COLLADA DOM 1.4.0 – Programming Guide

Please note that the specifications contained in this document are preliminary and subject to change without prior notice.

Table of Contents

1 Introduction

The COLLADA Document Object Model (DOM) is an application programming interface (API) that provides a C++ object representation of a COLLADA XML instance document.

Background

In August 2003, Sony Computer Entertainment Inc. (SCEI) initiated an industry collaboration design of an open, nonproprietary 3D digital asset schema. The collaboration, dubbed COLLADA™, quickly gained momentum as a core group of industry partners contributed to a first version of the digital asset schema.

This core group chose the XML Schema Language for the COLLADA specification format to promote openness. This choice builds on the broad acceptance of XML as an interchange format and on the wide availability of XML-based tools.

To validate the design work and to promote real applications of the COLLADA specification, Emdigo, Inc. designed and implemented the COLLADA DOM and reference importer/exporter.

Overview

The COLLADA DOM is a comprehensive framework for the development of COLLADA applications. The DOM provides a C++ programming interface to load, query, and translate COLLADA instance data. The DOM loads COLLADA data into a runtime database consisting of structures that mirror those defined in the COLLADA schema. These runtime structures are auto-generated from the current schema, eliminating inconsistency and error.

Developers interact with the runtime database through a query API and plugin points for integration code. Understanding the details of the schema and instance data formats are not required to successfully integrate with the API. Developers can also directly use the data structures loaded into the COLLADA runtime database within their application.

Initial customers include the COLLADA core partners and any industry participants who wish to contribute to or leverage the ongoing collaboration.

Design Considerations

The COLLADA DOM takes into account the following design considerations, which provide some of the desirable features of the DOM.

Simple Data Transformation

The COLLADA DOM provides a means for users to write translation code to transform data loaded into the COLLADA runtime database into data structures native to their own tools or engines. The API provides plugin points to eliminate the need to understand the underlying framework in order to write this translation.

Replaceable Backend

The DOM has implemented a repository-neutral strategy to allow for future uses of the COLLADA DOM with database systems based on XML or binary data representations. In this way, the COLLADA DOM eliminates any dependence on a particular underlying specification format.

Schema Driven

The runtime database uses structures derived directly from the COLLADA schema. The C++ definitions of these structures are generated automatically and are therefore always consistent and accurate. This correspondence means that the DOM can be kept in sync with the COLLADA schema as it is further developed.

COLLADA DOM Advantages

In addition to the preceding advantages, the COLLADA DOM provides several other advantages over using a standard XML DOM parser to read COLLADA instance documents.

- With the COLLADA DOM, you need concern yourself only with the specific elements that you want to use or modify. The other elements in an instance document are automatically preserved and are written back to the document when the data is saved.
- The COLLADA DOM automatically resolves URIs upon loading a COLLADA instance document; there is no need to write your own resolver or to search through the data to find referenced URIs.
- The API converts text strings within the COLLADA instance document into their appropriate binary forms. For example, it converts the text form of a number such as "1.345" into a C++ floating-point number.

Architecture Modules

The COLLADA DOM framework includes four basic components:

- **Object Model**. Includes the Digital Asset Exchange (DAE) Object Model, which is a reflective object model allowing for easy creation, manipulation, reading, and writing of COLLADA elements; and the COLLADA Object Model, a custom C++ DOM based on the DAE Object Model and the COLLADA schema.
- **Runtime Database**. Manages COLLADA elements. A reference implementation is provided via the Standard Template Library (STL). The Runtime Database includes the C++ structures for specific instances of the COLLADA Object Model, a mechanism for converting COLLADA Object Model elements to user-defined data structures, and a database query manager.
- **Database Interface**. Allows users to interact with COLLADA files and elements.
- **Backend**: The COLLADA backend consists of the components responsible for translating external COLLADA instance data into C++ runtime COLLADA objects.

These components are described in more detail in Chapter 3, "Architecture."

2 Getting Started with the COLLADA DOM

Technology and Tools

The COLLADA DOM is developed and packaged using VisualStudio™.NET 2003. The entire framework is written in C++. A custom, stream-based XML parser enables schema-driven parsing of instance documents. A code generation tool generates import/export C++ structures in exact correspondence to the current schema. The release of COLLADA DOM 1.4 is based on the COLLADA 1.4.0 schema.

Setting Up the Compiler

VisualStudio.NET requires additional path and dependency information to properly compile and link applications using the COLLADA DOM. Add the following information to your project:

- To the "Additional include directories" field, add:

```
<COLLADA_DOM-path>\include
<COLLADA_DOM-path>\include\1.4
```

- Under the "General" tab in the "Additional library directories" field, add:

```
<COLLADA_DOM-path>\lib\1.4
<COLLADA_DOM-path>\external-libs\libxml2\win32\lib
```

- Under the "Input" tab in the "Additional dependencies" field, add:

```
libcollada_dae.lib
libcollada_dom.lib
libcollada_STLDatabase.lib
libcollada_LIBXMLPlugin.lib
libxml2_a.lib
iconv_a.lib
zlib.lib
```

If you want to use the older and less standards compliant XMLPlugin, also add `libcollada_XMLPlugin.lib`. If you want to use this plugin only, see the next section, "Rebuilding the COLLADA DOM."

When you compile your application code, the VisualStudio C++ compiler "runtime library" setting must match the "runtime library" setting used for compiling the projects that create the COLLADA DOM libraries. To check this setting within VisualStudio, go to the project properties, open C/C++, open Code Generation, and click the "runtime library" property to pop up a list of possible settings. If your application code and the COLLADA DOM are compiled with different "runtime library" settings, the linking process produces a large number of nonintuitive "unsatisfied reference" error messages.

VisualStudio has both debug and nondebug versions of each library setting. Be sure to use the one that matches the type of build you are doing. The debug versions of the COLLADA DOM are compiled with the "debug multi-thread DLL" setting. The release versions of the COLLADA DOM libraries are compiled with the "multi-thread DLL" setting. If you want to compile with a different library setting, you must ensure that the "runtime library" setting is the same in all the Visual Studio projects for your application code and in all the projects that create the COLLADA DOM libraries, and recompile everything before relinking.

Rebuilding the COLLADA DOM

Though most developers can use the prebuilt libraries, the distribution of the COLLADA DOM includes full source code and VisualStudio projects for rebuilding the libraries from scratch.

By default, these projects compile all parts of the DOM, including integration classes and all I/O plugins. A windows binary distribution of libxml2 2.6.20 and its dependencies iconv and zlib are included in the COLLADA DOM distribution. These are taken directly from the Internet distribution site without changes. The default I/O plugin is `daeLIBXMLPlugin`. If you want to use the older `daeXMLPlugin` as the default, change the DAE project (or makefile) to define "`DEFAULT_DAEXMLPLUGIN`" on the compiler command line and rebuild the COLLADA DOM. For more information on this, see the comments at the beginning of `dae.cpp`.

Files

The files required to use the COLLADA DOM are as follows. Additional files required to use the integration templates are described in Chapter 6, "Using Integration Templates."

File name	Description
`dae.h`	The DAE header file.
`libcollada_dae.lib`	The DAE library file.
`libcollada_dom.lib`	The COLLADA Object Model library file.
`libcollada_STLDatabase.lib`	The runtime database library file.
`libcollada_XMLPlugin.lib`	The original XML parser library file, deprecated because it was not fully XML standards compliant.
`libcollada_LIBXMLPlugin.lib`	The XML parser library based on `libxml2`, the default I/O plugin because it is fully XML standards compliant.

Sample Programs

The following sample programs are provided with the COLLADA DOM in the samples directory:

Conditioners/Animation

Loads a COLLADA instance document that contains key frame animation channels, and uses a linear key frame interpolation to sample scalar function curves at regular intervals.

Conditioners/Common

Contains a generic function that parses command-line arguments and generates help and usage messages. The other conditioners use this function to provide a consistent interface.

Conditioners/Deindexer

Deindexes `<geometry>` elements, and optimizes them for rendering with GL vertex arrays. Demonstrates how to reorganize the contents of `<triangles>` and `<controller>` elements, and how to resolve references to `<source>` elements.

