COLLADA DOM 1.4.0
プログラミング・ガイド

現在の仕様は暫定的なものであり、このドキュメントに記載されている内容は予告なく変更される場合があります。ご了承ください。

目次

1 概要

COLLADA 文書オブジェクトモデル（DOM）は、COLLADA XML インスタンス文書の C++オブジェクト表現を提供するアプリケーション・プログラミング・インタフェース（API）です。

背景

2003 年 8 月に、株式会社ソニー・コンピュータエンタテインメント（SCEI）は、業界との共同作業により、特定の企業に帰属しない、オープンな 3D デジタルアセット・スキーマの設計を始めました。この共同作業（COLLADA™）は、まもなく、デジタルアセット・スキーマの最初のバージョンを制定する業界パートナーの中心団体として、活躍を始めました。
この中心団体は、COLLADA 仕様をよりオープンなものにするために、フォーマットとして XML スキーマ言語を選択しました。この選択が行われたのは、交換形式として XML が幅広く採用されていることと、利用できる XML ベースのツールがたくさんあることが理由です。
また、設計の出来具合を検証したり、COLLADA 仕様の実際の応用を促進したりするために、Emdigo 社によって、COLLADA DOM、および参照用のインポータとエクスポータが設計・実装されました。

概要

COLLADA DOM は、COLLADA アプリケーション開発用の包括的なフレームワークです。DOM は、COLLADA インスタンス・データのロード、クエリー、変換などを行うための C++プログラミング・インターフェイスを提供します。DOM では、COLLADA のデータを、COLLADA スキーマの中で定義された構造体と同じ構造体を持つランタイム・データベースにロードします。このランタイム構造体は、現在のスキーマから、矛盾やエラーを取り除くことにより、自動的に生成されます。
ディベロッパは、クエリーAPI や、コード統合のためのプラグイン・ポイントを介して、ランタイム・データベースと相互にやりとりします。この API を正しく組み込むために、スキーマやインスタンス・データのフォーマット詳細を理解しておく必要はありません。また、ディベロッパは、COLLADA ランタイム・データベースにロードされたデータ構造体を、アプリケーションの中で直接利用することができます。
初期の COLLADA ユーザには、COLLADA の中心的なパートナーだけでなく、進行中の共同作業に貢献もしくは利用することを希望する、業界からの参加者も含まれています。

設計上の目標

COLLADA DOM では、DOM に必要な機能を提供するため、以下のような設計上の課題が考慮されています。

データ変換が簡単

COLLADA DOM には、COLLADA ランタイム・データベースにロードされたデータを、ユーザ独自のツールやエンジン専用のデータ構造体に変換するための、変換コードを書く手段が用意されています。基本的なフレームワークを理解しなくてもこの変換を書くことができるように、API にはプラグイン・ポイントが用意されています。

交換可能なバックエンド

COLLADA DOM は、将来的に、XML やバイナリ・データ表現に基づくデータ・ベース・システムを使用して利用できるように、特定のリポジトリに依存しない方法で実装されています。このため、COLLADA DOM からは、特定の基本仕様フォーマットに対するあらゆる依存性が排除されています。

スキーマ駆動方式

COLLADA ランタイム・データベースでは、COLLADA スキーマから直接導き出した構造体を使います。この構造体の C++での定義は、自動的に生成されるので、常に正確で他と矛盾することがありません。この両者の一致が保証されるということは、DOM の開発が進行しても、COLLADA スキーマとの同期が維持されるということを意味します。

COLLADA DOM の利点

COLLADA DOM の利点は上述のものばかりでなく、COLLADA インスタンス文書を標準の XML DOM パーサを使って読み込むのに比較して、以下のようにさまざまな利点があります。

- COLLADA DOM では、インスタンス文書中の他の要素は、自動的に保持されて、データの保存時に文書に書き戻されるので、意識する必要があるのは、使用もしくは変更したい特定の要素だけです。
- COLLADA DOM では、COLLADA インスタンス文書をロードすると、自動的に URI を解決します。したがって、ユーザ独自の URI 解決プログラムを書いたり、参照先の URI を見つけるために、データを検索したりする必要はありません。
- この API は、COLLADA インスタンス文書の中のテキスト文字列を、適切なバイナリ形式に変換します。たとえば、「1.345」のようなテキスト形式の数値は、C++の浮動小数点数に変換されます。

モジュールの構造

COLLADA DOM フレームワークは、以下の 4 つの基本的な要素から構成されます。

- **オブジェクトモデル**：COLLADA 要素の簡単な作成、操作、および読み書きを可能にする自己反映オブジェクトモデルであるデジタルアセット交換(DAE)オブジェクトモデルと、DAE オブジェクトモデル、および、COLLADA スキーマに基づくカスタム C++ DOM である COLLADA オブジェクトモデルが含まれます。
- **ランタイム・データベース**：COLLADA 要素を管理します。参照実装としては、標準テンプレート・ライブラリ（STL）を使ったものが提供されています。このランタイム・データベースには、COLLADA オブジェクトモデルの特定のインスタンスに対応する C++の構造体、COLLADA オブジェクトモデルの要素をユーザ定義のデータ構造体に変換するためのメカニズム、および、データベース・クエリー・マネージャが含まれています。
- **データベース・インタフェース**：ユーザと、COLLADA のファイルや要素との間のやりとりを可能にします。
- **バックエンド**：COLLADA バックエンドは、外部の COLLADA インスタンス・データから、C++ランタイム COLLADA オブジェクトへの変換を担当するコンポーネントから構成されています。

これらのコンポーネントについては、第 3 章「アーキテクチャ」の節でさらに詳しく説明します。

2 COLLADA DOM 入門

テクノロジーとツール

COLLADA DOM は、VisualStudio™.NET 2003 を使って開発され、パッケージ化されています。全体のフレームワークが、C++で書かれています。ストリーム・ベースの専用 XML パーサは、スキーマ駆動によるインスタンス文書の構文解析を可能にしています。コード生成ツールは、現在のスキーマを忠実に反映する、インポート／エクスポート用の C++構造体を生成します。COLLADA DOM 1.4 のリリースは、COLLADA 1.4.0 のスキーマをベースにしています。

