created in the FX Runtime and are available to compilers and functions at run time.

Attributes

The **<newparam>** element has the following attribute:

sid	**xs:NCName** (in GLES and COMMON) **xs:token** (elsewhere)	Identifier for this parameter (that is, the variable name). Required.

Related Elements

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

Parent elements	**effect**, **technique** (FX) (in **profile_CG**, **profile_COMMON**, **profile_GLES**, **profile_GLSL**), **profile_CG**, **profile_COMMON**, **profile_GLSL**, **profile_GLES**
Child elements	See the following subsections.
Other	**param** (FX), **setparam**

Child Elements in profile_CG/newparam and profile_CG/technique/newparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
<semantic>	See main entry.	None	0 or 1
<modifier>	See main entry.	none	0 or 1
parameter_type	The parameter's type. Must be exactly one of the following: • **<array>**: See main entry. • **cg_value_type**: See "Value Types" at the end of the chapter for valid value types in CG scope. • **<usertype>**: See main entry.	None	1

Child Elements in profile_GLSL/newparam and profile_GLSL/technique/newparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
<semantic>	See main entry.	None	0 or 1

Name/example	Description	Default	Occurrences
<modifier>	See main entry.	None	0 or 1
parameter_type	The parameter's type. Must be exactly one of the following: • **<array>**: See main entry. • **glsl_value_type**: See "Value Types" at the end of the chapter for valid value types in GLSL scope.	None	1

Child Elements in profile_GLES/newparam, profile_GLES/technique/newparam, and effect/newparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
<semantic>	See main entry.	None	0 or 1
<modifier>	See main entry.	None	0 or 1
parameter_type	The parameter's type. Must be exactly one of the following: • **core_value_type**: See "Value Types" at the end of the chapter for valid value types in **<effect>**. • **gles_value_type**: See "Value Types" at the end of the chapter for valid value types in GLES scope.	None	1

Child Elements in profile_COMMON/newparam and profile_COMMON/technique/newparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<semantic>	See main entry.	None	0 or 1
parameter_type	The parameter's type. Must be exactly one of the following: • **<float>** • **<float2>** • **<float3>** • **<float4>** • **<surface>** • **<sampler2D>**	None	1

Details

Example

```
<newparam sid="diffuseColor">
  <annotate name="UIWidget"><string>none</string></annotate>
  <semantic>DIFFUSE</semantic>
  <modifier>EXTERN</modifier>
  <float3> 0.30 0.56 0.12 </float>
</newparam>
```

param

(FX)

Category: **Parameters**

Profile: **COMMON**, **CGs**, **GLSL**

Introduction

References a predefined parameter in shader binding declarations.

For this element in **<accessor>** or **<bind_material>**, see "**<param>** (core)".

Concepts

Parameters are typed data objects that are created in the FX Runtime and are available to compilers and functions at run time.

Attributes

See the "Details" subsection.

Related Elements

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

Parent elements	See the "Details" subsection.
Child elements	None
Other	**modifier**, **newparam**, **setparam**, **usertype**

Details

<param> refers to the sid of an existing parameter that was created using **<newparam>**. The method of specifying the sid varies depending on the **<param>**'s parent elements.

In Shader Attributes and CG <bind>

In the shader attribute elements (**<ambient>**, **<diffuse>**, and so on) and in **profile_CG**/**technique**/**pass**/**shader**/**bind**, the **<param>** element does not contain any information. In other words, **<param></param>** is always empty.

The **<param>** element has the following attribute:

ref	**xs:NCName**	Required. The sid of an existing parameter.

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

Parent elements	**ambient** (FX), **diffuse**, **emission**, **reflective**, **specular**, **transparent**, **index_of_refraction**, **reflectivity**, **shininess**, **transparency**, **profile_CG**/**technique**/**pass**/**shader**/**bind** (shader)

In GLSL <bind>

The **<param>** element does not contain any information. In other words, **<param></param>** is always empty.

The **<param>** element has the following attribute:

ref	**xs:string**	Required. The sid of an existing parameter.

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

Parent elements	**profile_GLSL**/**technique**/**pass**/**shader**/**bind** (shader)

In <texture*>

The **<param>** element contains information of type **xs:NCName**, which represents the sid of an existing parameter.

The **<param>** element has no attributes.

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

Parent elements	**texture1D**, **texture2D**, **texture3D**, **textureCUBE**, **textureRECT**, **textureDEPTH**

Example

Here is an example in a shader:

```
<shader stage="VERTEX">
  <compiler_target>ARBVP1</compiler_target>
  <name source="ThinFilm2">main</name>
  <bind symbol="lightpos">
    <param ref="LightPos_03"/>
  </bind>
</shader>
```

pass

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Provides a static declaration of all the render states, shaders, and settings for one rendering pipeline.

Concepts

<pass> describes all the render states and shaders for a rendering pipeline, and is the element that the FX Runtime is asked to "apply" to the current graphics state before the program can submit geometry.

A static declaration is one that requires no evaluation by a scripting engine or runtime system in order to be applied to the graphics state. At the time that a **<pass>** is applied, all render state settings and uniform parameters are precalculated and known.

Attributes

The **<pass>** element has the following attribute:

sid	**xs:NCName**	The optional label for this pass, allowing passes to be specified by name and, if desired, reordered by the application as the technique is evaluated. Optional.

Related Elements

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

Parent elements	**technique** (FX) (in **profile_CG**, **profile_GLES**, **profile_GLSL**)
Child elements	See the following subsections.
Other	None

Child Elements in GLES Scope

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	None	0 or 1
<color_target>	See main entry.	None	0 or 1
<depth_target>	See main entry.	None	0 or 1
<stencil_target>	See main entry.	None	0 or 1
<color_clear>	See main entry.	None	0 or 1
<depth_clear>	See main entry.	None	0 or 1
<stencil_clear>	See main entry.	None	0 or 1
<draw>	See main entry.	None	0 or 1
render_states	See "Render States" subsection.	Varies	0 or 1
<extra>	See main entry.	N/A	0 or more

Child Elements in CG or GLSL Scope

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	None	0 or 1
<color_target>	See main entry.	None	0 or 1
<depth_target>	See main entry.	None	0 or 1
<stencil_target>	See main entry.	None	0 or 1
<color_clear>	See main entry.	None	0 or 1
<depth_clear>	See main entry.	None	0 or 1
<stencil_clear>	See main entry.	None	0 or 1
<draw>	See main entry.	None	0 or 1
render_states	See “Render States” subsection.	Varies	0 or 1
<shader>	See main entry.	N/A	0 ore more
<extra>	See main entry.	N/A	0 or more

Details

Reordering passes can be useful when a single pass is applied repetitively, for example, a “blur” lowpass convolution may need to be applied to an offscreen texture several times to create the desired effect.

Example

Here is an example of a **<pass>** contained in a **<profile_CG>**:

```
<pass sid="PixelShaderVersion">
     <depth_test_enable value="true"/>
     <depth_func value="LEQUAL"/>
     <shader stage="VERTEX">
       <name>goochVS</name>
       <bind symbol="LightPos">
         <param ref="effectLightPos"/>
       </bind>
     </shader>
     <shader stage="FRAGMENT">
       <name>passThruFS</name>
     </shader>
   </pass>
```

Render States

Different FX profiles have different sets of render states available for use within the **<pass>** element.

In general, each render state element conforms to this declaration:

```
<render_state value="some_value" param="param_reference"/>
```

where the value attribute allows you to specify a value specific to the render state and the param attribute allows you to use a value stored within a param for the state. Each render state must have either a value or a param.

Some elements have an additional index attribute, as noted in the following table, that specifies to which element it belongs:

```
<render_state value="some_value" param="param_reference" index="name"/>
```

Further descriptions of these states are in the OpenGL specification.

The following table shows the render states for **<profile_CG>**, **<profile_GLSL>**, and **<profile_GLES>**. Render states are identical in all profiles except for differences noted for the GLES profile.

Render states and their child elements	Valid values or types, and index attribute	In GLES?
alpha_func **func** **value**	 **NEVER**, **LESS**, **LEQUAL**, **EQUAL**, **GREATER**, **NOTEQUAL**, **GEQUAL**, **ALWAYS** Float value 0.0 – 1.0 inclusive	
blend_func **src** **dest**	(both **src** and **dest**) **ZERO**, **ONE**, **SRC_COLOR**, **ONE_MINUS_SRC_COLOR**, **DEST_COLOR**, **ONE_MINUS_DEST_COLOR**, **SRC_ALPHA**, **ONE_MINUS_SRC_ALPHA**, **DST_ALPHA**, **ONE_MINUS_DST_ALPHA**, **CONSTANT_COLOR**, **ONE_MINUS_CONSTANT_COLOR**, **CONSTANT_ALPHA**, **ONE_MINUS_CONSTANT_ALPHA**, **SRC_ALPHA_SATURATE**	
blend_func_separate **src_rgb** **dest_rgb** **src_alpha** **dest_alpha**	Same as **blend_func** values	No
blend_equation	**FUNC_ADD**, **FUNC_SUBTRACT**, **FUNC_REVERSE_SUBTRACT**, **MIN**, **MAX**	No
blend_equation_separate **rgb** **alpha**	Same as blend_equation values	No
color_material **face** **mode**	 **FRONT**, **BACK**, **FRONT_AND_BACK** **EMISSION**, **AMBIENT**, **DIFFUSE**, **SPECULAR**, **AMBIENT_AND_DIFFUSE**	No
cull_face	**FRONT**, **BACK**, **FRONT_AND_BACK**	
depth_func	**NEVER**, **LESS**, **LEQUAL**, **EQUAL**, **GREATER**, **NOTEQUAL**, **GEQUAL**, **ALWAYS**	
fog_mode	**LINEAR**, **EXP**, **EXP2**	
fog_coord_src	**FOG_COORDINATE**, **FRAGMENT_DEPTH**	No
front_face	**CW**, **CCW**	
light_model_color_control	**SINGLE_COLOR**, **SEPARATE_SPECULAR_COLOR**	No
logic_op	**CLEAR**, **AND**, **AND_REVERSE**, **COPY**, **AND_INVERTED**, **NOOP**, **XOR**, **OR**, **NOR**, **EQUIV**, **INVERT**, **OR_REVERSE**, **COPY_INVERTED**, **NAND**, **SET**	
polygon_mode **face** **mode**	 **FRONT**, **BACK**, **FRONT_AND_BACK** **POINT**, **LINE**, **FILL**	No

Render states and their child elements	Valid values or types, and index attribute	In GLES?
shade_model	**FLAT**, **SMOOTH**	
stencil_func **func** **ref** **mask**	 **NEVER**, **LESS**, **LEQUAL**, **EQUAL**, **GREATER**, **NOTEQUAL**, **GEQUAL**, **ALWAYS** Unsigned byte Unsigned byte	
stencil_op **fail** **zfail** **zpass**	(For **fail**, **zfail**, and **zpass**) **KEEP**, **ZERO**, **REPLACE**, **INCR**, **DECR**, **INVERT**, **INCR_WRAP**, **DECR_WRAP**	
stencil_op_separate **face** **fail** **zfail** **zpass**	 **FRONT**, **BACK**, **FRONT_AND_BACK** (For **fail**, **zfail**, and **zpass:**) **KEEP**, **ZERO**, **REPLACE**, **INCR**, **DECR**, **INVERT**, **INCR_WRAP**, **DECT_WRAP**	No
stencil_func_separate **front** **back** **ref** **mask**	(For front and back) **NEVER**, **LESS**, **LEQUAL**, **EQUAL**, **GREATER**, **NOTEQUAL**, **GEQUAL**, **ALWAYS** Unsigned byte Unsigned byte	No
stencil_mask_separate **face** **mask**	 **FRONT**, **BACK**, **FRONT_AND_BACK** Unsigned byte	No
light_enable	Boolean Index attribute specifies which light. Required.	
light_ambient	**float4** Index attribute specifies which light. Required.	
light_diffuse	**float4** Index attribute specifies which light. Required.	
light_specular	**float4** Index attribute specifies which light. Required.	
light_position	**float4** Index attribute specifies which light. Required.	
light_constant_attenuation	**float** Index attribute specifies which light. Required.	
light_linear_attenuation	**float** Index attribute specifies which light. Required.	
light_quadratic_attenuation	**float** Index attribute specifies which light. Required.	

Render states and their child elements	Valid values or types, and index attribute	In GLES?
light_spot_cutoff	**float** Index attribute specifies which light. Required.	
light_spot_direction	**float3** Index attribute specifies which light. Required.	
light_spot_exponent	**float** Index attribute specifies which light. Required.	
texture1D	**sampler1D** type Index attribute specifies which texture unit. Required.	No
texture2D	**sampler2D** type Index attribute specifies which texture unit. Required.	No (see **<texture_pipeline>**)
texture3D	**sampler3D** type Index attribute specifies which texture unit. Required.	No
textureCUBE	**samplerCUBE** type Index attribute specifies which texture unit. Required.	No
textureRECT	**samplerRECT** type Index attribute specifies which texture unit. Required.	No
textureDEPTH	**samplerDEPTH** type Index attribute specifies which texture unit. Required.	No
texture1D_enable	Boolean Index attribute specifies which texture unit. Optional.	No
texture2D_enable	Boolean Index attribute specifies which texture unit. Optional.	No (see **<texture_pipeline>**)
texture3D_enable	Boolean Index attribute specifies which texture unit. Optional.	No
textureCUBE_enable	Boolean Index attribute specifies which texture unit. Optional.	No
textureRECT_enable	Boolean Index attribute specifies which texture unit. Optional.	No
textureDEPTH_enable	Boolean Index attribute specifies which texture unit. Optional.	No
texture_env_color	**float4** Index attribute specifies which texture unit. Optional.	No (see **<texture_pipeline>**)

Render states and their child elements	Valid values or types, and index attribute	In GLES?
texture_env_mode	**string** Index attribute specifies which texture unit. Optional.	No (see **<texture_pipeline>**)
clip_plane	**float4** Index attribute specifies which clip plane. Required.	**bool4**
clip_plane_enable	Boolean Index attribute specifies which clip plane. Optional.	
blend_color	**float4**	No
clear_color	**float4**	
clear_stencil	**int**	
clear_depth	**float**	
color_mask	**bool4**	
depth_bounds	**float2**	No
depth_mask	Boolean	
depth_range	**float2**	
fog_density	**float**	
fog_start	**float**	
fog_end	**float**	
fog_color	**float4**	
light_model_ambient	**float4**	
lighting_enable	Boolean	
line_stipple	**int2**	No
line_width	**float**	
material_ambient	**float4**	
material_diffuse	**float4**	
material_emission	**float4**	
material_shininess	**float**	
material_specular	**float4**	
model_view_matrix	**float4x4**	
point_distance_attenuation	**float3**	
point_fade_threshold_size	**float**	
point_size	**float**	
point_size_min	**float**	
point_size_max	**float**	
polygon_offset	**float2**	
projection_matrix	**float4x4**	
scissor	**int4**	
stencil_mask	**int**	
alpha_test_enable	Boolean	
auto_normal_enable	Boolean	No
blend_enable	Boolean	
color_logic_op_enable	Boolean	

Render states and their child elements	Valid values or types, and index attribute	In GLES?
color_material_enable Enables or disables the use of <color_material>. That is, indicates when runtimes should perform **glEnable** (**GL_COLOR_MATERIAL**) or **glDisable** (**GL_COLOR_MATERIAL**) (or equivalants).	Boolean	
cull_face_enable	Boolean	
depth_bounds_enable	Boolean	No
depth_clamp_enable	Boolean	No
depth_test_enable	Boolean	
dither_enable	Boolean	
fog_enable	Boolean	
light_model_local_viewer_enable	Boolean	No
light_model_two_side_enable	Boolean	
line_smooth_enable	Boolean	No
line_stipple_enable	Boolean	No
logic_op_enable	Boolean	No
multisample_enable	Boolean	
normalize_enable	Boolean	
point_smooth_enable	Boolean	No
polygon_offset_fill_enable	Boolean	
polygon_offset_line_enable	Boolean	No
polygon_offset_point_enable	Boolean	No
polygon_smooth_enable	Boolean	No
polygon_stipple_enable	Boolean	No
rescale_normal_enable	Boolean	
sample_alpha_to_coverage_enable	Boolean	
sample_alpha_to_one_enable	Boolean	
sample_coverage_enable	Boolean	
scissor_test_enable	Boolean	
stencil_test_enable	Boolean	
texture_pipeline	String – the name of the **<texture_pipeline>** parameter.	GLES only
texture_pipeline_enable	Boolean	GLES only

phong

Category: **Shaders**

Profile: **COMMON**

Introduction

Produces a specularly shaded surface where the specular reflection is shaded according the Phong BRDF approximation.