Conditioners/Filename

A simple example that finds all the `<image>` elements in an instance document and modifies their source= attributes by doing simple string substitution on the URI. As is, this example converts a simple absolute URI into a relative URI. With additional user code, you can effect more detailed transformations. This is useful because some DCC tools still output system-dependent URIs that must be reworked to make them more general-purpose.

Conditioners/Optimizer

Revises the order of triangles in a COLLADA file to optimize efficiency for rendering on hardware that has a simple FIFO vertex cache. The code that simulates the operation of the cache is replaceable, so the conditioner can be extended to handle other cache configurations. Demonstrates how to reorganize the `<p>` elements within a `<triangles>` element.

Conditioners/Triangulation

Converts the primitives in a COLLADA file to triangles using a simple fanning algorithm and outputs a COLLADA file in which all `<polygons>` elements have been replaced with `<triangles>` elements. Demonstrates the creation of a new `<triangles>` element that uses the same parameters and inputs as an existing `<polygons>` element. Also demonstrates how to insert and delete elements in the COLLADA Object Model.

Reference Materials

COLLADA DOM Reference.

Table 1: URL Sources for COLLADA Information

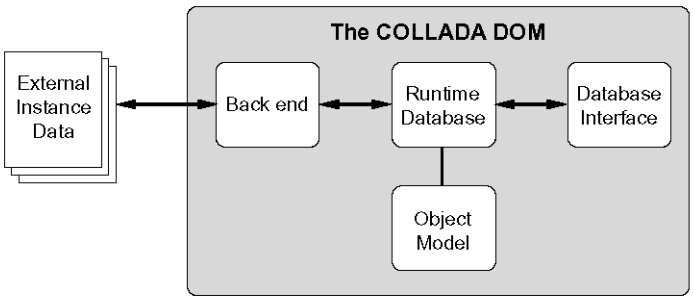
Reference	URL
COLLADA website	http://www.khronos.org/collada
XML schema API specification	http://www.w3.org/Submission/2004/SUBM-xmlschema-api-20040309/
COLLADA schema website	http://www.collada.org/2005/COLLADASchema/
COLLADA schema document	http://www.collada.org/2005/COLLADASchema.xsd
W3C XML website	http://www.w3.org/XML/

3 Architecture

Overview

The following diagram shows the components and communication flow of COLLADA DOM.

Figure 1 The COLLADA DOM



Database Interface

The Database interface provides the means through which applications load and query COLLADA elements from the runtime database. This interface is also used to register integration objects and request application-equivalent data structures from the objects in the runtime database.

Object Model

The COLLADA Object Model is a set of C++ objects (COLLADA DAE objects) generated directly from the COLLADA schema. Each COLLADA DAE object derives from a specialized base class designed to provide reflectivity and robustness to the API. The generated object model is the basis of the runtime database, and is a C++ representation of the COLLADA schema elements. Generated integration objects that exactly mirror each COLLADA DAE object provide plugin points for user code that can be written to intelligently translate data for relevant COLLADA DAE objects into application-specific structures.

Runtime Database

The Runtime Database is a resident, active cache of COLLADA elements. This cache is based on C++ COLLADA elements handed to the cache by the COLLADA Backend. The runtime database provides a mechanism for converting COLLADA elements to application-specific structures. This integration with the COLLADA DOM is accomplished by translation code inserted at predefined plugin points in the integration libraries.

Backend

The COLLADA backend consists of the components responsible for translating external COLLADA instance data into C++ runtime COLLADA objects. The backend uses a meta-parser built into the object model to intelligently parse and translate instance data.

External Instance Data

The nonresident storage of COLLADA instance documents or databases.

Database Interface

The COLLADA database interface is implemented as a C++ class, `daeInterface`, and has a virtual interface supporting the loading, storing, translating, and querying of COLLADA elements.

The `DAE` class provides a simple, general-purpose interface to the replaceable COLLADA backend and runtime database. This class serves as a wrapper for the entire pipeline, ensuring a consistent interface regardless of extensions to or replacements for the various DOM components.

Creating a new `DAE` interface object automatically creates and initializes default versions of the COLLADA backend, the COLLADA runtime database, and registered integration libraries.

The following is a subset of the public database-interface API. The complete API is defined in `dae.h`.

```
virtual daeInt setDatabase(daeDatabase* database);
virtual daeIntegrationLibraryFunc getIntegrationLibrary();
virtual daeInt load(daeString name);
virtual daeInt saveAs(daeString fileName,
                      daeString collectionName);
virtual daeInt unload(daeString name);
```

The COLLADA DOM was designed to allow the underlying database mechanism to be supplemented or replaced without affecting performance of the rest of the API components. The included front-end interface hands off generic queries through the API to any configured backend. Query results assembled by the backend are handed back to the runtime cache for delivery in response to the original query. In addition, the front-end interface itself could be extended to provide a more robust or specific query feature set.

Object Model

The COLLADA DOM is built upon a C++ object model that includes a reflective object system (ROS) and an object model that corresponds directly to the COLLADA XML schema definition (see "Figure 2 The COLLADA Object Model").

Reflective Object System (ROS)

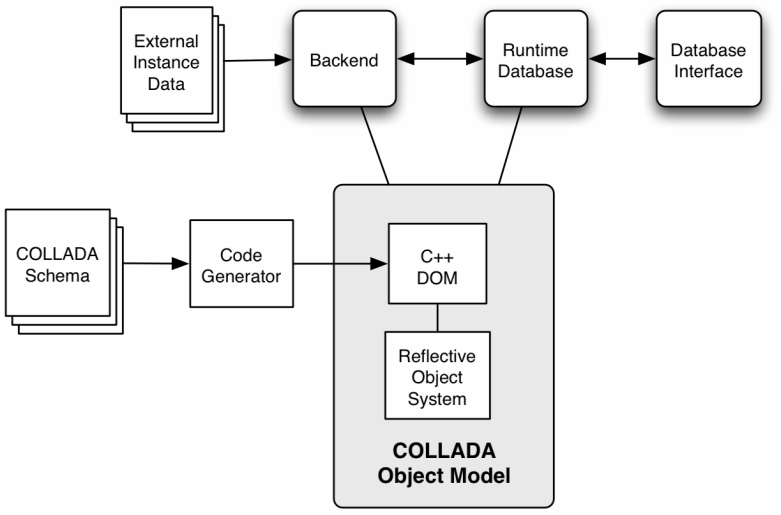
The COLLADA DOM is built upon a set of functionality that defines a self-managed, reflective object system (ROS). The ROS creates, manages, and manipulates DOM objects and the data that define their structure (meta-data) and contents. In the code, any class prefixed with `dae` is part of the reflective object system. The ROS is configured with metadata that, for COLLADA, defines the COLLADA Object Model. The COLLADA Object Model classes are created by the ROS at runtime, loaded with instance data, and passed to the runtime database (see "Runtime Database").

COLLADA Object Model

The COLLADA Object Model is a C++ equivalent representation of the elements defined in the COLLADA schema. The entire Object Model class hierarchy is auto-generated using a code generator that translates the COLLADA schema element definitions into C++ classes. Code generation of the COLLADA Object Model structures guarantees consistency with the schema and accuracy of the code, and eliminates the need for additional work as the schema is updated. Every COLLADA Object Model class is prefixed with `dom`.

The code generation builds in registration of meta-information about the structure of each element in the COLLADA Object Model. Via the static method `registerElement()` contained in every COLLADA Object Model class, the metadata about an element and its subelements are registered with the ROS, allowing it to understand, track, and manage each COLLADA DAE object.