コンパイラの設定

COLLADA DOM を使ったアプリケーションを、VisualStudio.NET を使って正しくコンパイル、リンクするには、追加パスや依存関係の情報が必要です。そのために、プロジェクトに以下の情報を追加してください。

- ［追加のインクルード ディレクトリ］フィールドに、以下を追加します。

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

- ［追加のライブラリ ディレクトリ］フィールドの［全般］タブの下に、以下を追加します。

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

- ［追加の依存ファイル］フィールドの［入力］タブの下に、以下を追加します。

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

また、昔の規格準拠レベルの低い XMLPlugin を使いたい場合には、さらに、libcollada_XMLPlugin.lib も追加します。また、このプラグインだけを使いたい場合には、次の「COLLADA DOM の再構築」を参照してください。
アプリケーション・コードをコンパイルする際の VisualStudio C++コンパイラの「ランタイムライブラリ」設定は、COLLADA DOM ライブラリを構築するプロジェクトをコンパイルする際に使われる「ランタイムライブラリ」設定と一致している必要があります。VisualStudio の中でこの設定をチェックするには、プロジェクトのプロパティに移動して、[C/C++] を開き、さらに［コード生成］を開いて、「ランタイムライブラリ」プロパティをクリックし、可能な設定の一覧を開きます。アプリケーション・コードと COLLADA DOM が、異なる「ランタイムライブラリ」設定でコンパイルされている場合には、リンク時に、意味不明な「満たされてない参照」エラー・メッセージが大量に生成されます。
VisualStudio のライブラリ設定には、デバッグ用とそうでないものとがあるので、実行するビルドの種類に合ったライブラリを使うようにしてください。COLLADA DOM のデバッグ・バージョンは、[マルチスレッド DLL をデバッグする] 設定でコンパイルします。COLLADA DOM ライブラリのリリース・バージョンは、[マルチスレッド DLL］設定でコンパイルします。コンパイル時にライブラリ設定を変更する場合には、アプ

リケーション・コード用の Visual Studio プロジェクトと、COLLADA DOM ライブラリ構築用のプロジェクトのすべてにおいて、「ランタイムライブラリ」設定が同一であることを確認し、全プロジェクトを再コンパイルしてから、再リンクを行ってください。

COLLADA DOM の再構築

ディベロッパは、ほとんどの場合、ビルド済みのライブラリだけを使っていればよいのですが、COLLADA DOM のディストリビューションには、ライブラリを一から再構築するための、ソースコードや VisualStudio プロジェクトもすべて含まれています。
この再構築用のプロジェクトは、デフォルトでは、統合用クラスや入出力プラグインのすべてを含めた DOM 全体をコンパイルします。COLLADA DOM ディストリビューショには、libxml2 2.6.20 のウインドウ用バイナリ・ディストリビューション、および、その依存ファイルである iconv と zlib が含まれています。これらのファイルは、インターネット上のディストリビューション・サイトから、直接入手することができます。デフォルトの入出力プラグインは、daeLIBXMLPlugin です。昔の daeXMLPlugin をデフォルトで使いたい場合には、DAE プロジェクト（もしくは makefile）を変更して、コンパイラ・コマンドの行に「DEFAULT_DAEXMLPLUGIN」を指定します。この操作の詳細については、dae.cpp の冒頭にあるコメントを参照してください。

ファイル

COLLADA DOM を利用するために必要なファイルは、以下の通りです。統合用のテンプレートを利用するために必要な追加ファイルについては、「統合用テンプレートの使用」で説明します。

ファイル名	説明
dae.h	DAE ヘッダ・ファイル
libcollada_dae.lib	DAE ライブラリ・ファイル。
libcollada_dom.lib	COLLADA オブジェクトモデル用のライブラリ・ファイル。
libcollada_STLDatabase.lib	ランタイム・データベース用のライブラリ・ファイル。
libcollada_XMLPlugin.lib	元の XML パーサ用のライブラリ・ファイルであるが、XML 規格に完全に準拠していないので、将来的には削除される予定。
libcollada_LIBXMLPlugin.lib	完全に XML 規格に準拠した、デフォルトの入出力プラグインである libxml2 をベースにした XML パーサ用ライブラリ。

サンプル・プログラム

COLLADA DOM の samples ディレクトリには、以下のようなサンプル・プログラムが付属しています。

Conditioners/Animation

キー・フレーム・アニメーション・チャンネルを含んでいて、キー・フレームの線型補間を使ってスカラ関数曲線を一定間隔でサンプリングするような、COLLADA インスタンス文書をロードします。

Conditioners/Common

コマンドライン引数を解析して、ヘルプや使用法のメッセージを生成する、汎用関数が含まれています。この関数は、他の Conditioner が統一されたインターフェイスを提供するために使っています。

Conditioners/Deindexer

<geometry>要素のインデックスを外して、GL 頂点配列を使ったレンダリング用に最適化します。この例には、<triangles>要素や<controller>要素の内容を再編成する方法、および、<source>要素への参照を解決する方法が示されています。

Conditioners/Filename

インスタンス文書の中から、あらゆる<image>要素を探し出して、その URI に対して簡単な文字列置換を行うことにより、source 属性を変更するという簡単な例です。この例では、単純な絶対 URI を、相対 URI に変換しています。この例にユーザのコードを追加すれば、より細かい変換を行うこともできます。DCC ツールの中には、システムに依存していて、修正しないと汎用的に利用できない URI を出力するものがあるので、このコードはそのような場合に役立ちます。

