COLLADA – Digital Asset Schema Release 1.4.1

Specification

June 2006

Editor: Mark Barnes, Sony Computer Entertainment Inc.

Table of Contents

About This Document

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.

The file extension chosen for documents that use the COLLADA schema is "`.dae`" (an acronym for Digital Asset Exchange). When an Internet search was completed, no preexisting usage was found.

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®.

Organization of this Document

This document consists of the following chapters:

Chapter/Section	Description
Ch. 1: Design Considerations	Issues concerning the COLLADA design
Ch. 2: Tool Requirements and Options	COLLADA tool requirements for implementors
Ch. 3: Schema and Reference Overview	A general description of the schema, its design, and how it is presented in the reference chapters. Includes a description of the COLLADA COMMON profile.
Ch. 4: COLLADA Core Elements Reference	Detailed reference description of the core elements in the COLLADA schema
Ch. 5: COLLADA Physics Reference	Detailed reference description of COLLADA Physics elements
Ch. 6: COLLADA FX Reference	Detailed reference description of COLLADA FX elements
Appendix A: COLLADA Example	An example COLLADA instance document
Glossary	Definitions of terms used in this document, including XML terminology
Index	

Other Sources of Information

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

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

For more information on COLLADA, visit www.khronos.org/collada.

Typographic Conventions

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

Conventions	Description
Regular text	Descriptive text
Courier-type font	References to class, method, and variable names
Courier bold	File names
> Courier bold	Commands, which are preceded with a right angle bracket (>)
blue	Hyperlinks

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).
- End users will use the Microsoft Windows® and Linux® operating systems in their production environments. In the case of end-user developers, we assume that they will program in the C/C++ languages, predominantly. Therefore, COLLADA example source code uses primarily these languages.

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.

  COLLADA currently supports polygon-based models in a common format. As we discuss new issues, COLLADA will attempt to cover these as well, such as shader effects, physics, and parametric surfaces.

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.

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

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

Materials and Textures

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 (e.g., 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

Data	Must be possible to export
Variable sampling rate	Using a variable sampling rate for animations. This allows a user to specify different sampling rates for different portions of the animation to be exported
Bones	Boned animations
Skeletal animation	Skeletal animations
Skeletal animation with smooth binding	Skeletal animations with smooth binding
Animation of light parameters	Animated light parameters
Camera animation	Animated cameras
Key-frame animation of transforms	Animated transforms with key frames
Animation function curves	Animation function curves

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 and Reference Overview

Introduction

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

Concepts

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:

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

The *id* attributes are addressed using the URI fragment identifier notation. The XML specification defines the syntax for a URI fragment identifier within an XML document. The URI fragment identifier must conform to the XPointer syntax. As COLLADA 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" />
```

The *target* attribute syntax has several parts:

- The first part is the ID of an element in the instance document or a dot segment (".") indicating that this is a relative address.

- One or more subidentifiers follow. Each is preceded by a literal slash (/) as a path separator. The subidentifiers are taken from a child of the element identified by the first part. For nested elements, multiple subidentifiers 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:

  o 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.

  o 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. Actually, some of these 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.

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 needs to provide a definition for the common profile as the schema evolves.

---

[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.

Naming Conventions

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

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

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

- 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, this means that 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, this means that 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 Profiles

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 **`<polygon>`** element is not in the common profile; rather, it is invariant to all techniques.

Parameter as an Interface

In COLLADA, a **`<param>`** or **`<newparam>`** element declares a symbol whose *name* or *semantics* declare 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.

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>`** *name* attribute and **`<newparam>`** semantic values are:

Name attribute value	Type	Typical Context	Description	Default
A	float	**`<material>`**, **`<texture>`**	Alpha color component	N/A
ANGLE	float	**`<animation>`**, **`<light>`**	Euler angle	N/A

Name attribute value	Type	Typical Context	Description	Default
B	float	`<material>`, `<texture>`	Blue color component	N/A
DOUBLE_SIDED	float	`<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 `<input>` *semantic* attribute values are:

Semantic attribute value	Description
`BINORMAL`	Geometric binormal (bitangent) vector
`COLOR`	Color coordinate vector
`CONTINUITY`	Continuity constraint at the CV
`IMAGE`	Raster or MIP-level input
`INPUT`	Sampler input
`IN_TANGENT`	Tangent vector for preceding control point
`INTERPOLATION`	Sampler interpolation type
`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
`MORPH_TARGET`	Morph targets for mesh morphing
`MORPH_WEIGHT`	Weights for mesh morphing
`NORMAL`	Normal vector
`OUTPUT`	Sampler output
`OUT_TANGENT`	Tangent vector for succeeding control point
`POSITION`	Geometric coordinate vector
`TANGENT`	Geometric tangent vector

Semantic attribute value	Description
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

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

Name	Type	Description
'(' # ')'['(' # ')']	float	Matrix or vector field
A	float	Alpha color component
ANGLE	float	Euler angle
B	float	Blue color component
G	float	Green color component
P	float	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.

Schema Reference Organization

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	Element constraints and relationships
Child Elements	Details about valid child elements
Remarks	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 element:

- If there is a main entry for the child element in one of the Reference chapters, refer there 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>`	Description of child element. *(A main Reference entry exists for `camera`, so look there for details.)*	N/A (not applicable or none assigned)	1 or more
`<yfov sid="...">`	The `sid` is optional. *(There is no main Reference entry for `yfov`. Details are given here.)*		
`<technique_common`	See the following subsection. *(Details are given here but in a separate table.)*		

This page intentionally left blank.

Chapter 4:
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 section includes the elements that describe geometry, animation, skinning, assets, and scenes.

accessor

Introduction

Declares an access pattern to one of the array elements `<float_array>`, `<int_array>`, `<Name_array>`, `<bool_array>`, and `<IDREF_array>`.

The `<accessor>` element describes access to arrays that are organized in either an interleaved or noninterleaved manner, depending on the `offset` and `stride` attributes.

Concepts

The `<accessor>` element describes a stream of values from an array data source. 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 URL expression. Required.
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:

Occurrences	Number of elements defined in the schema
Parent elements	`technique_common`
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
`<param>`	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`.	N/A	0 or more

Remarks

The number and order of `<param>` elements define the output of the `<accessor>` element. Parameters are bound to values in the order they are specified. No reordering of the data can occur. A `<param>` element without a name attribute is unbound. This is an indication that the value is not part of the output.

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

The `stride` attribute must have a value equal to or greater 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>
```

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

```
<source>
  <int_array name="values" count="9">
    1 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="C" type="int"/>
    </accessor>
  </technique_common>
</source>
```

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="A" type="int"/>
      <param name="B" type="int"/>
    </accessor>
  </technique_common>
</source>
```

ambient (core)

Introduction

Describes an ambient light source.

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`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. It can also have an `sid` attribute.		1

Remarks

The **`<ambient>`** element can be used only as a child of **`<technique_common>`** under a **`<light>`** element.

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

Introduction

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

Concepts

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**library_animations, animation**
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>			0 or 1
<animation>			0 or more (see "Remarks")
<source>			0 or more
<sampler>			0 or more (see "Remarks")

Name/example	Description	Default	Occurrences
\<channel\>			0 or more (see "Remarks")
\<extra\>			0 or more

Remarks

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.

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=" elliot's 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

Introduction

Defines a section of the 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:

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<instance_animation>`			1 or more
`<extra>`			0 or more

Remarks

Example

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

```xml
<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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**camera, COLLADA, light, material, technique, source, geometry, image, animation, animation_clip, controller, extra, node, visual_scene, library_*, effect, force_field, physics_material, physics_scene, physics_model**
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>`	Data related to a contributor that worked on the parent element.		0 or more
`<created>`	Date and time that the parent element was created. Represented in an ISO 8601 format as per the XML Schema <u>dateTime</u> primitive type.		1
`<keywords>`	A list of words used as search criteria for the parent element.		0 or 1
`<modified>`	Date and time that the parent element was last modified. Represented in an ISO 8601 format as per the XML Schema <u>dateTime</u> primitive type.		1
`<revision>`	Revision information for the parent element.		0 or 1
`<subject>`	A description of the topical subject of the parent element.		0 or 1
`<title>`	Title information for the parent element.		0 or 1

Name/example	Description	Default	Occurrences

`<unit`
    `meter=…`
    `name=…`
`/>`
	Descriptive information about unit of measure. Its optional attributes are:
	
	name: The name of the unit.
	
	meter: The measurement with respect to the meter. For more information, see "About Physical Units" in the "COLLADA Physics Reference" chapter.
		name:
		meter
		
		meter:
		1.0
			0 or 1

`<up_axis>`	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 "Remarks."	`Y_UP`	0 or 1

Remarks

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

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:

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

Remarks

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

Introduction

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

Concepts

A camera embodies the eye point of the viewer looking into the 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 into an image. The image is focused onto the plane of the camera's imager or film. The imager records the resulting image.

Attributes

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

`id`	`xs:ID`	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:

Occurrences	Number of elements defined in the schema
Parent elements	`library_cameras`
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>`			0 or 1
`<optics>`	Describes the field of view and viewing frustum using canonical parameters.		1
`<imager>`			0 or 1
`<extra>`			0 or more

Remarks

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

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</yfov>
        <aspect_ratio>1.33333</aspect_ratio>
        <znear>1.0</znear>
        <zfar>1000.0</zfar>
      </perspective>
    </technique_common>
  </optics>
</camera>
```

channel

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 2. Required.

Related Elements

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

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

Remarks

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

Example

Here is an example of a **`<channel>`** element that targets the 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

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 a "document root element" or document entity to be a well-formed XML document.

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`	The XML Schema namespace attribute `xmlns` 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:

Occurrences	One time
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: **<asset>**, **<library_*>**, **<scene>**, **<extra>**.

Name/example	Description	Default	Occurrences
<asset>			1
<library_animations>			0 or more
<library_animation_clips>			0 or more
<library_cameras>			0 or more
<library_controllers>			0 or more
<library_effects>			0 or more
<library_force_fields>			0 or more
<library_geometries>			0 or more
<library_images>			0 or more
<library_lights>			0 or more
<library_materials>			0 or more

Name/example	Description	Default	Occurrences
`<library_nodes>`			0 or more
`<library_physics_materials>`			0 or more
`<library_physics_models>`			0 or more
`<library_physics_scenes>`			0 or more
`<library_visual_scenes>`			0 or more
`<scene>`			0 or 1
`<extra>`			0 or more

Remarks

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

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

Attributes

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

`sid`	**`xs:NCName`**	For **`<profile_COMMON>`** 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:

Occurrences	Number of elements defined in the schema
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

Remarks

Example

contributor

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.

Attributes

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

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<authoring_tool>`			0 or 1
`<comments>`			0 or 1
`<copyright>`			0 or 1
`<source_data>`			0 or 1

Remarks

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`library_controllers`**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present.

Note: Exactly one of the child elements **`<morph>`** or **`<skin>`** must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<asset>`			0 or 1
`<skin>`			see Note
`<morph>`			see Note
`<extra>`			0 or more

Remarks

A **`<controller>`** element contains the 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>
```

directional

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`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. It can also have an **`sid`** attribute.		1

Remarks

A **`<directional>`** element can occur only as a child of **`<technique_common>`** under a **`<light>`** element.

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

Introduction

Declares additional information regarding its parent element.

Concepts

An extensible schema requires a means for users to specify arbitrary information. This extra information can represent additional real data or semantic (meta) data to the application.

COLLADA represents extra information as techniques containing 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`	The type of the value data. This text string must be understood by the application. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`animation, animation_clip, attachment, box, camera, capsule, COLLADA, controller, control_vertices, convex_mesh, cylinder, effect, force_field, geometry, image, imager, instance_*, joints, library_*, light, lines, linestrips, material, mesh, morph, node, optics, physics_material, physics_model, physics_scene, plane, polygons, polylist, profile_COMMON, profile_COMMON::technique, ref_attachment, rigid_body, rigid_constraint, scene, shape, skin, sphere, spline, tapered_capsule, tapered_cylinder, targets, texture, 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>`			0 or 1
`<technique>`			1 or more

Remarks

Example

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

```xml
<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>
```

float_array

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:

Occurrences	Number of elements defined in the schema
Parent elements	`source`
Child elements	None
Other	None

Remarks

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

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

Introduction

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

Concepts

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

- B-Spline
- Bezier
- 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`library_geometries`**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present.

Note: Exactly one of the child elements **`<convex_mesh>`**, **`<mesh>`**, or **`<spline>`** must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<asset>`			0 or 1
`<convex_mesh>`			see Note
`<mesh>`			see Note

Name/example	Description	Default	Occurrences
\<spline\>			see Note
\<extra\>			0 or more

Remarks

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.

Examples

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

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:

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

Remarks

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>
```

image

Introduction

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

Concepts

Digital imagery comes in three main forms of data: raster, vector, and hybrid. Raster imagery 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.

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. Optional.
height	uint	An integer value that indicates the height of the image in pixel units. Optional.
width	uint	An integer value that indicates the width of the image in pixel units. Optional.
depth	uint	An integer value that indicates the depth of the image in pixel units. A 2-D image has a depth of 1, which is also the default. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`library_images, effect, profile_CG, profile_GLSL, profile_COMMON, profile_GLES, technique`
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present.

Note: Exactly one of the child elements `<data>` or `<init_from>` must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<asset>`			0 or 1
`<data>`	Contains a sequence of hexadecimal encoded binary octets representing the embedded image data.		see Note

Name/example	Description	Default	Occurrences
`<init_from>`	Specifies an external image file.		see Note
`<extra>`	Contains embedded image data.		0 or more

Remarks

The `<image>` element allows you either to specify an external image file with `<init_from>` or to embed image data with `<data>`.

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>
```

imager

Introduction

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

Concepts

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

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

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

Display devices use a much more limited dynamic range and they usually consider only 3 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 will store the actual pixel colors as float3s, 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:

Occurrences	Number of elements defined in the schema
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\>			1 or more
\<extra\>			0 or more

Remarks

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

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

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

Introduction

Declares the input semantics of a data source.