Figure 2 The COLLADA Object Model

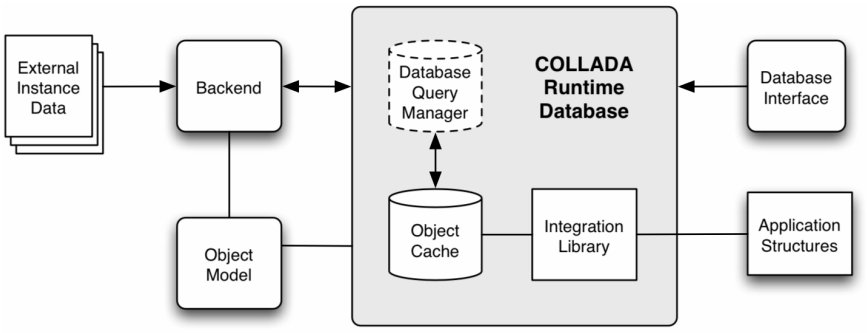


This registered meta-information is used by a parser built directly into the ROS to both parse and validate instance data. This "metaparser" intelligently uses the schema to drive document parsing rather than using it only as an (optional) validation mechanism. This deep connection with the metadata defined by the schema allows the entire COLLADA DOM framework to automatically track the schema as it evolves.

Runtime Database

The core of the COLLADA DOM is a runtime database, a resident cache of COLLADA elements. The runtime database is implemented as a C++ object, `daeDatabase`, with a virtual interface.

Figure 3 The COLLADA Runtime Database



C++ Object Cache

Each COLLADA element in the cache consists of a C++ object hierarchy of COLLADA-defined structures that mirror those defined in the COLLADA schema. The runtime database allows for simple requests of COLLADA objects by name through the database interface. The cache automatically patches references to other COLLADA elements, creating cache lines that move in and out of memory as needed.

Integration Library

The COLLADA integration library is a set of generated template classes that exactly mirror the DOM objects. Every integration class is prefixed with `int`. Each integration class derives from a base class, `daeIntegrationObject`, which provides a virtual interface that defines the semantics for translation of COLLADA data to application structures. Starting with the COLLADA DOM integration library, developers need only define object equivalence by filling in translation functions from one C++ structure

to another. See Chapter 6, “Using Integration Templates,” for an example of how the integration library templates can be used.

Database Query Manager

The COLLADA DOM does not include a query manager at this time, but it warrants mention because the addition of this component may be a useful extension to the runtime database. The COLLADA backend is essentially a wrapper providing access to a persistent data store. While the provided mechanism uses local XML files, it is expected that extensions for binary file formats as well as full database systems will be implemented. An intelligent query manager in the runtime database could cooperate with a particular backend data store to optimize query and retrieval of data for a particular application. The API has attempted to facilitate these extensions.

COLLADA Backend

The COLLADA backend consists of an abstract class, `daeIOPlugin`, with a virtual interface for accepting queries and returning instanced COLLADA DAE objects to the runtime database. Included with the API are two implementations, `daeXMLPlugin`, which provides access to local UTF-8 XML files, and `daeLIBXMLPlugin`, which uses the `libxml2` library to provide fully standards-compliant XML document loading. It is expected that other implementations of this interface would yield I/O modules for other types of persistent data stores, for example: binary files and database systems.

4 Using the Runtime Database

This section describes how to load COLLADA instance documents into the COLLADA runtime database, how to save the runtime database into COLLADA instance documents, and how to query the database. The terms database, collection, and element are defined.

Concepts

The term *database* refers to the runtime database, the resident cache of COLLADA *elements* represented by a `daeDatabase` object. The elements in a database are grouped into *collections*.

In the COLLADA DOM, an element represents any COLLADA element, for example, `<geometry>`, `<image>`, `<scene>`, and so on. The base class defining an element is `daeElement`; all COLLADA DAE objects derive from `daeElement`. Each element may have other elements as children, or may be a child of another element. An element is part of a collection, and each element is found in only one collection.

A collection represents a set of COLLADA elements that are grouped based on a URL. The COLLADA DOM allows a database to contain any number of collections, represented by a `daeCollection` object. Using the XML instance document representation for COLLADA data, each time you use the `DAE::load(xmlUrl)` method to load in a new COLLADA XML instance document, a new collection is created in the database. The method `DAE::saveAs(url,name)` writes out a specific collection to an instance document based on the collection name. The method `DAE::saveAs(url,idx)` writes out a specific collection to an instance document based on the collection index.

Loading and Saving COLLADA Data

To load COLLADA data, begin by creating a `DAE` object, the basic class that initializes the COLLADA DOM and facilitates a connection between the database and the COLLADA backend. This class defines the methods for loading data into and saving data from the database.

```
DAE *daeObject = new DAE;
```

Use the `DAE::load()` method to read a collection of existing COLLADA elements from an instance document. When you load a collection from a URI, the COLLADA DOM also uses this URI as the name of the collection when searching or doing other operations where the collection needs to be identified. In this example, the name of the collection created by the load process is "`file:///input_url_name.xml`", the same as the URI it was loaded from.

```
error = daeObject->load("file:///input_url_name.xml");
```

After you have completed any changes to the elements in the database, write the contents of the database to a new URL using the `DAE::saveAs()` method to save the specific collection. You can use either the collection name or the collection index to choose which collection to save.

```
error = daeObject->saveAs("output_url_name.xml",
                                "file:///input_url_name.xml");
```

Because XML is a text format that can be easily (and potentially incorrectly) hand edited, always check the errors returned from DOM functions; otherwise, some types of errors may go unnoticed and lead to lost data.

Note. *The COLLADA DOM XML parser does not fully validate the data. It detects some types of errors in XML input but not all of them. You may want to run new COLLADA data through an XML validator before loading it into the DOM. If you are developing programs that write COLLADA data, you should run a validator on the output of your programs during testing.*

Adding a New Collection

You may wish to add a new collection to an existing database by hand. For example, you might want to read a single COLLADA instance document, and then separate the single document into two separate documents: one to hold the libraries, and one to hold the scene information.

To add a new collection to an existing database, use the `daeDatabase::insertCollection()` method, which creates a new `daeCollection` object. Every collection requires a root `domCOLLADA` object; `insertCollection()` also creates and initializes this. You also need to provide a name for the new collection. Normally this should be the URI where you eventually plan to save the document. This URI is used in resolving references to this document, so if you plan to have other loaded documents reference this new one it is important that the URI you provide to insertCollection be correct. When the new collection exists, use `daeElement::placeElement()` to move elements from one collection to the other. The process of moving elements from one collection to another is discussed in more detail in "Working with DOM Objects and Multiple Collections".

For example, to insert a new collection, you could do the following:

```
// The DAE* object is defined by the daeObject variable.

// Define the daeCollection variable
daeCollection *collection;

// insert a new collection into the database.
int res = daeObject->getDatabase()->insertCollection("file:///myDom",

&collection);

// The new collection is now ready for adding objects to.
```

Creating COLLADA Data from Scratch

You can use the COLLADA DOM to create a set of elements, add them to an empty database, and write the database to a COLLADA instance document. To do this, you need to create a new database and add a collection to it. Each collection requires a root DOM node of type `domCOLLADA`. The method `DAE::insertCollection()` takes care of creating the collection and creating and inserting a `domCOLLADA` object as the root of the collection. All other DOM objects are added underneath the root node, by using the `createAndPlace()` or `placeElement()` methods. (See "Adding and Removing Elements".)

The following example shows the steps required to set up the framework for creating a DOM from scratch and saving it to an XML instance document.

```
// create a DAE object:
DAE * daeObject = new DAE;

// Create a new database
daeObject->setDatabase(NULL);

// Define the daeCollection variable
daeCollection *collection;

// insert a new collection into the database.
int res = daeObject->getDatabase()->insertCollection(

        "file:///myColladaDoc.xml",
                                                &collection);

if (collection) {
```

```
  // getDomRoot returns the domCOLLADA object
  domCOLLADA *domRoot = (domCollada *)collection->getDomRoot();

  if (domRoot) {
    // Then add subsequent DOM objects underneath domRoot using
    // createAndPlace().  For example, add a library…
    domLibrary_geometries *newLib = (domLibrary_geometries
*)domRoot->createAndPlace

(COLLADA_ELEMENT_LIBRARY_GEOMETRIES);
    ...

    //Write out the new collection, no arguments are required because
    // save defaults to saving collection 0 (the only one loaded) and
    // saves it to the URI specified in insertCollection.  saveAs
    // could also be used to save the document to a different URI.
    int res = daeObject->save();
  }
}
```

Querying the Database

The current version of the COLLADA DOM provides two methods to query the database to get information about specific elements, `daeDatabase::getElementCount()` and `daeDatabase::getElement()`.