Conditioners/Optimizer

COLLADA ファイルの中の三角形の順序を修正して、単純な FIFO の頂点キャッシュを持つハードウェア上でのレンダリング効率を最適化します。キャッシュの操作をシミュレートするコードは交換可能なので、この conditioner を拡張して、これ以外の構造を持つキャッシュを扱うこともできます。また、この例では、<triangles>要素の中の<p>要素を再編成する方法も示されています。

Conditioners/Triangulation

COLLADA ファイルの中のプリミティブを、簡単なファンニング・アルゴリズムを使って三角形に変換し、<polygons>要素のすべてが<triangles>要素に置き換えられた COLLADA ファイルを出力します。この例では、既存の<polygons>要素と同一のパラメータや入力を持つ、新たな<triangles>要素の作成方法を示しています。また、COLLADA オブジェクトモデルに、要素を挿入したり削除したりする方法も示しています。

参考資料

「COLLADA DOM リファレンス」

表 1：COLLADA の情報源となる URL

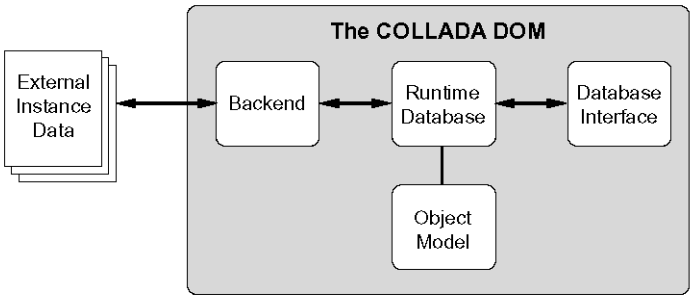
リファレンス	URL
COLLADA ウェブサイト	http://www.khronos.org/collada
XML Schema API 仕様	http://www.w3.org/Submission/2004/SUBM-xmlschema-api-20040309/
COLLADA スキーマのウェブサイト	http://www.collada.org/2005/COLLADASchema/
COLLADA スキーマのドキュメント	http://www.collada.org/2005/COLLADASchema.xsd
W3C XML ウェブサイト	http://www.w3.org/XML/

3 アーキテクチャ

概要

下記の図は、COLLADA DOM の構成要素、および、コミュニケーションの流れを示しています。

図 1 COLLADA DOM



データベース・インタフェース

データベース・インタフェースは、ランタイム・データベースから COLLADA 要素をロードしたり問い合わせを行ったりする手段を、アプリケーションに提供します。また、このインターフェイスは、統合用オブジェクトを登録したり、ランタイム・データベース中のオブジェクトと同等のアプリケーション用データ構造体を要求したりするのにも使われます。

オブジェクトモデル

COLLADA オブジェクトモデルは、COLLADA スキーマから直接生成された C++ オブジェクト（COLLADA DAE オブジェクト）の集合です。各 COLLADA DAE オブジェクトは、API に自己反映性やロバストネスを与えるために設計された、特殊なベースクラスから導出されます。生成されたオブジェクトモデルは、ランタイム・データベースの基盤であると同時に、COLLADA スキーマ要素の C++ 表現でもあります。各 COLLADA DAE オブジェクトを忠実に反映するように生成された統合用オブジェクトには、対応する COLLADA DAE オブジェクトのデータから、アプリケーション固有のデータ構造体への、インテリジェントな変換を行うために書かれたユーザコードのための、プラグイン・ポイントが用意されています。

ランタイム・データベース

ランタイム・データベースは、COLLADA 要素の、動的な常駐のキャッシュとして働きます。このキャッシュは、COLLADA バックエンドからキャッシュに渡される C++ COLLADA 要素を基盤としています。また、ランタイム・データベースは、COLLADA 要素をアプリケーション固有の構造体に変換するメカニズムを提供しています。この COLLADA DOM との統合は、統合用ライブラリ内にあらかじめ定義されたプラグイン・ポイントに、変換コードを挿入することによって実現されます。

バックエンド

COLLADA バックエンドは、外部の COLLADA インスタンス・データから、C++ ランタイム COLLADA オブジェクトへの変換を担当するコンポーネントから構成されています。バックエンドでは、オブジェクトモデルに組み込まれたメタ・パーサを使って、インスタンス・データをインテリジェントに解析・変換します。

外部インスタンス・データ

COLLADA インスタンス文書やデータベース用の、非常駐のストレージです。

データベース・インタフェース

COLLADA データベース・インタフェースは、daeInterface という C++ のクラスとして実装されており、COLLADA 要素のロード、保存、変換、照会などをサポートする仮想インターフェイスを備えています。
DAE クラスは、交換可能な COLLADA バックエンドやランタイム・データベースに対し、単純で汎用的なインターフェイスを提供します。このクラスは、パイプライン全体のラッパとして働き、DOM コンポーネントに対して拡張や交換が行われても、インターフェイスの一貫性が損なわれないことを保証します。
新しい DAE インターフェイス・オブジェクトを作成すると、デフォルト・バージョンの COLLADA バックエンド、COLLADA ランタイム・データベース、および、登録された統合用ライブラリが、自動的に構築され、初期化されます。
下に示すのは、パブリックなデータベース・インタフェース API のサブセットです。完全な API は、dae.h の中に定義されています。

```
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);
```

COLLADA DOM は、他の API コンポーネントのパフォーマンスに影響を及ぼすことなく、基盤となるデータベース・メカニズムの拡張や交換を行うことができるように設計されています。COLLADA DOM に含まれるフロントエンドのインターフェイスは、その API を通じて、汎用的なクエリーを、任意の構成のバックエンドに渡すことができます。バックエンドによって生成されたクエリーの結果は、元のクエリーに対する応答として返すために、ランタイム・キャッシュに返されます。さらに、このフロントエンドのインターフェイス自体も、より堅牢もしくは具体的なクエリー機能セットを提供するために、拡張することができます。

オブジェクトモデル

COLLADA DOM は、自己反映オブジェクト・システム（ROS）を持つ C++ オブジェクトモデル、および、COLLADA XML Schema の定義に直接対応するオブジェクトモデルの上に構築されています（「図 2 COLLADA オブジェクトモデル」を参照）。