Concepts

Used inside a **<profile_COMMON>** effect, declares a fixed-function pipeline that produces a specularly shaded surface that reflects ambient, diffuse, and specular reflection, where the specular reflection is shaded according the Phong BRDF approximation.

The **<phong>** shader uses the common Phong shading equation, that is:

$$color = <emission> + <ambient> * al + <diffuse> * \max(N \bullet L, 0) + <specular> * \max(R \bullet I, 0)^{<shininess>}$$

where:

- al — A constant amount of ambient light contribution coming from the scene. In the COMMON profile, this is the sum of all the **<light><technique_common><ambient><color>** values in the **<visual_scene>**.
- N — Normal vector
- L — Light vector
- I — Eye vector
- R — Perfect reflection vector (reflect (L around N))

Attributes

The **<phong>** element has no attributes.

Related Elements

Parent elements	**technique** (FX) in **profile_COMMON**
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<emission>	Declares the amount of light emitted from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1
<ambient> (FX)	Declares the amount of ambient light emitted from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1

Name/example	Description	Default	Occurrences
<diffuse>	Declares the specularity or roughness of the specular reflection lobe. See **common_float_or_param_type**	N/A	0 or 1
<specular>	Declares the color of light specularly reflected from the surface of this object. See **common_color_or_texture_type**	N/A	0 or 1
<shininess>	Declares the specularity or roughness of the specular reflection lobe. See **common_float_or_param_type**	N/A	0 or 1
<reflective>	Declares the color of a perfect mirror reflection. See **common_color_or_texture_type**	N/A	0 or 1
<reflectivity>	Declares the amount of perfect mirror reflection to be added to the reflected light as a value between 0.0 and 1.0. See **common_float_or_param_type**	N/A	0 or 1
<transparent>	Declares the color of perfectly refracted light. See **common_color_or_texture_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<transparency>	Declares the amount of perfectly refracted light added to the reflected color as a scalar value between 0.0 and 1.0. See **common_float_or_param_type** and "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.	N/A	0 or 1
<index_of_refraction>	Declares the index of refraction for perfectly refracted light as a single scalar index. See **common_float_or_param_type**	N/A	0 or 1

Details

Example

This example has the following properties:

- It is an effect that takes its diffuse color as a parameter. Diffuse is defaulted to (0.2 0.56 0.35) but can be overridden in the material.
- It does not emit any light or absorb any indirect lighting (ambient).
- It has a little white shiny spot. 50 is a moderately high shininess power term, so the shiny spot should be fairly sharp.
- It is reflective and will reflect the environment at 5% intensity on top of the standard surface color calculations.
- It is not transparent. See "Determining Transparency (Opacity)" in Chapter 7: Getting Started with COLLADA FX.

```
<profile_COMMON>
   <newparam sid="myDiffuseColor">
      <float3> 0.2 0.56 0.35 </float3>
   </newparam>
   <technique sid="phong1">
      <phong>
```

```
            <emission><color>0.0 0.0 0.0 1.0</color></emission>
            <ambient><color>0.0 0.0 0.0 1.0</color></ambient>
            <diffuse><param ref="myDiffuseColor"/></diffuse>
            <specular><color>1.0 1.0 1.0 1.0</color></specular>
            <shininess><float>50.0</float></shininess>
            <reflective><color>1.0 1.0 1.0 1.0</color></reflective>
            <reflectivity><float>0.051</float></reflectivity>
            <transparent><color>0.0 0.0 0.0 1.0</color></transparent>
            <transparency><float>1.0</float></transparency>
        </phong>
    </technique>
</profile_COMMON>
```

profile_CG

Category: **Profiles**

Profile: **CG**

Introduction

Declares platform-specific data types and **<technique>**s for the Cg language.

Concepts

The **<profile_CG>** elements encapsulate all the platform-specific values and declarations for a particular profile. In **<effect>** scope, parameters are available to all platforms, but parameters declared inside a **<profile_CG>** block are available only to shaders that are also inside that profile.

The **<profile_CG>** element defines the clear interface between concrete, platform-specific data types and the abstract COLLADA data types used in the rest of the document. Parameters declared outside of this barrier may require casting when used inside a **<profile_CG>** block.

For more information, see “Using Profiles for Platform-Specific Effects” in Chapter 7: Getting Started with COLLADA FX.

Attributes

The **<profile_CG>** element has the following attribute:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
platform	**xs:NCName**	The type of platform. This is a vendor-defined character string that indicates the platform or capability target for the technique. The default is “PC”. Optional.

Related Elements

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

Parent elements	**effect**
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present, with the following exceptions:

- **<include>** and **<code>** are interchangeable
- **<newparam>**, **<setparam>**, and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<code>	See main entry.	N/A	0 or more
<include>	See main entry.	N/A	0 or more
<newparam>	See main entry.	N/A	0 or more
<image>	See main entry.	N/A	0 or more
<technique> (FX)	See main entry for attributes and description and the following subsection for child element details.	N/A	1 or more

Name/example	Description	Default	Occurrences
<extra>	See main entry.	N/A	0 or more

Child Elements for profile_CG / technique

Child elements must appear in the following order if present, with the following exceptions:

- **<include>** and **<code>** are interchangeable
- **<newparam>**, **<setparam>**, and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<annotate>	See main entry.	N/A	0 or more
<include>	See main entry.	N/A	0 or more
<code>	See main entry.	N/A	0 or more
<newparam>	See main entry.	N/A	0 or more
<setparam>	See main entry.	N/A	0 or more
<image>	See main entry.	N/A	0 or more
<pass>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

```
<profile_CG>
     <newparam sid="color">
       <float3> 0.5 0.5 0.5 </float3>
     </newparam>
     <newparam sid="lightpos">
       <semantic>LIGHTPOS0</semantic>
       <float3> 0.0 10.0 0.0 </float3>
     </newparam>
     <newparam sid="world">
       <semantic>WORLD</semantic>
       <float4x4> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </float4x4>
     </newparam>

     <newparam sid="worldIT">
       <semantic>WORLD_INVERSE_TRANSPOSE</semantic>
       <float4x4> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </float4x4>
    </newparam>

     <newparam sid="worldViewProj">
       <semantic>WORLD_VIEW_PROJECTION</semantic>
       <float4x4> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </float4x4>
     </newparam>

     <technique id="default" sid="default">
         <code>
    void VS (
         in varying float4 pos,
         in varying float3 norm,
         in uniform float3 light_pos,
```

```
      in uniform float4x4 w: WORLD,
      in uniform float4x4 wit: WORLD_INVERSE_TRANSPOSE,
      in uniform float4x4 wvp: WORLD_VIEW_PROJECTION,
      out varying float4 oPosition : POSITION,
      out varying float3 oNormal : TEXCOORD0,
      out varying float3 oToLight : TEXCOORD1 )
{   oPosition = mul(wvp, pos);
    oNormal =  mul(wit, float4(norm, 1)).xyz;
    oToLight = light_pos - mul(w, pos).xyz;
    return;
}

float3 diffuseFS (
       in uniform float3 flat_color,
       in varying float3 norm : TEXCOORD0,
       in varying float3 to_light : TEXCOORD1 ) : COLOR
     {  return flat_color * saturate(NdotL),
          0.0, 1.0);
     }
     </code>
     <pass sid="single_pass">
       <shader stage="VERTEX">
         <name source="diffuse-code-1">VS</name>
         <bind symbol="light_pos">
           <param ref="lightpos"/>
         </bind>
         <bind symbol="w">
           <param ref="world"/>
         </bind>
         <bind symbol="wit">
           <param ref="worldIT"/>
         </bind>
          <bind symbol="wvp">
            <param ref="worldViewProj"/>
          </bind>
       </shader>
       <shader stage="FRAGMENT">
         <name source="diffuse-code-1">diffuseFS</name>
         <bind symbol="flat_color">
           <param ref="color"/>
         </bind>
       </shader>
     </pass>
   </technique>
 </profile_CG>
```

profile_COMMON

Category: **Profiles**

Profile: **COMMON**

Introduction

Opens a block of platform-independent declarations for the common, fixed-function shader.

Concepts

The **<profile_COMMON>** elements encapsulate all the values and declarations for a platform-independent fixed-function shader. All platforms are required to support **<profile_COMMON>**. **<profile_COMMON>** effects are designed to be used as the reliable fallback when no other profile is recognized by the current effects runtime.

For more information, see “Using Profiles for Platform-Specific Effects” in Chapter 7: Getting Started with COLLADA FX.

Attributes

The **<profile_COMMON>** element has the following attribute:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.

Related Elements

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

Parent elements	**effect**
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present, with the following exception:

- **<newparam>** and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<image> Example: **<image id="myID"** **name="./BrickTexture"** **format="R8G8B8A8"** **height="64"** **width="128"** **depth-"1">**	Declares a standard COLLADA image resource. See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
<newparam> Example: **<newparam sid="mySID">** **<semantic>** **DIFFUSECOLOR** **</semantic>** **<float3>** **1 2 3** **</float3>** **</newparam>**	Creates a new parameter from a constrained set of types recognizable by all platforms – **<float>**, **<float2>**, **<float3>**, **<float4>**, **<surface>**, and **<sampler2D>**, with an additional semantic. See main entry.	N/A	0 or more
<technique> (FX)	Declares the only technique for this effect. See main entry for attributes and description and the following subsection for child element details.	N/A	1
<extra>	See main entry.	N/A	0 or more

Child Elements for profile_COMMON / technique

Child elements must appear in the following order if present, with the following exception:

- **<newparam>** and **<image>** are interchangeable.

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<newparam>	See main entry.	N/A	0 or more
<image>	See main entry.	N/A	0 or more
shader_element	One of **<constant>** (FX), **<lambert>**, **<phong>**, or **<blinn>**. See main entries.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

```
<profile_COMMON>
   <newparam sid="myDiffuseColor">
      <float3> 0.2 0.56 0.35 </float3>
    </newparam>
    <technique sid="phong1">
      <phong>
        <emission><color>1.0 0.0 0.0 1.0</color></emission>
        <ambient><color>1.0 0.0 0.0 1.0</color></ambient>
        <diffuse><param ref="myDiffuseColor"/></param></diffuse>
        <specular><color>1.0 0.0 0.0 1.0</color></specular>
        <shininess><float>50.0</float></shininess>
        <reflective><color>1.0 1.0 1.0 1.0</color></reflective>
        <reflectivity><float>0.5</float></reflectivity>
        <transparent><color>0.0 0.0 1.0 1.0</color></transparent>
        <transparency><float>1.0</float></transparency>
      </phong>
    </technique>
</profile_COMMON>
```

profile_GLES

Category: **Profiles**

Profile: **GLES**

Introduction

Declares platform-specific data types and **<technique>**s for OpenGL ES.

Concepts

The **<profile_GLES>** elements encapsulate all the platform-specific values and declarations for a particular profile. In **<effect>** scope, parameters are available to all platforms, but parameters declared inside a **<profile_GLES>** block are available only to shaders that are also inside that profile.

The **<profile_GLES>** element defines the clear interface between concrete, platform-specific data types and the abstract COLLADA data types used in the rest of the document. Parameters declared outside of this barrier may require casting when used inside a **<profile_GLES>** block.

For more information, see “Using Profiles for Platform-Specific Effects” in Chapter 7: Getting Started with COLLADA FX.

Attributes

<profile_GLES> has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
platform	**xs:NMTOKEN**	The type of platform. This is a vendor-defined character string that indicates the platform or capability target for the technique. Optional.

Related Elements

The **<profile_GLES>** elements relate to the following elements:

Parent elements	**effect**
Child elements	See the following subsections.
Other	None

<newparam> contains new objects for GLES. GLES-specific **VALUE_TYPE**s for **<newparam>** include **<texture_pipeline>** and **<sampler_state>**.

Child Elements

Child elements must appear in the following order if present, with the following exception:

- **<newparam>** and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<image> Example: **<image id="myID" name="./**	Declare a standard COLLADA image resource. See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
BrickTexture" format="R8G8B8A8" height="64" width="128" depth-"1">			
<newparam> Example: **<newparam sid="mySID"> <semantic> DIFFUSECOLOR </semantic> <float3> 1 2 3 </float3> </newparam>**	Create a new parameter from a constrained set of types recognizable by all platforms – **<float>**, **<float2>**, **<float3>**, **<float4>**, **<surface>** and **<sampler2D>**, with an additional semantic. See main entry.	N/A	0 or more
<technique> (FX)	Declares a technique for this effect. See main entry for attributes and description and the following subsection for child element details.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Child Elements for profile_GLES / technique

Child elements must appear in the following order if present, with the following exceptions:

- **<newparam>**, **<setparam>**, and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<annotate>	See main entry.	N/A	0 or more
<newparam>	See main entry.	N/A	0 or more
<setparam>	See main entry.	N/A	0 or more
<image>	See main entry.	N/A	0 or more
<pass>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

The following example shows terrain rendering to transitions between two different ground textures. It combines gravel texture and grass texture with an alpha transition texture that dictates the per-texel percentages of how they will blend.

```
<profile_GLES>
  <newparam sid="gravel">
    <texture_unit/>
  </newparam>
  <newparam sid="grass">
    <texture_unit/>
  </newparam>
```

```
  <newparam sid="transition">
    <texture_unit/>
  </newparam>
  <technique sid="main">
    <pass sid="p0">
      <texture_pipeline_enable value="true" />
      <texture_pipeline>
        <value>
          <texcombiner>
            <constant> 0.0f, 0.0f, 0.0f, 1.0f </constant>
            <RGB operator="INTERPOLATE">
               <argument source="TEXTURE" operand="SRC_RGB" unit="gravel"/>
              <argument source="TEXTURE" operand="SRC_RGB" unit="grass"/>
              <argument source="TEXTURE" operand="SRC_ALPHA" unit="transition"/>
            </RGB>
            <alpha operator="INTERPOLATE">
              <argument source="TEXTURE" operand="SRC_ALPHA" unit="gravel"/>
              <argument source="TEXTURE" operand="SRC_ALPHA" unit="grass"/>
              <argument source="TEXTURE" operand="SRC_ALPHA" unit="transition"/>
            </alpha>
          </texcombiner>
          <texcombiner>
            <RGB operator="MODULATE">
              <argument source="PRIMARY" operand="SRC_RGB"/>
              <argument source="PREVIOUS" operand="SRC_RGB"/>
            </RGB>
            <alpha operator="MODULATE">
              <argument source="PRIMARY" operand="SRC_ALPHA"/>
              <argument source="PREVIOUS" operand="SRC_ALPHA"/>
            </alpha>
          </texcombiner>
        </value>
      </texture_pipeline>
    </pass>
  </technique>
</profile_GLES>
```

profile_GLSL

Category: **Profiles**

Profile: **GLSL**

Introduction

Declares platform-specific data types and **<technique>**s for OpenGL Shading Language.