The method `getElementCount()` returns the number of elements of a particular type. For example, the following code asks how many `<image>` elements are in the database:

```
imageCount = daeObject->getDatabase()->getElementCount
                                                    (NULL,
COLLADA_ELEMENT_IMAGE, NULL);
```

The additional parameters to `getElementCount()` make the request more specific. The first parameter represents the name or ID of the element. The third parameter represents the name of the collection, and might be used if you have loaded elements from multiple instance documents.

The method `getElement()` requests that the database return a specific element. For example, the following code returns an image element with an index that matches the value in `el_number`.

```
error = daeObject->getDatabase()->getElement
     ((daeElement**)&thisImage, el_number, NULL, COLLADA_ELEMENT_IMAGE, NULL);
```

The element itself is returned in the first parameter. The third parameter restricts the query to a specific element by name, and the fifth parameter restricts the search to a specific collection.

Typically, `getElementCount()` and `getElement()` are used as a pair, first getting the number of elements that match a particular name, type, or collection query, and then iterating through those by using the `getElement()` method. For example, if you want to take an action on all of the images in the database, you could do the following:

```
imageCount = daeObject->getDatabase()->getElementCount
                              (NULL, COLLADA_ELEMENT_IMAGE, NULL);
for (unsigned int i=0; i<imageCount; i++)
{ error = daeObject->getDatabase()->getElement
     ((daeElement**)&thisImage, i, NULL, COLLADA_ELEMENT_IMAGE, NULL);
 /* take action on this image */
}
```

As another example, if you want to take action on all images with the name "`landImage`", you could do the following:

```
imageCount = daeObject->getDatabase()->getElementCount
                          ("landImage", COLLADA_ELEMENT_IMAGE, NULL);
for (unsigned int i=0; i<imageCount; i++)
{ error = daeObject->getDatabase()->getElement
     ((daeElement**)&thisImage, i, "landImage", COLLADA_ELEMENT_IMAGE, NULL);
 /* take action on this land image */
}
```

The index passed to `getElement()` is not directly associated with the element within the database. The index relates to the query itself. The queries used for `getElementCount()` and `getElement()` must match in order for the index to make sense when passed to the `getElement()` method.

5 Using the COLLADA Object Model

This chapter describes how to use and manipulate element information in the COLLADA runtime database using the COLLADA dom* classes directly. The COLLADA DOM can also be set up to automatically create user-defined structures and to copy information from the COLLADA DAE objects to them. This is covered in the next chapter.

Representation of COLLADA Object Model Elements

The structure of COLLADA Object Model elements closely matches the element structure found in the COLLADA instance document. Every element defined in the COLLADA specification maps onto a corresponding dom* class, which is derived from the daeElement class. The factory creates the name of the class based on the element's tag.

For example, the <camera> element maps onto the domCamera class, the <bool_array> element maps onto domBool_array class, and the <source> element maps on the domSource class.

Attributes of the element map onto data members of the dom* object, and the children of the element map into data members that are typed to match the type of the child element. When an element can have multiple children of the same type, the data member is an array.

For example, the <geometry> element as defined in the COLLADA specification includes two attributes: ID and name, and it can have one <mesh> child and any number of <extra> elements as children. The domGeometry class includes *attrName* and *attrId* data members for the attributes, provides a *elemMesh* data member to hold the object's domMesh child, and a *elemExtra_array* to hold any number of domExtra child elements.

Note. *The DOM doesn't perfectly preserve the structure of the original COLLADA instance document; some pieces of information that don't affect the interpretation of the file may be rearranged or changed. This means that, if you read a COLLADA instance document and immediately write it back out without making any changes, the documents may not be identical, though they will be functionally equivalent. You may notice differences in the order of attributes within an XML tag, or in the precision or output format of floating-point numbers. If an optional attribute is set to its default value, that attribute is suppressed when writing to the output document.*

Reference Counting Classes

To facilitate tracking of dom* objects within the DOM, the DOM uses reference counting classes that correspond to each dom* class. The daeSmartRef<T> class template is used to define a reference counting object that derives from both <T> and daeElementRef. So, for example, the definition:

```
typedef daeSmartRef<domCOLLADA> domCOLLADARef;
```

defines domCOLLADARef, which derives from domCOLLADA and daeElementRef. Every dom* class provides one of these reference counting classes. In most cases you do not need to use the domCOLLADARef reference directly; however, whenever a daeElement is placed into the tree using the createAndPlace() method (described in the following "Adding and Removing Elements" section), the DOM creates the daeElementRef version of the object, and this object can be cast to either the dom* reference or the daeElementRef reference.

For example, when you create a domLight object using the daeElement::createAndPlace() method, the resulting object can be cast to domLight*, domLightRef, or daeElementRef.

Note*. The DOM makes extensive use of the* daeElementRef *reference counting classes. If you use a creation method that returns a* daeElementRef *(such as those in* daeMetaElement*), be sure to receive the return value into a* daeElementRef *rather than into a dom* element class pointer. If the value is not returned into a*

daeElementRef, *it can result in incorrect reference counting, which may cause the object to immediately try to delete itself, leaving a pointer to uninitialized memory. Generally you will not want to change the reference count of an object on your own; avoid calling the* daeElement::ref() *or* release() *methods.*

Adding and Removing Elements

The method daeElement::createAndPlace() creates a new element as a child of the current element, and places the new element in the appropriate field of its parent element based on the className parameter provided in the method invocation. To remove a child element from its parent element, you can use the method daeElement::removeChildElement(), or alternatively, you can use the static method daeElement::removeFromParent().

The following example demonstrates how to add a new light to an existing library of lights.

```
/* DAE *daeObject */
daeDatabase *db = daeObject->getDatabase();

// Find the light library, in this example, it has the name "lightLib".
error = db->getElement((daeElement**)&myLib, 0,
                       "lightLib", COLLADA_ELEMENT_LIBRARY);

// Add a new light to the library
domLight *newLight = (domLight *)myLib->createAndPlace(COLLADA_ELEMENT_LIGHT);

// Now you can add data to the new light.
```

The following example demonstrates how you might remove a light from a library of lights.

```
/* DAE *daeObject */
daeDatabase *db = daeObject->getDatabase();

// Find the light we want to remove, the one with name "frontLight"
error = db->getElement((daeElement**)&myLight, 0,
                       "frontLight", COLLADA_ELEMENT_LIGHT);

// Get its parent
daeElement* lightParent = myLight->getXMLParentElement();

// Remove the light
daeBool removed = lightParent->removeChildElement(myLight);
```

Or, alternatively, you can use removeFromParent() to remove a light from a library of lights, which finds the parent object automatically. The method removeFromParent() is a static method, since the act of removing an object from its parent can result in the deletion of the object.

```
/* DAE *daeObject */
daeDatabase *db = daeObject->getDatabase();

// Find the light we want to remove, the one with name "frontLight"
error = db->getElement((daeElement**)&myLight, 0,
                       "frontLight", COLLADA_ELEMENT_LIGHT);

// Remove the light from its parent
daeBool removed = daeElement::removeFromParent(myLight);
```

Using these methods to add elements to and remove elements from the database ensures that the element bookkeeping information maintained by the COLLADA DOM is appropriately updated. For example, many DOM classes contain a *_contents* array, which records the creation order of all the child elements of that DOM class so they can be written in the same order that they were read or created. The DOM

maintains the *_contents* array automatically through the use of these methods when creating and deleting new child elements.

Note. *We do not recommend manually updating or directly using the* `_contents` *array, as the process is error-prone. In addition, future versions of the DOM may not have the* `_contents` *array exposed for public use.*

Copying Elements

To copy a DOM element, rather than using the assignment operator, use the `daeElement::clone()` method, which provides a deep copy operation. After the element is cloned, you need to resolve any URIs that have been copied, as `clone()` does not do that automatically.