自己反映オブジェクト・システム（ROS）

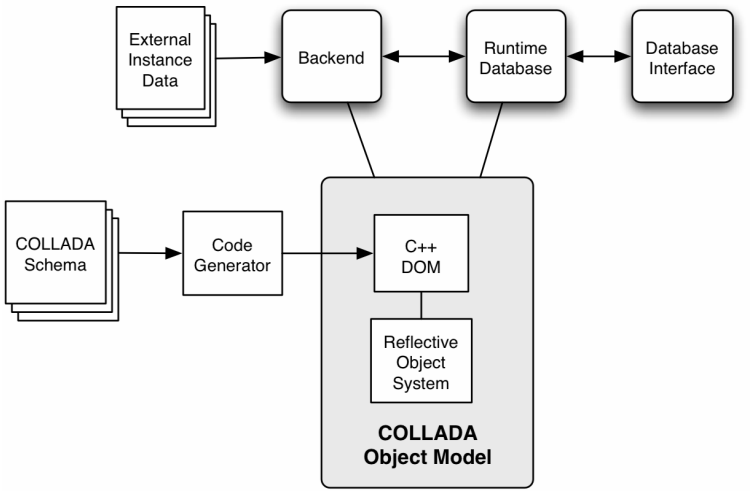
COLLADA DOM は、自己管理・自己反映オブジェクト・システム（ROS）を定義する機能の集合の上に構築されています。ROS は、DOM オブジェクト、および、その構造(メタデータ)や内容を定義するデータの作成、管理、操作を行います。コード中で接頭辞 dea を持つクラスは、すべて、自己反映オブジェクト・システムの一部です。ROS は、COLLADA 用の COLLADA オブジェクトモデルを定義するメタデータによって構成され

ます。COLLADA オブジェクトモデル・クラスは、実行時に ROS によって生成され、インスタンス・データをロードされてから、ランタイム・データベース（「ランタイム・データベース」を参照）に渡されます。

COLLADA オブジェクトモデル

COLLADA オブジェクトモデルは、COLLADA スキーマで定義された要素の、C++ における等価表現です。COLLADA オブジェクトモデルのクラス階層は、COLLADA スキーマ要素定義を C++ クラスに変換するコードジェネレータを使うことにより、すべて、自動的に生成されます。この COLLADA オブジェクトモデル構造体コードの自動生成は、スキーマとの整合性やコードの正確さを保証し、スキーマ更新時の追加作業の必要性を省きます。COLLADA オブジェクトモデルのクラスには、すべて、接頭辞に dom がついています。生成されたコードには、COLLADA オブジェクトモデルの各要素の構造に関するメタ情報の登録機能が組み込まれています。要素およびそのサブ要素についてのメタデータは、どの COLLADA オブジェクトモデルにも用意されている `registerElement()` 静的メソッドによって ROS に登録され、それにより、それぞれの COLLADA DAE オブジェクトの理解、追跡、管理が可能になります。

図 2 COLLADA オブジェクトモデル

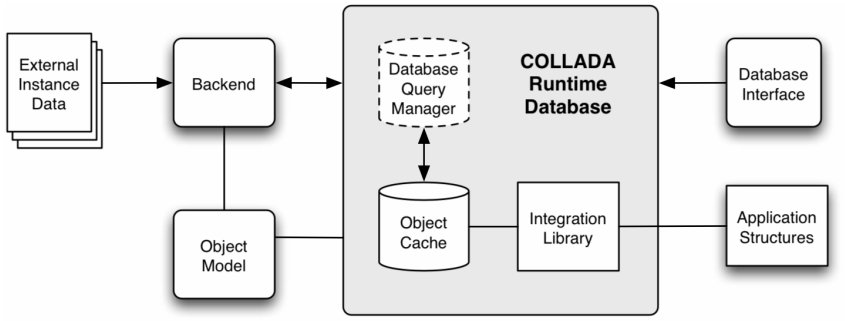


この登録されたメタ情報は、ROS に直接組み込まれたパーサによって、インスタンス・データの解析や検証のために使われます。この「メタ・パーサ」は、スキーマを単に（オプションの）検証メカニズムとして使うだけでなく、ドキュメントの構文をインテリジェントに解析するためにも使います。このような、スキーマによって定義されたメタデータとの緊密な連携は、スキーマが進化するにつれ、COLLADA DOM フレームワーク全体を自動的にスキーマに追従させることを可能にしています。

ランタイム・データベース

COLLADA DOM の中核を成しているのは、COLLADA 要素の常駐のキャッシュである、ランタイム・データベースです。このランタイム・データベースは、daeDatabase という仮想インターフェイスを持つ C++ オブジェクトとして実装されています。

図 3 COLLADA ランタイム・データベース



C++ オブジェクト・キャッシュ

キャッシュ内の各 COLLADA 要素は、COLLADA スキーマの中で定義された構造体を反映した COLLADA 定義の構造体の、C++ オブジェクト階層により構成されています。このランタイム・データベースでは、データベース・インタフェースを通して、名前によって簡単に COLLADA オブジェクトを要求することができます。このキャッシュは、必要に応じてメモリに出入りするキャッシュ・ラインから構成されており、他の COLLADA 要素に対する参照を自動的に解決します。

統合用ライブラリ

COLLADA 統合用ライブラリは、DOM オブジェクトを忠実に反映するように生成された、テンプレート・クラスの集合です。統合用のクラスには、すべて、接頭辞に int が付きます。各統合用クラスは、daeIntegrationObject というベースクラスから導出されます。このベースクラスは、COLLADA データをアプリケーション固有の構造体に変換するためのセマンティクスを定義する、仮想インターフェイスを提供しています。COLLADA DOM 統合用ライブラリを出発点にすれば、ディベロッパがやらなくてはならないのは、特定の C++ の構造体を別の構造体に変換する変換関数を書き込んで、オブジェクトの対応を定義することだけです。統合用ライブラリのテンプレートの使い方の例については、第 6 章「統合用テンプレートの使用」を参照してください。