Concepts

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

Attributes

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

`offset`	**`uint`**	The offset into the list of indices. If two **`<input>`** elements share the same offset, they will be indexed the same. This works as a simple form of compression for the list of indices as well as defining the order in which the inputs should be used. Required attribute if **`<input>`** is a child of a Group B parent; not valid for Group A parents.
`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.
`source`	**`xs:URIFragmentType`**	The location of the data source. Required.
`set`	**`uint`**	Which inputs should be grouped together as a single set. This is helpful when multiple inputs share the same semantics. Optional if **`<input>`** is a child of a Group B parent; not valid for Group A parents.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	Group A: **`joints, sampler, targets, vertices, control_vertices`** Group B: **`lines, linestrips, polygons, polylist, triangles, trifans, tristrips, vertex_weights`**
Child elements	None
Other	None

Remarks

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

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

Example

Here is an example of 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:

```xml
<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>
```

instance_animation

Introduction

Declares the instantiation of a COLLADA animation resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**animation_clip**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<extra>			0 or more

Remarks

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_animation>`** element that refers to a locally defined **`<animation>`** element identified by the ID "anim". The instance is translated some distance from the original:

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

instance_camera

Introduction

Declares the instantiation of a COLLADA camera resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`node`**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
`<extra>`			0 or more

Remarks

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_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

Introduction

Declares the instantiation of a COLLADA controller resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate.
		Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**node**
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
<skeleton>	Indicates where a skin controller is to start to search for the joint nodes it needs. This element is meaningless for morph controllers.		0 or more
<bind_material>			0 or 1

Name/example	Description	Default	Occurrences
<extra>			0 or more

Remarks

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

Introduction

Declares the instantiation of a COLLADA geometry resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**node, 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
`<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.		0 or 1

Name/example	Description	Default	Occurrences
`<extra>`			0 or more

Remarks

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

Introduction

Declares the instantiation of a COLLADA **light** resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**node**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<extra>			0 or more

Remarks

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

Introduction

Declares the instantiation of a COLLADA **node** resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**node**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
<extra>			0 or more

Remarks

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

Introduction

Declares the instantiation of a COLLADA visual_scene resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`scene`**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
`<extra>`			0 or more

Remarks

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

Introduction

Declares the storage for a homogenous array of integer values.

Concepts

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

Attributes

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

count	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:

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

Remarks

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

Example

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

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

joints

Introduction

The **\<joints\>** element 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:

Occurrences	Number of elements defined in the schema
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\>	Must *not* have the **offset** attribute when it is the child of a **\<joints\>** element. One of the two **\<input\>** elements 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.		2 or more
\<extra\>			0 or more

Remarks

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

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or more
`<animation_clip>`			1 or more
`<extra>`			0 or more

Remarks

Example

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

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

library_animations

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<animation>`			1 or more
`<extra>`			0 or more

Remarks

Example

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

```
<library_animations>
  <animation/>
</library_animations>
```

library_cameras

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<camera>`			1 or more
`<extra>`			0 or more

Remarks

Example

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

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

library_controllers

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<controller>`			1 or more
`<extra>`			0 or more

Remarks

Example

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

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

library_geometries

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or 1
<geometry>			1 or more
<extra>			0 or more

Remarks

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_images

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or 1
<image>			1 or more
<extra>			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_lights

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_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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<light>`			1 or more
`<extra>`			0 or more

Remarks

Example

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

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

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

library_nodes

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or 1
<node>			1 or more
<extra>			0 or more

Remarks

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

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<visual_scene>`			1 or more
`<extra>`			0 or more

Remarks

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

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:

- 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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`library_lights`**
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>`			0 or 1
`<technique_common>`	As a child of **`<light>`**, **`<technique_common>`** must contain exactly one **`<ambient> (core)`**, **`<directional>`**, **`<point>`**, or **`<spot>`** element.		1
`<technique>`			0 or more
`<extra>`			0 or more

Remarks

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:

```xml
<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>
      <instance_light url="#sun"/>
      <rotate>1 0 0 -10</rotate>
    </node>
  </visual_scene>
</library_visual_scenes>
```

lines

Introduction

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

Concepts

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

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

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

Attributes

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

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. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**mesh, 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>			0 or more
<p>	("**p**" stands for primitives.) Describes the vertex attributes for an arbitrary number of individual lines. 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 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.		0 or 1
<extra>			0 or more

Remarks

Example

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

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

linestrips

Introduction

Declares a binding of geometric primitives and vertex attributes for a `<mesh>` element.

Concepts

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

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

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

Attributes

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

`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. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`mesh, 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>`			0 or more
`<p>`	("`p`" stands for primitive.) Describes the vertex attributes for an arbitrary number of connected line segments. 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 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.		0 or more
`<extra>`			0 or more

Remarks

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 are using 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

Introduction

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

Concepts

The **`<lookat>`** element contains 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:

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

Remarks

The **`<lookat>`** element contains a list of 9 floating-point values. As in the OpenGL® Utilities (GLU) implementation, these values are organized into three vectors as follows:

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.

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">
  <instance_camera url="#camera1"/>
  <lookat>
     2.0  0.0  3.0  <!-- eye position (X,Y,Z)        -->
     0.0  0.0  0.0  <!-- interest position (X,Y,Z)  -->
     0.0  1.0  0.0  <!-- up-vector position (X,Y,Z) -->
  </lookat>
  ...
```

Figure 4-1: <lookat> element; the 3-D "cross-hair" represents the interest-point position
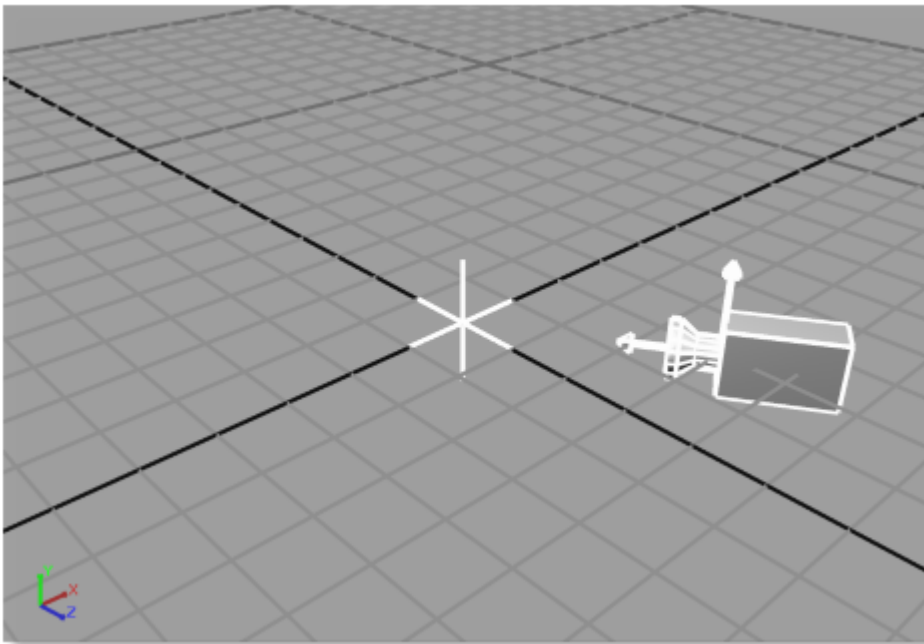
matrix

Introduction

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

Concepts

The `<matrix>` element contains 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 below.

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:

Occurrences	Number of elements defined in the schema
Parent elements	`node`
Child elements	None
Other	None

Remarks

The `<matrix>` element contains a list of 16 floating-point values. These values are organized into a 4-by-4 column-order matrix suitable for matrix composition.

Example

Here is an example of a `<matrix>` element forming a translation matrix that translates 2 units along the *x* axis, 3 units along the *y* axis, and 4 units along the *z* axis:

```
<matrix>
  1.0 0.0 0.0 2.0
  0.0 1.0 0.0 3.0
  0.0 0.0 1.0 4.0
  0.0 0.0 0.0 1.0
</matrix>
```

mesh

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:

Occurrences	Number of elements defined in the schema
Parent elements	**geometry**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present: **\<source\>**, **\<vertices\>**, **primitive elements**, **\<extra\>** (where primitive elements is any combination of **\<lines\>**, **\<linestrips\>**, **\<polygons\>**, **\<polylist\>**, **\<triangles\>**, **\<trifans\>**, or **\<tristrips\>**).

Name/example	Description	Default	Occurrences
\<source\>	Provides the bulk of the mesh's vertex data.		1 or more
\<vertices\>	Describes the mesh-vertex attributes and establishes their topological identity.		1
\<lines\>	Contains line primitives.		0 or more
\<linestrips\>	Contains line-strip primitives.		0 or more
\<polygons\>	Contains polygon primitives which may contain holes.		0 or more
\<polylist\>	Contains polygon primitives that cannot contain holes.		0 or more

Name/example	Description	Default	Occurrences
\<triangles\>	Contains triangle primitives.		0 or more
\<trifans\>	Contains triangle-fan primitives.		0 or more
\<tristrips\>	Contains triangle-strip primitives.		0 or more
\<extra\>			0 or more

Remarks

To describe geometric primitives that are formed from the vertex data, the **\<mesh\>** element may contain zero or more of the following 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\>** element with a **semantic** attribute whose value is **POSITION**.

For texture coordinates, COLLADA's right-handed coordinate system applies; therefore, an "ST" coordinate of [0,0] maps to the lower-left texel of a texture image, when loaded in a professional 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>
```

morph

Introduction

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

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).

Concepts

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

Attributes

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

source	**xs:anyURI**	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:

Occurrences	Number of elements defined in the schema
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>			2 or more
\<targets>			1
\<extra>			0 or more

Remarks

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

Introduction

Declares the storage for a homogenous array of symbolic name values.

Concepts

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

Attributes

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

count	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 the element. Optional.

Related Elements

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

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

Remarks

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

Example

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

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

node

Embodies the hierarchical relationship of elements in the scene.

Introduction

The **`<node>`** element declares a point of interest in the scene. A node denotes one point on a branch of the scene graph. The **`<node>`** element is essentially the root of a 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 the 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 'foreground' layer and the 'glowing' layer. 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`library_nodes, node, visual_scene`**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present: **`<asset>`**, any combination of transform elements, **`<instance_camera>`**, **`<instance_controller>`**, **`<instance_geometry>`**, **`<instance_light>`**, **`<instance_node>`**, **`<node>`**, **`<extra>`**, where the transform elements are **`<lookat>`**, **`<matrix>`**, **`<rotate>`**, **`<scale>`**, **`<skew>`**, **`<translate>`**.

Name/example	Description	Default	Occurrences
`<asset>`	Allows the node to express asset management information.		0 or 1
`<lookat>`			0 or more
`<matrix>`			0 or more

Name/example	Description	Default	Occurrences
`<rotate>`			0 or more
`<scale>`			0 or more
`<skew>`			0 or more
`<translate>`			0 or more
`<instance_camera>`	Allows the node to instantiate a camera object.		0 or more
`<instance_controller>`	Allows the node to instantiate a controller object.		0 or more
`<instance_geometry>`	Allows the node to instantiate a geometry object.		0 or more
`<instance_light>`	Allows the node to instantiate a light object.		0 or more
`<instance_node>`	Allows the node to instantiate a hierarchy of other nodes.		0 or more
`<node>`	Allows the node to recursively define hierarchy.		0 or more
`<extra>`	Allows the node to define extra information		0 or more

Remarks

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

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

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

Example

The following example shows a simple outline of a **`<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

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:

Occurrences	Number of elements defined in the schema
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 the optics information for the common profile that all COLLADA implementations must support. See the following subsection.		1
`<technique> (core)`	Non-COMMON profile techniques.		0 or more
`<extra>`			0 or more

Child Elements for optics / technique_common

Note: Exactly one of the child elements **`<orthographic>`** or **`<perspective>`** must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<orthographic>`			see Note
`<perspective>`			see Note

Remarks

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

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.

Attributes

The `<orthographic>` element has no attributes.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`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
`<xmag sid="...">`	A floating-point number describing the horizontal (X) magnification of the view. The `sid` is optional.		See "Remarks"
`<ymag sid="...">`	A floating-point number describing the vertical (Y) magnification of the view. The `sid` is optional.		See "Remarks"
`<aspect_ratio sid="...">`	A floating-point number describing the aspect ratio of the field of view. The `sid` is optional.		See "Remarks"
`<znear sid="...">`	A floating-point number that describes the distance to the near clipping plane. The `sid` is optional.		1
`<zfar sid="...">`	A floating-point number that describes the distance to the far clipping plane. The `sid` is optional.		1

Remarks

The `<orthographic>` element can occur only in `<technique_common>` under `<camera><optics>`.

The `<orthographic>` element must contain either a single `<xmag>` element, a single `<ymag>` element, both `<xmag>` and `<ymag>` elements, or one of `<xmag>` or `<ymag>` and the `<aspect_ratio>` element. 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.