The DOM type `daeString`, used in many objects derived from the daeElement class, is a C-style string and not a C++ string class. If you use the assignment operator to copy a `daeString`, you can end up with a dangling pointer if the original string is deleted.

Working with daeTArrays

The DOM uses the templated class `daeTArray` object as the base class for many of the data members of objects derived from the `daeElement` class. It is used to represent the links between a parent and a child, and is also used to hold lists of other types of data. When you work with a `daeTArray` object, you need to use different methods to manipulate the array depending on what type of data the array contains.

When a `daeTArray` contains something derived from a `daeElement`, such as a `domInput_Array`, `domParam_Array`, or `domP_Array`, use `createAndPlace()` to add elements to the array, and `removeChildElement()` to remove elements from the array. If you need to explicitly place an element at a specific index within the array of child elements, the method `daeElement::createAndPlaceAt()` does the same thing as `createAndPlace()`, but places the element at a specific index in the *_contents* array.

When a `daeTArray` contains something other than an object derived from `daeElement`, use the `append()` method to append a new array element. The `appendUnique()` method appends the array element only if it doesn't already exist in the array. The `insertAt()` method allows you to insert a new element at a specific index within the array, growing the array if necessary. You can also use the `set()` method to change the value of a specific index. Use the `remove()` method to remove an array element by value, or `removeIndex()` to remove an array element at a specific index.

Note. *Avoid using the* `daeArray::clear()` *method on* any *array member of a* `daeElement` *derived class. The* `clear()` *method does not update the* `_contents` *array, so while it may appear to delete data, the data will still be written to the COLLADA instance document when the database is saved.*

The following example shows how to add data to both types of arrays. In this example, `thisTriangles` is a pointer to `domTriangles`, and we want to add a new triangle to its `p_array`. The content of the `p_array` is derived from a `daeElement`, while the `_value` array in the `p_array` is a simple list of integers.

```
// Add a new triangle: use createAndPlace because domP is derived
// from a daeElement.  This adds a new element to p_array.
domP* p_triangles =
     (domP*)thisTriangles->createAndPlace(COLLADA_ELEMENT_P);

// Update the count data, which tracks the number of p elements
thisTriangles->setCount( thisTriangles.getCount()+1);

// Then use append to add indices to the p's _value array, which
// which is a domListOfInts
```

```
p_triangles->getValue().append(1);
p_triangles->getValue().append(2);
p_triangles->getValue().append(3);
```

Note. *The* `count` *data field found in some dom objects, such as* `domPolygons`, `domTriangles`, `domAccessor`, *or* `domFloat_Array`, *is not updated automatically when you call* `createAndPlace()` *or* `removeChildElement()`. *For example, if you change the number of elements in the* `p_array` *of a* `domPolygons`, *you need to update the* `count` *data member yourself.*

To get the data from either type of array, use the `get() or index operator[]` and `getCount()` methods. You can also use the `find()` method to find a specific value within the array. The following example shows iterating through a set of polygons on a mesh. The object `thisMesh` is a pointer to `domMesh`.

```
int polygonCnt = thisMesh->getPolygons_array().getCount();
for (int currPoly = 0; currPoly < polygonCnt; currPoly++){
   domPolygons *thisPoly = thisMesh->getPolygons_array().get(currPoly);
   …
}
```

Working with URI References

When COLLADA elements refer to other elements, they use Uniform Resource Identifiers (URIs). For example, within the COLLADA instance document, a light defined with the ID "`Lt-Light`" can be referenced using `<instance_light url = "#Lt-Light">`. In the following example, the light node element refers to the light element found in the LIGHT library.

```
<library type="LIGHT">
  <light id="Lt-Light" name="light">
    ...
  </light>
</library>

<node id="Light" name="Light">
   <translate>-5.000000 10.000000 4.000000</translate>
  <rotate>0 0 1 0</rotate>
  <rotate>0 1 0 0</rotate>
  <rotate>1 0 0 0</rotate>
  <instance_light url="#Lt-Light" />
</node>
```

The COLLADA DOM uses the `daeURI` data structure to represent URIs. This structure allows you to retrieve the `daeElement` that a `daeURI` refers to, and also allows you to change a `daeURI` to point to a new `daeElement`.

The *`daeURI::element`* data member points to the `daeElement` associated with the `daeURI` object. Since all URI references are resolved when a collection is loaded, usually the element data member already contains the `daeElement` reference.

If the URI has not been resolved, you can use the method `daeURI::resolveElement()`, which puts a reference to the `daeElement` in the *`daeURI::element`* member. If the URI is associated with an instance document that is not loaded into the database, this method loads the instance document into a new collection, and then retrieves the `daeElement` from the new collection. The resolve status is stored in *`daeURI::state`*; the value `daeURI::uri_success` indicates that the URI has been successfully resolved.

The following example demonstrates retrieving an element associated with a URI, and resolving the reference, if necessary. In this case, the code finds the daeElement associated with an <instance> element in a light <node>.

```
/* DAE *daeObject */
daeDatabase *db = daeObject->getDatabase();

// Get the node, the one with name "Light"
error = db->getElement((daeElement**)&lightNode, 0, "Light",
COLLADA_ELEMENT_NODE);

// Get the first element of the instance array in lightNode.
int instCnt = lightNode->getInstance_light_array().getCount();
if (instCnt > 0) {
  domInstance_light *thisInst = lightNode->getInstance_light_array().get(0);

  // Get the url associated with the instance
  daeURI *lightURI = &thisInst->getUrl();

  // Get the element, resolve the reference if necessary
  if (lightURI->getState() != daeURI::uri_success) {
     lightURI->resolveElement();
  }
  domLight *lightRef = (domLight*)(daeElement*)lightURI->getElement();
```

If you want to set a URI to point to a specific daeElement, set the *daeURI::element* to the target element and call daeURI::resolveURI() to fill in a text version of the URI. This method uses the daeElement's base URI and ID information to fill in the daeURI data members.

If you need to copy one daeURI to another, use the daeURI::copyFrom() method. If you use the assignment operator to copy, problems may result if the source daeURI is deleted.

Working with DOM Objects and Multiple Collections

With multiple collections loaded into a database simultaneously, you can use DOM function calls to move elements between collections. The method daeElement::placeElement() removes the element from its current parent and collection, attaches it to the new parent, and puts the daeElement and all its children into the same collection as the new parent.

For example, you can load a single COLLADA instance document, and then create a new collection and use placeElement() to move the scene node and all its children to the new collection.

```
DAE *daeObject = new DAE;
// Load in an instance document that contains a scene and libraries
int error = daeObject->load("file:///full_url.xml");

// get the database
daeDatabase *database = NULL;
database = daeObject->getDatabase();

// create a new collection to move the scene to,
// and insert it into the database
daeCollection *collection;
int error = database->insertCollection(
                                          "file:///new_collection.xml",
                                          &collection);

// Get the scene node to move. Only one scene node per collection,
// so just need to get the first scene in "full_url.xml" using index 0.
```

```
// It is returned into thisScene.
int error = database->getElement((daeElement**)&thisScene, 0,
                                                 NULL,
                                                 COLLADA_ELEMENT_SCENE,
                                                 "file:///full_url.xml");

// Move the scene node and its children to the new collection
domCOLLADA *domRoot = collection->getDomRoot();
domRoot->placeElement(thisScene);
```

When you move an element from one collection to another, you need to manually resolve any URL references that change as a result of the move. In the preceding example, after you have moved the scene node and all its children to the new collection, you need to iterate through each node in the scene, changing the `url` of any `domInstance_*` elements that are no longer found in the same collection.

If you move elements into a collection that already contains elements, you also need resolve any ID collisions. Many COLLADA elements contain an ID attribute that must be unique within the instance document. The preceding example does not require ID resolution, because the destination collection is empty to start with.

If you require a `daeElement` to appear in multiple collections, you need to duplicate the element and add the duplicate to a collection. Use `daeElement::clone()` to duplicate the element.