データベース・クエリー・マネージャ

現時点では、COLLADA DOM にはクエリー・マネージャが含まれていませんが、このようなコンポーネントの追加は、ランタイム・データベースの拡張として効果的であると考えられるので、ここで言及しておきます。COLLADA バックエンドは、基本的には、持続性をもつデータ・ストアへのアクセスを提供するラッパです。現在提供されているメカニズムでは、ローカルな XML ファイルを利用していますが、将来的には、バイナリ・ファイル形式や完全なデータ・ベース・システム用の拡張を実装することが予定されています。ランタイム・データベース中にインテリジェントなクエリー・マネージャがあれば、特定のバックエンド・データ・ストアと協調して、クエリーやデータの取得を、特定のアプリケーション用に最適化することができます。クエリー・マネージャの API は、このような拡張を容易にすることを狙っています。

COLLADA バックエンド

COLLADA バックエンドは、クエリーを受け取って、インスタンス化された COLLADA DAE オブジェクトをランタイム・データベースに返す仮想インターフェイスをもつ、daeIOPlugin という抽象クラスから構成されています。この API には、ローカルな UTF-8 XML ファイルへのアクセスを提供する daeXMLPlugin と、libxml2 ライブラリを使って、完全に規格に準拠した XML 文書のロード機能を提供する daeLIBXMLPlugin の 2 種類の実装が含まれています。このインターフェイスの将来の実装では、バイナリ・ファイルやデータ・ベース・システムのようなこれ以外の種類の持続性を持つデータ・ストアに対する入出力モジュールも提供される予定です。

4 ランタイム・データベースの利用

この節では、COLLADA ランタイム・データベースに COLLADA インスタンス文書をロードする方法、ランタイム・データベースを COLLADA インスタンス文書に保存する方法、および、データベースに対して問い合わせを行う方法について説明します。また、データベース、コレクション、要素などの用語の定義も行います。

コンセプト

「データベース」という用語は、daeDatabase オブジェクトによって表された、COLLADA 要素の常駐のキャッシュであるランタイム・データベースを指します。データベース中の要素は、コレクションというグループにグループ化されます。
COLLADA DOM では、「要素」という用語は、<geometry>、<image>、<scene> など、任意の COLLADA 要素を指します。要素を定義するベースクラスは daeElement で、COLLADA DAE オブジェクトはすべて daeElement から導出されます。各要素は、他の要素を子要素として持ったり、他の要素の子要素になることができます。要素はコレクションの一部で、1 つの要素は特定の 1 つのコレクションにのみ所属します。
コレクションは、URL に基づいてグループ化された、COLLADA 要素の集合を表します。COLLADA DOM のデータベースには、daeCollection オブジェクトによって表された、任意の数のコレクションを含むことができます。COLLADA データの XML インスタンス文書表現を利用し、DAE::load(xmlUrl) メソッドを使って新しい COLLADA XML インスタンス文書をロードすると、そのたびに、データベースの中に新しいコレクションが生成されます。DAE::saveAs(url,name) メソッドは、コレクション名の指定に基づいて、特定のコレクションをインスタンス文書に書き込みます。DAE::saveAs(url,idx) メソッドは、コレクション・インデックスの指定に基づいて、特定のコレクションをインスタンス文書に書き込みます。

COLLADA データのロードとセーブ

COLLADA データをロードするには、まず、COLLADA DOM を初期化する基本的なクラスである DAE オブジェクトを生成し、データベースと COLLADA バックエンドとの間の接続を可能にします。このクラスには、データの、データベースへのロードや、データベースからのセーブを行うメソッドが定義されています。

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

インスタンス文書から既存の COLLADA 要素のコレクションを読み込むには、DAE::load()メソッドを使います。コレクションを URI からロードした場合、COLLADA DOM は、検索などのコレクションを特定する必要のある操作を行う際に、その URI をコレクションの名前として使います。この例では、ロード処理によって生成されるコレクションの名前は、ロード元の URI と同じ「file:\\\input_url_name.xml」になります。

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

データベース中の要素に対してなんらかの変更を行った後、DAE::saveAs() メソッドを使って、データベースの内容を新たな URL に書き込み、特定のコレクションをセーブします。セーブするコレクションの選択は、コレクション名やコレクション・インデックスを使って行うことができます。

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

XML はテキスト形式であり、手作業で簡単に編集することができる（したがって、間違う可能性がある）ので、DOM 関数からの戻り値のエラーは、かならずチェックするようにしてください。そうしないと、エラーの一部が見落とされ、データ損失につながる可能性があります。

注意：COLLADA DOM XML パーサは、データを完全には検証しないので、XML 入力中のエラーの一部は検出しても、すべてのエラーを検出するわけではありません。新しい COLLADA データを DOM にロードする際には、その前に、データに対して XML バリデータを実行したほうがよいかもしれません。COLLADA データを書き込むプログラムを開発する際には、テスト期間の間に、そのプログラムの出力に対して、バリデータを実行したほうがよいでしょう。