Concepts

The **<profile_GLSL>** elements encapsulate all the platform-specific values and declarations for a particular profile. In **<effect>** scope, parameters are available to all platforms, but parameters declared inside a **<profile_GLSL>** block are available only to shaders that are also inside that profile.

The **<profile_GLSL>** element defines the clear interface between concrete, platform-specific data types and the abstract COLLADA data types used in the rest of the document. Parameters declared outside of this barrier may require casting when used inside a **<profile_GLSL>** block.

For more information, see “Using Profiles for Platform-Specific Effects” in Chapter 7: Getting Started with COLLADA FX.

Attributes

<profile_GLSL> has the following attributes:

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.

Related Elements

The **<profile_GLSL>** elements relate to the following elements:

Parent elements	**effect**
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present, with the following exceptions:

- **<include>** and **<code>** are interchangeable
- **<newparam>** and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<asset>	See main entry.	N/A	0 or 1
<code>	See main entry.	N/A	0 or more
<include>	See main entry.	N/A	0 or more
<image> Example: **<image id="myID" name="./BrickTexture" format="R8G8B8A8"**	Declares a standard COLLADA image resource. See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
height="64" **width="128"** **depth-"1">**			
<newparam> Example: **<newparam** **sid="mySID">** **<semantic>** **DIFFUSECOLOR** **</semantic>** **<float3>** **1 2 3** **</float3>** **</newparam>**	Creates a new parameter from a constrained set of types recognizable by all platforms – **<float>**, **<float2>**, **<float3>**, **<float4>**, **<surface>** and **<sampler2D>**, with an additional semantic. See main entry.	N/A	0 or more
<technique> (FX)	Declares a technique for this effect. See main entry for attributes and description and the following subsection for child element details.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Child Elements for profile_GLSL / technique

Child elements must appear in the following order if present, with the following exceptions:

- **<include>** and **<code>** are interchangeable
- **<newparam>**, **<setparam>**, and **<image>** are interchangeable

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
<include>	See main entry.	N/A	0 or more
<code>	See main entry.	N/A	0 or more
<newparam>	See main entry.	N/A	0 or more
<setparam>	See main entry.	N/A	0 or more
<image>	See main entry.	N/A	0 or more
<pass>	See main entry.	N/A	1 or more
<extra>	See main entry.	N/A	0 or more

Details

Example

See Appendix B, “Example: <profile_GLSL>.”

render

Category: **Rendering**

Introduction

Describes one effect pass to evaluate a scene.

Concepts

Attributes

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

camera_node	**xs:anyURI**	Refers to a node that contains a camera describing the viewpoint from which to render this compositing step. Required.

Related Elements

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

Parent elements	**visual_scene**/evaluate_scene
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<layer>	Specifies which layer or layers to render in this compositing step while evaluating the scene. Contains layer names of type **xs:NCName**. This element has no attributes.	None	0 or more
<instance_effect>	Specifies which effect to render in this compositing step while evaluating the scene. See main entry.	N/A	0 or 1

Details

Example

See **<visual_scene>**.

RGB

Category: **Texturing**

Profile: **GLES**

Introduction

Defines the RGB portion of a **<texture_pipeline>** command. This is a combiner-mode texturing operation.

Concepts

See **<texcombiner>** for details about assignments and overall concepts.

Attributes

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

operator	**REPLACE | MODULATE | ADD | ADD_SIGNED | INTERPOLATE | SUBTRACT | DOT3_RGB | DOT3_RGBA**	Infers the use of **glTexEnv(TEXTURE_ENV, COMBINE_RGB, operator)**. See **<texcombiner>** for details.
scale	**float**	Infers the use of **glTexEnv(TEXTURE_ENV, RGB_SCALE, scale)**. See **<texcombiner>** for details.

Related Elements

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

Parent elements	**texcombiner**
Child elements	See the following subsections.
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<argument>	Sets up the arguments required for the given operator to be executed. See main entry.	None	1 to 3

Details

See **<texcombiner>** for details.

Example

See **<texture_pipeline>**.

sampler1D

Category: **Texturing**

Profile: **COMMON**, **CG**, **GLSL**

Introduction

Declares a one-dimensional texture sampler.

Concepts

Attributes

The **<sampler1D>** element has no attributes.

Related Elements

The **<sampler1D>** element relates to the following elements:

Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind** (shader)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source> (FX)	An **xs:NCName**, which is the sid of a **<surface>**. A **<sampler*>** is a definition of how a shader will resolve a color out of a **<surface>**. **<source>** identifies the **<surface>** to read.	None	1
<wrap_s>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1
<mipfilter>	MIPmap filter. Enumerated type **fx_sampler_filter_common**.	NONE	0 or 1
<border_color>	When reading past the edge of the texture address space based on the wrap modes involving clamps, this color takes over. Type **fx_color_common** (four floating-point numbers in RGBA order).	None	0 or 1

Name/example	Description	Default	Occurrences
<mipmap_maxlevel>	An **xs:unsignedByte**, which is the maximum number of progressive levels that the sampler will evaluate.	0	0 or 1
<mipmap_bias>	An **xs:float**, which biases the gamma (level of detail parameter) that is used by the sampler to evaluate the MIPmap chain.	0	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Example

This example repeats a texture across a surface regardless of any UVs exceeding the 0-to-1 range. It linearly magnifies the texture if it needs to be enlarged. It does trilinear filtering if the texels are smaller than the pixels being rasterized. This reads from a one-dimensional surface, that is, a surface that is N by 1 (height=1).

```
<sampler1D>
   <source>mySurface</source>
   <wrap_s>WRAP</wrap_s>
   <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
   <magfilter>LINEAR</magfilter>
</sampler1D>
```

sampler2D

Category: **Texturing**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLSL**

Introduction

Declares a two-dimensional texture sampler.

Concepts

Attributes

The **<sampler2D>** element has no attributes.

Related Elements

The **<sampler2D>** element relates to the following elements:

Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind** (shader)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source> (FX)	An **xs:NCName**, which is the sid of a **<surface>**. A **<sampler*>** is a definition of how a shader will resolve a color out of a **<surface>**. **<source>** identifies the **<surface>** to read.	None	1
<wrap_s>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_t>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1
<mipfilter>	MIPmap filter. Enumerated type **fx_sampler_filter_common**.	NONE	0 or 1
<border_color>	When reading past the edge of the texture address space based on the wrap modes involving clamps, this color takes over. Type **fx_color_common** (four floating-point numbers in RGBA order).	None	0 or 1

Name/example	Description	Default	Occurrences
<mipmap_maxlevel>	An **xs:unsignedByte**, which is the maximum number of progressive levels that the sampler will evaluate.	255	0 or 1
<mipmap_bias>	An **xs:float**, which biases the gamma (level of detail parameter) that is used by the sampler to evaluate the MIPmap chain.	0	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Example

This is an example of the most common sampler type. It repeats a texture across a surface regardless of any UVs exceeding the 0-to-1 range. It linearly magnifies the texture if it needs to be enlarged. It does trilinear filtering if the texels are smaller than the pixels being rasterized.

```
<sampler2D>
   <source>mySurface</source>
   <wrap_s>WRAP</wrap_s>
   <wrap_t>WRAP</wrap_t>
   <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
   <magfilter>LINEAR</magfilter>
</sampler2D>
```

sampler3D

Category: **Texturing**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLSL**

Introduction

Declares a three-dimensional texture sampler.

Concepts

Attributes

The **<sampler3D>** element has no attributes.

Related Elements

The **<sampler3D>** element relates to the following elements:

Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind** (shader)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source> (FX)	An **xs:NCName**, which is the sid of a **<surface>**. A **<sampler*>** is a definition of how a shader will resolve a color out of a **<surface>**. **<source>** identifies the **<surface>** to read.	None	1
<wrap_s>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_t>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_p>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1
<mipfilter>	MIPmap filter. Enumerated type **fx_sampler_filter_common**.	NONE	0 or 1

Name/example	Description	Default	Occurrences
<border_color>	When reading past the edge of the texture address space based on the wrap modes involving clamps, this color takes over. Type **fx_color_common** (four floating-point numbers in RGBA order).	None	0 or 1
<mipmap_maxlevel>	An **xs:unsignedByte**, which is the maximum number of progressive levels that the sampler will evaluate.	255	0 or 1
<mipmap_bias>	An **xs:float**, which biases the gamma (level of detail parameter) that is used by the sampler to evaluate the MIPmap chain.	0	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

For more details about all **<sampler*>** child elements, refer to the OpenGL specification.

Example

This example repeats a texture across a surface regardless of any UVs exceeding the 0-to-1 range. It linearly magnifies the texture if it needs to be enlarged. It does trilinear filtering if the texels are smaller than the pixels being rasterized.

This example does this typical sampling operation from a three-dimensional texture, that is, from a volume. This is common for reading from noise, patterns such as wood, and medical imaging.

```
<sampler3D>
   <source>mySurface</source>
   <wrap_s>WRAP</wrap_s>
   <wrap_t>WRAP</wrap_t>
   <wrap_p>WRAP</wrap_p>
   <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
   <magfilter>LINEAR</magfilter>
</sampler3D>
```

samplerCUBE

Category: **Texturing**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLSL**

Introduction

Declares a texture sampler for cube maps.

Concepts

Attributes

The **<samplerCUBE>** element has no attributes.

Related Elements

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

Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind** (shader)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source> (FX)	An **xs:NCName**, which is the sid of a **<surface>**. A **<sampler*>** is a definition of how a shader will resolve a color out of a **<surface>**. **<source>** identifies the **<surface>** to read.	None	1
<wrap_s>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_t>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_p>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1
<mipfilter>	MIPmap filter. Enumerated type **fx_sampler_filter_common**.	NONE	0 or 1

Name/example	Description	Default	Occurrences
<border_color>	When reading past the edge of the texture address space based on the wrap modes involving clamps, this color takes over. Type **fx_color_common** (four floating-point numbers in RGBA order).	None	0 or 1
<mipmap_maxlevel>	An **xs:unsignedByte**, which is the maximum number of progressive levels that the sampler will evaluate.	255	0 or 1
<mipmap_bias>	An **xs:float**, which biases the gamma (level of detail parameter) that is used by the sampler to evaluate the MIPmap chain.	0	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Example

This example reads from a cube map surface. The shader passes in a 3D vector that is a normal. That normal points to a location on one of the six sides of a cube map. Samples around the coordinate that it points to are filtered and returned.

```
<samplerCUBE>
   <source>mySurface</source>
   <wrap_s>WRAP</wrap_s>
   <wrap_t>WRAP</wrap_t>
   <wrap_p>WRAP</wrap_p>
   <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
   <magfilter>LINEAR</magfilter>
</samplerCUBE>
```

samplerDEPTH

Category: **Texturing**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLSL**

Introduction

Declares a texture sampler for depth maps.

Concepts

Attributes

The **<samplerDEPTH>** element has no attributes.

Related Elements

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

Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind** (shader)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source> (FX)	An **xs:NCName**, which is the sid of a **<surface>**. A **<sampler*>** is a definition of how a shader will resolve a color out of a **<surface>**. **<source>** identifies the **<surface>** to read.	None	1
<wrap_s>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_t>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Example

This example repeats a texture across a surface regardless of any UVs exceeding the 0-to-1 range. It linearly magnifies the texture if it needs to be enlarged. It does trilinear filtering if the texels are smaller than the pixels being rasterized. If the surface is depth data, it performs percentage closest filtering. This technique provides better results when sampling depth maps for uses such as shadow maps.

```
<samplerDEPTH>
   <source>mySurface</source>
   <wrap_s>WRAP</wrap_s>
   <wrap_t>WRAP</wrap_t>
   <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
   <magfilter>LINEAR</magfilter>
</samplerDEPTH>
```

samplerRECT

Category: **Texturing**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLSL**

Introduction

Declares a RECT texture sampler.

Concepts

RECT textures are an a OpenGL extension; they are not the same as nonsquare 2D textures. It is typically used as a render target or screen space processing, not as a general nonsquare replacement for **<sampler2D>**. For more information, see www.opengl.org/registry/specs/ARB/texture_rectangle.txt.

Attributes

The **<samplerRECT>** element has no attributes.

Related Elements

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

Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind** (shader)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<source> (FX)	An **xs:NCName**, which is the sid of a **<surface>**. A **<sampler*>** is a definition of how a shader will resolve a color out of a **<surface>**. **<source>** identifies the **<surface>** to read.	None	1
<wrap_s>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_t>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<wrap_p>	See **fx_sampler_wrap_common** Type.	WRAP	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1

Name/example	Description	Default	Occurrences
<mipfilter>	MIPmap filter. Enumerated type **fx_sampler_filter_common**.	NONE	0 or 1
<border_color>	When reading past the edge of the texture address space based on the wrap modes involving clamps, this color takes over. Type **fx_color_common** (four floating-point numbers in RGBA order).	None	0 or 1
<mipmap_maxlevel>	An **xs:unsignedByte**, which is the maximum number of progressive levels that the sampler will evaluate.	255	0 or 1
<mipmap_bias>	An **xs:float**, which biases the gamma (level of detail parameter) that is used by the sampler to evaluate the MIPmap chain.	0	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

RECT reflects OpenGL RECT samplers. It is not supported in DirectX. RECT is two dimensional. It does not support MIP-mapping. Samples use a **float2** that is in the range [0-to-width, 0-to-height] as opposed to the 2D 0-to-1 range.

Example

RECT samplers are very limited. They do not support MIP-mapping, so this trivial example is actually the most common usage:

```
<samplerRECT>
   <source>mySurface</source>
</samplerRECT>
```

sampler_state

Category: **Texturing**

Profile: **GLES**

Introduction

Provides a two-dimensional texture sampler state for **<profile_GLES>**.

Note: For **<texture_unit>** / **<sampler_state>**, see **<texture_unit>**.

Concepts

This is a bundle of sampler-specific states that will be referenced by one or more **<texture_pipeline>**s.

Attributes

The **<sampler_state>** element has the following attribute:

sid	**xs:NCName**	Identifier for this state. Optional.