The methods `removeFromParent()` and `removeChildElement()` remove a `daeElement` from its parent and the parent's collection. In general, the removal results in the automatic deletion of the child element, unless there are additional reference pointers to the element. If removal doesn't result in automatic deletion, the element and all its children are still in the database but are not bound to any collection.

Note. *When a* `daeElement` *tree is not bound to any collection, these elements are returned by the* `getElement()` *and* `getElementCount()` *methods, unless a collection name is specified as one of the search keys. The standard* `saveAs()` *methods do not write* `daeElements` *unless they are bound to a collection.*

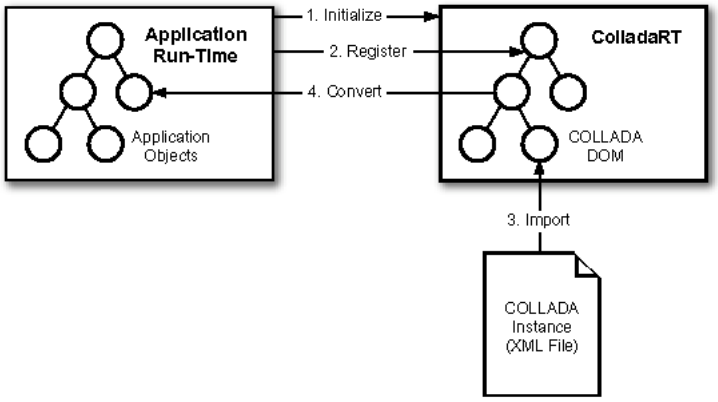
6 Using Integration Templates

This chapter describes how to use the integration templates to convert COLLADA DAE objects into application-specific objects and back again. It provides a step-by-step example of how to integrate application data structures with the COLLADA runtime infrastructure (COLLADA DOM). Detailed knowledge of the COLLADA DOM architecture is not necessary to understand this example or to successfully integrate your own application-specific data structures.

Overview

The COLLADA Object Model is a set of runtime COLLADA DAE objects that correspond to elements in a COLLADA XML instance document. For importing, they are built when a COLLADA XML instance document is parsed. For exporting, you might need to provide code to build them.To convert data contained in COLLADA DAE objects into your own application data structures, and the reverse, the DOM provides integration templates for every object. The integration templates provide plugin points where you can insert conversion code. After a document has been imported by the DOM into COLLADA DAE objects, or when you request an export, your integration code is called to perform the conversion.

Figure 4 COLLADA DOM Integration Usage Model for Importing



The basic steps during integration are the following:

(1) Initialize the COLLADA DOM by creating a new `DAE` object.

(2) Register application-specific integration libraries.

(3) To convert after importing:

(a) Import a COLLADA XML file by loading it; this places the data into runtime COLLADA DAE objects. COLLADA DAE objects with registered integration libraries automatically create their associated application data structures.

(b) The COLLADA runtime calls the conversion methods in the integration libraries to convert the content of COLLADA DAE objects into their associated data structures.

(4) To convert before exporting:

(a) If matching COLLADA DAE objects don't already exist, call `createTo` to create them.

(b) Export the COLLADA DAE structure by saving it; the COLLADA runtime calls the conversion methods in the integration libraries to convert your data structures into COLLADA DAE objects and then export them.

COLLADA DOM Integration Templates

At import time, elements in COLLADA instance documents are loaded into a run-time COLLADA Object Model. The COLLADA DOM provides integration templates to enable you to convert COLLADA data between the COLLADA Object Model and your own run-time structures.

Each element in the COLLADA Object Model has its own integration template consisting of a header (`.h`) file and a code (`.cpp`) file. Copy into your application directory the template files for the elements that you want to convert from DOM structures into application-specific structures or vice versa. These copies are the starting points for your customized *integration libraries* You can then add code to these integration libraries to convert data stored in the associated DOM object into your application object(s) and vice versa. The plugin points for your code are identified in comments in the template source.

The COLLADA DOM provides two sets of integration templates:

- **Simple Integration Templates:** These templates are in the `templates/integrationSimple` directory, and provide plugin points for the most commonly converted COLLADA DAE objects. These templates do not provide plugin points for the nested XML elements, such as the `<author>` element found in `<asset>` that is defined by the `domAuthor` class.
- **Full Integration Templates:** These templates are in the `templates/integrationFull` directory, and provide plugin points for all COLLADA DAE objects.

The DOM provides a pair of template files for each basic COLLADA DAE object. For example, the template files to convert `domNode` objects are `intNode.cpp` and `intNode.h`.

Integration Objects

Your integration library for each element in the COLLADA Object Model provides the code that defines an *integration object* for the element. The integration object serves as the focal point for all information about the conversion. It defines methods that perform the conversion, and it provides data members that bind the COLLADA Object Model element with the application-specific data structure being converted to.

Integration objects are represented with the `daeIntegrationObject` class. Every class derived from `daeElement` provides for an integration object through the data member *`daeElement::_intObject`*.

The integration object class for each element is defined with the prefix "`int`". For example, the `domGeometry` class provides an `intGeometry` integration object.

When you define integration code for an element, your code binds the element with your application-specific object, via the *`_element`* and *`_object`* data members of the integration object. You can use methods `daeIntegrationObject::getElement()` and `getObject()` to access these data members.

When you load a COLLADA instance document into a new collection, the DOM automatically creates the appropriate integration objects. To get the integration object for an element, use the method `daeElement::getIntObject()`, which also initiates the conversion process for any objects that have not yet been converted.

Note. *The beta release of the COLLADA DOM does not automatically call the plugin code to update the COLLADA Object Model data structures from your application objects prior to saving the database. If you wish to write the changed application data out to a new COLLADA instance document, you must ensure that the COLLADA Object Model structures are updated by calling the "to" plugin points in your application code, which can usually be accomplished by calling the* `daeElement::getIntObject()` *method, since this method converts application objects both to and from their associated COLLADA DAE Objects.*

Integration Template Plugin Points

The integration class for each COLLADA Object Model element provides six plugin points into which you can add conversion code. The plugin points are implemented as methods; you need to provide the code for the method bodies. You need to implement only the body of the methods for those plugin points that are relevant to your application.

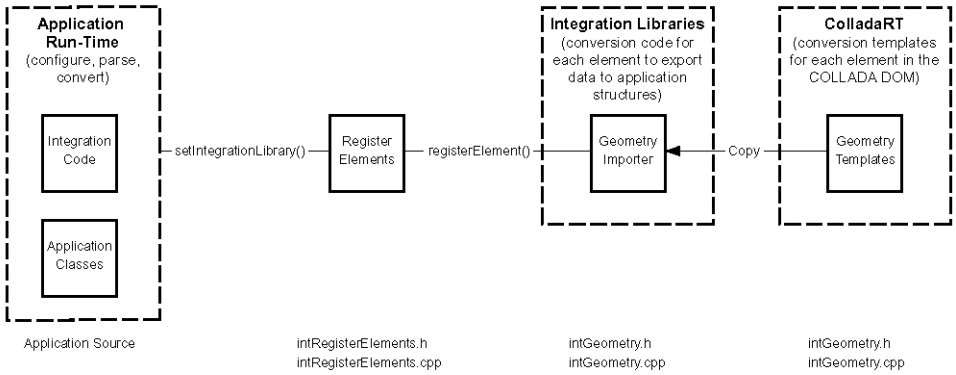
- `createFrom()`: Defines the code to create the application-specific data structure associated with the DOM class for this template. This method sets up the integration object for the DOM class.
- `fromCOLLADA()`: Defines the code to covert the COLLADA Object Model data structure into your application-specific data structure.
- `fromCOLLADAPostProcess()`: Defines any postprocessing code that must execute after the basic conversion to your application data structure.
- `createTo()`: Defines code to create the COLLADA Object Model data structure associated with the DOM class for this template if such a structure does not already exist (for example, if it was not created while importing).
- `toCOLLADA()`: Defines the code to covert the content of your application's data structures into COLLADA Object Model data structures.
- `toCOLLADAPostProcess()`: Defines any postprocessing code that must execute after the basic conversion from your application's data structure.

Geometry Integration Example

This example demonstrates integration with the `<geometry>` element.