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

Introduction

Declares parametric information regarding its parent element.

Concepts

A functional or programmatical format requires a means for users to specify parametric information. This information represents function parameter (argument) data.

Material shader programs may contain code representing vertex or pixel programs. These programs require parameters as part of their state information.

The basic declaration of a parameter describes the name, data type, and value data of the parameter. That parameter name identifies it to the function or program. The parameter type indicates the encoding of its value. The parameter value is the actual data.

Attributes

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

`name`	`xs:NCName`	The text string name of the 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:

Occurrences	Number of elements defined in the schema
Parent elements	`accessor, bind_material`
Child elements	None
Other	None

Remarks

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

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.

Attributes

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

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`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
`<xfov sid="...">`	A floating-point number describing the horizontal field of view in degrees. The **`sid`** is optional.		See "Remarks"
`<yfov sid="...">`	A floating-point number describing the vertical field of view in degrees. The **`sid`** is optional.		See "Remarks"
`<aspect_ratio sid="...">`	A floating-point number describing the aspect ratio of the field of view. The **`sid`** is optional.		See "Remarks"
`<znear sid="...">`	A floating-point number that describes the distance to the near clipping plane. The **`sid`** is optional.		1
`<zfar sid="...">`	A floating-point number that describes the distance to the far clipping plane. The **`sid`** is optional.		1

Remarks

The **`<perspective>`** element can occur only in **`<technique_common>`** under **`<camera><optics>`**.

The **`<perspective>`** element must contain either a single **`<xfov>`** element, a single **`<yfov>`** element, both **`<xfov>`** and **`<yfov>`** elements, or one of **`<xfov>`** or **`<yfov>`** and the **`<aspect_ratio>`** element. These describe the field of view of the camera.

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 a pixel's height over the pixel's width;

therefore, the aspect ratio can be derived from, or be used to derive, the field of view parameters:
$aspect\_ratio = yfov / xfov$.

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

Example

Here is an example of a **`<perspective>`** element specifying a horizontal field-of-view of 90 degrees:

```
<perspective>
  <xfov sid="animated_zoom">1.0</xfov>
  <aspect_ratio>1.333</aspect_ratio>
  <znear>0.1</znear>
  <zfar>1000.0</zfar>
</perspective>
```

point

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:

Occurrences	Number of elements defined in the schema
Parent elements	`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 sid="...">`	Three floating-point numbers specifying the color of the light. The **sid** is optional.		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

Remarks

The `<point>` element can occur only as a child of `<technique_common>` under a `<light>` element.

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:

```xml
<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

Introduction

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

Concepts

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

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>` elements
- standard (geometric) tangents: specified with the **TANGENT** and **BINORMAL** semantics on the `<input>` 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. 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:

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>`			0 or more
`<p>`			0 or more
`<ph>`			0 or more
`<extra>`			0 or more

Remarks

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

A polygon that contains one or more holes is specified as a `<ph>` element. Each `<ph>` element must contain one `<p>` element and one or more `<h>` elements. The `<p>` element specifies the indices of the polygon and each `<h>` element specifies the indices of a hole.

The indices in a `<p>` (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>` 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

Introduction

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

Concepts

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

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>` elements
- standard (geometric) tangents: specified with the **TANGENT** and **BINORMAL** semantics on the `<input>` 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. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or more
\<vcount>			0 or 1
\<p>			0 or 1
\<extra>			0 or more

Remarks

The **\<vcount>** element contains a list of integers describing the number of sides for each polygon described by the **\<polylist>** element.

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 quads and a triangle. The **\<polylist>** element contains two **\<source>** elements that contain the position and normal data, according to the **\<input>** element semantics. The **\<p>** element index values indicate the order that the input values are used:

```
<mesh>
  <source 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

Introduction

Contains an angle and a mathematical vector that represents the axis of rotation.

Concepts

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

Attributes

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

Occurrences	Number of elements defined in the schema
Parent elements	**`node, mass_frame, shape, ref_attachment, attachment`**
Child elements	None
Other	None

Remarks

The **`<rotate>`** element contains a list of four floating-point values, similar to rotations in the OpenGL® and RenderMan® specification. These values are organized into a column vector [X, Y, Z] specifying the axis of rotation followed by an angle in degrees.

Example

Here is an example of a **`<rotate>`** element forming a rotation of 90 degrees about the *y* axis:

```
<rotate>
  0.0 1.0 0.0 90.0
</rotate>
```

sampler

Introduction

Declares an N-dimensional function.

Concepts

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

Sampling points (key frames) are input data sources to the sampler, as are interpolation type symbolic names. Animation channels direct the output data values of the sampler to their targets.

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:

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

Child Elements

Name/example	Description	Default	Occurrences
<input>			1 or more

Remarks

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**, and **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.

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

Introduction

Contains a mathematical vector that represents the relative proportions of the *x*, *y*, and *z* axes of a coordinated system.

Concepts

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

Attributes

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

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

Remarks

The **`<scale>`** element contains a list of three floating-point values. These values are organized into a column vector suitable for matrix composition.

Example

Here is an example of a **`<scale>`** element that describes a uniform increase in size of an object (or coordinated system) by a factor of two:

```
<scale>
  2.0 2.0 2.0
</scale>
```

scene

Introduction

Embodies the entire set of information that can be visualized from the contents of a COLLADA resource.

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

Concepts

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

Attributes

The `<scene>` element has no attributes.

Related Elements

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

Occurrences	Zero or one time
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>`			0 or more
`<instance_visual_scene>`			0 or 1
`<extra>`			0 or more

Remarks

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 instances a visual scene with the id 'world':

```
<COLLADA>
  <scene>
    <instance_visual_scene url="#world"/>
  </scene>
</COLLADA>
```

skeleton

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:

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

Remarks

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

Introduction

Contains an angle and two mathematical vectors that represent the axis of rotation and the axis of translation.

Concepts

Skew (shear) deforms an object along one axis of a coordinated system. It translates values along the affected axis in a direction that is parallel to that axis. Computer graphics techniques apply a skew or shear transformation in order to deform objects or to correct distortion in images.

Attributes

The **`<skew>`** element has the following 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:

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

Remarks

As in the RenderMan® specification, the **`<skew>`** element contains a list of seven floating-point values. These values are organized into an angle in degrees followed by two column vectors specifying the axes of rotation and translation.

Example

Here is an example of a **`<skew>`** element forming a displacement of points along the *x* axis due to a rotation of 45 degrees around the *y* axis:

```
<skew>
  45.0 0.0 1.0 0.0 1.0 0.0 0.0
</skew>
```

skin

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:

- pre-processing, known as "binding the skeleton to the skin"
- running the *skinning algorithm* to modify the shape of the skin as the *pose* of the skeleton changes The results of the pre-processing, or "skinning information" consists of the following:
- **bind-shape:** also called "default shape". This is the shape of the skin when it was *bound* to the skeleton. This includes positions (required) for each corresponding **`<mesh>`** vertex and may optionally include additional vertex attributes.
- **influences:** a variable-length lists of node + weight pairs for each **`<mesh>`** vertex.
- **bind-pose**: the transforms of all influences at the time of binding. This per-node information is usually represented by a "bind-matrix", which is the *local-to-world* matrix of a node at the time of binding.

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

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

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

The binding process usually involves:

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

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

Attributes

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

Occurrences	Number of elements defined in the schema
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
`<bind_shape_matrix>`	Provides extra information about the position and orientation of the base mesh before binding. If **`<bind_shape_matrix>`** is not specified then an identity matrix may be used as the **`<bind_shape_matrix>`**.		0 or 1
`<source>`	Provides most of the data required for skinning the given base mesh.		3 or more
`<joints>`	Aggregates the per-joint information needed for this skin.		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.		1
`<extra>`			0 or more

Remarks

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

Introduction

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

Concepts

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

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

Attributes

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

`id`	**`xs:ID`**	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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`morph, animation, mesh, convex_mesh, skin, spline`**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present.

Note: No more than one of the array elements (**`<bool_array>`**, **`<float_array>`**, **`<int_array>`**, **`<Name_array>`**, or **`<IDREF_array>`**) may occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<asset>`			0 or 1
`<IDREF_array>`			see Note
`<Name_array>`			see Note
`<bool_array>`			see Note
`<float_array>`			see Note
`<int_array>`			see Note
`<technique_common>`	As a child of **`<source>`**, this element must contain exactly one **`<accessor>`** element.		0 or 1
`<technique>`			0 or more

Remarks

Example

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

```xml
<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

Introduction

Contains information sufficient to describe a multisegment spline with control vertex (CV) and segment information.

Concepts

Both per-CV and per-segment information are stored on the CVs. Per-segment data applies to the spline segment starting at the given CV.

Each CV may have an arbitrary number of attributes, but the number and type of the attributes must be uniform for each CV within a spline.

Per CV/segment information may be:

- CV position (2d, 3d, or 4d – for NURBS)
- Tangents to control the curvature of a segment
- Tangent constraint to control continuity between two consecutive segments
- Segment types – each segment of a spline may use a different interpolation (LINEAR, BEZIER, etc.)
- Number of steps for piece-wise linear approximation (tessellation)
- Custom per-CV data

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.

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:

Occurrences	Number of elements defined in the schema
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.		1 or more
\<control_vertices>	Describes the CVs of the spline.		1
\<extra>			0 or more

Remarks

CVs must have at least one `<input>` element with a semantic attribute whose value is **POSITION**.

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

The COMMON profile defines the following `<input>` semantics for `<spline>` / `<control_vertices>`:

Name	Type	Description	Default
POSITION	`float2` `float3` `float4`	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`, `BSPLINE`, `STEP`, and `NURBS`	`LINEAR`
IN_TANGENT	`float2` `float3`	The tangent that controls the shape of the segment preceding the CV (`BEZIER` and `HERMITE`)	N/A
OUT_TANGENT	`float2` `float3`	The tangent that controls the shape of the segment following the CV (`BEZIER` and `HERMITE`)	N/A
CONTINUITY	`Name`	Defines the continuity constraint at the CV. The common-profile types are: C0, C1, G1	C1
LINEAR_STEPS	`int`	The number of piece-wise linear approximation steps to use for the spline segment that follows this CV	N/A

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

Introduction

Describes a spot light source.

Concepts

The `<spot>` element declares the parameters required to describe a spot light source. 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.

The position of the light is defined by the transform of the node in which it is instantiated. 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 `<spot>` element has no attributes.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`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 sid="...">`	Contains three floating spot numbers specifying the color of the light. The `sid` is optional.		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

Remarks

The `<spot>` element can only occur as a child of `<technique_common>` under a `<light>` element.

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

Introduction

Declares the 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:

Occurrences	Number of elements defined in the schema
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>	Must occur once with **semantic="MORPH_TARGET"** and once with **semantic="MORPH_WEIGHT"**.		2 or more
<extra>			0 or more

Remarks

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)

Introduction

Declares the information used to process some portion of the content. Each technique conforms to an associated profile.

For **`<technique>`** in FX elements, see "**`technique`** (FX)."

Concepts

A technique describes information needed by a specific platform or program. This is represented by its profile. 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.

The **`<technique>`** element can also have a xmlns attribute (as per XML Schema Language) describing an additional schema to use for validating its contents.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`extra, source, light, optics, imager, force_field, physics_material, physics_scene, rigid_body, rigid_constraint, instance_rigid_body, bind_material`**
Child elements	See "Remarks"
Other	None

Remarks

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 can't 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>
```

technique_common

Introduction

Specifies the information for a specific element for the COMMON profile that all COLLADA implementations must support.

Concepts

Each element's **`<technique_common>`** arguments and children are unique. Refer to each parent element for details.

Attributes

See Reference chapter entries for each parent element.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`bind_material`**, **`instance_rigid_body`**, **`light`**, **`optics`**, **`physics_material`**, **`physics_scene`**, **`rigid_body`**, **`rigid_constraint`**, **`source`**
Child elements	See Reference chapter entries for each parent element.
Other	None

Rearks

Example

translate

Introduction

Contains a mathematical vector that represents the distance along the *x*, *y*, and *z* axes.

Concepts

Translations change the position of objects in a coordinate system without any rotation. Computer graphics techniques apply a translation transformation in order to position or, move values with respect to a coordinate system. Conversely, translation means to move the origin of the local coordinate system.

Attributes

The **<translate>** element has the following 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:

Occurrences	Number of elements defined in the schema
Parent elements	**node, shape, mass_frame, ref_attachment, attachment**
Child elements	None
Other	None

Remarks

The **<translate>** element contains a list of three floating-point values. These values are organized into a column vector suitable for a matrix composition.

Example

Here is an example of a **<translate>** element forming a displacement of 10 units along the *x* axis:

```
<translate>
   10.0 0.0 0.0
</translate>
```

triangles

Introduction

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

Concepts

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

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

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

Attributes

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

`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. Optional. If not specified then the lighting and shading results are application defined.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`mesh, 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>`			0 or more
`<p>`	("**p**" stands for primitive.) Describes the vertex attributes for a number of triangles.		0 or 1
`<extra>`			0 or more

Remarks

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

Introduction

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

Concepts

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

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

Each triangle described by the mesh has three vertices. The first triangle is formed from 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. Optional. If not specified then the lighting and shading results are application defined.

Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**mesh, 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>			0 or more
<p>	("**p**" stands for primitive.) Describes the vertex attributes for an arbitrary number of connected triangles.		0 or more
<extra>			0 or more

Remarks

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

Introduction

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

Concepts

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

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

Each triangle described by the mesh has three vertices. The first triangle is formed from 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. Optional. If not specified then the lighting and shading results are application defined.

Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**mesh, 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>			0 or more
<p>	("**p**" stands for primitive.) Describes the vertex attributes for an arbitrary number of connected triangles.		0 or more
<extra>			0 or more

Remarks

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

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:

Occurrences	Number of elements defined in the schema
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>`	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.		2 or more
`<vcount>`	Describes the number of bones associated with each vertex.		0 or 1
`<v>`	Describes 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.		0 or 1
`<extra>`			0 or more

Remarks

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

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 the element. Optional.

Related Elements

The **`<vertices>`** 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>`	One input must specify **`semantic="POSITION"`** to establish the topological identity of each vertex in the mesh. Must *not* have the **`offset`** attribute when it is the child of a **`<vertices>`** element.		1 or more
`<extra>`			0 or more

Remarks

Example

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

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

visual_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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`library_visual_scenes`
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>`			0 or 1
`<node>`			1 or more
`<evaluate_scene>`			0 or more
`<extra>`			0 or more

Remarks

There may 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.

The `<visual_scene>` element forms the root of the scene graph topology.

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
<?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>
```

This page intentionally left blank.

Chapter 5:
COLLADA Physics Reference

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 (non-dynamic) 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_o, y_o, z_o) 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 1.4.0 adds some primitive geometry types, such as **\<box\>**, **\<sphere\>** etc., as well as **\<convex_mesh\>**.

Other than **\<convex_mesh\>**, these shapes are not meant for rendering because meshes, subdivision surfaces, etc., are much better-suited for that purpose.

This system – of analytical shapes plus the convex mesh – can describe any shape, so the **\<bounding_box\>** element is deprecated.

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>
```
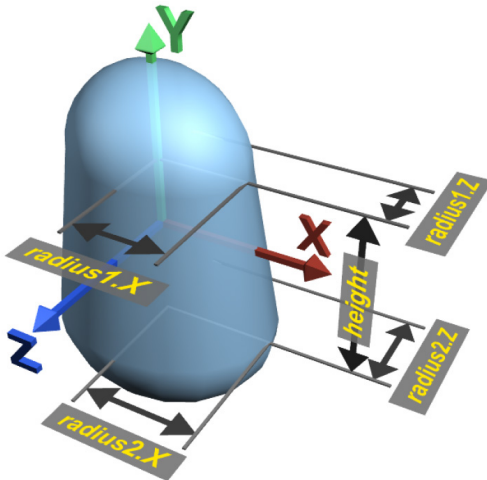
box

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.

Attributes

The **\<box>** element has no attributes.

Related Elements

The **\<box>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
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>	3 float values that represent the extents of the box	N/A	1
\<extra>		N/A	0 or more

Example

```
<box>
  <half_extents> 2.5  1.0  1.0 </half_extents>
</box>
```

capsule

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:

Occurrences	Number of elements defined in the schema
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>	A float value that represents the length of the line segment connecting the centers of the capping hemispheres.	N/A	1
<radius>	Two float values that represent the radii of the capsule (it may be elliptical)	N/A	1
<extra>		N/A	0 or more

Example

```
<capsule>
  <height> 2.0 </height>
  <radius> 1.0 1.0 </radius>
</capsule>
```

convex_mesh

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>** to compute the convex hull of. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**geometry**
Child elements	See the following subsection.
Other	None

Child Elements

Child elements must appear in the following order if present: **<source>**, **<vertices>**, **primitive elements**, **<extra>** (where primitive elements is any combination of **<lines>**, **<linestrips>**, **<polygons>**, **<polylist>**, **<triangles>**, **<trifans>**, or **<tristrips>**).

Name/example	Description	Default	Occurrences
<source>	Provides the bulk of the mesh's vertex data.		1 or more
<vertices>	Describes the mesh-vertex attributes and establishes their topological identity.		1
<lines>	Contains line primitives.		0 or more
<linestrips>	Contains line-strip primitives.		0 or more
<polygons>	Contains polygon primitives which may contain holes.		0 or more
<polylist>	Contains polygon primitives that cannot contain holes.		0 or more
<triangles>	Contains triangle primitives.		0 or more
<trifans>	Contains triangle-fan primitives.		0 or more

Name/example	Description	Default	Occurrences
`<tristrips>`	Contains triangle-strip primitives.		0 or more
`<extra>`			0 or more

Remarks

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

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.

Attributes

The **\<cylinder\>** element has no attributes.

Related Elements

The **\<cylinder\>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
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\>	A float value that represents the length of the cylinder along the *y* axis	N/A	1
\<radius\>	Two float values that represent the radii of the cylinder (it may be elliptical)	N/A	1
\<extra\>		N/A	0 or more

Example

```
<cylinder>
  <height> 2.0 </height>
  <radius> 1.0 1.0 </radius>
</cylinder>
```

force_field

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:

Occurrences	Number of elements defined in the schema
Parent elements	**library_force_fields**
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>		N/A	0 or 1
<technique>		N/A	1 or more
<extra>		N/A	0 or more

Remarks

Currently there is no COMMON technique/profile for **<force_field>**. The **<technique>** element can contain any well-formed XML data.

Example

```xml
<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

Introduction

Declares the instantiation of a **`<force_field>`**.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`instance_physics_model, physics_scene`**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
`<extra>`			0 or more

Remarks

Example

instance_physics_material

Introduction

Declares the instantiation of a **`<physics_material>`**.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`technique_common`** in **`rigid_body`**, **`technique_common`** in **`instance_rigid_body`**, **`shape`**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
`<extra>`			0 or more

Remarks

Example

instance_physics_model

Introduction

Allows the instantiation of 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.

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. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.
`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:

Occurrences	Number of elements defined in the schema
Parent elements	**`physics_scene, physics_model`**
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_force_field>`	Instantiates a **`<force_field>`** element to influence this physics model	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.	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.	N/A	0 or more
`<extra>`		N/A	0 or more

instance_physics_scene

Introduction

Declares the instantiation of a **`<physics_scene>`**.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`scene`**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
`<extra>`			0 or more

Remarks

Example

instance_rigid_body

Introduction

Allows the instantiation of a **`<rigid_body>`** within an **`<instance_physics_model>`**.

Concepts

Rigid-bodies will 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

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. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**instance_physics_model**
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 the common-profile representation for the rigid body. See the following subsection.	N/A	1

Name/example	Description	Default	Occurrences
`<technique>`	Specifies a target profile for the rigid body to allow for multirepresentation.	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).	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>`	Specifies 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>`	Specifies the initial linear velocity of the **`rigid_body`** instance.	0, 0, 0	0 or 1
`<dynamic>false</dynamic>`	If false, the **`rigid_body`** is not moveable	N/A	0 or 1
`<mass>0.5</mass>`	The total mass of the rigid-body.	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*).
		"identity" (center of mass is at the local origin and the principal axes are the local axes)	0 or 1

`<inertia> 1 1 1 </inertia>`	**`float3`** – The diagonal elements of the inertia tensor (*moments of inertia*), which is represented in the local frame of the center of mass. See above.	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`**.	N/A	0 or 1
`<shape>`			0 or more

The **`<instance_rigid_body>`** element and its techniques have the same children as that of **`<rigid_body>`** and its technique element adds the following:

Name/example	Description	Default	Occurrences
`<velocity>`	Specifies the initial linear velocity of the **`rigid_body`** instance.	0, 0, 0	0 or 1
`<angular_velocity>`	Specifies 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

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>
```

library_force_fields

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or 1
<force_field>			1 or more
<extra>			0 or more

Remarks

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

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<physics_material>`			1 or more
`<extra>`			0 or more

Remarks

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

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<physics_model>`			1 or more
`<extra>`			0 or more

Remarks

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

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>`			0 or 1
`<physics_scene>`			1 or more
`<extra>`			0 or more

Remarks

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**library_physics_materials, shape, technique_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
<asset>		N/A	0 or 1
<technique_common>	Defines the common technique for a physics material. See the following subsection.	N/A	1
<technique>	Defines a profile for a physics material.	N/A	0 or more
<extra>		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> 0.23 </static_friction>	Dynamic friction coefficient.	0	0 or 1
<restitution> 0.2 </restitution>	Also known as "bounciness" or "elasticity": the proportion of the kinetic energy preserved in the impact (typically ranges from 0.0 to 1.0).	0	0 or 1
<static_friction> 0.23 </static_friction>	Static friction coefficient.	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

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`library_physics_models`**
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>`		N/A	0 or 1
`<rigid_body sid="...">`	Defines a **`<rigid_body>`** element and sets its non-default properties.	N/A	0 or more
`<rigid_constraint sid="...">`	Defines a **`<rigid_constraint>`** element and allows for overriding some or all of its properties.	N/A	0 or more
`<instance_physics_model sid="..." url="#...">`	Instantiates a physics model from the given url, and assigns an sid to it, to distinguish it from other child elements.	N/A	0 or more
`<extra>`		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

Introduction

Specifies an environment in which physical objects are instantiated and simulated.

Concepts

COLLADA allows for multiple simulations to run independently for two 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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**library_physics_scenes**
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>		N/A	0 or 1
<instance_force_field>	Instantiates a **<force_field>** element to influence this physics scene.	N/A	0 or more
<instance_physics_model>	Instantiates a **<physics_model>** element and allows for overriding some or all of its children.	N/A	0 or more
<technique_common>	Specifies the common technique parameters for the **physics_scene**. See the following subsection.	N/A	1

Name/example	Description	Default	Occurrences
`<technique>`	Specifies custom techniques.	N/A	0 or more
`<extra>`		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

```xml
<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

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:

Occurrences	Number of elements defined in the schema
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\>	Four float values that represent the coefficients for the plane's equation: Ax + By + Cz + D = 0	N/A	1
\<extra\>		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>
```

rigid_body

Introduction

Describes simulated bodies that do not deform. These bodies may or may not be connected by constraints (hinge, ball-joint, and so on).

Rigid bodies, constraints, and so on are encapsulated in **`<physics_model>`** elements to allow the instantiation of complex models.

Concepts

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**physics_model**
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
`<technique_common>`	Specifies the common-profile representation for the rigid-body. See the following subsection.	N/A	1
`<technique>`	Specifies a target profile for the rigid-body to allow for multirepresentation.	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).	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>false</dynamic>`	If false, the `rigid_body` is not moveable	N/A	0 or 1
`<mass>0.5</mass>`	The total mass of the rigid-body.	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*).
		"identity" (center of mass is at the local origin and the principal axes are the local axes)	0 or 1

`<inertia> 1 1 1 </inertia>`	`float3` – The diagonal elements of the inertia tensor (*moments of inertia*), which is represented in the local frame of the center of mass. See above.	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`.	N/A	0 or 1

`<shape>`			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.
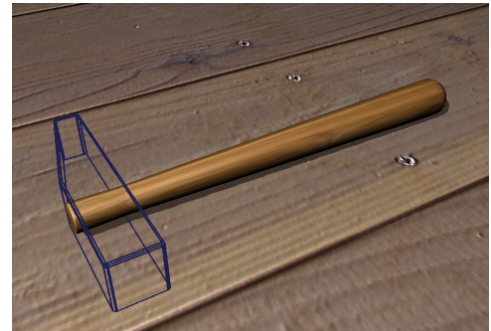
Example

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>
```
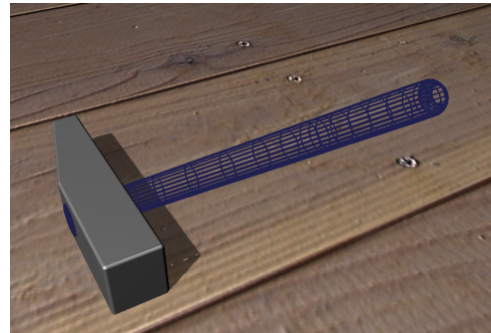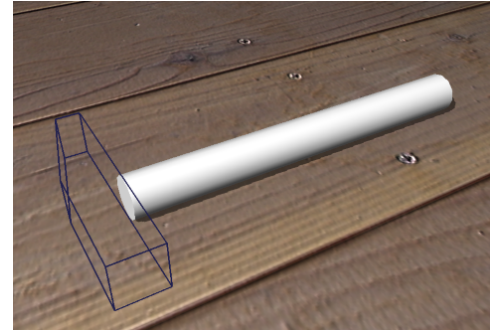


```
  <geometry id="hammerHandleToRender">

    <mesh>
...
    </mesh>
  </geometry>
```



```
  <geometry id="hammerHeadToRender">

    <mesh>
...
    </mesh>
  </geometry>
<library_geometries>
```
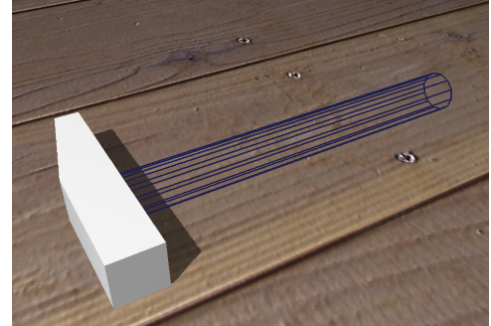
```
<library_physics_models>
  <physics_model id="HammerPhysicsModel">
    <rigid_body sid="HammerHandleRigidBody">
      <technique_common>

        <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

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 6 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 the element. Optional.