Related Elements

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

Parent elements	**newparam**, **setparam**
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<wrap_s>	See **fx_sampler_wrap_common** Type.	REPEAT	0 or 1
<wrap_t>	See **fx_sampler_wrap_common** Type.	REPEAT	0 or 1
<minfilter>	Texture minimization. Enumerated type **fx_sampler_filter_common**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	NONE	0 or 1
<magfilter>	Texture magnification. Enumerated type **fx_sampler_filter_common**. When gamma indicates magnification, this value determines how the texture value is obtained.	NONE	0 or 1
<mipfilter>	MIPmap filter. Enumerated type **fx_sampler_filter_common**.	NONE	0 or 1
<mipmap_maxlevel>	An **xs:unsignedByte**, which is the maximum number of progressive levels that the sampler will evaluate.	255	0 or 1

Name/example	Description	Default	Occurrences
<mipmap_bias>	An **xs:float**, which biases the gamma (level of detail parameter) that is used by the sampler to evaluate the MIPmap chain.	0	0 or 1
<extra>	Contains additional application-specific information, such as OpenGL ES extensions. See main entry.	N/A	0 or more

Details

Example

semantic

Category: **Effects**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Provides metainformation that describes the purpose of a parameter declaration.

Concepts

Semantics describe the intention or purpose of a parameter declaration in an effect, using an overloaded concept. Semantics have been used historically to describe three different type of metainformation:

- A hardware resource allocated to a parameter, for example, **TEXCOORD2**, **NORMAL**.
- A value from the scene graph or graphics API that is being represented by this parameter, for example, **MODELVIEWMATRIX**, **CAMERAPOS**, **VIEWPORTSIZE**.
- A user-defined value that will be set by the application at run time when the effect is being initialized, for example, **DAMAGE_PERCENT**, **MAGIC_LEVEL**.

Semantics are used by the **<instance_geometry>** declaration inside **<node>** to bind effect parameters to values and data sources that can be found in the scene graph, using the **<bind_material>** mechanism used to disambiguate this mapping.

Attributes

The **<semantic>** element has no attributes.

Related Elements

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

Parent elements	**newparam**
Child elements	None
Other	None

Details

There is currently no standard set of semantics. This element can contain any **xs:NCName** defined by your application.

See “The Common Profile” in Chapter 3: Schema Concepts.

Example

```
<newparam sid="diffuseColor">
  <annotate name="UIWidget"><string>none</string></annotate>
  <semantic>DIFFUSE</semantic>
  <modifier>EXTERN</modifier>
  <float3> 0.30 0.56 0.12 </float>
</newparam>
```

setparam

Category: **Parameters**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Assigns a new value to a previously defined parameter.

Concepts

Parameters can be defined at run time as **<newparam>** or can be discovered as global parameters in source code or precompiled binaries at compile/link time. Each **<setparam>** is a speculative call, saying in effect:

- Search for a symbol called "X". If you find one in the current scope, attempt to assign a value of this data type to it. If you do not find the symbol or cannot assign the value, ignore and continue loading.

Not all instance of **<setparam>** are equal. **<setparam>** inside a specific **<profile_*>** has access to platform-specific data types and definitions, whereas a **<setparam>** inside an **<instance_material>** block can assign values only from the pool of common COLLADA data types.

<setparam> is one method for adding annotations to parameters that were previously unannotated. Under advanced language profiles, **<setparam>** can be used to assign concrete array sizes to previously unsized arrays using the **<array length="N"/>** element as well as connect instances of **<usertype>** parameters to abstract interface typed parameters.

Attributes

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

ref	**xs:token** (**xs:NCName** in **profile_GLES**/**technique**)	Attempts to reference the predefined parameter that will have its value set. Required.
program	**xs:NCName**	Optional in **<usertype>** and in **<technique>** for GLSL and CG profiles; not valid in GLES profile, **<generator>**, or **<instance_effect>**.

Related Elements

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

Parent elements	**technique** (FX) (in **profile_CG**, **profile_GLES**, **profile_GLSL**), **generator**, **instance_effect**, **usertype**
Child elements	See the following subsections.
Other	None

Child Elements in profile_GLSL/technique/setparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
param_value	Must contain exactly one of the following: • **glsl_value_type**: See "Value Types" at the end of the chapter for value types valid in GLSL scope. • **<array>**: See main entry.	N/A	1

Child Elements in profile_CG/technique/setparam and usertype/setparam

Note: Must contain exactly one of the following:

Name/example	Description	Default	Occurrences
cg_value_type	See "Value Types" at the end of the chapter for value types valid in CG scope.	N/A	See "Note"
<usertype>	See main entry.	N/A	See "Note"
<array>	See main entry.	N/A	See "Note"
<connect_param>	See main entry.	None	See "Note"

Child Elements in profile_GLES/technique/setparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
gles_value_type	See "Value Types" at the end of the chapter for value types valid in GLES scope.	N/A	1

Child Elements in <instance_effect>/setparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
core_value_type	See "Value Types" at the end of the chapter for valid core value types.	N/A	1

Child Elements in <generator>/setparam

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
value_type	See "Value Types" at the end of the chapter for valid value types; either GLSL or CG value types depending on context.	N/A	1

Details

FX Runtime loaders are free to report failed **<setparam>** attempts, but should not abort loading an effect on failure.

Example

```
<setparam ref="light_Direction">
   <annotate name="UIWidget"> <string>text</string> </annotate>
   <float3> 0.0 1.0 0.0 </float3>
</setparam>
```

shader

Category: **Shaders**

Profile: **CG**, **GLSL**

Introduction

Declares and prepares a shader for execution in the rendering pipeline of a **<pass>**.

Concepts

Executable shaders are small functions or programs that execute at a specific stage in the rendering pipeline. Shaders can be built from preloaded, precompiled binaries or dynamically generated at run time from embedded source code. The **<shader>** declaration holds all the settings necessary for compiling a shader and binding values or predefined parameters to the uniform inputs.

COLLADA FX allows declarations of both source code shaders and precompiled binaries, depending on support from the FX Runtime. Precompiled binary shaders already have the target profile specified for them at compile time, but to allow COLLADA readers to validate declarations involving precompiled shaders without having to load and parse the binary headers, profile declarations are still required.

Previously defined parameters, shader source, and binaries are considered merged into the same namespace / symbol table/source code string so that all symbols and functions are available to shader declarations, allowing common functions to be used in several shaders in a **<technique>**, for example, common lighting code. FX Runtimes that use the concept of “translation units” are allowed to name each source code block to break up the namespace.

Shaders with uniform input parameters can bind either previously defined parameters or literal values to these values during shader declaration, allowing compilers to inline literal and constant values.

Attributes

The **<shader>** element has the following attribute:

stage	Platform-specific enumeration	In which pipeline stage this programmable shader is designed to execute. Initial defined values in GLSL scope are **VERTEXPROGRAM** and **FRAGMENTPROGRAM**; in CG scope, they are **VERTEX** and **FRAGMENT**. Optional.

Related Elements

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

Parent elements	**pass** (in **profile_GLSL**/**technique** and **profile_CG**/**technique**)
Child elements	See the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>	See main entry.	N/A	0 or more
<compiler_target>	See main entry. If you specify **<compiler_options>**, you must also specify this element.	None	0 or 1
<compiler_options>	See main entry.	None	0 or 1

Name/example	Description	Default	Occurrences
<name>	See main entry.	None	1
<bind> (shader)	See main entry.	N/A	0 or more

Details

Example

```
<shader stage="VERTEX">
  <compiler_target>ARBVP1</compiler_target>
  <name source="ThinFilm2">main</entry>
  <bind symbol="lightpos">
    <param ref="LightPos_03"/>
  </bind>
</shader>
```

stencil_clear

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Specifies whether a render target surface is to be cleared, and which value to use.

Concepts

Before drawing, render target surfaces may need resetting to a blank canvas or default. These **<stencil_clear>** declarations specify which value to use. If no clearing statement is included, the target surface will be unchanged as rendering begins.

Attributes

The **<stencil_clear>** element has no attributes in GLES scope.

It has the following attribute in GLSL and CG scope:

index	**xs:nonNegativeInteger**	Which of the multiple render targets is being set. The default is 0. Optional.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	None

Details

This element contains an **xs:byte** that is the value used to clear a resource.

When this element exists inside a pass, it a cue to the runtime that a particular backbuffer or render-target resource should be cleared. This means that all existing image data in the resource should be replaced with the value provided. This puts the resource into a fresh and known state so that other operations with this resource execute as expected.

The index attribute identifies the resource that you want to clear. An index of 0 identifies the primary resource. The primary resource may be the backbuffer or the override provided with an appropriate **<*_target>** element (**<color_target>**, **<depth_target>**, or **<stencil_target>**).

Current platforms have fairly restrictive rules for setting up MRTs; for example, only four color buffers, which must be all of the same size and pixel format, and only one depth buffer and one stencil buffer active for all color buffers. The COLLADA FX declaration is designed to be looser in its restrictions, so an FX runtime must validate that a particular MRT declaration in a **<pass>** is possible before attempting to apply it, and flag it as an error if it fails.

Example

```
<stencil_clear index="0">0.0</stencil_clear>
```

stencil_target

Category: **Rendering**

Profile: **CG**, **GLES**, **GLSL**

Introduction

Specifies which **<surface>** will receive the stencil information from the output of this pass.

Concepts

Multiple Render Targets (MRTs) allow fragment shaders to output more than one value per pass, or to redirect the standard depth and stencil units to read from and write to arbitrary offscreen buffers. These elements tell the FX Runtime which previously defined surfaces to use.

Attributes

The **<stencil_target>** element has no attributes in GLES scope.

It has the following attributes in CG and GLSL scope:

index	**xs:nonNegativeInteger**	Indexes one of the Multiple Render Targets. Optional.
slice	**xs:nonNegativeInteger**	Indexes a subimage inside a target **<surface>**, including a single MIP-map level, a unique cube face, or a layer of a 3-D texture. Optional.
mip	**xs:nonNegativeInteger**	The default is 0. Optional.
face	Enumeration	Valid values are **POSITIVE_X**, **NEGATIVE_X**, **POSITIVE_Y**, **NEGATIVE_Y**, **POSITIVE_Z**, and **NEGATIVE_Z**. The default is **POSITIVE_X**. Optional.

Related Elements

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

Parent elements	**pass**
Child elements	None
Other	None

Details

Current platforms have fairly restrictive rules for setting up MRTs; for example, only four color buffers, which must be all of the same size and pixel format, and only one depth buffer and one stencil buffer active for all color buffers. The COLLADA FX declaration is designed to be looser in its restrictions, so an FX runtime must validate that a particular MRT declaration in a **<pass>** is possible before attempting to apply it, and flag it as an error if it fails.

If no **<stencil_target>** is specified, the FX Runtime will use the default stencil buffer set for its platform.

This element contains an **xs:NCName** which identifies the surface to receive the stencil output.

Example

```
<newparam sid="surfaceTex">
  <surface type="2D"/>
</newparam>
```

```
<pass>
  <stencil_target>surfaceTex</stencil_target>
</pass>
```

surface

Category: **Texturing**

Profile: **External**, **Effect**, **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Declares a resource that can be used both as the source for texture samples and as the target of a rendering pass.

Concepts

<surface> is an abstract generalization of **<image>** for GPU rendering that can link multiple **<image>** resources into a single object. For example, the object could be a MIP-mapped image with n prefiltered levels, or a cube map formed by joining six square textures.

<surface> objects:

- Have a data format describing the size and layout of fields in each pixel.
- Can be sized either in absolute numbers of pixels using **<size>** or as some fractional size of the viewport using **<viewport_ratio>**.
- Can declare a fixed number of MIP-map levels using **<mip_levels>**.

<surface> objects can be initialized from a set of preexisting **<image>** objects by providing an ordered list of their IDs to **<init_from>**. **<surface>** objects can also be initialized programmatically by evaluating source code over each pixel in the surface, using the **<generator>** element.

Attributes

The **<surface>** element has the following attribute:

type	**fx_surface_type_enum**	The type of this surface. Must be one of **UNTYPED**, **1D**, **2D**, **3D**, **CUBE**, **DEPTH**, **RECT**. Required in the COMMON, GLES, and GLSL scope; not valid in the CG scope (or in **<texture_unit>** within GLES).

Related Elements

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

Parent elements	COMMON: **newparam**, **setparam** CG: **newparam**, **setparam**, **array**, **bind** (shader), **usertype** GLES: **newparam**, **setparam**, **texture_unit** GLSL: **newparam**, **setparam**, **array**, **bind** (shader)
Child elements	None in GLES **<texture_unit>** Otherwise, see the following subsections.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
initialization option	An initialization option for this surface. Choose which is appropriate for your surface based on the type attribute and other characteristics described in the "Details" subsection.	N/A	0 or 1
<format>	Contains a string representing the texel format for this surface. If this element is not specified or understood by the application, then the application will attempt to use **<format_hint>** if it is provided; otherwise, it should use a common format linear R8G8B8A8. This element has no attributes.	None	0 or 1
<format_hint>	An application uses **<format_hint>** if **<format>** does not exist or is not understood by the application and **<format_hint>** exists. This element describes the important features intended by the author so that the application can pick a format that best represents what the author wanted. This element has no attributes. See the following subsection, "Child Elements in <surface>/<format_hint>".	N/A	0 or 1
<size>	Contains three integer values. If specified, the surface is sized to these exact dimensions in texels. Surfaces of type 1D and CUBE use only the first value. Surfaces of type 2D and RECT use only the first two values, representing width and then height. Type 3D uses all three values, representing width, height, and depth. This element has no attributes. Invalid if **<viewport_ratio>** is used.	0 0 0	0 or 1
<viewport_ratio>	Contains two floating-point values representing width and then height. If specified, the surface is sized to a dimension based on these ratios of the viewport's (backbuffer's) dimensions. For example, the following scales the surface's width to half the viewport's width and its height to twice the viewport's height: **<viewport_ratio>0.5 2</viewport_ratio>** This element is valid only for surfaces of type 2D or RECT. Invalid if **<size>** is used. This element has no attributes.	1 1	0 or 1
<mip_levels>	Contains the number of MIP levels in the surface. A value of 0 assumes that all MIP levels exist until a dimension becomes 1 texel. To create a surface that has only one level of MIP maps (mip=0), set this to 1. This element has no attributes.	0	0 or 1
<mipmap_generate>	Contains a Boolean. If false and not all subsurfaces are initialized because you have not provided MIP-map levels, the generated surface will have profile- and platform-specific behavior. If true, the application is responsible for initializing the remainder of the subsurfaces; this is typically done through a graphics API render state or function that does this automatically, such as `glGenerateMipmap()`. This element has no attributes.	False	0 or 1
<extra>	See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
<generator>	Valid in CG, GLES, and GLSL scopes. Specifies a procedural surface generator. See main entry.	N/A	0 or 1