Creating the example has three basic steps. You copy the corresponding integration templates to application-specific versions. You register these classes with the COLLADA DOM. In the application runtime, you add code to initialize the COLLADA DOM, call the file load, and request application objects from the integration classes.

Figure 5 Files Used in this Example



Make Copies of Relevant Integration Templates

The first step in the process of creating conversion code is to copy the relevant integration template files from the integration subdirectory into your application's directory. For this example, the relevant integration templates are `intGeometry.cpp` and `intGeometry.h`. You will modify the copied files to contain your conversion code.

Register Integration Libraries with the DOM

For the DOM to create integration objects during parse time or before exporting, you must register the integration objects with the infrastructure. To do this, call the `registerElement()` method on each integration library that you are using. A convenience function to register these classes with the COLLADA DOM is defined in `intRegisterElements.cpp`, which is also found in the template directory. You can copy this function into your application's directory. The relevant code for this example is shown here:

```
void intRegisterElements()
{
        intGeometry::registerElement();
}
```

The integration library file provides the definition for the `intGeometry` class. Its contents are explained in a later step.

After the `intRegisterElements()` function is defined, pass a handle to this function into the `DAE::setIntegrationLibrary()` method, as described in the "Invoke Integration Libraries Registration" section.

Setting up the Integration Library Header File

Now we begin the process of editing the new integration library for the objects we want to convert. We start with the `intGeometry.h` integration library copied from the `intGeometry.h` template. The template must be modified to add information about the application object(s) to or from which the COLLADA Object Model structure will be converted. For the header, we need to declare the class we are converting into or from:

```
// class myGeometry is fully defined in an application header file.
class myGeometry;
```

We also add code to define the relevant structures and provide a method to return those structures. These definitions fall within the body of the `intGeometry` integration object declaration:

```
class intGeometry : public daeIntegrationObject
{
// intGeometry template declarations provided here
. . .

public: // USER CODE
     virtual ~intGeometry();
     // define the accessor for the myGeometry object
     myGeometry *getGeometry() { return _object; }

private: // USER CODE
     // declare the types for the integration object data members
     myGeometry *_object;
     daeElement *_element;
};
```

Defining Your Application Data Structure

Within the integration library files, you need to provide the code to create your application-specific data structure for the COLLADA DAE objects that you want to convert.

For this example, we have an application-specific data structure used to represent geometry, defined in an application header file, in this case, `myGeometry.h`:

```
//  Definition of application's myPologon class also included here
class myGeometry
{
public:
   unsigned int _iVertexCount;
   float *_pVertices;
   std::vector<myPolygon> _vPolygons;
};
```

Provide the Plugin Code to Create an Application Object for Importing

The plugin method relevant to this step for importing is the `createFrom()` method.Within the `createFrom()` method, we add code to create a new `myGeometry` object, initialize it, and initialize the `intGeometry` integration object data members. Within `intGeometry.cpp`:

```
void intGeometry::createFrom(daeElementRef element)
{
   // create class to hold geometry information and
   // initialize the new object as empty
   _object = new myGeometry();
   _object->pVertices = NULL;

   // set up the _element data member of the integration object
   _element = element;
}
```

Now, when a COLLADA instance document is loaded into the COLLADA runtime database and a `domGeometry` object is encountered, the `createFrom()` method automatically creates a new `myGeometry` object, because the `intGeometry` integration library has been registered with the DOM.

Build Conversion Code for Importing

Now we must add code to the integration library to translate the data stored in a DOM object to its associated application object. The `fromCOLLADA()` method is the plugin point for the basic application-specific conversion code. The `fromCOLLADAPostProcess()` method provides additional flexibility for the conversion process.

Depending on the COLLADA DAE object and the application object, the code may vary significantly. For this example, we use the `intGeometry::fromCOLLADA()` method to create new vertex buffers from the COLLADA DAE geometry object.

The following code retrieves the `mesh` object from the COLLADA `domGeometry` object:

```
// Get the geometry element from this integration object
domGeometry* geomElement = (domGeometry*)(domElement*)getElement();
domMesh *meshEl = geomElement->getMesh();
```

When we have the mesh, we can construct our application-specific data structure `iVertices` for the `myGeometry` object. The following code shows how to create the new vertex buffer.

```
// Get a pointer to the application-defined geometry object that was
// automatically created during load by calling createFrom.
myGeometry *local = (myGeometry *)_object;

// Get a pointer to the domPolygons in this domMesh. To simplify this example,
// we will handle only a domMesh that has a single domPolygons.
```

```
if(meshElement->getPolygons_array().getCount() != 1)
{
        fprintf(stderr,
           "This example supports only one domPolygons per domMesh\n");
        return;
}
domPolygons *polygons = meshElement->getPolygons_array()[0];
int           polygonCount = polygons->getCount();

// To simplify this example, we assume the domPolygons has only one domInput.

if(polygons->getInput_array().getCount() != 1)
{
        fprintf(stderr,
           "This example supports only one domInput per domPolygons\n");
        return;
}

// Loop over all the polygons in the domPolygons element

for (int i=0;i<polygonCount;i++)
{
        myPolygon myPoly;
        // Get pointer to this polygon (domP).
        domPolygons::domP *poly = polygons->getP_array()[i];
        // Get the number of indices from the domP and save it in my structure.
        myPoly._iIndexCount = poly->getValue().getCount();
        // You can modify the data as you copy it from
        // the COLLADA object to your object.
        // Here we repeat the first index in list as the last index,
        // to form a closed loop that can be drawn as a line strip.
        myPoly._iIndexCount++;
        myPoly._pIndexes = new unsigned short[myPoly._iIndexCount];
        // Copy all the indices from the domP into my structure.
        for (int j=0;j<myPoly._iIndexCount-1;j++)
              myPoly._pIndexes[j] = poly->getValue()[j];
        // Repeat the first index at the end of the list to create a closed loop.
        myPoly._pIndexes[j] = myPoly._pIndexes[0];
        // Push this polygon into the list of polygons in my structure.
        local->_vPolygons.push_back(myPoly);
}

// Copy the vertices we are going to use into myGeometry. To keep things simple,
// we will assume there is only one domSource and domFloatArray in the domMesh,
// that it is the array of vertices, and that it is in X, Y, Z format. A real
// app would find the vertices by starting with domPolygons and following
// the links through the domInput, domVertices, domSource, domFloat_array,
// and domTechnique.

if(meshElement->getSource_array().getCount() != 1)
{
        fprintf(stderr,
           "This example supports only one source array per domMesh\n");
        return;
}
domSource *source = meshElement->getSource_array()[0];

if(source->getFloat_array_array().getCount() != 1)
{
        fprintf(stderr,
           "This example supports only one float array per source\n");
```

```
}
domFloat_array *floatArray = source->getFloat_array_array()[0];

// Assume there are 3 values per vertex with a stride of 3.
local->_iVertexCount = floatArray->getCount()/3;
local->_pVertices = new float[local->_iVertexCount*3];

// Copy the vertices into my structure one-by-one
// (converts from COLLADA's doubles to floats).
for ( unsigned int i = 0; i < local->_iVertexCount*3; i++ ) {
        local->_pVertices[i] = floatArray->getValue()[i];
}
```

Access COLLADA Objects from the Application

We can now put together the application code that registers the integration libraries, loads the COLLADA data into the in-memory DOM structures, and accesses the converted data.

Invoke Integration Libraries Registration

You defined the `intRegisterElements()` function as described in "Register Integration Libraries with the DOM." Now you must pass a handle to this function into the `DAE::setIntegrationLibrary()` method.

With this step, the elements that you want to convert are registered with the DOM, and are converted from COLLADA Object Model structures to application-specific structures when a COLLADA instance document is loaded into the DOM, or the reverse when an object is saved.