Related Elements

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

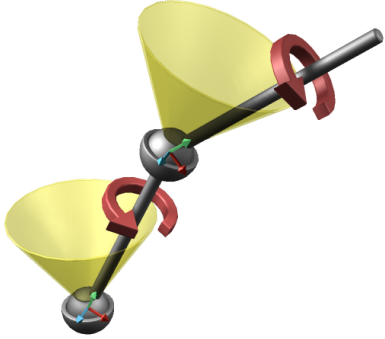|	
Occurrences	Number of elements defined in the schema
Parent elements	**physics_model**
Child elements	See the following subsections
Other	None

Child Elements

Child elements must appear in the following order if present:

| Name/example	Description	Default	Occurrences

`<ref_attachment`
`rigid_body="./SomeRigidBody">`
`  <translate/>`
`  <rotate/>`
`  <extra/>`
`</ref_attachment>`
	Defines the attachment (to a `rigid_body` or a node) to be used as the reference frame.
	The `rigid_body` attribute is a relative reference to a `<rigid_body>` within the same `<physics_model>`.
	The child elements can occur in any order and can each occur 0 or more times.
		N/A	1

`<attachment`
`rigid_body="./SomeRigidBody">`
`  <translate/>`
`  <rotate/>`
`  <extra/>`
`</attachment>`
	Defines an attachment to a rigid body or a node.
	The `rigid_body` attribute is a relative reference to a `<rigid_body>` within the same `<physics_model>`.
	The child elements can occur in any order and can each occur 0 or more times.
		N/A	1

`<technique_common>`	Specifies the common target profile for the `rigid_constraint`. See the following subsection.	N/A	1
`<technique>`	Specifies the target profile for the `rigid_constraint` to allow for multirepresentation.	N/A	0 or more
`<extra>`	User-defined, multirepresentable data.	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>true</enabled>`	If false, the `<constraint>` doesn't exert any force or influence on the rigid bodies.	true	0 or 1
`<interpenetrate>true` `</interpenetrate>`	If true, the attached rigid bodies may inter-penetrate.	false	0 or 1

Two constraints with "swing-cone and twist"-type angular limits:
 	
`<limits>`
`  <linear>`
`    <min sid="...">`
`       0 0 0 </min>`
`    <max sid="...">`
`       0 0 0 </max>`
`  </linear>`
`  <swing_cone_and_twist>`
`    <min sid="...">`
`       -15.0 -15.0 -INF`
`      </min>`
`    <max sid="...">`
`      15.0  15.0  INF`
`      </max>`
`  </swing_cone_and_twist>`
`</limits>`
	The `<limits>` element provides a flexible way to specify the constraint limits (degrees of freedom and ranges).
	This element may contain `<swing_cone_and_twist>` and `<linear>` child elements.
	As new ways to express the limits become standardized, new, strongly typed child elements will be added.
	Until a specific limit description has such XML elements, a custom `<technique>` should be used.
	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.
		`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
	The **sid** attributes are optional. The *x* and *y* limits describe a "swing cone" and the *z* limits describe the "twist angle" range (see diagram on the left). A value of INF and -INF correspond to +/- infinity, indicating that there is no limit along that axis. Limits are expressed in the space of **ref_attachment**.		

```<spring>```
  ```<linear>```
    ```<stiffness>5.4544```
      ```</stiffness>```
    ```<damping>0.4132```
      ```</damping>```
    ```<target_value>3```
      ```</target_value>```
  ```</linear>```
```</spring>```
 	
```<spring>```
  ```<angular>```
    ```<stiffness>5.4544```
      ```</stiffness>```
    ```<damping>0.4132```
      ```</damping>```
    ```<target_value>90```
      ```</target_value>```
    ```</angular>```
```</spring>```
	Spring, based on distance ("**LINEAR**") or angle ("**ANGULAR**"), can have three optional child elements, which must appear in the order shown if used.
	
	The **<stiffness>** (also called spring coefficient) has units of force/distance (or force/angle in degrees).
	
	Expressed in the space of **ref_attachment**.
		stiffness: 1.0
		
		damping: 0.0
		
		target_value: 0.0
		
		("Infinitely rigid" constraint, that is, no spring.)
			0 or 1

Examples
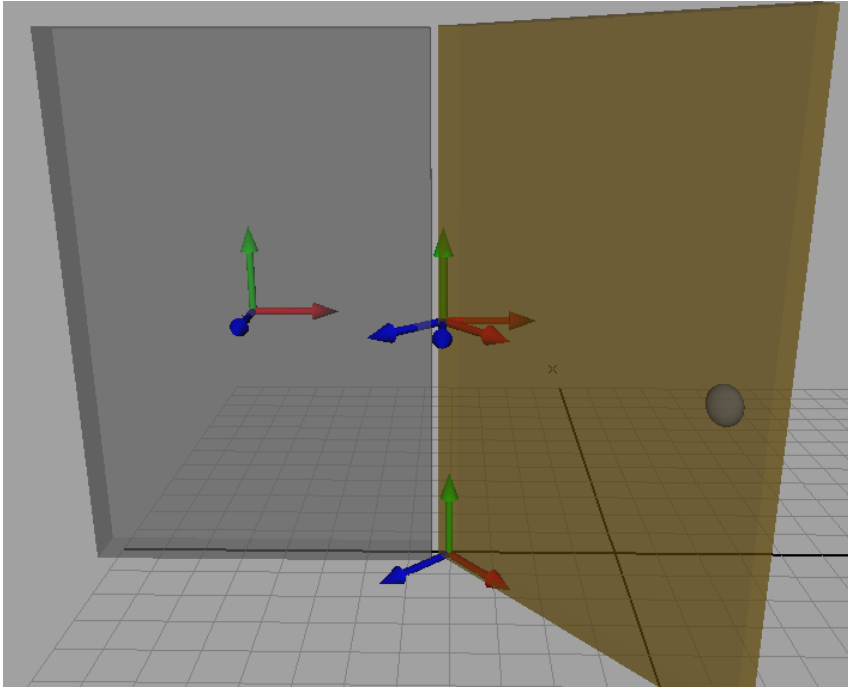
This example demonstrates a door with a hinge. The wall rigid body (in gray, on the right) has its local space frame in its center. The door has its local space on the floor and rotated 45 degrees on the *y* axis. The hinge constraint is limited to rotate +/- 90 degrees on its *y* axis. Each attachment 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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`rigid_body`**'s **`technique_common, instance_rigid_body`**'s **`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>`	If true, the mass is distributed along the surface of the shape. The **`sid`** is optional,	N/A	0 or 1
`<mass sid="..."> 0.5 </mass>`	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>` The `sid` is optional,	A floating-point number specifying the density of the shape	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
inline definition or instance for the geometry: **`<box>...`** **`</box>`** or **`<instance_geometry url=""/>`**	The geometry of the shape. Both inlining and referencing a geometry are allowed. **`<plane>`**, **`<box>`**, **`<sphere>`**, **`<cylinder>`**, **`<tapered_cylinder>`**, **`<capsule>`**, and **`<tapered_capsule>`** are inlined, while other geometry types (**`<mesh>`**, **`<convex_mesh>`**, **`<spline>`** etc.) are referenced via an **`<instance_geometry>`** element.	N/A	1
`<rotate>`, **`<translate>`**	Same as under **`<node>`**	No transforms	0 or more
`<extra>`		N/A	0 or more

Remarks

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

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:

Occurrences	Number of elements defined in the schema
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\>	A float value that represents the radius of the sphere.	N/A	1
\<extra\>		N/A	0 or more

Example

```
<sphere>
  <radius> 1.0 </radius>
</sphere>
```

tapered_capsule

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:

Occurrences	Number of elements defined in the schema
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>	A float value that represents the length of the line segment connecting the centers of the capping hemispheres.	N/A	1
<radius1>	Two float 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.	N/A	1
<radius2>	Two float 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.	N/A	1
<extra>		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

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:

Occurrences	Number of elements defined in the schema
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>	A float value that represents the length of the cylinder along the *y* axis.	N/A	1
<radius1>	Two float 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.	N/A	1
<radius2>	Two float 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.	N/A	1
<extra>		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>
```

This page intentionally left blank.

Chapter 6:
COLLADA FX Reference

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 the **`<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.

Organization of FX Elements

The specific attributes and child elements of each FX element may vary depending on the scope in which it is used. The profile scopes can be grouped as follows:

- Shared (elements are valid in all scopes)
- External to FX
- Common profile
- Cg profile
- Open GL ES profile (abbreviated to GLES)
- GLSL profile

The following lists show which elements fall into which group; an asterisk (*) indicates elements that fall into multiple groups and might have different attributes and content depending on context.

Shared

- **<annotate>**
- **<bind> ***
  - **<bind>/<param> ***
- **<code>**
- **<color_clear>**
- **<color_target>**
- **<compiler_options>**
- **<compiler_target>**
- **<depth_clear>**
- **<depth_target>**
- **<draw>**
- **<effect>**
- **<generator>**
  - **<generator>/<name> ***
- **<include>**
- **<modifier>**
- **<newparam>**
  - **<newparam>/<semantic>**
- **<pass> ***
- Render states
- **<sampler1D>**
- **<sampler2D>**
- **<sampler3D>**
- **<samplerRECT>**
- **<samplerCUBE>**
- **<setparam>**
- **<shader>**
  - **<shader>/<name> ***
- **<stencil_clear>**
- **<stencil_target>**
- **<surface>**
- **<technique> ***
- Value Types

External to FX

- **<bind> ***
- **<bind_material>**
- **<instance_effect>**
- **<instance_material>**
- **<param> ***
- **<technique_hint>**

Profile Common

- **`<blinn>`**
- **`<color>`** (see also "**`common_color_or_texture_type`**")
- **`<param>`** (see also "**`common_float_or_param_type`**")
- **`<lambert>`**
- **`<phong>`**
- **`<profile_COMMON>`**
- **`<texture>`** (see also "**`common_color_or_texture_type`**")
- **`<technique> *`**

Profile Cg

- **`<array>`**
- **`<connect_param>`**
- **`<pass> *`**
- **`<profile_CG>`**
- **`<surface> *`**… note also support **`<generator>`**
- **`<technique> *`**… note also support **`<code>`** and **`<include>`**
- **`<usertype>`**
- **`<technique> *`**

Profile GLSL

- **`<pass> *`**
- **`<profile_GLSL>`**
- **`<surface> *`**… note also support **`<generator>`**
- **`<technique> *`**

Profile OpenGL ES

- **`<alpha>`**
- **`<argument>`**
- **`<constant>`**
- **`<newparam> *`**… note also support **`texture_pipeline`**, **`texture_unit`**, and **`sampler_state`**
- **`<pass> *`**
- **`<profile_GLES>`**
- **`<RGB>`**
- **`<sampler_state>`**
- **`<setparam>*`**… note also support **`texture_pipeline`**, **`texture_unit`**, and **`sampler_state`**
- **`<technique> *`**
- **`<texcombiner>`**
- **`<texenv>`**
- **`<texture_pipeline>`**
- **`<texture_unit>`**

alpha

Introduction

Defines the alpha portion of a `<texture_pipeline>` command. This is a combiner-mode texturing operation.

Concepts

See `<texture_pipeline>` 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.
`scale`	`float`	Infers the use of `glTexEnv`(`TEXTURE_ENV, ALPHA_SCALE, scale)`. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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.		1 to 3

Remarks

Example

See `<texture_pipeline>`.

annotate

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`effect, technique, pass, newparam, setparam, generator, shader`**
Child elements	See the following subsection
Other	None

Child Elements

Name/example	Description	Default	Occurrences
VALUE_TYPES	A strongly typed value that represents the *VALUE* in an object of the form *SYMBOL = VALUE*. Valid types are: **`bool, bool2, bool3, bool4, int, int2, int3, int4, float, float2, float3, float4, float2x2, float3x3, float4x4, string`**	N/A	1

Remarks

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**RGB, alpha**
Child elements	None
Other	None

Remarks

`<argument>` sets up the arguments required for the given operator to be executed.

Example

See `<texture_pipeline>`.

array

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, with multidimensional arrays being declared as arrays 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`newparam, setparam`**
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
`VALUE_TYPE`	See "CG Scope Value Types" in "Value Types" at the end of the chapter.		0 or more
`<connect_param>`	Valid only in **`<newparam>`**.		0 or more
`<usertype>`			0 or more
`<array>`			0 or more

Child Elements in GLSL Scope

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
`VALUE_TYPE`	See "GLSL Scope Value Types" in "Value Types" at the end of the chapter.		0 or more
`<array>`			0 or more

Remarks

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".

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

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 values at execution time. These values can be literal values, constant parameters or uniform parameters. In the case of constant values, these declarations 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. Valid and required only when **<bind>** is a child of **<shader>**.
semantic	**xs:NCName**	Which effect parameter to bind. Valid only when **<bind>** is a child of **<instance_material>**.
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. Valid and required only when **<bind>** is a child of **<instance_material>**.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**shader**, **instance_material**
Child elements	See the following subsection **CG_PARAM_TYPE, GLSL_PARAM_TYPE, param**
Other	None

Child Elements

Note: Exactly one of the child elements **<param>** or a value type must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
<param>	Valid only when **<bind>** is a child of **<shader>**.		see Note
CG_PARAM_TYPE or GLSL_PARAM_TYPE	Valid only when **<bind>** is a child of **<shader>**. See "Value Types" at the end of the chapter.		see Note

Remarks

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>
  ...
<instance_material symbol="RedMat" target="#RedCGEffect">
  <bind semantic="LIGHTPOS0" target="LightNode/translate"/>
</instance_material>
```

bind_material

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 has a particular material, for example,

```
<polygons name="leftarm" count="2445" material="bluePaint">
```

This abstract symbol needs to be bound to a particular material instance, and this is done during the **`<instance_geometry>`** with the **`<bind_material>`** block. The geometry is scanned for material requests by name and actual material are bound to these symbols.

While a material is bound, shader parameters can 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:

Occurrences	Number of elements defined in the schema
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>`	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>`**.		0 or more
`<technique_common>`	See the following subsection.		1
`<technique> (FX)`			0 or more
`<extra>`		N/A	0 or more

Child Elements for bind_material / technique_common

Name/example	Description	Default	Occurrences
\<instance_material\>			1 or more

Remarks

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>
```

blinn

Introduction

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 to the Blinn BRDF approximation.