Child Elements in <surface>/<format_hint>

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<channels>	Contains an enumeration value that describes the per-texel layout of the format. The length of the enumeration string indicates how many channels there are and each letter represents the name of a channel. There are typically 1 to 4 channels. This element has no attributes. Valid enumeration values are: • **RGB**: Red/Green/Blue color map. • **RGBA**: Red/Green/Blue/Alpha map, often used for color and transparency or other things packed into channel A, such as specular power. • **L**: Luminance map, often used for light mapping. • **LA**: Luminance/Alpha map, often used for light mapping. • **D**: Depth map, often used for displacement, parallax, relief, or shadow mapping. • **XYZ**: Typically used for normal maps or three-component displacement maps. • **XYZW**: Typically used for normal maps, where W is the depth for relief or parallax mapping.	N/A	1
<range>	Contains an enumeration that describes the range of texel channel values. Each channel represents a range of values. Some example ranges are signed or unsigned integers, or are within a clamped range such as 0.0f to 1.0f, or are a high dynamic range via floating point. This element has no attributes. Valid enumeration values are: • **SNORM**: Format represents a decimal value that remains within the -1 to 1 range. Implementation could be integer, fixed-point, or float. • **UNORM**: Format represents a decimal value that remains within the 0 to 1 range. Implementation could be integer, fixed-point, or float. • **SINT**: Format represents signed integer numbers; for example, 8 bits can represent -128 to 127. • **UINT**: Format represents unsigned integer numbers. For example, 8 bits can represent 0 to 255. • **FLOAT**: Format should support full floating-point ranges typically used for high dynamic range.	N/A	1

Name/example	Description	Default	Occurrences
<precision>	Contains an enumeration that identifies the precision of the texel channel value.Each channel of the texel has a precision. Typically, channels have the same precision. An exact format may lower the precision of an individual channel but applying a higher precision by linking the channels may still convey the same information. This element has no attributes. Valid enumeration values are: • **LOW**: For integers, this typically represents 8 bits. For floats, typically 16 bits. • **MID**: For integers, this typically represents 8 to 24 bits. For floats, typically 16 to 32 bits. • **HIGH**: For integers, this typically represents 16 to 32 bits. For floats, typically 24 to 32 bits.	N/A	0 or 1
<option>	Contains additional hints about data relationships and other things to help an application pick the best format. This element has no attributes. Valid enumeration values are: • **SRGB_GAMMA**: Colors are stored with respect to the sRGB 2.2 gamma curve rather than linear. • **NORMALIZED3**: The texel's XYZ/RGB should be normalized such as in a normal map. • **NORMALIZED4**: The texel's XYZW/RGBA should be normalized such as in a normal map. • **COMPRESSABLE**: The surface may use run-time compression. Consider the best compression based on desired **<channels>**, **<range>**, **<precision>**, and **<option>**s.	N/A	0 or more
<extra>	See main entry.	N/A	0 or more

Details

Determining Surface Dimensions

If neither the **<size>** nor the **<viewport_ratio>** element is present then the surface's dimensions are taken from its initialization images (**<init_*>**). The dimensions are taken from the images that are loaded into MIP level 0.

The **<viewport_ratio>** element is useful for screenspace effects and render targets. For example, it's useful for an effect that needs to copy to or from the screen with per-pixel accuracy, or for a downsampler for generating HDR blooms after a bright-pass.

Determining Surface Type

When a surface's type attribute is set to UNTYPED, its type is initially unknown and is established later by the context in which it is used, such as by a texture sampler that references it. A surface of any other type may be changed later into an UNTYPED surface at run time, as if it were created by **<newparam>**, using **<setparam>**. If there is a type mismatch between a **<setparam>** operation and what the runtime decides the type should be, the result is profile- and platform-specific behavior.

Determining Surface Format Using <format> and <format_hint>

DCC tools will likely write either nothing or only **<format_hint>**.

Game-engine tools will likely add **<format>** for design cases such as DirectX 4CC texture codes; that is, it is very specific to be exact, to minimize memory space, or to maximize performance or quality.

Applications creating the surface should use the following:

- If **<format>** exists and its string is understood, use that string.
- If **<format_hint>** exists, use features and characteristics described there to select an appropriate format.
- If initialized from **<image>**, use something compatible with the **<image>**s.
- Otherwise, if none of the preceding exist, the application should use a reasonable default format based on your platform and profile.

Initializing a Surface

The optional initialization option specifies whether to initialize the surface and how to do so.

Although the initialization option is optional in the schema, all surfaces should have one such element. An uninitialized surface should include the **<init_as_null>** element, while the **<init_as_target>** element indicates that that images will be rendered or copied into later by an effect or some other system.

The **<init_cube>**, **<init_volume>**, and **<init_planar>** elements enable the initialization of surface types with compound images such as DDS, animated GIF, or openEXR. In COLLADA, all 3D images are considered to be compound images, because the schema considers them to be an array of slices. Also in COLLADA, 1D images are also planar images, with y value of 1. For other images, **<init_from>** can be used.

To maximize compatibility with more applications, it is suggested, that developers support a single **<init_from>** with a DDS file as a complete initialization of the surface, because it could include the MIPmapping information.

The option can be any one of the following elements:

Name/example	Description
<init_as_null />	This surface is intended to be initialized later externally by a **<setparam>** element. If it is used before being initialized, there is profile- and platform-specific behavior. Most elements on the **<surface>** element that contains this will be ignored, including **<mip_levels>**, **<mipmap_generate>**, **<size>**, **<viewport_ratio>**, and **<format>**. This element has no attributes and contains no data.
<init_as_target />	Initializes this surface as a target for depth, stencil, or color. It does not need image data. If this element is used, **<mipmap_generate>** is ignored. This element has no attributes and contains no data.

<init_cube>
<all ref=… />
<primary ref= …>
<order />
</primary>
<face ref=… />
</init_cube>
	Initializes the entire surface with a CUBE from a compound image such as DDS.
	Contains one of:
	• **<all>**: Initializes the surface with one compound image such as DDS. This element contains no data; the ref attribute, referencing an image, is required.
	• **<primary>**: Initializes all primary MIP level 0 subsurfaces with one compound image such as DDS. Its ref attribute, referencing an image, is required. Use of this element expects the surface to have element **<mip_levels>=0** or **<mipmap_generate>**.
	Its subelement **<order>** occurs 0 or 6 times and has no attributes. Image formats that natively describe the face ordering, such as DDS, do not need this element. For other images, such as animated GIFs, this series of ordered elements describes which face the index belongs to. Each **<order>** contains one of the enumerated values of type **fx_surface_face_enum**. For example:
	`<primary ="file:///c:/foo.gif">`
	`<order>POSITIVE_X</order>`
	`<order>POSITIVE_Y</order>`

Name/example	Description
	<order>POSITIVE_Z</order> <order>NEGATIVE_X</order> <order>NEGATIVE_Y</order> <order>NEGATIVE_Z</order> • **</primary><face>**: Occurs 6 times. Initializes each face MIP chain with one compound image such as DDS. This element contains no data; the ref attribute, referencing an image, is required.
<init_volume> **<all ref=… />** **<primary ref=… />** **</init_volume>**	Initializes this surface with a 3D from a compound image such as DDS. Choose one of the following; for both, the ref attribute, referencing an image, is required: • **<all>**: Initializes the surface with one compound image such as DDS. • **<primary>**: Initializes MIP level 0 of the surface with one compound image such as DDS. Use of this element expects the surface to have element **<mip_levels>=0 or <mipmap_generate>**.
<init_planar> **<all ref=… />** **</init_planar>**	Initializes this surface with a 1D, 2D, RECT, or DEPTH from a compound image such as DDS. Must include: • **<all>**: Initializes the surface with one compound image such as DDS. This element contains no data; the ref attribute, referencing an image, is required.
<init_from **mip=…** **slice=…** **face=… />**	Contains a reference to a 1D or 2D image. Initializes the surface one subsurface at a time by specifying combinations of mip, face, and slice that make sense for a particular surface type. Each subsurface is initialized by a common 1-D or 2-D image, not a complex compound image such as DDS. If not all subsurfaces are initialized, the surface is invalid and will result in profile- and platform-specific behavior unless **<mipmap_generate>** is responsible for initializing the remaining subsurfaces. All attributes are optional: • **mip**: An **xs:unsignedInt** that specifies the MIP level. The default is 0. • **slice**: An **xs:unsignedInt** that specifies which 2D layer within a volume to initialize. There are anywhere from 0 to n slices in a volume, where n is the volume's depth slice. This attribute is used in combination with mip because a volume might have MIPmaps The default is 0. • **face**: An enumerated value of type **fx_surface_face_enum** that specifies which surface of a cube to initialize from the specified image. This attribute is used in combination with mip because a cubemap might have MIPmaps. The default is **POSITIVE_X**.

Example

```
<surface type="CUBE">
   <init_cube><all ref="sky_dds"></init_cube>
</surface>

<surface type="2D">
   <init_as_target/>
   <format>R5G6B5</format>
   <format_hint>
       <channel>RGB</channel>
       <range>UNORM</range>
       <precision>LOW</precision>
   </format_hint>
</surface>
```

technique

(FX)

Category: **Effects**

Profile: **CG**, **COMMON**, **GLES**, **GLSL**

Introduction

Holds a description of the textures, samplers, shaders, parameters, and passes necessary for rendering this effect using one method.

For **<technique>** in elements other than **<profile_*>**, see "**<technique>** (core)."

Concepts

Techniques hold all the necessary elements required to render an effect. Each effect can contain many techniques, each of which describes a different method for rendering that effect. There are three different scenarios for which techniques are commonly used:

- One technique might describe a high-LOD version while a second technique describes a low-LOD version of the same effect.
- Describe an effect in different ways and use validation tools in the FX Runtime to find the most efficient version of an effect for an unknown device that uses a standard API.
- Describe an effect under different game states, for example, a daytime and a nighttime technique, a normal technique, and a "magic-is-enabled" technique.

Attributes

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

id	**xs:ID**	A text string containing the unique identifier of the element. This value must be unique within the instance document. Optional.
sid	**xs:NCName**	A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element. Required.

Related Elements

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

Parent elements	**profile_CG**, **profile_COMMON**, **profile_GLSL**, **profile_GLES**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements vary by profile. See the parent element main entries for details. The following list summarizes valid child elements. The child elements must appear in the following order if present, with the following exceptions: **<code>** and **<include>**; **<newparam>**, **<setparam>**, and **<image>** are choices, can appear in any order in that position; **<blinn>**, **<constant>**, **<lambert>**, **<phong>** are choices, can appear in any order in that position:

Name	profile_CG	profile_COMMON	profile_GLES	profile_GLSL	Occurrences
<asset>	yes	yes	yes	-	0 or 1
<annotate>	yes	-	yes	yes	0 or more

Name	profile_CG	profile_COMMON	profile_GLES	profile_GLSL	Occurrences
<include>	yes	-	-	yes	0 or more
<code>	yes	-	-	yes	0 or more
<newparam>	yes	yes	yes	yes	0 or more
<setparam>	yes	-	yes	yes	0 or more
<image>	yes	yes	yes	yes	0 or more
<blinn>	-	yes	-	-	1
<constant> (FX)	-	yes	-	-	
<lambert>	-	yes	-	-	
<phong>	-	yes	-	-	
<pass>	yes	-	yes	yes	1 or more
<extra>	yes	yes	yes	yes	0 or more

Details

Techniques can be managed as first-class **<asset>**s, allowing tools to automatically generate techniques for effects and track their creation time, freshness, parent-child relationships, and the tools used to generate them.

Example

```
<effect id="BumpyDragonSkin">
  <profile_GLSL>
    <technique sid="HighLOD">
      ...
    </technique>
    <technique sid="LowLOD">
      ...
    </technique>
  </profile_GLSL>
</effect>
```

technique_hint

Category: **Effects**

Profile: **External**

Introduction

Adds a hint for a platform of which technique to use in this effect.

Concepts

Shader editors require information on which technique to use by default when an effect is instantiated. Subject to validation, the suggested technique should be used if your FX Runtime recognizes the platform string.

Attributes

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

platform	**xs:Name**	Defines a string that specifies for which platform this hint is intended. Optional.
ref	**xs:NCName**	A reference to the name of the platform. Required.
profile	**xs:NCName**	A string that specifies for which API profile this hint is intended. It is the name of the profile within the effect that contains the technique. Profiles are constructed by appending this attribute's value to "**profile_**". For example, to select **profile_CG**, specify **profile="CG"**. Optional.

Related Elements

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

Parent elements	**instance_effect**
Child elements	None
Other	None

Details

Example

```
<technique_hint platform="PS3" ref="HighLOD"/>
<technique_hint platform="OpenGL|ES" ref="twopass"/>
<technique_hint profile="CG" platform="GL" ref="HighLOD"/>
<technique_hint profile="GLES" platform="NOKIA_SW" ref="OneLight"/>
```

texcombiner

Category: **Texturing**

Profile: **GLES**

Introduction

Defines a **<texture_pipeline>** command for combiner-mode texturing.

Concepts

This element sets the combiner states for the texture unit to which it is assigned.

See **<texture_pipeline>** for details about assignments and overall concepts.

Attributes

The **<texcombiner>** element has no attributes.

Related Elements

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

Parent elements	**newparam**/**texture_pipeline**, **setparam**/**texture_pipeline**, **pass**/**texture_pipeline**/**value**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<constant **value= ...** **param= ... >** **(combiner)**	A static or parameter float4 that may be passed to the OpenGL ES texturing unit. This value is then combined with the color sampled from the texture to produce the final color output from that texturing unit. The equations are dependent on the texture pipeline setup. Contains no data. The arguments are optional; use only one: • **value**: Specifies a float4 for **glTexEnv(TEXTURE_ENV, TEXTURE_ENV_COLOR, value)**. • **param**: Specifies a parameter that contains a value for the **glTexEnv** function.	N/A	0 or 1
<RGB>	Sets up the RGB component of the texture combiner command. See main entry.	N/A	0 or 1
<alpha>	Sets up the alpha component of the texture combiner command. See main entry.	N/A	0 or 1

Details

This is a complex stage command in a **<texture_pipeline>**. It is used for more customized operations than available via **<texenv>**.

Read about **<texenv>** first, as the following information builds upon that basic knowledge.

The **<RGB>** and **<alpha>** children elements are roughly the same; **<alpha>** is simply a subset of **<RGB>**.

While **<texenv>** allows you to specify one operator equation that will be used for the entire state, **<texcombiner>** adds flexibility by allowing you to specify different equations for the **<RGB>** channel and **<alpha>** channel.

The equations specified consist of up to 3 arguments. The arguments are specified in a series (Arg0, Arg1, Arg2). Each channel may specify its own **<argument>**s to the equation. Each **<argument>** has a source, operand, and unit. The **<argument>** source attribute determines where the value for that equation's argument comes from:

- **TEXTURE**: From the texture_unit specified in the unit attribute.
- **CONSTANT**: The **<texcombiner>** schema also allows each stage to have its own **<constant>** that may be used by the operators.
- **PRIMARY**: The incoming fragment color from the material.
- **PREVIOUS**: The incoming color from the previous texture pipeline stage.

The **<argument>** operand attribute determines which part of the value selected by the source will be used in the equation:

- **SRC_COLOR**: The RGB portion of the source.
- **ONE_MINUS_SRC_COLOR**: the per-component inverse (one minus) of **SRC_COLOR**.
- **SRC_ALPHA**: the alpha portion of the source.
- **ONE_MINUS_SRC_ALPHA**: the inverse (one minus) of **SRC_ALPHA**.