新しいコレクションの追加

既存のデータベースに対し、新しいコレクションを手作業で追加したいことがあります。たとえば、単一の COLLADA インスタンス文書を読み込んで、その単一の文書を、ライブラリを保持する文書と、シーン情報を保持する文書との、2 つの独立した文書に分けたいことがあります。
既存のデータベースに新しいコレクションを追加するには、daeDatabase::insertCollection() メソッドを使って、新しい daeCollection オブジェクトを生成します。コレクションには、すべて、ルートとなる domCOLLADA オブジェクトが必要です。insertCollection() は、このルートの作成や初期化も行います。また、新しいコレクションの名前を指定する必要もあります。この名前は、通常は、文書を最終的にセーブする URI にします。この URI は、この文書への参照を解決する際に使われるので、ロードした他の文書の中でこの新しい文書を参照したい場合には、insertCollection に指定する URI が間違っていないことが重要です。新しいコレクションが生成された後、特定のコレクションから別のコレクションに要素を移動する際には、daeElement::placeElement() を使います。特定のコレクションから別のコレクションに要素を移動する処理については、「DOM オブジェクト、および、複数のコレクションでの作業」でより詳しく説明します。
新しいコレクションを挿入するには、たとえば、以下のようにします。

```
// DAE* オブジェクトは、daeObject 変数によって定義される。

// daeCollection 変数の定義
daeCollection *collection;

// データベースに新しいコレクションを挿入する。
int res = daeObject->getDatabase()->insertCollection("file:///myDom",
&collection);

// これで、新しいコレクションにオブジェクトを追加することができる。
```

COLLADA データを白紙の状態から作成する

COLLADA DOM を使って要素の集合を生成し、それを空のデータベースに追加して、そのデータベースを COLLADA インスタンス文書に書き込みます。そのためには、新しいデータベースを生成して、そこにコレクションを追加する必要があります。各コレクションには、domCOLLADA 型のルート DOM ノードが必要です。DAE::insertCollection() メソッドを使うと、コレクションと domCOLLADA オブジェクトを生成して、domCOLLADA オブジェクトをコレクションのルートとして挿入する処理を行います。これ以外の DOM オブジェクトは、createAndPlace() や placeElement() メソッドを使って、すべてルートノードの下に追加されます（「要素の追加と削除」を参照してください）。

以下の例は、白紙の状態から DOM を生成して、XML インスタンス文書に保存するための枠組みを設定する手順を示しています。

```
// DAE オブジェクトの生成
DAE *daeObject = new DAE;

// 新しいデータベースの生成
daeObject->setDatabase(NULL);

// daeCollection 変数の定義
daeCollection *collection;

// データベースに新しいコレクションを挿入する。
int res = daeObject->getDatabase()->insertCollection(
        "file:///myColladaDoc.xml", &collection);

if (collection) {
// getDomRoot が domCOLLADA オブジェクトを返す
domCOLLADA *domRoot = (domCollada *)collection->getDomRoot();

if (domRoot) {
// そして、createAndPlace() を使って、残りの DOM オブジェクトを、
// domRoot の下に追加する。たとえば、ライブラリを追加するなら、
domLibrary_geometries *newLib = (domLibrary_geometries
*)domRoot->createAndPlace

(COLLADA_ELEMENT_LIBRARY_GEOMETRIES);
    ...

// 新しいコレクションに書き込出す。引数は不要。なぜなら、
// デフォルトでは、save はコレクション 0 (ロードされた唯一のコレクション) を保存して、
// insertCollection で指定される URI に書き込むから。
// また、異なる URI に文書をセーブしたい場合には、saveAs を使ってもよい。
int res = daeObject->save();
}
}
```

データベースへの問い合わせ

現在の COLLADA DOM のバージョンでは、データベースに問い合わせを行って、特定の要素に関する情報を得るために、daeDatabase::getElementCount()および daeDatabase::getElement()の 2 種類のメソッドを提供しています。
getElementCount() メソッドは、特定の種類の要素の数を返します。たとえば、以下のコードでは、データベースの中に <image> 要素がいくつあるかを問い合わせます。

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

型以外の getElementCount() のパラメータは、より細かい指定の要求を行うのに使われます。最初のパラメータは、要素の名前や ID を表します。3 番目のパラメータは、コレクションの名前を表しており、要素が複数のインスタンス文書からロードされている場合に、使われることがあります。
getElement() メソッドは、データベースに対して特定の要素を返すことを要求します。たとえば、以下のコードでは、インデックス値が el_number に一致する「image」要素を返します。

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

要素自体は、最初のパラメータに返されます。3 番目パラメータには、クエリーを特定の名前の要素に限定し、5 番目パラメータは、検索を特定のコレクションに限定します。
getElementCount() と getElement() は通常はペアで使われ、まず、特定の名前・型・コレクションの問い合わせに一致する要素の数を取得してから、getElement() メソッド を使って、それらの要素に対する処理を繰り返します。たとえば、データベース中の全画像に対して処理を行いたい場合には、以下のようにします。

```
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);
/* この画像に対する処理 */
}
```

また別の例として、「landImage」という名前の画像のすべてに対して処理を行いたい場合には、以下のようにします。

```
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);
 /* この画像に対する処理 */
}
```

getElement() に渡されるインデックスは、データベース中の要素と直接結びついているわけではありません。インデックスはクエリー自体と関連しています。getElement()メソッドに渡されるインデックスが意味をもつには、getElementCount() と getElement() で使われるクエリーが一致している必要があります。

5 COLLADA オブジェクトモデルを使う

この章では、直接 COLLADA dom* クラスを使って、COLLADA ランタイム・データベース中の要素の情報を利用したり操作したりする方法を説明します。また、COLLADA DOM では、ユーザ定義の構造体を自動的に生成して、そこに COLLADA DAE オブジェクトの情報がコピーされるように設定することもできます。これについては、次の章で扱います。

COLLADA オブジェクトモデル要素の表現

COLLADA オブジェクトモデル要素の構造は、COLLADA インスタンス文書の中の要素構造に、極めてよく一致しています。COLLADA 仕様で定義された要素は、それぞれ、daeElement クラスから導出された、対応する dom* クラスに割り当てられます。クラスの名前は、対応する要素のタグに基いて生成されます。
たとえば、<camera> 要素は domCamera クラスに、<bool_array> 要素は domBool_array クラスに、<source> 要素は domSource クラスに割り当てられます。
また、要素の属性は、dom* オブジェクトのデータ・メンバーに、要素の子要素は、子要素の型に一致するように型付けされたデータ・メンバーに割り当てられます。同じ型の子要素を複数の持つことのできる要素の場合には、このデータ・メンバーは配列になります。
たとえば、COLLADA 仕様の定義によれば、<geometry> 要素には id と name の 2 種類の属性が含まれており、1 個の <mesh> 要素と、任意の数の <extra> 要素を、子要素として持つことができます。
domGeometry クラスには、属性用のデータ・メンバーとして attrName と attrId を持ち、オブジェクトの domMesh 子要素を保持するためのデータ・メンバーとして elemMesh、任意の数の domExtra 子要素を保持するデータ・メンバーとして elemExtra_array を備えています。