For example, your main application code could pass a handle to the integration library registration function as follows:

```
// Instantiate the reference implementation
daeObject = new DAE;
//register the integration objects
daeObject->setIntegrationLibrary(&intRegisterElements);
```

Parse a COLLADA File

Now we are ready to parse a COLLADA instance into the run-time COLLADA Object Model. The load step creates the integration objects and invokes the `createFrom()` and `fromCOLLADA()` methods to convert the data from the COLLADA Object Model structures into the application-defined structures.

```
//load the COLLADA file
int res = daeObject->load(filename);
```

Access a COLLADA Object

With the COLLADA file parsed and loaded into in-memory database objects, it is now possible to request objects from the database. Here we request that the database return the first geometry element, placing it into *`pElem`*.

```
//query the runtime to retrieve an element

int res = daeObject->getDatabase()->getElement

        ((daeElement**)&pElem,0,NULL,COLLADA_ELEMENT_GEOMETRY);
```

Acquire the Converted Application Object

In the following code, the `myGeometry` class represents the application object containing geometry data. The COLLADA DAE object retrieved above into `pElem` contains a reference to its associated integration object (thanks to its earlier registration). The `intGeometry` class converts the COLLADA data and returns the converted data as a `myGeometry` instance via the `getGeometry()` method, which was defined in `importGeometry.h`.

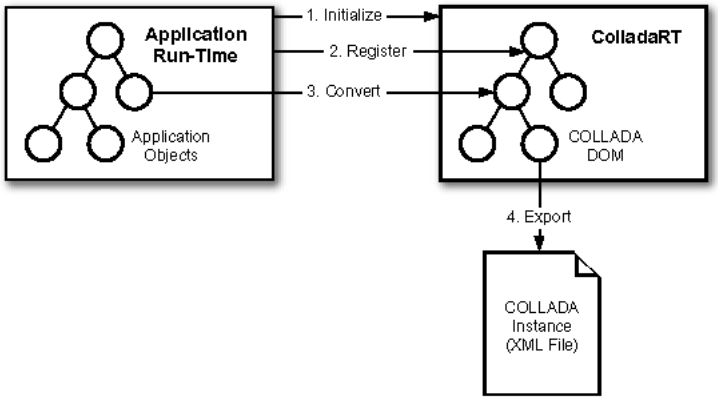
```
// Get the integration object from the element
daeIntegrationObject *pIntegrationObj = pElem->getIntObject();
intGeometry *importGeometry =(intGeometry *)pIntegrationObj;

//extract the user data from the integration object
myGeometry geom = importGeometry->getGeometry();
```

Exporting Using Integration

The preceding sections showed how to set up your integration objects, import a COLLADA instance document, and ensure that the data is converted into data structures specific to your application.

If you modify data values and want to write it to a COLLADA instance document, the process is similar to reversing the import-and-convert process, although there are some differences.

Figure 6 COLLADA DOM Integration Usage Model for Exporting



Modifying a COLLADA DOM Object Structure

In some cases, your application might modify the structure of your application objects that you loaded from a COLLADA DOM object. If you want to export the data back to a COLLADA instance document, you must change the structure of the COLLADA DOM object to match your changes before converting from your application-specific structures. The example code does not show how to do this.

Creating a New COLLADA DOM Object Using createTo

In some cases, you might need to export your data to an entirely new COLLADA document, such as when the data originated in your application rather than by being imported from an existing COLLADA instance document. In this cases, to use the integration methods to convert from your application structure to a DOM object and then export to a new COLLADA instance document, you must create the matching COLLADA DOM structure before you can convert from your application's structure.

Compare with the import process: When your application loads a COLLADA instance document, the COLLADA DOM automatically creates DAE objects in which to load the COLLADA data, then calls your customized `createFrom` for each element that has an integration class, which creates your

application-specific objects. Then it calls `fromCOLLADA`, which copies the data from the DAE objects to your application objects.

If you create a new application object, there are no COLLADA elements (no DAE objects) associated with it. If you simply save the file, this new data is not written. The purpose of `createTo` is to create these COLLADA elements and associate them with the application object. `createTo` is *not* called automatically when a new application object is created; you must call it explicitly. After `createTo` has been called, the rest of the saving process is automatic, that is, if you have registered your integration library, when you call save, `toCOLLADA` is called for every COLLADA element with an integration class to copy the data from the application objects into the COLLADA objects.

This example code does not show how to do this. The `createTo` method looks like this:

```
void
intGeometry::createTo(void *userData)
{
        // This function would create new COLLADA elements from an
        // application- defined object
        // It is NOT called automatically by the COLLADA DOM.
        // The user needs to call this when creating a new
        // application-specific object.
}
```

Exporting Data Value Modifications

If you have modified only the values of the COLLADA data and want to export to a revised COLLADA instance document, this example shows how you might convert the data.

The saving process for this example is automatic because you have registered your integration library. When you call save, `toCOLLADA` is called for every COLLADA element with an integration class. This copies the data from the application objects into the COLLADA ones.

Note that the code is essentially the reverse of the `fromCOLLADA` code in the importing example.

```
void
intGeometry::toCOLLADA()
{
        // INSERT CODE TO TRANSLATE TO YOUR RUNTIME HERE
        // The following lines are example code from the template:
        // myRuntimeClassType* local = (myRuntimeClassType*)_object;
        // element->foo = local->foo;
        // element->subelem[0]->bar = local->bar;

        // This code takes data from an application-defined object and
        // puts it back into the appropriate collada objects.

        // Get a pointer to the COLLADA domGeometry element
        // (this is the element that called us)
        domGeometry* geometryElement = (domGeometry*)(domElement*)_element;

        // Get a pointer to the domGeometry's domMesh element
        domMesh *meshElement = geometryElement->getMesh();

        // Get a pointer to my object that's assocated with this collada object
        myGeometry *local = (myGeometry *)_object;

        // Get a pointer to the domPolygons in this domMesh.
        // To simplify this example,
        // we will handle only a domMesh that has a single domPolygons
        if(meshElement->getPolygons_array().getCount() != 1)
```

```
        {
            fprintf(stderr,
                "this example supports only one domPolygons per domMesh\n");
            return;
        }
        domPolygons *polygons = meshElement->getPolygons_array()[0];
        int          polygonCount = local->_vPolygons.size();

        // To simplify this example, we assume that the domPolygons has
        // only one domInput

        if(polygons->getInput_array().getCount() != 1)
        {
            fprintf(stderr,
                "this example supports only one domInput per domPolygons\n");
            return;
        }
        // Loop over all the polygons in the domPolygons element and
        // put the data from
        // myGeometry back into it. For the purposes of the example,
        // assume the number of polygons and indices hasn't changed
        // so we can just update the values in place

        polygons->setCount(polygonCount);
        for (int i=0;i<polygonCount;i++)
        {
            // Get pointer to this polygon (domP)
            domPolygons::domP *poly = polygons->getP_array()[i];
            // Copy all the indices from my structure back to the domP
            for (int j=0;j< local->_vPolygons[i]._iIndexCount-1;j++)
                poly->getValue()[j] =
                local->_vPolygons[i]._pIndexes[j] ;
        }

        // Now copy the vertices from myGeometry back to the source.
        // Assume that the number of
        // vertices hasn't changed so we can just update them in place.
        if(meshElement->getSource_array().getCount() != 1)
        {
            fprintf(stderr,
                "this example supports only one source array per domMesh\n");
            return;
        }
        domSource *source = meshElement->getSource_array()[0];

        if(source->getFloat_array_array().getCount() != 1)
        {
            fprintf(stderr,
                "this example supports only one float array per source\n");
        }
        domFloat_array *floatArray = source->getFloat_array_array()[0];

        // Copy the vertices into from myGeometry back into the
        // COLLADA float array
        floatArray->setCount(local->_iVertexCount*3);
        for ( unsigned int i = 0; i < local->_iVertexCount*3; i++ ) {
            floatArray->getValue()[i] = local->_pVertices[i];
        }
    }
```

Changes Since Last Release

The following changes were made to the Programming Guide:

Chapter 1, “Beta Release Notes” section

- Removed the “Beta Release Notes” section.

Chapter 2, “Reference Materials” section

- Updated the COLLADA website’s URL address.

Footer

- Removed “PLAYSTATION®3 Programmer Tool Runtime Library Release 0.8.0”.
- Added “©SCEI”