Concepts

The **`<blinn>`** shader uses the common Blinn shading equation, that is:

```
color = emissive + <ambient>*ambient_light + <diffuse>*max(N . L, 0) +
<specular>*max(H . I, 0)<shininess>
```

Note the use of the half angle vector H which is calculated as halfway between the unit Eye and Light vectors, using the equation `H=(I+L)/2`.

Attributes

The **`<blinn>`** element has no attributes.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**`technique`**
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>`	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

Name/example	Description	Default	Occurrences
`<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`**	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`**	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

Remarks

Example

code

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:

Occurrences	Number of elements defined in the schema
Parent elements	**technique, generator, profile_CG, profile_GLSL**
Child elements	None
Other	None

Remarks

Inlined source code must escape all XML identifier characters, for example, converting "**<**" to "**<**".

Example

```
<code sid="lighting_code">
atrix4x4 mat : MODELVIEWMATRIX;
float4 lighting_fn( varying float3 pos : POSITION,
   ...
</code>
```

color_clear

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 the following attribute:

`index`	`xs:nonNegativeInteger`	Which of the multiple render targets is being set. The default is 0. Optional in GLSL and CG scope; not valid in GLES scope.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`pass`
Child elements	None
Other	None

Remarks

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, and 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

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 the following attributes:

`index`	`xs:nonNegativeInteger`	Indexes one of the Multiple Render Targets. The default is 0. Not valid in GLES scope; optional otherwise.
`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. Not valid in GLES scope; optional otherwise.
`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 `<color_target>` element relates to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	`pass`
Child elements	None
Other	None

Remarks

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 `<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

Introduction

A type that describes color attributes of fixed-function shader elements inside `<profile_COMMON>` effects.

Concepts

Attributes

Elements of type `common_color_or_texture_type` have no attributes.

Related Elements

Elements of type `common_color_or_texture_type` relate to the following elements:

Occurrences	Number of elements defined in the schema
Parent elements	`constant, lambert, phong, blinn`
Child elements	See the following subsection
Other	Elements of type `common_color_or_texture_type`: ambient, diffuse, emission, reflective, specular, transparent (within each of the parent elements)

Child Elements

Note: Exactly one of the child elements `<color>`, `<param>`, or `<texture>` must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<color sid="mySID">`	The value is a literal color, specified by four floating-point numbers in RGBA order.	N/A	see Note
`<param ref="myParam">`	The value is specified by a reference to a previously defined parameter in the current scope that can be cast directly to a `<float4>`.	N/A	see Note

`<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>`.	N/A	see Note

Remarks

Example

common_float_or_param_type

Introduction

A type that describes the scalar attributes of fixed-function shader elements inside **`<profile_COMMON>`** effects.

Concepts

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`constant, lambert, phong, blinn`**
Child elements	See the following subsection
Other	Elements of type **`common_float_or_param_type`**: **`index_of_refraction`**, **`reflectivity`**, **`shininess`**, **`transparency`** (within each of the parent elements)

Child Elements

Note: Exactly one of the child elements **`<float>`** or **`<param>`** must occur. 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>`**	N/A	see Note
`<param ref="myParam">`	The value is represented by a reference to a previously defined parameter that can be directly cast to a floating-point scalar.	N/A	see Note

Remarks

Example

compiler_options

Introduction

Contains command-line options for the 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:

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

Remarks

The **<compiler_options>** value text is the command-line options given to the tool as a text string.

Example

```
<compiler_options> -o3 -finline_level 6 </compiler_options>
```

compiler_target

Introduction

Declares which profile or platform the compiler is targeting this shader for.

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:

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

Remarks

Example

connect_param

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:

Occurrences	Number of elements defined in the schema
Parent elements	**setparam, array, usertype**
Child elements	None
Other	None

Remarks

Example

```
<setparam ref="scene.light[2]">
  <connect_param ref="OverheadSpotlight_B"/>
</setparam>
```

constant

Introduction

Used inside a `<profile_COMMON>` effect, declares a fixed-function pipeline that produces a constantly shaded surface that is independent of lighting.

Concepts

The reflected color is calculated simply as:

```
color = emission
```

Attributes

The `<constant>` element has no attributes.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`technique`
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
`<transparent>`	Declares the color of perfectly refracted light. See `common_color_or_texture_type`	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`	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

Remarks

Example

depth_clear

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 the following attribute:

index	**xs:nonNegativeInteger**	Which of the multiple render targets (MRTs) is being set. The default is 0. Optional in GLSL and CG scope; not valid in GLES scope.

Related Elements

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

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

Remarks

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

```
<depth_clear index="0">0.0</depth_clear>
```

depth_target

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 the following attributes:

`index`	**`xs:nonNegativeInteger`**	Indexes one of the Multiple Render Targets (MRTs). The default is 0. Optional in GLSL and CG scope; not valid in GLES scope.
`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 in GLSL and CG scope; not valid in GLES scope.
`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:

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

Remarks

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.

Example

```
<newparam sid="depthSurface">
  <surface type="2D"/>
</newparam>
<pass>
  <depth_target>depthSurface</depth_target>
  <depth_clear>0.0</depth_clear>
</pass>
```

draw

Introduction

Specifies a user-defined string instructing 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:

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

Remarks

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 **uv**s match.
- FULL_SCREEN_QUAD_PLUS_HALF_PIXEL: Positions are 0,0 to 1,1 and the **uv**s are off by plus ½ of a pixel's **uv** size.

Example

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`library_effects`**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present.

Note: At least one profile must appear, but any number of any types of profiles can be included.

Name/example	Description	Default	Occurrences
`<asset>`			0 or 1
`<annotate>`			0 or more
`<image>`			0 or more

Name/example	Description	Default	Occurrences
`<newparam>`			0 or more
`<profile_CG>`			see Note
`<profile_GLSL>`			see Note
`<profile_COMMON>`			see Note
`<extra>`		N/A	0 or more

Remarks

Example

generator

Introduction

Describes a procedural surface generator.

Concepts

Attributes

The **\<generator\>** element has no attributes.

Related Elements

The **\<generator\>** element relates to the following elements:

Occurrences	Number of elements defined in the schema
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.

Note: At least one of either **\<code\>** or **\<include\>** must occur.

Name/example	Description	Default	Occurrences
\<annotate\>			0 or more
\<code\>			see Note
\<include\>			see Note
\<name\>			1
\<setparam\>			0 or more

Remarks

Example

include

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:

Occurrences	Number of elements defined in the schema
Parent elements	`technique, generator, profile_CG, profile_GLSL`
Child elements	None
Other	None

Remarks

Example

```
<include sid="ShinyShader" url="file://assets/source/shader.glsl"/>
```

instance_effect

Introduction

Declares the instantiation of a COLLADA material resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

Attributes

The **<instance_effect>** 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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**material, render**
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_hint>			0 or more
<setparam>			0 or more
<extra>			0 or more

Remarks

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

Introduction

Declares the instantiation of a COLLADA material resource.

Concepts

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

Each instance of the object may be unique or share data with other instances. A unique instance has its own copy of the object's data and can be manipulated independently. A 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*.

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 object to instantiate. Required. Refers to a local instance using a relative URI fragment identifier that begins with the "#" character. The fragment identifier is an XPointer shorthand pointer that consists of the URI of the element to instantiate. Refers to an external reference using an absolute or relative URL when it contains a path to another resource.
`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:

Occurrences	Number of elements defined in the schema
Parent elements	**`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
\<bind\>			0 or more
\<bind_vertex_input\>	The **\<bind_vertex_input\>** element binds vertex inputs to effect parameters upon instantiation. This 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_attribute\>**. See the following subsection for details.		0 or more
\<extra\>			0 or more

Attributes for instance_material/bind_vertext_input

\<bind_vertex_input\> 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.

When locating by semantic, search in the following order:

- COLLADA FX parameters
- Directly in the shader by semantic if supported by the shader
- Directly in the shader by param-name if the semantic is not supported

Remarks

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

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>
```

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 **<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>
```

lambert

Introduction

Used inside a `<profile_COMMON>` effect, declares a fixed-function pipeline that produces a constantly shaded surface that is independent of lighting.

Concepts

The reflected color is calculated simply as:

```
color = emission
```

Attributes

The `<lambert>` element has no attributes.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`technique`
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>`	Declares the amount of ambient light emitted from the surface of this object. See `common_color_or_texture_type`		0 or 1
`<diffuse>`	Declares the amount of light diffusely reflected from the surface of this object. See `common_color_or_texture_type`		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`	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`	N/A	0 or 1

Name/example	Description	Default	Occurrences
`<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

Remarks

Example

library_effects

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or 1
<effect>			1 or more
<extra>			0 or more

Remarks

Example

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

```
<library_effects>
  <effect id="fullscreen_effect1">
     ...
  </effect>
</library_effects>
```

library_materials

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
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>			0 or 1
<material>			1 or more
<extra>			0 or more

Remarks

Example

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

```
<library_materials>
  <material id="mat1">
    ...
  </material >
```

```
    <material id="mat2">
      ...
    </material>

</library_materials>
```

material

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 the element. Optional.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`library_materials`
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>`			0 or 1
`<instance_effect>`			1
`<extra>`			0 or more

Remarks

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

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:

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

Remarks

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**shader, generator**
Child elements	None
Other	None

Remarks

The **<name>** element contains the name of the entry-point function; **source** specifies where the entry point exists.

Example

newparam

Introduction

Creates a new, named **`<param>`** 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`effect, technique, profile_CG, profile_COMMON, profile_GLSL, profile_GLES`**
Child elements	See the following subsections
Other	None

Child Elements in profile_CG and profile_CG / technique

Child elements must appear in the following order if present.

Note: Exactly one of the child elements **`<array>`**, **`<usertype>`**, or **`VALUE_TYPE`** must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<annotate>`			0 or more
`<semantic>`			0 or 1
`<modifier>`			0 or 1
`<array>`			see Note
`VALUE_TYPE`	See the "Value Types" section for valid value types in CG scope.		see Note
`<usertype>`			see Note

Child Elements in profile_GLSL and profile_GLSL / technique

Child elements must appear in the following order if present.

Note: Exactly one of the child elements `<array>` or `VALUE_TYPE` must occur. They are mutually exclusive.

Name/example	Description	Default	Occurrences
`<annotate>`			0 or more
`<semantic>`			0 or 1
`<modifier>`			0 or 1
`<array>`			see Note
`VALUE_TYPE`	See the "Value Types" section for valid value types in GLSL scope.		see Note

Child Elements in profile_GLES, profile_GLES/technique, and effect

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
`<annotate>`			0 or more
`<semantic>`			0 or 1
`<modifier>`			0 or 1
`VALUE_TYPE`	See the 'Value Types" section for valid value types in GLES; the COLLADA scope values apply in `<effect>`.		1

Child Elements in profile_COMMON and profile_COMMON/technique

Child elements must appear in the following order if present.

Name/example	Description	Default	Occurrences
`<semantic>`			0 or 1
`VALUE_TYPE`	Only `float`, `float2`, `float3`, `surface`, and `sampler2D` are valid.		1

Remarks

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

Introduction

References a predefined parameter in shader binding declarations.

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 **`<param>`** element has the following attributes in **`<bind_material>`**:

name	xs:NCName	
sid	xs:NCName	
semantic	ns:NMTOKEN	
type	ns:NMTOKEN	

The **`<param>`** element has the following attributes in all other parents:

ref	xs:NCName	Required.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	**bind, bind_material, texture1D, texture2D, texture3D, textureCUBE, textureRECT, textureDEPTH**
Child elements	None
Other	None

Remarks

Example

```
<shader stage="VERTEX">
  <compiler_target>ARBVP1</compiler_target>
  <name source="ThinFilm2">main</name>
  <bind symbol="lightpos">
    <param ref="LightPos_03"/>
  </bind>
</shader>
```

pass

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:

Occurrences	Number of elements defined in the schema
Parent elements	**technique**
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>			0 or more
<color_target>			0 or 1
<depth_target>			0 or 1
<stencil_target>			0 or 1
<color_clear>			0 or 1
<depth_clear>			0 or 1
<stencil_clear>			0 or 1
<draw>			0 or 1
RENDER_STATES	See "Render States" subsection.		0 or 1
<extra>			0 or more

Child Elements in CG, GL, or GLSL Scope

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
\<annotate>			0 or more
\<color_target>			0 or more
\<depth_target>			0 or more
\<stencil_target>			0 or more
\<color_clear>			0 or more
\<depth_clear>			0 or more
\<stencil_clear>			0 or more
\<draw>			0 or more
RENDER_STATES	See "Render States" subsection. In GL scope, cannot appear if **\<shader>** is used.		0 or more
\<shader>	In GL scope, cannot appear if a **render_state** is used.		0 or more
\<extra>		N/A	0 or more

Remarks

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. 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 except for differences noted for the GLES profile.