注意：DOM は、元の COLLADA インスタンス文書の構造を完全に維持するわけではありません。ファイルの解釈に影響を与えない情報の一部は、再配置されたり、変更されたりする可能性があります。つまり、COLLADA インスタンス文書を読み込んで、なんの変更もせずにすぐ書き出したとしても、読み込む前の文書と書き出した後の文書は、(機能的には同等ですが) まったく同じになるとは限らないということです。ですからたとえば、XML タグ内の属性の順序であるとか、浮動小数点数の精度や出力形式が異なっていることがあります。デフォルト値のままに設定された任意の属性は、出力文書に書き込む際には省略されます。

参照カウント・クラス

DOM では、DOM 中の dom* オブジェクトの追跡を容易にするために、各 dom* クラスに対応する参照カウント用のクラスを使います。daeSmartRef<T> クラス・テンプレートは、<T>および daeElementRef から導出される、参照カウント用のオブジェクトを定義するために使われます。したがって、たとえば、

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

で定義された domCOLLADARef は、domCOLLADA および daeElementRef から導出されます。dom*クラスには、すべて、このような参照カウント用のクラスが用意されています。ほとんどの場合、この domCOLLADARef 参照を直接使う必要はありませんが、createAndPlace()メソッドを使って daeElement をツリーに配置（「要素の追加と削除」を参照）したときには、そのオブジェクトの daeElementRef 版が DOM によって生成されるので、このオブジェクトは、dom*の参照や daeElementRef の参照に型変換することができます。

たとえば、daeElement::createAndPlace() メソッドを使って domLight オブジェクトを生成した場合、生成されたオブジェクトは、domLight*、domLightRef、もしくは daeElementRef に型変換することができます。

注意 : DOM では、daeElementRef 参照カウント用クラスが幅広く利用されています。(daeMetaElement の生成メソッドのように) daeElementRef を返す生成メソッドを使う場合には、その戻り値は、dom* 要素クラスのポインタではなく、daeElementRef で受け取るようにしてください。戻り値を daeElementRef に代入しないと、参照カウントが正しく計算されず、その結果、オブジェクトが自分自身を削除して、初期化されていないメモリへのポインタだけが残るといった事態が引き起こされる可能性があります。通常は、オブジェクトの参照カウントをユーザが自分で変更する必要はありません。したがって、daeElement::ref() や release() などのメソッドを呼び出すことは避けてください。

要素の追加と削除

daeElement::createAndPlace() メソッドは、現在の要素の子要素として新しい要素を生成し、その新しい要素をメソッド呼出し時に与えられた className パラメータに基づいて親要素の適切なフィールドに配置します。親要素から子要素を削除するには、メソッド daeElement::removeChildElement() を使うか、もしくは静的メソッド daeElement::removeFromParent() を使います。

以下の例は、既存の光源ライブラリに、新しい光源を追加する方法を示しています。

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

// 光源ライブラリ (この例では「lightlib」という名前の) を探す。
error = db->getElement((daeElement**)&myLib, 0,
                       "lightLib", COLLADA_ELEMENT_LIBRARY);

// 新しい光源をライブラリに追加する
domLight *newLight = (domLight *)myLib->createAndPlace(COLLADA_ELEMENT_LIGHT);

// これで、新しい光源にデータを追加することができます。
```

以下の例は、光源ライブラリから光源を削除する方法を示しています。

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

// 取り除きたい光源 (「frontLight」という名前の) を探す
error = db->getElement((daeElement**)&myLight, 0,
                       "frontLight", COLLADA_ELEMENT_LIGHT);

// その親を取得する
daeElement* lightParent = myLight->getXMLParentElement();

// 光源を削除する
daeBool removed = lightParent->removeChildElement(myLight);
```

また、光源のライブラリからの光源の削除は、removeFromParent() を使って行うこともできます。このメソッドの場合、親オブジェクトを自動的に探し出します。この場合、オブジェクトを親から削除することにより、そのオブジェクト自体が削除される可能性があるので、removeFromParent() メソッドは静的メソッドになっています。

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

// 取り除きたい光源 (「frontLight」という名前の) を探す
error = db->getElement((daeElement**)&myLight, 0,
                       "frontLight", COLLADA_ELEMENT_LIGHT);