The following operator equations are available for texcombiners:

- **REPLACE**: `Arg0`
- **MODULATE**: `Arg0 * Arg1`
- **ADD**: `Arg0 + Arg1`
- **ADD_SIGNED**: `Arg0 + Arg1- 0.5`
- **INTERPOLATE**: `Arg0 * Arg2 + Arg1 * (1 - Arg2)`
- **SUBTRACT**: `Arg0 - Arg1`
- **DOT3** (for **<RGB>** only): `4 × ((Arg0.r - 0.5) * (Arg1.r - 0.5) + (Arg0.g - 0.5) * (Arg1.g - 0.5) + (Arg0.b - 0.5) * (Arg1.b - 0.5))`

Lastly, each channel may be scaled. The RGB and alpha results of the equation are then multiplied by the scale attribute (if specified) to compute the final values per channel.

The RGB and alpha channels are then placed back together as a 4-component color as fed to the next stage as its **PREVIOUS** source.

For more information about any of these enumerations, refer to the OpenGL and OpenGL ES specifications.

Commands are eventually assigned to OpenGL ES hardware texture units. For this command type, each texture unit must be changed into texture-combiner mode with the following command:

```
glTexEnv(TEXTURE_ENV, TEXTURE_ENV_MODE, COMBINE)
```

See **<texture_pipeline>** for more details about OpenGL ES hardware texture-unit assignments.

Example

See **<texture_pipeline>**.

texenv

Category: **Texturing**

Profile: **GLES**

Introduction

Defines a **<texture_pipeline>** command for simple, noncombiner-mode texturing.

Concepts

This element sets the states for the texture unit to which it is assigned.

See **<texture_pipeline>** for details about assignments and overall concepts.

Attributes

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

operator	**REPLACE | MODULATE | DECAL | BLEND | ADD**	The operation to execute upon the incoming fragment. Optional.
unit	**xs:NCName**	The texturing unit. Optional.

Related Elements

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

Parent elements	**newparam** / **texture_pipeline**, **setparam** / **texture_pipeline**, **pass** / **texture_pipeline**/**value**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<constant **value= ...** **param= ... >** (combiner)	A static or parameter float4 that may be passed to the OpenGL ES texturing unit. This value is then combined with the color sampled from the texture to produce the final color output from that texturing unit. The equations are dependent on the texture pipeline setup. The element contains no data. The arguments are optional: value: Specifies a static float4 for **glTexEnv(TEXTURE_ENV, TEXTURE_ENV_COLOR, value)**. param: Specifies a parameter **float4**.	N/A	0 or 1

Details

This is a simple stage command in a **<texture_pipeline>**. It is used for very common operations with less setup.

Infers a call to **glTexEnv(TEXTURE_ENV, TEXTURE_ENV_MODE, operator**) for the texture unit to which it is assigned.

The equation used by this operation is specified via the operator attribute. The following equations are available:

- **REPLACE**: The output is the value sampled by the texture_unit specified in the unit attribute regardless of the alpha value.
- **MODULATE**: The output is the multiplication of the incoming value and the value sampled by the texture_unit specified in the unit attribute
- **DECAL**: The output is a blend (based on the alpha) of the color sampled from the texture_unit specified in the unit attribute and the input of the previous stage or material.
- **BLEND**: The output is a blend (based on each color component) of the color sampled from the texture_unit specified in the unit attribute and the input of the previous stage or material.
- **ADD**: The output is the addition of the input of the previous stage or material and the color sampled from the texture_unit specified in the unit attribute

For more information about any of these enumerations. refer to the OpenGL and OpenGL ES specifications.

Example

See **<texture_pipeline>**.

texture_pipeline

Category: **Texturing**

Profile: **GLES**

Introduction

Defines a set of texturing commands that will be converted into multitexturing operations using **glTexEnv** in regular and combiner mode.

Concepts

This element contains an ordered sequence of commands which together define all of the GLES multitexturing states.

Each command will eventually be assigned to a texture unit:

- The **<texcombiner>** defines a texture-unit setup in combiner mode.
- The **<texenv>** element defines a texture-unit setup in noncombiner mode.

Commands are assigned to texture units in a late binding step based on texture-unit names and usage characteristics of commands.

A pass will use the **<texture_pipeline>** and **<texture_pipeline_enable>** states to activate a fragment shader.

The ordering of the commands is 1:1 on which hardware texture unit they are assigned to. Depending on whether the texturing crossbar is supported (GLES 1.1), the named texture-unit objects (**<texture_unit>**) from each command are assigned into appropriate hardware texture units. On GLES 1.0, the texture must come from the existing unit, so two arguments with source="texture" would not be valid unless they referenced the same **<texture_unit>** element.

Attributes

The **<texture_pipeline>** element has the following attribute:

sid	**xs:NCName**	Optional under **newparam** and **setparam**. Not valid for **pass**.
param (FX)	**xs:NCName**	Optional under **pass**. Not valid for **newparam** and **setparam**.

Related Elements

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

Parent elements	In GLES scope: **newparam**, **setparam**, **pass** (render state)
Child elements	See the following subsection.
Other	None

Child Elements in newparam / texture_pipeline and setparam / texture_pipeline

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
<texcombiner>	See main entry.	N/A	0 or more

Name/example	Description	Default	Occurrences
<texenv>	See main entry.	N/A	0 or more
<extra>	Contains application-specific additional information, such as OpenGL ES extensions. See main entry.	N/A	0 or more

Child Elements in pass / texture_pipeline

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
<value sid=" "> **<texcombiner/>** **<texenv/>** **<extra/>**	Child elements are optional, and can appear in any order and as often as desired. See main entries for additional information. The sid attribute is optional.	N/A	0 or 1

Details

The **<texture_pipeline>** is used to create a sequence of commands that describe how the user would like to combine material color, textures, and destination (backbuffer) data. Each stage in the texture pipeline is a command. There are two available command types.

- **<texcombiner>**
- **<texenv>**

The purpose of each stage in the pipeline is to combine existing and/or new input data to produce a new output color. The output color from each stage is then available as input data to the next stage in the pipeline.

Please read about **<texenv>** and **<texcombiner>** before proceeding to understand how the **<texture_pipeline>** is converted to GL calls.

The stage commands in the API typically translate to **glTexEnv** function calls. The main difference between these APIs and what you have seen here is that that **glTexEnv** calls are paired with a particular texture unit. The design here has freed the operation from the texture unit so that the author can design the operations more easily and enable the importer or conditioner to assign appropriate indices in a resolve operation. The index of the stage in the texturing pipeline becomes the index of the texture unit where the **glTexEnv** calls are to be assigned. In Open GL ES 1.0, the texture referenced by the unit attribute must be placed into that same texture unit. In Open GL ES 1.1, the textures can be placed anywhere to utilize the crossbar, although pairing them like in 1.0 may be perform better on some hardware.

Note: some texture_pipelines may resolve directly under OpenGL ES 1.1 but not under OpenGL ES 1.0 due to support for texturing crossbars. Additionally. some may not resolve on certain hardware due to usage of too many textures.

Example

```
<texture_pipeline sid="terrain-transition-shader">
 <texcombiner>
  <constant> 0.0f, 0.0f, 0.0f, 1.0f </constant>
   <RGB operator="INTERPOLATE">
    <argument source="TEXTURE" operand="SRC_RGB" unit="gravel"/>
    <argument source="TEXTURE" operand="SRC_RGB" unit="grass"/>
    <argument source="TEXTURE" operand="SRC_ALPHA" unit="transition"/>
   </RGB>
   <alpha operator="INTERPOLATE">
    <argument source="TEXTURE" operand="SRC_ALPHA" unit="gravel"/>
    <argument source="TEXTURE" operand="SRC_ALPHA" unit="grass"/>
```

```
    <argument source="TEXTURE" operand="SRC_ALPHA" unit="transition"/>
   </alpha>
 </texcombiner>
 <texcombiner>
   <RGB operator="MODULATE">
    <argument source="PRIMARY" operand="SRC_RGB"/>
    <argument source="PREVIOUS" operand="SRC_RGB"/>
   </RGB>
   <alpha operator="MODULATE">
    <argument source="PRIMARY" operand="SRC_ALPHA"/>
    <argument source="PREVIOUS" operand="SRC_ALPHA"/>
   </alpha>
 </texcombiner>
 <texenv unit="debug-decal-unit" operator="DECAL"/>
</texture_pipeline>
```

texture_unit

Category: **Texturing**

Profile: **GLES**

Introduction

Defines a texture unit that will be mapped to hardware texture units based on its usage in **<texture_pipeline>** commands.

Concepts

A texture unit is a named representation of a collection of texturing-related data that must come together to use texturing in OpenGL ES 1.x. The GL API combines many items into the texture object that might otherwise be maintained separately in other APIs and profiles in COLLADA.

There may be defined more texture units than available hardware but, as long as a **<texture_pipeline>** uses fewer than the hardware limit at one time, the fragment shader is valid. (Conformance also depends on the version of GLES.)

Attributes

The **<texture_unit>** element has the following attribute:

sid	**xs:NCName**	Optional.

Related Elements

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

Parent elements	**setparam**, **newparam**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<surface> (in **texture_unit**)	Contains an **xs:NCName** that references the SID of a surface parameter whose surface image data will be used for textures, including MIPs and other complicated image representations. This element has no attributes.	None	0 or 1
<sampler_state> (in **texture_unit**)	Contains an **xs:NCName** that references the SID of the parameter containing the sampling state that is to be used when reading from the surface image data. This includes wrap modes and filters. This element has no attributes.	None	0 or 1

Name/example	Description	Default	Occurrences
<texcoord semantic=... >	Includes a semantic attribute that provides a semantic name for the texcoord channel that the texture unit must use to read from in the mesh. The channel (**array**) is mapped here using **<bind_material>**. In shader-based programming, **<texcoord>**s can be calculated in the shader, but for fixed-function APIs such as OpenGL ES 1.x, the texture coordinates must come parameterized with the mesh. This element contains no data.	None	0 or 1
<extra>	See main entry.	N/A	0 or more

Details

Inside the hardware the texture unit uses the **<texcoord>** address in combination with the sampling states to determine which samples it must read and combine from the surface when reading texture data is required. For more information about textures and sampling, refer to the OpenGL and OpenGL ES specifications.

Example

See **<texture_pipeline>**.

usertype

Category: **Parameters**

Profile: **CG**

Introduction

Creates an instance of a structured class for a parameter.

Concepts

Interface objects declare the abstract interface for a class of objects. Interface objects declare only the function signatures required and make no requirements for specific member data.

User types are concrete instances of these interfaces, structures that contain function declarations that provide implementations for each function declared in the interface along with any necessary member data.

User types can be declared only inside source code or included shaders, and so **<usertype>** declarations can take place only after all source code has been declared for a technique.

Attributes

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

name	**xs:token**	The identifier for the struct declaration that will be found inside the current source-code translation unit. Required.
source	**xs:NCName**	References a code or include element that defines the usertype. Required.

Related Elements

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

Parent elements	**newparam** (in **profile_CG**), **array**, **usertype**, **setparam**
Child elements	See the following subsection.
Other	None

Child Elements

Note: The **<usertype>** element does not require any child elements. However, if any child element occurs, you must choose either:

- One or more **<setparam>**
- One or more of any combination of the other elements

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
<array>	See main entry.	N/A	See “Note”
cg_value_type	See “Value Types” at the end of the chapter for value types valid in CG scope.	N/A	See “Note”
<connect_param>	See main entry.	N/A	See “Note”
<usertype>	As documented here.	N/A	See “Note”
<setparam>	Use a series of these to set the members by name. The ref attribute is relative to the usertype you are in right now. See main entry.	N/A	See “Note”

Details

Elements of a **<usertype>** can be initialized at creation time in **<newparam>** by traversing every leaf node in order and setting its value, or by accessing each leaf node by name using a series of **<setparam>** declarations.

Use a combination of **<array>**, value types, **<connect_param>**, and **<usertype>** to initialize the usertype in an order-dependent manner.

Some usertypes do not have data. They can be used only to implement interface functions.

Example

```
<include sid="simple_cg_source" url="simple.cgfx"/>

<newparam sid="lightsource0" source="simple_cg_source">
  <usertype name="spotlight">
    <float3> 10 12 10 </float3>
    <float3> 0.3 0.3 0.114 </float3>
  </usertype>
</newparam>

<newparam sid="lightsource1" source="simple_cg_source">
  <usertype name="spotlight">
    <setparam ref="position"><float3> 10 12 10 </float3></setparam>
    <setparam ref="direction"><float3> 0.3 0.3 0.114 </float3></setparam>
  </usertype>
</newparam>
```

Value Types

Introduction

Different FX profiles have different sets of strongly typed parameter types available for use.

For definitions of the types, see Chapter 9: COLLADA Types.

COLLADA Core Value Types (CORE_PARAM_TYPE)

bool, **bool2**, **bool3**, **bool4**, **int**, **int2**, **int3**, **int4**, **float**, **float2**, **float3**, **float4**, **float1x1**, **float1x2**, **float1x3**, **float1x4**, **float2x1**, **float2x2**, **float2x3**, **float2x4**, **float3x1**, **float3x2**, **float3x3**, **float3x4**, **float4x1**, **float4x2**, **float4x3**, **float4x4**, **surface**, **sampler1D**, **sampler2D**, **sampler3D**, **samplerCUBE**, **samplerRECT**, **samplerDEPTH**, **enum**

GLSL Value Types (GLSL_PARAM_TYPE)

bool, **bool2**, **bool3**, **bool4**, **int**, **int2**, **int3**, **int4**, **float**, **float2**, **float3**, **float4**, **float2x2**, **float3x3**, **float4x4**, **surface**, **sampler1D**, **sampler2D**, **sampler3D**, **samplerCUBE**, **samplerRECT**, **samplerDEPTH**, **enum**

CG Value Types (CG_PARAM_TYPE)

bool, **bool2**, **bool3**, **bool4**, **bool1x1**, **bool1x2**, **bool1x3**, **bool1x4**, **bool2x1**, **bool2x2**, **bool2x3**, **bool2x4**, **bool3x1**, **bool3x2**, **bool3x3**, **bool3x4**, **bool4x1**, **bool4x2**, **bool4x3**, **bool4x4**, **int**, **int2**, **int3**, **int4**, **int1x1**, **int1x2**, **int1x3**, **int1x4**, **int2x1**, **int2x2**, **int2x3**, **int2x4**, **int3x1**, **int3x2**, **int3x3**, **int3x4**, **int4x1**, **int4x2**, **int4x3**, **int4x4**, **float**, **float2**, **float3**, **float4**, **float1x1**, **float1x2**, **float1x3**, **float1x4**, **float2x1**, **float2x2**, **float2x3**, **float2x4**, **float3x1**, **float3x2**, **float3x3**, **float3x4**, **float4x1**, **float4x2**, **float4x3**, **float4x4**, **half**, **half2**, **half3**, **half4**, **half1x1**, **half1x2**, **half1x3**, **half1x4**, **half2x1**, **half2x2**, **half2x3**, **half2x4**, **half3x1**, **half3x2**, **half3x3**, **half3x4**, **half4x1**, **half4x2**, **half4x3**, **half4x4**, **fixed**, **fixed2**, **fixed3**, **fixed4**, **fixed1x1**, **fixed1x2**, **fixed1x3**, **fixed1x4**, **fixed2x1**, **fixed2x2**, **fixed2x3**, **fixed2x4**, **fixed3x1**, **fixed3x2**, **fixed3x3**, **fixed3x4**, **fixed4x1**, **fixed4x2**, **fixed4x3**, **fixed4x4**, **surface**, **sampler1D**, **sampler2D**, **sampler3D**, **samplerCUBE**, **samplerRECT**, **samplerDEPTH**, **enum**

GLES Value Types (GLES_PARAM_TYPE)

bool, **bool2**, **bool3**, **bool4**, **int**, **int2**, **int3**, **int4**, **float**, **float2**, **float3**, **float4**, **float1x1**, **float1x2**, **float1x3**, **float1x4**, **float2x1**, **float2x2**, **float2x3**, **float2x4**, **float3x1**, **float3x2**, **float3x3**, **float3x4**, **float4x1**, **float4x2**, **float4x3**, **float4x4**, **texture_unit**, **surface**, **sampler_state**, **texture_pipeline**, **enum**

Chapter 9: COLLADA Types

Introduction

Note: For a list of which types are valid in various FX profiles, see "Value Types" at the end of Chapter 8: COLLADA FX Reference.