Render states	Valid values, value types, and index attribute	GLES Differences

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, DEST_ALPHA, ONE_MINUS_DEST_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	Not in GLES

blend_equation	**FUNC_ADD, FUNC_SUBTACT, FUNC_REVERSE_SUBTRACT, MIN, MAX**	Not in GLES

blend_equation_separate
    rgb
    alpha
	Same as blend_equation values	Not in GLES

color_material
    face
 	
    mode
	FRONT, BACK, FRONT_AND_BACK
	
	EMISSION, AMBIENT, DIFFUSE, SPECULAR, AMBIENT_AND_DIFFUSE
		Not in GLES

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**	Not in GLES
front_face	**CW, CCW**	
light_model_color_control	**SINGLE_COLOR, SEPARATE_SPECULAR_COLOR**	Not in GLES
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
		Not in GLES

shade_model	**FLAT, SMOOTH**	

Render states	Valid values, value types, and index attribute	GLES Differences

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, DECT_WRAP	

stencil_func_separate
    front
    back
    ref
    mask
	(For front and back)
	NEVER, LESS, LEQUAL, EQUAL, GREATER, NOTEQUAL, GEQUAL, ALWAYS
	Unsigned byte
	Unsigned byte
		Not in GLES

stencil_mask_separate
    face
    mask
	FRONT, BACK, FRONT_AND_BACK
	Unsigned byte
		Not in GLES

light_enable	Boolean Index attribute specifies which light.	
light_ambient	**float4** Index attribute specifies which light	
light_diffuse	**float4** Index attribute specifies which light	
light_specular	**float4** Index attribute specifies which light	
light_position	**float4** Index attribute specifies which light	
light_constant_attenuation	**float** Index attribute specifies which light	
light_linear_attenuation	**float** Index attribute specifies which light	
light_quadratic_attenuation	**float** Index attribute specifies which light	
light_spot_cutoff	**float** Index attribute specifies which light	
light_spot_direction	**float3** Index attribute specifies which light	
light_spot_exponent	**float** Index attribute specifies which light	
texture1D	**sampler1D** type Index attribute specifies which texture unit.	Not in GLES
texture2D	**sampler2D** type Index attribute specifies which texture unit.	Not in GLES (see **<texture_pipeline>**)
texture3D	**sampler3D** type Index attribute specifies which texture unit.	Not in GLES
textureCUBE	**samplerCUBE** type Index attribute specifies which texture unit.	Not in GLES
textureRECT	**samplerRECT** type Index attribute specifies which texture unit.	Not in GLES

Render states	Valid values, value types, and index attribute	GLES Differences
textureDEPTH	**samplerDEPTH** type Index attribute specifies which texture unit.	Not in GLES
texture1D_enable	Boolean Index attribute specifies which texture unit.	Not in GLES
texture2D_enable	Boolean Index attribute specifies which texture unit.	Not in GLES (see **\<texture_pipeline>**)
texture3D_enable	Boolean Index attribute specifies which texture unit.	Not in GLES
textureCUBE_enable	Boolean Index attribute specifies which texture unit.	Not in GLES
textureRECT_enable	Boolean Index attribute specifies which texture unit.	Not in GLES
textureDEPTH_enable	Boolean Index attribute specifies which texture unit.	Not in GLES
texture_env_color	**float4** Index attribute specifies which texture unit.	Not in GLES (see **\<texture_pipeline>**)
texture_env_mode	**string** Index attribute specifies which texture unit.	Not in GLES (see **\<texture_pipeline>**)
clip_plane	**float4** Index attribute specifies which clip plane.	
clip_plane_enable	Boolean Index attribute specifies which clip plane.	
blend_color	**float4**	
clear_color	**float4**	
clear_stencil	**int**	
clear_depth	**float**	
color_mask	**bool4**	
depth_bounds	**float2**	
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**	
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**	

Render states	Valid values, value types, and index attribute	GLES Differences
`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	Not in GLES
`blend_enable`	Boolean	
`color_logic_op_enable`	Boolean	
`cull_face_enable`	Boolean	
`depth_bounds_enable`	Boolean	Not in GLES
`depth_clamp_enable`	Boolean	Not in GLES
`depth_test_enable`	Boolean	
`dither_enable`	Boolean	
`fog_enable`	Boolean	
`light_model_local_viewer_enable`	Boolean	Not in GLES
`light_model_two_side_enable`	Boolean	
`line_smooth_enable`	Boolean	Not in GLES
`line_stipple_enable`	Boolean	Not in GLES
`logic_op_enable`	Boolean	
`multisample_enable`	Boolean	
`normalize_enable`	Boolean	
`point_smooth_enable`	Boolean	Not in GLES
`polygon_offset_fill_enable`	Boolean	
`polygon_offset_line_enable`	Boolean	Not in GLES
`polygon_offset_point_enable`	Boolean	Not in GLES
`polygon_smooth_enable`	Boolean	Not in GLES
`polygon_stipple_enable`	Boolean	Not in GLES
`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	
`gl_hook_abstract`	An element to allow for render states from GL extensions to be added.	Not in GLES
`texture_pipeline`	String – the name of the `<texture_pipeline>` parameter	GLES only
`texture_pipeline_enable`	Boolean	GLES only

phong

Introduction

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.

Concepts

The `<phong>` shader uses the common Phong shading equation, that is:

```
color = emissive + <ambient>*ambient_light + <diffuse>*max(N.L, 0) +
<specular>*max(R.I, 0)<shininess>
```

Note the use of the perfect reflection vector `R` rather than the half-angle vector `H` that is used in Blinn shading.

Attributes

The `<phong>` element has no attributes.

Related Elements

Occurrences	Number of elements defined in the schema
Parent elements	**technique**
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>`	Declare the amount of light emitted from the surface of this object. See `common_color_or_texture_type`	N/A	0 or 1
`<ambient>`	Declare the amount of ambient light emitted from the surface of this object. See `common_color_or_texture_type`	N/A	0 or 1
`<diffuse>`	Declare the amount of light diffusely reflected from the surface of this object. See `common_color_or_texture_type`	N/A	0 or 1
`<specular>`	Declare the color of light specularly reflected from the surface of this object. See `common_color_or_texture_type`	N/A	0 or 1
`<shininess>`	Declare the specularity or roughness of the specular reflection lobe. See `common_float_or_param_type`	N/A	0 or 1
`<reflective>`	Declare the color of a perfect mirror reflection. See `common_color_or_texture_type`	N/A	0 or 1

Name/example	Description	Default	Occurrences
\<reflectivity\>	Declare 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\>	Declare the color of perfectly refracted light. See **common_color_or_texture_type**	N/A	0 or 1
\<transparency\>	Declare 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**	N/A	0 or 1
\<index_of_refraction\>	Declare the index of refraction for perfectly refracted light as a single scalar index. See **common_float_or_param_type**	N/A	0 or 1

Remarks

Example

profile_CG

Introduction

Opens a block of platform-specific data types and **`<technique>`** declarations.

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.

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:

Occurrences	Number of elements defined in the schema
Parent elements	**effect**
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>`			0 or 1
`<code>`			0 or more
`<include>`			0 or more
`<image>`			0 or more
`<newparam>`			0 or more
`<technique>` (FX)			1 or more
`<extra>`		N/A	0 or more

Remarks

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

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.

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:

Occurrences	Number of elements defined in the schema
Parent elements	**effect**
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present: xx

Name/example	Description	Default	Occurrences
<asset>		N/A	0 or 1
<image id="myID" name="./BrickTexture" format="R8G8B8A8" height="64" width="128" depth-"1">	Declares a standard COLLADA image resource.	N/A	0 or more

<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.	N/A	0 or more

<technique> (FX)	Declares the only technique for this effect. This node contains **<asset>**, **<image>**, and **<extra>**, along with one of **<constant>**, **<lambert>**, **<phong>**, or **<blinn>**.	N/A	1
<extra>		N/A	0 or more

Remarks

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>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

Introduction

Opens a block of platform-specific data types and **<technique>** declarations.

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.

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:

Occurrences	Number of elements defined in the schema
Parent elements	**effect**
Child elements	See the following subsection
Other	None

<newparam> contains new objects for GLES. GLES-specific **VALUE_TYPES** for **<newparam>** include **<texture_pipeline>** and **<sampler_state>**.

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<asset>		N/A	0 or 1
<image id="myID" name="./BrickTexture" format="R8G8B8A8" height="64" width="128" depth-"1">	Declare a standard COLLADA image resource	N/A	0 or more

Name/example	Description	Default	Occurrences

`<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.	N/A	0 or more

`<technique> (FX)`	Declare the one and only technique for this effect. This node contains `<asset>`, `<image>` and `<extra>`, along with one of `<constant>`, `<lambert>`, `<phong>,` or `<blinn>`.	N/A	1 or more
`<extra>`		N/A	0 or more

Remarks

Example

profile_GLSL

Introduction

Opens a block of platform-specific data types and **`<technique>`** declarations.

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.

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`effect`**
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>`		N/A	0 or 1
`<code>`			0 or more
`<include>`			0 or more
`<image id="myID" name="./BrickTexture" format="R8G8B8A8" height="64" width="128" depth-"1">`	Declare a standard COLLADA image resource	N/A	0 or more

`<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.	N/A	0 or more

Name/example	Description	Default	Occurrences
`<technique>` (FX)	Declare the one and only technique for this effect. This node contains `<asset>`, `<image>`, and `<extra>`, along with one of `<constant>`, `<lambert>`, `<phong>,` or `<blinn>`.	N/A	1 or more
`<extra>`		N/A	0 or more

Remarks

Example

RGB

Introduction

Defines the RGB portion of a **<texture_pipeline>** command. This is a combiner-mode texturing operation.

Concepts

See **<texture_pipeline>** 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***)**
scale	**float**	Infers the use of **glTexEnv(TEXTURE_ENV, RGB_SCALE,** *scale***)**

Related Elements

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

Occurrences	Number of elements defined in the schema
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.		1 to 3

Remarks

Example

See **<texture_pipeline>**.

sampler1D

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`newparam`, `setparam`, `usertype`, `array`, `bind`**
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>`			1
`<wrap_s>`		WRAP	0 or 1
`<minfilter>`		NONE	0 or 1
`<magfilter>`		NONE	0 or 1
`<mipfilter>`		NONE	0 or 1
`<border_color>`		0	0 or 1
`<mipmap_maxlevel>`		0	0 or 1
`<mipmap_bias>`		0	0 or 1
`<extra>`		N/A	0 or more

Remarks

Example

sampler2D

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`newparam`**, **`setparam`**, **`usertype`**, **`array`**, **`bind`**
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>`			1
`<wrap_s>`		WRAP	0 or 1
`<wrap_t>`		WRAP	0 or 1
`<minfilter>`		NONE	0 or 1
`<magfilter>`		NONE	0 or 1
`<mipfilter>`		NONE	0 or 1
`<border_color>`		0	0 or 1
`<mipmap_maxlevel>`		255	0 or 1
`<mipmap_bias>`		0	0 or 1
`<extra>`		N/A	0 or more

Remarks

Example

sampler3D

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:

Occurrences	Number of elements defined in the schema
Parent elements	**newparam**, **setparam**, **usertype**, **array**, **bind**
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>			1
<wrap_s>		WRAP	0 or 1
<wrap_t>		WRAP	0 or 1
<wrap_p>		WRAP	0 or 1
<minfilter>		NONE	0 or 1
<magfilter>		NONE	0 or 1
<mipfilter>		NONE	0 or 1
<border_color>		0	0 or 1
<mipmap_maxlevel>		255	0 or 1
<mipmap_bias>		0	0 or 1
<extra>		N/A	0 or more

Remarks

Example

samplerCUBE

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`newparam`, `setparam`, `usertype`, `array`, `bind`**
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>`			1
`<wrap_s>`		WRAP	0 or 1
`<wrap_t>`		WRAP	0 or 1
`<wrap_p>`		WRAP	0 or 1
`<minfilter>`		NONE	0 or 1
`<magfilter>`		NONE	0 or 1
`<mipfilter>`		NONE	0 or 1
`<border_color>`		0	0 or 1
`<mipmap_maxlevel>`		255	0 or 1
`<mipmap_bias>`		0	0 or 1
`<extra>`		N/A	0 or more

Remarks

Example

samplerDEPTH

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`newparam`**, **`setparam`**, **`usertype, array, bind`**
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>`			1
`<wrap_s>`		WRAP	0 or 1
`<wrap_t>`		WRAP	0 or 1
`<minfilter>`		NONE	0 or 1
`<magfilter>`		NONE	0 or 1
`<extra>`		N/A	0 or more

Remarks

Example

samplerRECT

Introduction

Declares a two-dimensional texture sampler.