The following table lists most simple types defined in the COLLADA schema. Types based on **xs:** refer to the XML schema (http://www.w3.org/TR/xmlschema-2/#built-in-datatypes) for their definitons and lexical representations.

Type	Description	Definition
bool	A Boolean value as defined in the XML schema.	**xs:restriction base="xs:boolean"**
bool2	Contains two Booleans	**xs:restriction base="ListOfBools"**
bool3	Contains three Booleans	**xs:restriction base="ListOfBools"**
bool4	Contains four Booleans	**xs:restriction base="ListOfBools"**
dateTime	Contains a combination of a date and time as defined in the XML schema.	**xs:restriction base="xs:dateTime"**
float	A floating-point number with optional exponent as defined in the XML schema.	**xs:restriction base="xs:double"**
float2	Contains two floating-point numbers	**xs:restriction base="ListOfFloats"**
float2x2	Contains four floating-point numbers representing a 2x2 matrix	**xs:restriction base="ListOfFloats"**
float2x3	Contains six floating-point numbers representing a 2x3 matrix	**xs:restriction base="ListOfFloats"**
float2x4	Contains eight floating-point numbers representing a 2x4 matrix	**xs:restriction base="ListOfFloats"**
float3	Contains three floating-point numbers	**xs:restriction base="ListOfFloats"**
float3x2	Contains six floating-point numbers representing a 3x2 matrix	**xs:restriction base="ListOfFloats"**
float3x3	Contains nine floating-point numbers representing a 3x3 matrix	**xs:restriction base="ListOfFloats"**
float3x4	Contains twelve floating-point numbers representing a 3x4 matrix	**xs:restriction base="ListOfFloats"**
float4	Contains four floating-point numbers	**xs:restriction base="ListOfFloats"**
float4x2	Contains eight floating-point numbers representing a 4x2 matrix	**xs:restriction base="ListOfFloats"**
float4x3	Contains twelve floating-point numbers representing a 4x3 matrix	**xs:restriction base="ListOfFloats"**

Type	Description	Definition
float4x4	Contains sixteen floating-point numbers representing a 4x4 matrix	**xs:restriction base="ListOfFloats"**
float7	Contains seven floating-point numbers	**xs:restriction base="ListOfFloats"**
int	Contains an integer as described in the XML schema.	**xs:restriction base="xs:long"**
int2	Contains two integers	**xs:restriction base="ListOfInts"**
int2x2	Contains four integers representing a 2x2 matrix	**xs:restriction base="ListOfInts"**
int3	Contains three integers	**xs:restriction base="ListOfInts"**
int3x3	Contains nine integers representing a 3x3 matrix	**xs:restriction base="ListOfInts"**
int4	Contains four integers	**xs:restriction base="ListOfInts"**
int4x4	Contains sixteen integers representing a 4x4 matrix	**xs:restriction base="ListOfInts"**
ListOfBools		**xs:list itemType="bool"**
ListOfFloats		**xs:list itemType="float"**
ListOfHexBinary		**xs:list itemType="xs:hexBinary"**
ListOfInts		**xs:list itemType="int"**
ListOfNames		**xs:list itemType="Name"**
ListOfTokens		**xs:list itemType="token"**
ListOfUInts		**xs:list itemType="uint"**
MorphMethodType	An enumuerated type specifying the acceptable morph methods. Valid values are: • **NORMALIZED** • **RELATIVE**	**xs:restriction base="xs:string"**
Name	A valid **Name** as described in the XML schema.	**xs:restriction base="xs:Name"**
NCName	A valid **NCName** as described in the XML schema.	
NodeType	An enumerated type specifying the acceptable node types. Valid values are: • **JOINT** • **NODE**	**xs:restriction base="xs:string"**
string		**xs:restriction base="xs:string"**
token		**xs:restriction base="xs:token"**
uint		**xs:restriction base="xs:unsignedLong"**

Type	Description	Definition
UpAxisType	An enumerated type specifying the acceptable up-axis values. Valid values are: • **Y_UP** • **Z_UP**	**xs:restriction base="xs:string"**
URIFragmentType	This type is used for URI reference which can only reference a resource declared within its same document. Valid values are: **xs:pattern value="(#(.*))"**	**xs:restriction base="xs:string"**
VersionType	An enumerated type specifying the acceptable COLLADA document versions. Valid values are: • **1.4.0** • **1.4.1**	**xs:restriction base="xs:string"**
fx_sampler_filter _common	An enumerated type. Valid values are: **NONE**, **NEAREST**, **LINEAR,** **NEAREST_MIPMAP_NEAREST**, **LINEAR_MIPMAP_NEAREST**, **NEAREST_MIPMAP_LINEAR**, **LINEAR_MIPMAP_LINEAR**. Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment.	
fx_sampler_wrap _common	Wrap modes that affect the interpretation of s, t, and p texture coordinates in the **<sampler_*>** elements. See the detailed type description later in this section.	
fx_surface_face_enum	• **POSITIVE_X** • **NEGATIVE_X** • **POSITIVE_Y** • **NEGATIVE_Y** • **POSITIVE_Z** • **NEGATIVE_Z**	**xs:restriction base="xs:string"**

Type	Description	Definition
fx_surface_type_enum	**UNTYPED**: When a surface's type attribute is set to **UNTYPED**, its type is initially unknown and established later by the context in which it is used, such as by a texture sampler that references it. A surface of any other type may be changed into an **UNTYPED** surface at run-time, as if it were created by <newparam>, using <setparam>. If there is a type mismatch between a <setparam> operation and what the run-time decides the type should be, the result is profile- and platform-specific behavior. • 1D • 2D • 3D • RECT • CUBE • DEPTH	xs:restriction base="xs:string"
fx_opaque_enum		xs:restriction base="xs:string"
fx_surface_format_hint_channels_enum		xs:restriction base="xs:string"
fx_surface_format_hint_precision_enum		xs:restriction base="xs:string"
fx_surface_format_hint_range_enum		xs:restriction base="xs:string"
fx_surface_format_hint_option_enum		xs:restriction base="xs:string"

fx_sampler_wrap_common Type

Wrap modes that affect the interpretation of s, t, and p texture coordinates in **<sampler_*>** elements.

The wrap mode enums map to the following OpenGL symbols.

Wrap Mode	OpenGL symbol	Description
WRAP	GL_REPEAT	Ignores the integer part of texture coordinates, using only the fractional part.
MIRROR	GL_MIRRORED_REPEAT	First mirrors the texture coordinate. The mirrored coordinate is then clamped as described for CLAMP_TO_EDGE.
CLAMP	GL_CLAMP_TO_EDGE	Clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. **Note:** GL_CLAMP takes any texels beyond the sampling border and substitutes those texels with the border color. So CLAMP_TO_EDGE is more appropriate. This also works much better with OpenGL ES where the GL_CLAMP symbol was removed from the OpenGL ES specification.

Wrap Mode	OpenGL symbol	Description
BORDER	**GL_CLAMP_TO_BORDER**	Clamps texture coordinates at all MIPmaps such that the texture filter always samples border texels for fragments whose corresponding texture coordinate is sufficiently far outside the range [0, 1].
NONE	**GL_CLAMP_TO BORDER**	The defined behavior for **NONE** is consistent with decal texturing where the border is black.Mapping this calculation to **GL_CLAMP_TO_BORDER** is the best approximation of this.

For more details about all **<sampler*>** child elements, refer to the OpenGL specification.

Appendix A: COLLADA Example

Example: Cube

This is a simple example of a COLLADA instance document that describes a simple white cube.

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema" version="1.4.1">
<asset>
    <created>2005-11-14T02:16:38Z</created>
    <modified>2005-11-15T11:36:38Z</modified>
    <revision>1.0</revision>
  </asset>
  <library_effects>
    <effect id="whitePhong">
      <profile_COMMON>
        <technique sid="phong1">
          <phong>
            <emission>
              <color>1.0 1.0 1.0 1.0</color>
            </emission>
            <ambient>
              <color>1.0 1.0 1.0 1.0</color>
            </ambient>
            <diffuse>
              <color>1.0 1.0 1.0 1.0</color>
            </diffuse>
            <specular>
              <color>1.0 1.0 1.0 1.0</color>
            </specular>
            <shininess>
              <float>20.0</float>
            </shininess>
            <reflective>
              <color>1.0 1.0 1.0 1.0</color>
            </reflective>
            <reflectivity>
              <float>0.5</float>
            </reflectivity>
            <transparent>
              <color>1.0 1.0 1.0 1.0</color>
            </transparent>
            <transparency>
              <float>1.0</float>
            </transparency>
          </phong>
        </technique>
      </profile_COMMON>
    </effect>
  </library_effects>
  <library_materials>
    <material id="whiteMaterial">
      <instance_effect url="#whitePhong"/>
    </material>
```

```
</library_materials>
<library_geometries>
  <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_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_common>
      </source>
      <source id="box-0-Normal">
        <float_array id="box-0-Normal-array" count="18">
         1.0  0.0  0.0
        -1.0  0.0  0.0
         0.0  1.0  0.0
         0.0 -1.0  0.0
         0.0  0.0  1.0
         0.0  0.0 -1.0
        </float_array>
        <technique_common>
          <accessor source="#box-0-Normal-array" count="6" stride="3">
            <param name="X" type="float"/>
            <param name="Y" type="float"/>
            <param name="Z" type="float"/>
          </accessor>
        </technique_common>
      </source>
      <vertices id="box-Vtx">
        <input semantic="POSITION" source="#box-Pos"/>
      </vertices>
      <polygons count="6" material="WHITE">
        <input semantic="VERTEX" source="#box-Vtx" offset="0"/>
        <input semantic="NORMAL" source="#box-0-Normal" offset="1"/>
        <p>0 4  2 4  3 4  1 4</p>
        <p>0 2  1 2  5 2  4 2</p>
        <p>6 3  7 3  3 3  2 3</p>
        <p>0 1  4 1  6 1  2 1</p>
        <p>3 0  7 0  5 0  1 0</p>
        <p>5 5  7 5  6 5  4 5</p>
      </polygons>
    </mesh>
  </geometry>
</library_geometries>
<library_visual_scenes>
  <visual_scene id="DefaultScene">
    <node id="Box" name="Box">
      <translate> 0  0  0</translate>
```

```
        <rotate> 0  0  1  0</rotate>
        <rotate> 0  1  0  0</rotate>
        <rotate> 1  0  0  0</rotate>
        <scale> 1  1  1</scale>
        <instance_geometry url="#box">
          <bind_material>
            <technique_common>
              <instance_material symbol="WHITE" target="#whiteMaterial"/>
            </technique_common>
          </bind_material>
        </instance_geometry>
      </node>
    </visual_scene>
  </library_visual_scenes>
  <scene>
    <instance_visual_scene url="#DefaultScene"/>
  </scene>
</COLLADA>
```

Appendix B: Profile GLSL Example

Example: <profile_GLSL>

This is a simple example of a COLLADA instance document that uses **<profile_GLSL>**.

```
<?xml version="1.0"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema" version="1.4.1">
   <asset>
          <contributor>
                 <author/>
                 <authoring_tool>RenderMonkey</authoring_tool>
                 <comments>Output from RenderMonkey COLLADA Exporter</comments>
                 <copyright/>
                 <source_data/>
          </contributor>
          <created>2007-12-11T14:24:00Z</created>
          <modified>2007-12-11T14:24:00Z</modified>
          <unit meter="0.01" name="centimeter"/>
          <up_axis>Y_UP</up_axis>
   </asset>
   <library_visual_scenes>
          <visual_scene id="VisualSceneNode" name="untitled">
                 <node id="Model_E0_MESH_0_REF_1" name="Model_E0_MESH_0_REF_1">
                        <instance_geometry url="#Model_E0_MESH_0_REF_1_lib">
                               <bind_material>
                                      <technique_common>
                                             <instance_material
                                               symbol="Textured_Bump_E0_MP_MAT"
                                               target="#Textured_Bump_E0_MP_MAT">
                                                  <bind_vertex_input
                                                    semantic="rm_Binormal"
                                                    input_semantic="BINORMAL"
                                                    input_set="2"/>
                                                  <bind_vertex_input
                                                    semantic="rm_Tangent"
                                                    input_semantic="TANGENT"
                                                    input_set="1"/>
                                             </instance_material>
                                      </technique_common>
                               </bind_material>
                        </instance_geometry>
                 </node>
          </visual_scene>
   </library_visual_scenes>
   <library_materials>
          <material id="Textured_Bump_E0_MP_MAT" name="Textured_Bump_E0_MP_MAT">
                 <instance_effect url="#Textured_Bump_E0_MP_FX">
                        <technique_hint platform="PC-OGL" profile="GLSL"
                               ref="Textured_Bump_E0_MP_TECH"/>
                        <setparam ref="fSpecularPower_E0_P0">
                               <float>25</float>
                        </setparam>
                        <setparam ref="fvAmbient_E0_P0">
```

```
                        <float4>0.368627 0.368421 0.368421 1</float4>
                    </setparam>
                    <setparam ref="fvDiffuse_E0_P0">
                        <float4>0.886275 0.885003 0.885003 1</float4>
                    </setparam>
                    <setparam ref="fvEyePosition_E0_P0">
                        <float3>0 0 100</float3>
                    </setparam>
                    <setparam ref="fvLightPosition_E0_P0">
                        <float3>-100 100 100</float3>
                    </setparam>
                    <setparam ref="fvSpecular_E0_P0">
                        <float4>0.490196 0.488722 0.488722 1</float4>
                    </setparam>
                    <setparam ref="baseMap_Sampler">
                        <sampler2D>
                            <source>baseMap_Surface</source>
                            <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
                            <magfilter>LINEAR</magfilter>
                        </sampler2D>
                    </setparam>
                    <setparam ref="baseMap_Surface">
                        <surface type="2D">
                            <init_from>base_E0</init_from>
                            <format>A8R8G8B8</format>
                        </surface>
                    </setparam>
                    <setparam ref="bumpMap_Sampler">
                        <sampler2D>
                            <source>bumpMap_Surface</source>
                            <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
                            <magfilter>LINEAR</magfilter>
                        </sampler2D>
                    </setparam>
                    <setparam ref="bumpMap_Surface">
                        <surface type="2D">
                            <init_from>bump_E0</init_from>
                            <format>A8R8G8B8</format>
                        </surface>
                    </setparam>
                </instance_effect>
            </material>
        </library_materials>
        <library_effects>
            <effect id="Textured_Bump_E0_MP_FX">
                <profile_COMMON>
                    <technique sid="phong">
                        <phong>
                            <emission>
                                <color>0 0 0 1</color>
                            </emission>
                            <ambient>
                                <color>0 0 0 1</color>
                            </ambient>
                            <diffuse>
                                <color>1 0.75 0.07 1</color>
                            </diffuse>
                            <specular>
                                <color>0.35 0.35 0.35 1</color>
                            </specular>
```