Concepts

Attributes

The `<samplerRECT>` element has no attributes.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`newparam, setparam, usertype, array, bind`
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>`			1
`<wrap_s>`		WRAP	0 or 1
`<wrap_t>`		WRAP	0 or 1
`<wrap_p>`		WRAP	0 or 1
`<minfilter>`		NONE	0 or 1
`<magfilter>`		NONE	0 or 1
`<mipfilter>`		NONE	0 or 1
`<border_color>`		0	0 or 1
`<mipmap_maxlevel>`		255	0 or 1
`<mipmap_bias>`		0	0 or 1
`<extra>`		N/A	0 or more

Remarks

Example

sampler_state

Introduction

Provides a two-dimensional texture sampler state for `<profile_GLES>`. This is a bundle of sampler-specific states that will be referenced by one or more `<texture_pipeline>`s

Concepts

Attributes

The `<sampler_state>` element has the following attribute:

`sid`	`xs:NCName`	Identifier for this state. Optional in `<newparam>` and `<setparam>`; not valid in `<texture_unit>`.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`newparam`, `setparam`, `texture_unit`
Child elements	None in `<texture_unit>`. For `<newparam>` and `<setparam>`, see the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
`<wrap_s>`		REPEAT	0 or 1
`<wrap_t>`		REPEAT	0 or 1
`<minfilter>`		NONE	0 or 1
`<magfilter>`		NONE	0 or 1
`<mipfilter>`		NONE	0 or 1
`<mipmap_maxlevel>`		255	0 or 1
`<mipmap_bias>`		0	0 or 1
`<extra>`			0 or more

Remarks

Example

semantic

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:

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

Remarks

There is currently no standard set of semantics.

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

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	Attempts to reference the predefined parameter that will have its value set. Required.
program	xs:NCName	Optional 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`technique, generator, instance_effect`**
Child elements	See the following subsection
Other	None

Child Elements in profile_GLSL/technique

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
`<annotate>`			0 or more
`VALUE_TYPES`	See GLSL value types in the "Value Types" section.		1
`<usertype>`			
`<array>`			

Child Elements in profile_CG/technique

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
VALUE_TYPES	See CG value types in the "Value Types" section.		1
<usertype>			
<array>			
<connect_param>			

Child Elements in profile_GLES/technique

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>			0 or more
VALUE_TYPES	See GLES value types in the "Value Types" section.		1

Child Elements in <instance_effect>

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
VALUE_TYPES	See core value types in the "Value Types" section.		1

Child Elements in <generator>

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
<annotate>			0 or more
VALUE_TYPES	See the "Value Types" section; either GLSL or CG value types depending on context		1

Remarks

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

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:

Occurrences	Number of elements defined in the schema
Parent elements	**pass**
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\>			0 or more
\<compiler_target\>			0 or 1
\<name\>			1
\<compiler_options\>			0 or 1

Name/example	Description	Default	Occurrences
<bind>			0 or more

Remarks

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

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 the following attribute:

`index`	`xs:nonNegativeInteger`	Which of the multiple render targets is being set. The default is 0. Optional in the GLSL and CG profiles; not valid in the GLES profile.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	`pass`
Child elements	None
Other	None

Remarks

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

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 the following attributes:

`index`	`xs:nonNegativeInteger`	Indexes one of the Multiple Render Targets. Optional in the GLSL and CG profiles; not valid in the GLES profile.
`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 in the GLSL and CG profiles; not valid in the GLES profile.
`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:

Occurrences	Number of elements defined in the schema
Parent elements	`pass`
Child elements	None
Other	None

Remarks

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.

Example

```
<newparam sid="surfaceTex">
  <surface type="2D"/>
</newparam>
<pass>
  <stencil_target>surfaceTex</stencil_target>
</pass>
```

surface

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 a MIP-mapped image with N prefiltered levels, or by joining six square textures into a cube map.

`<surface>` objects have a data format describing the size and layout of fields in each pixel, can be sized in absolute numbers of pixels using **`<size>`** or as some fractional size of the viewport using **`<viewport_ratio>`**, and 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`COMMON: newparam, setparam`** **`CG: newparam, setparam, array, bind, usertype`** **`GLES: newparam, setparam, texture_unit`** **`GLSL: newparam, setparam, array, bind`**
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 "Remarks" subsection.		0 or 1
`<format>`	Contains a string representing the profile- and platform-specific texel format that you want this surface to use. If this element is not specified, then the application will use a common format R8G8B8A8 with linear color gradient, not sRGB.		0 or 1

Name/example	Description	Default	Occurrences
`<format_hint>`	If the exact format cannot be resolved using `<format>` then the `<format_hint>` describes the important features of the format so that the application can select a compatible or similar format. See the following subsection, "Child Elements in `<surface>`/`<format_hint>`."		0 or 1
`<size>`	If specified, the surface is sized to these exact dimensions. Cannot appear if `<viewport_ratio>` is used.	0 0 0	0 or 1
`<viewport_ratio>`	If specified, the surface is sized to a dimension based on this ratio of the viewport's dimensions in pixels. Cannot appear if `<size>` is used.	1	0 or 1
`<mip_levels>`	If specified, the surface should contain the specified number of MIP levels. If this element is not present, it is assumed that all miplevels 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. If the value is 0, the result is the same as if mip_levels was unspecified, that is, all possible `mip_levels` will exist.	0	0 or 1
`<mipmap_generate>`	Boolean. Default is false, in which case, if not all subsurfaces are initialized because you have not provided mipmap levels, the generated surface will have profile- and platform-specific behavior. If true, `mipmap_generate` is responsible for initializing the remainder of the subsurfaces.	false	0 or 1
`<extra>`		N/A	0 or more
`<generator>`	Valid in CG, GLES and GLSL scopes. Specifies a procedural surface generator.		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>`	The per-texel layout of the format. The length of the string indicates how many channels there are and the letter represents the name of the channel. There are typically 0 to 4 channels. Valid enumeration values are: **RGB**: RGB color map. **RGBA**: RGB color + Alpha map often used for color + transparency or other things packed into channel A like 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 3component displacement maps. **XYZW**: Typically used for normal maps where W is the depth for relief or parallax mapping	N/A	1

Name/example	Description	Default	Occurrences
`<range>`	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. Valid enumeration values are: **SNORM**: Format is representing a decimal value that remains within the -1 to 1 range. Implementation could be integer, fixed-point, or float. **UNORM**: Format is representing a decimal value that remains within the 0 to 1 range. Implementation could be integer, fixed point, or float. **SINT**: Format is representing signed integer numbers. (ex. 8bits = -128 to 127) **UINT**: Format is representing unsigned integer numbers. For example, 8bits = 0 to 255. **FLOAT**: Format should support full floating-point ranges. HIGH precision is expected to be 32 bits. MID precision may be 16 to 32 bit. LOW precision is expected to be 16 bits.	N/A	1
`<precision>`	Each channel of the texel has a precision. Typically, these are linked together. An exact format may lower the precision of an individual channel but applying a higher precision by linking the channels together may still convey the same information. 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>`	Additional hints about data relationships and other things to help the application pick the best format. 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. Considering the best compression based on desired `<channels>`, `<range>`, `<precision>`, and `<option>`s.	N/A	0 or more
`<extra>`		N/A	0 or more

Remarks

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.

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.

Initialization Options

The optional *initialization option* can be any one of the following:

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>`.
`<init_as_target>`	Initializes this surface as a target for depth, stencil, or color. It does not need image data. Surface should not have `<mipmap_generate>` when using this.

`<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.
	Choose one of:
	`<all>`: Initializes the surface with one compound image such as DDS.
	`<primary>`: Initializes all primary mip level 0 subsurfaces with one compound image such as DDS. Use of this element expects the surface to have element `<mip_levels>=0` or `<mipmap_generate>`. Its subelement `<order>` occurs 0 or 6 times; if the image does not natively describe the face ordering then this series of ordered elements describes which face the index belongs to.
	`<face>`: Occurs 6 times. Initializes each face mipchain with one compound image such as DDS.

`<init_volume>`
   `<all ref=… />`
   `<primary ref=… />`
`</init_volume>`
	Initializes this surface with a 3D from a compound image such as DDS.
	Choose one of:
	`<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.

`<init_from`
`mip=…`
`slice=…`
`face=… />`
	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 2D image, not a complex compound image such as DDS. If not all subsurfaces are initialized, it is invalid and will result in profile- and platform-specific behavior unless `<mipmap_generate>` is responsible for initializing the remaining subsurfaces.

Use of <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 format string.
- If `<format_hint>` exists, use features and characteristics described there to select an appropriate format
- If initialized from `<images>`, use something compatible with the `<image>`s.
- Otherwise, there is no information to help the application, so use a well-known format such as R8G8B8A8_UNORM or R16G16B16A16_UNORM.

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)

Introduction

Holds a description of the textures, samplers, shaders, parameters, and passes necessary for rendering this effect using one method.

For **\<technique\>** in non-FX elements, 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:

Occurrences	Number of elements defined in the schema
Parent elements	**profile_CG, profile_COMMON, profile_GLSL, profile_GLES**
Child elements	**asset, annotate, blinn, include, newparam, setparam, pass**
Other	None

Child Elements

Child elements must appear in the following order if present:

Name/example	Description	Default	Occurrences
\<annotate\>			
\<asset\>			
\<blinn\>			
\<include\>			
\<newparam\>			
\<pass\>			
\<setparam\>			

Remarks

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

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:NCName`	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:

Occurrences	Number of elements defined in the schema
Parent elements	`instance_effect`
Child elements	None
Other	None

Remarks

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

Introduction

Defines a **<texture_pipeline>** command. This is a combiner-mode texturing operation.

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:

Occurrences	Number of elements defined in the schema
Parent elements	**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>	A **float4** for **glTexEnv(TEXTURE_ENV, TEXTURE_ENV_COLOR, *value*)**.		0 or 1
<RGB>	Sets up the RGB component of the texture combiner command.		0 or 1
<alpha>	Sets up the alpha component of the texture combiner command.		0 or 1

Remarks

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

Introduction

Defines a **`<texture_pipeline>`** command. It is a simple noncombiner mode of texturing operations.

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:

Occurrences	Number of elements defined in the schema
Parent elements	**`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>`	A **`float4`** for **`glTexEnv(TEXTURE_ENV, TEXTURE_ENV_COLOR, value)`**.		0 or 1

Remarks

Infers a call to **`glTexEnv(TEXTURE_ENV, TEXTURE_ENV_MODE, operator)`** for the texture unit to which it is assigned.

Example

See **`<texture_pipeline>`**.

texture_pipeline

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 for **`newparam`** and **`setparam`**.
		Not valid for pass.
`param`		Optional for pass.
		Not valid for **`newparam`** and **`setparam`**.

Related Elements

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

Occurrences	Number of elements defined in the schema
Parent elements	In GLES scope: **`newparam, setparam, pass`** (render state)
Child elements	See the following subsection
Other	None

Child Elements in newparam and setparam

Child elements can appear in any order:

Name/example	Description	Default	Occurrences
\<texcombiner>			0 or more
\<texenv>			0 or more
\<extra>		N/A	0 or more

Child Elements in pass

Child elements can appear in any order:

Name/example	Description	Default	Occurrences

\<value sid=" ">
  \<texcombiner/>
  \<texenv/>
  \<extra/>
	Child elements are optional, and can occur in any order and as often as desired.
	
	The sid attribute is optional.		

Remarks

Each element is a command type.

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

Introduction

Defines a texture unit. These texture units will be mapped to hardware texture units based on their usage in `<texture_pipeline>` commands.

Concepts

There may be defined more texture units than available hardware but, as long as a `<texture_pipeline>` uses less than the hardware limit at one time, the fragment shader is valid. (Conformance is also dependent 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:

Occurrences	Number of elements defined in the schema
Parent elements	`setparam, newparam`
Child elements	See the following subsection
Other	None

Child Elements

Child elements must appear in the following order if present: xx

Name/example	Description	Default	Occurrences
`<surface>`	References the surface parameter whose surface will be used for textures.		0 or 1
`<sampler_state>`	Includes a semantic attribute that provides a semantic name for the texcoord array that the texture unit must use. The array is mapped here using `<bind_material>`.		0 or 1
`<texcoord>`	Includes a semantic attribute that provides a semantic name for the texcoord array that the texture unit must use. The array is mapped here using `<bind_material>`.		0 or 1
`<extra>`		N/A	0 or more

Remarks

Example

See `<texture_pipeline>`.

usertype

Introduction

Creates an instance of a structured class.

Concepts

Interface objects were introduced as part of the Cg 1.4 language specification, and 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:

Occurrences	Number of elements defined in the schema
Parent elements	**`newparam (cg_newparam), array, usertype, setparam`**
Child elements	See the following subsection
Other	None

Child Elements

Child elements can appear in any order.

Name/example	Description	Default	Occurrences
`<array>`			0 or more
`VALUE_TYPE`	See the CG value types in the "Value Types" section.		0 or more
`<connect_param>`			0 or more
`<usertype>`			0 or more
`<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.		1 or more

Remarks

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.

COLLADA Scope Core Value Types

```
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 Scope Value Types

```
bool, bool2, bool3, bool4, int, int2, int3, int4, float, float2, float3,
float4, float2x2, float3x3, float4x4, surface, sampler1D, sampler2D,
sampler3D, samplerCUBE, samplerRECT, samplerDEPTH, enum
```

CG Scope Value Types

```
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 Scope Value Types

```
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
```

Appendix A:
COLLADA Example

Example: Cube

This is a simple example of a COLLADA instance document that describes a simple white cube.

```xml
<?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>
```

This page intentionally left blank.

Glossary

- **Attributes** – 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.

- **Comments** – *XML* files can contain comment text. Comments are identified by special markup of the following form:

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

- **DCC** – Digital content creation.

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

- **Name** – The name of an *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.

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

    ```
    <tagName>
    ```

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

    ```
    </tagName>
    ```

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

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

- **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.

This page intentionally left blank.

Index