```
                                    <shininess>
                                          <float>10</float>
                                    </shininess>
                                    <reflective>
                                          <color>0 0 0 1</color>
                                    </reflective>
                                    <reflectivity>
                                          <float>0.5</float>
                                    </reflectivity>
                                    <transparent>
                                          <color>0 0 0 1</color>
                                    </transparent>
                                    <transparency>
                                          <float>1</float>
                                    </transparency>
                                    <index_of_refraction>
                                          <float>1</float>
                                    </index_of_refraction>
                              </phong>
                        </technique>
                  </profile_COMMON>
                  <profile_GLSL>
                        <code sid="Vertex_Program_E0_P0_VP">uniform vec3
fvLightPosition;
uniform vec3 fvEyePosition;

varying vec2 Texcoord;
varying vec3 ViewDirection;
varying vec3 LightDirection;

attribute vec3 rm_Binormal;
attribute vec3 rm_Tangent;

void main( void )
{
 gl_Position = ftransform();
 Texcoord = gl_MultiTexCoord0.xy;

 vec4 fvObjectPosition = gl_ModelViewMatrix * gl_Vertex;

 vec3 fvViewDirection = fvEyePosition - fvObjectPosition.xyz;
 vec3 fvLightDirection = fvLightPosition - fvObjectPosition.xyz;

 vec3 fvNormal = gl_NormalMatrix * gl_Normal;
 vec3 fvBinormal = gl_NormalMatrix * rm_Binormal;
 vec3 fvTangent = gl_NormalMatrix * rm_Tangent;

 ViewDirection.x = dot( fvTangent, fvViewDirection );
 ViewDirection.y = dot( fvBinormal, fvViewDirection );
 ViewDirection.z = dot( fvNormal, fvViewDirection );

 LightDirection.x = dot( fvTangent, fvLightDirection.xyz );
 LightDirection.y = dot( fvBinormal, fvLightDirection.xyz );
 LightDirection.z = dot( fvNormal, fvLightDirection.xyz );

}</code>
                        <code sid="Fragment_Program_E0_P0_FP">uniform vec4
fvAmbient;
uniform vec4 fvSpecular;
uniform vec4 fvDiffuse;
```

```
uniform float fSpecularPower;

uniform sampler2D baseMap;
uniform sampler2D bumpMap;

varying vec2 Texcoord;
varying vec3 ViewDirection;
varying vec3 LightDirection;

void main( void )
{
 vec3 fvLightDirection = normalize( LightDirection );
 vec3 fvNormal = normalize( ( texture2D( bumpMap, Texcoord ).xyz * 2.0 ) - 1.0
);
 float fNDotL = dot( fvNormal, fvLightDirection );

 vec3 fvReflection = normalize( ( ( 2.0 * fvNormal ) * fNDotL ) -
fvLightDirection );
 vec3 fvViewDirection = normalize( ViewDirection );
 float fRDotV = max( 0.0, dot( fvReflection, fvViewDirection ) );

 vec4 fvBaseColor = texture2D( baseMap, Texcoord );

 vec4 fvTotalAmbient = fvAmbient * fvBaseColor;
 vec4 fvTotalDiffuse = fvDiffuse * fNDotL * fvBaseColor;
 vec4 fvTotalSpecular = fvSpecular * ( pow( fRDotV, fSpecularPower ) );

 gl_FragColor = ( fvTotalAmbient + fvTotalDiffuse + fvTotalSpecular );

}</code>
                        <newparam sid="fSpecularPower_E0_P0">
                               <float>25</float>
                        </newparam>
                        <newparam sid="fvAmbient_E0_P0">
                               <float4>0.368627 0.368421 0.368421 1</float4>
                        </newparam>
                        <newparam sid="fvDiffuse_E0_P0">
                               <float4>0.886275 0.885003 0.885003 1</float4>
                        </newparam>
                        <newparam sid="fvEyePosition_E0_P0">
                               <float3>0 0 100</float3>
                        </newparam>
                        <newparam sid="fvLightPosition_E0_P0">
                               <float3>-100 100 100</float3>
                        </newparam>
                        <newparam sid="fvSpecular_E0_P0">
                               <float4>0.490196 0.488722 0.488722 1</float4>
                        </newparam>
                        <newparam sid="baseMap_Sampler">
                               <sampler2D>
                                      <source>baseMap_Surface</source>
                                      <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
                                      <magfilter>LINEAR</magfilter>
                               </sampler2D>
                        </newparam>
                        <newparam sid="baseMap_Surface">
                               <surface type="2D">
                                      <init_from>base_E0</init_from>
                                      <format>A8R8G8B8</format>
                               </surface>
```

```
</newparam>
<newparam sid="bumpMap_Sampler">
       <sampler2D>
              <source>bumpMap_Surface</source>
              <minfilter>LINEAR_MIPMAP_LINEAR</minfilter>
              <magfilter>LINEAR</magfilter>
       </sampler2D>
</newparam>
<newparam sid="bumpMap_Surface">
       <surface type="2D">
              <init_from>bump_E0</init_from>
              <format>A8R8G8B8</format>
       </surface>
</newparam>
<technique sid="Textured_Bump_E0_MP_TECH">
       <pass sid="Pass_0">
              <shader stage="VERTEXPROGRAM">
                     <compiler_target>110
                       </compiler_target>
                     <name
                       source="Vertex_Program_E0_P0_VP">
                       main</name>
                     <bind symbol="fvEyePosition">
                            <param
                              ref="fvEyePosition_E0_P0"/>
                     </bind>
                     <bind symbol="fvLightPosition">
                            <param
                             ref="fvLightPosition_E0_P0"/>
                     </bind>
              </shader>
              <shader stage="FRAGMENTPROGRAM">
                     <compiler_target>110
                       </compiler_target>
                     <name
                       source="Fragment_Program_E0_P0_FP">
                       main</name>
                     <bind symbol="fSpecularPower">
                            <param
                              ref="fSpecularPower_E0_P0"/>
                     </bind>
                     <bind symbol="fvAmbient">
                            <param ref="fvAmbient_E0_P0"/>
                     </bind>
                     <bind symbol="fvDiffuse">
                            <param ref="fvDiffuse_E0_P0"/>
                     </bind>
                     <bind symbol="fvSpecular">
                            <param
                              ref="fvSpecular_E0_P0"/>
                     </bind>
                     <bind symbol="baseMap">
                            <param ref="baseMap_Sampler"/>
                     </bind>
                     <bind symbol="bumpMap">
                            <param ref="bumpMap_Sampler"/>
                     </bind>
              </shader>
       </pass>
</technique>
```

```
                </profile_GLSL>
                <extra>
                      <technique profile="RenderMonkey">
                            <RenderMonkey_TimeCycle>
                                  <param type="float">120.000000</param>
                            </RenderMonkey_TimeCycle>
                      </technique>
                </extra>
          </effect>
    </library_effects>
    <library_images>
          <image id="base_E0" name="base_E0">
                <init_from>./Textured_Bump/Fieldstone.tga</init_from>
          </image>
          <image id="bump_E0" name="bump_E0">
                <init_from>./Textured_Bump/FieldstoneBumpDOT3.tga</init_from>
          </image>
    </library_images>
    <library_geometries>
          <geometry id="Model_E0_MESH_0_REF_1_lib" name="Model_E0_MESH_0_REF_1">
                <mesh>
                      <source id="Model_E0_MESH_0_REF_1_lib_positions"
                        name="position">
                            <float_array
                              id="Model_E0_MESH_0_REF_1_lib_positions_array"
                              count="9">
                                  -50 -50 0 50 -50 0 0 50 0
                            </float_array>
                            <technique_common>
                                  <accessor count="3"
                            source="#Model_E0_MESH_0_REF_1_lib_positions_array"
                                    stride="3">
                                        <param name="X" type="float"/>
                                        <param name="Y" type="float"/>
                                        <param name="Z" type="float"/>
                                  </accessor>
                            </technique_common>
                      </source>
                      <source id="Model_E0_MESH_0_REF_1_lib_normals"
                        name="normal">
                            <float_array
                              id="Model_E0_MESH_0_REF_1_lib_normals_array"
                              count="9">
                                  0 0 -1 0 0 -1 0 0 -1
                            </float_array>
                            <technique_common>
                                  <accessor count="3"
                            source="#Model_E0_MESH_0_REF_1_lib_normals_array"
                                    stride="3">
                                        <param name="X" type="float"/>
                                        <param name="Y" type="float"/>
                                        <param name="Z" type="float"/>
                                  </accessor>
                            </technique_common>
                      </source>
                      <source id="Model_E0_MESH_0_REF_1_lib_texcoords"
                        name="texcoords">
                            <float_array
                              id="Model_E0_MESH_0_REF_1_lib_texcoords_array"
                              count="6">0 0 1 0 0.5 1</float_array>
```

```
                        <technique_common>
                              <accessor count="3"
                        source="#Model_E0_MESH_0_REF_1_lib_texcoords_array"
                                    stride="2">
                                          <param name="X" type="float"/>
                                          <param name="Y" type="float"/>
                              </accessor>
                        </technique_common>
                  </source>
                  <source id="Model_E0_MESH_0_REF_1_lib_tangents"
                    name="tangent">
                        <float_array
                          id="Model_E0_MESH_0_REF_1_lib_tangents_array"
                          count="9">1 0 0 1 0 0 1 0 0</float_array>
                        <technique_common>
                              <accessor count="3"
                        source="#Model_E0_MESH_0_REF_1_lib_tangents_array"
                                    stride="3">
                                          <param name="X" type="float"/>
                                          <param name="Y" type="float"/>
                                          <param name="Z" type="float"/>
                              </accessor>
                        </technique_common>
                  </source>
                  <source id="Model_E0_MESH_0_REF_1_lib_binormals"
                    name="binormal">
                        <float_array
                          id="Model_E0_MESH_0_REF_1_lib_binormals_array"
                          count="9">0 1 0 0 0 0 0 1 0</float_array>
                        <technique_common>
                              <accessor count="3"
                        source="#Model_E0_MESH_0_REF_1_lib_binormals_array"
                                    stride="3">
                                          <param name="X" type="float"/>
                                          <param name="Y" type="float"/>
                                          <param name="Z" type="float"/>
                              </accessor>
                        </technique_common>
                  </source>
                  <vertices id="Model_E0_MESH_0_REF_1_lib_vertices">
                        <input semantic="POSITION"
                          source="#Model_E0_MESH_0_REF_1_lib_positions"/>
                        <input semantic="NORMAL"
                          source="#Model_E0_MESH_0_REF_1_lib_normals"/>
                        <input semantic="TEXCOORD"
                          source="#Model_E0_MESH_0_REF_1_lib_texcoords"/>
                  </vertices>
                  <triangles count="1" material="Textured_Bump_E0_MP_MAT">
                        <input offset="0" semantic="VERTEX"
                          source="#Model_E0_MESH_0_REF_1_lib_vertices"/>
                        <input offset="0" semantic="TANGENT"
                          source="#Model_E0_MESH_0_REF_1_lib_tangents"/>
                        <input offset="0" semantic="BINORMAL"
                          source="#Model_E0_MESH_0_REF_1_lib_binormals"/>
                        <p>0 1 2</p>
                  </triangles>
            </mesh>
      </geometry>
</library_geometries>
<scene>
```

```
            <instance_visual_scene url="#VisualSceneNode"/>
        </scene>
    </COLLADA>
```

Glossary

animation curve – A 2D function defined by a set of *key frames* and the interpolation among them.

arc – A connection between *nodes*.

backbuffer – The viewport buffer into which the computer normally renders in a double-buffered system.

attribute – An *XML* element can have zero or more attributes. Attributes are given within the start *tag* and follow the tag *name*. Each attribute is a name-*value* pair. The value portion of an attribute is always surrounded by quotation marks (" "). Attributes provide semantic information about the *element* on which they are bound. For example:

```
<tagName attribute="value">
```

COLLADA – *Colla*borative *D*esign *A*ctivity.

COLLADA document – A file containing COLLADA *XML* elements that describe certain digit assets.

COLLADA schema – An *XML* schema document that defines all valid COLLADA elements.

comment – *XML* files can contain comment text. Comments are identified by special markup of the following form:

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

CV – Control vertex. A control point on a spline curve.

DCC – Digital content creation.

effect scope – The declaration space that is inside an **<effect>** element but not within any specific **<profile_*>** element.

element – 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.

function curve – Same as *animation curve*.

frustum – see *viewing frustum*.

FX runtime – The assumed underlying library of code that handles the creation, use, and management of shaders, source code, parameters, and other effects properties.

HDR – High dynamic range.

id – An element's identifier, which can be referenced as part of a URI and which is unique within an *instance document*. See "Address Syntax" in Chapter 3: Schema Concepts.

IDREF – A reference to an *id*. See "Address Syntax" in Chapter 3: Schema Concepts.

instance – An occurrence of an object, the result of instantiating a copy or version of the object.

instance document – A *COLLADA document*.

instantiation – The creation of a copy (instance) of an object.

key frame – The beginning or ending point of an animated object. Consists of a 2D data sampling, consisting of the "input" (usually a point in time) and the "output" (the value being animated).

MIP map – An optimized collection of bitmap images for a texture.

morph target – A mesh that can be blended with other meshes.

multiple render targets (MRT) – Rendering to multiple drawing buffers simultaneously.

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

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

This shows an element named Array with two attributes, size and type. The size attribute specifies how large the array is and the type attribute specifies that the array contains floating-point data.

node – Points of information within a *scene graph*. COLLADA uses node to refer to interior (branch) nodes rather than to exterior (leaf) nodes.

path – see *arc*.

profile – A structure in which to gather effects information for a specific platform or environment.

scene graph – The hierarchical structure of a scene, represented in COLLADA by the **<scene>** element's content. Specifically, a directed acyclic graph (DAG) or tree data structure that contains nodes of visual information and related data.

shorthand pointer – The value of the *id* attribute of an element in an instance document. This is a URI fragment identifier that conforms to XPointer syntax.

sid – An element's scoped identifier; similar to an *id* except that it is unique only within a certain scope, not necessarily in an entire *instance document*. See "Address Syntax" in Chapter 3: Schema Concepts.

tag – 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.

tone mapping – The combination of spectral sampling and dynamic range remapping, performed as the last step of image synthesis (rendering).

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 rules given in the schema document, an XML parser can validate the instance document's syntax and semantics. This process is called validation.

value – In XML, the value of an *attribute* is always textual data during parsing.

viewing frustum – The region of space that appears in a camera's view.

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.

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-based formats.

General Index

NOTE: This index includes concepts, terms, and values. For a list of COLLADA elements, see the separate "Index of COLLADA Elements."

Index of COLLADA Elements

Note: This index lists the main definition entry for each element, not use within other elements.