// 光源を親から削除する
daeBool removed = daeElement::removeFromParent(myLight);
```

このメソッドを使ってデータベースへの要素の追加や、データベースからの要素の削除を行えば、COLLADA DOM によって管理されている要素管理情報が、適切に更新されることが保証されます。たとえば、多くの DOM クラスに含まれる _contents 配列には、その DOM クラスの子要素すべての生成順序が記録されており、読み込みや生成のときと同じ順序で、子要素を出力できるようになっています。DOM は、新しい子要素を生成・削除する際に、このメソッドを利用することによって _contents 配列を自動的に管理します。

注意：この処理はエラーを起こしやすいので、_contents 配列を手作業で更新したり、直接操作したりすることはお勧めできません。さらに、DOM の将来バージョンでは、_contents 配列がパブリック・メンバーとして公開されない可能性もあります。

要素のコピー

DOM 要素をコピーする際には、代入演算子の代わりに、ディープ・コピー操作を提供する daeElement::clone() メソッドを使います。clone() はコピーされた URI の解決を自動的には行わないので、要素の複製が済んだ後、コピーされた URI をすべて解決する必要があります。
DOM の daeString 型は、daeElement クラスから導出されるさまざまなオブジェクトで使われていますが、C 方式の文字列で、C++ の string クラスではありません。代入演算子を使って daeString をコピーすると、元の文字列が削除されたときに、宙ぶらりんのポインタが生じる可能性があります。

daeTArrays の操作

DOM では、daeElement クラスから導出されたオブジェクトのデータ・メンバーの多くのベースクラスとして、テンプレート・クラス daeTArray のオブジェクトを使います。daeTArray は親要素と子要素の間のリンクを表したり、異なるデータ型のリストを保持したりするのに使われます。daeTArray オブジェクトを操作する際には、配列の中に含まれるデータの型によって、配列を操作するメソッドを変える必要があります。
daeTArray の中に、daeElement から派生したオブジェクト (domInput_Array、domParam_Array、domP_Array など) が含まれている場合、配列への要素の追加には createAndPlace()、配列からの要素の削除には removeChildElement() を使います。子要素の配列中の特定のインデックスに、明示的に要素を配置する必要がある場合には、createAndPlace() と同じことを行いつつ、_contents 配列中の特定のインデックスに要素を配置する、daeElement::createAndPlaceAt() メソッドを使います。
daeTArray に、daeElement から派生したのではないオブジェクトが含まれている場合には、append() メソッドを使って新しい配列要素を追加します。appendUnique() メソッドは、該当する用度が配列の中に存在しない場合にのみ、配列要素を追加します。insertAt() メソッドを使うと、配列中の特定のインデックス位置に新しい要素を挿入することができます。このとき、必要に応じて、配列のサイズも変更されます。また、set() メソッドを使うと、特定のインデックス位置の値を変更することができます。特定の値をもつ配列要素を削除するには remove() メソッドを、特定のインデックス位置にある配列要素を削除するには removeIndex() を使います。

注意：daeElement から派生したクラスのメンバーである配列に対して、daeArray::clear() メソッドを使うことは避けてください。clear() メソッドは、_contents 配列を更新しないので、削除されるように見えるデータも、データベースの保存時に、COLLADA インスタンス文書に書き込まれてしまいます。

以下の例では、この 2 種類の配列にデータを追加する方法を示しています。この例の中の thisTriangles は、domTriangles へのポインタで、ここでは、その p_array に新しい三角形を追加します。p_array の内容は daeElement の派生クラスですが、p_array の中の _value 配列は、単純な整数のリストです。

```
// 新しい三角形を追加する、domP は daeElement の派生クラスなので、
// createAndPlace を使う。これにより、p_array に新しい要素が追加される。
domP* p_triangles =
         (domP*)thisTriangles->createAndPlace(COLLADA_ELEMENT_P);

// p 要素の数を追跡しているカウントのデータを更新する
thisTriangles->setCount( thisTriangles.getCount()+1);

// append を使って、domListOfInts 型の p の _value 配列に
// インデックスを追加する
p_triangles->getValue().append(1);
p_triangles->getValue().append(2);
p_triangles->getValue().append(3);
```

注意:createAndPlace() や removeChildElement() を呼んでも、domPolygons、domTriangles、domAccessor や domFloat_Array のような一部の DOM オブジェクトの中のカウント・データのフィールドが、自動的に更新されるわけではありません。たとえば、domPolygons の p_array の要素の数を変更した場合、カウント・データのメンバーをディベロッパが更新する必要があります。

どちらの種類の配列でも、データを取得する際には、get()、もしくはインデックス演算子[] と getCount()メソッドを使います。また、配列の中から特定の値を探すには、find() メソッドを使います。以下の例は、mesh 上のポリゴン集合に対して繰り返しを行う方法を示しています。thisMesh オブジェクトは、domMesh へのポインタです。

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

URI 参照の操作

COLLADA 要素が他の要素を参照する際には、ユニフォーム・リソース識別子（URI）が使われます。たとえば、COLLADA インスタンス文書の中で定義された、ID「Lt-Light」をもつ光源は、<instance_light url = "#Lt-Light">という構文を使って参照することができます。以下の例では、light ノードの要素は、LIGHT ライブラリの中の light 要素を参照しています。

```
<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>
```

COLLADA DOM では、URI を表すためのデータ構造体として、daeURI を使います。この構造体を使うと、daeURI が参照している daeElement を取得したり、さらに、daeURI の参照先を新たな daeElement に変更したりすることができます。
daeURI::element データ・メンバーは、daeURI オブジェクトによって表される daeElement を指しています。URI 参照は、すべて、コレクションのロード時に解決されるので、通常、element データ・メンバーには、daeElement への参照が割り当て済みになっているはずです。
URI が解決されていない場合には、daeURI::resolveElement() メソッドを使えば、daeElement への参照を daeURI::element メンバーに割り当てることができます。URI が、データベースにロードされていないインスタンス文書を指している場合には、このメソッドは、そのインスタンス文書を新しいコレクションにロードして、その新しいコレクションから daeElement を取得します。その解決ステータスは、*daeURI::state* に入れられます。この値が daeURI::uri_success である場合には、URI が無事解決されたことを示しています。
以下の例では、URI の示す要素を取得して、必要に応じて、参照を解決する方法を示しています。このコードでは、light <node> の <instance> 要素の示す daeElement を探しています。

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

// 「Light」という名前のノードを取得する
error = db->getElement((daeElement**)&lightNode, 0, "Light",
COLLADA_ELEMENT_NODE);

// lightNode のインスタンス配列の最初の要素を取得する
int instCnt = lightNode->getInstance_light_array().getCount();
if (instCnt > 0) {
  domInstance_light *thisInst = lightNode->getInstance_light_array().get(0);

// インスタンスの示す URL を取得する
daeURI *lightURI = &thisInst->getUrl();

// 要素を取得し、必要に応じて参照を解決する
if (lightURI->getState() != daeURI::uri_success) {
   lightURI->resolveElement();
}
domLight *lightRef = (domLight*)(daeElement*)lightURI->getElement();
```

URI が特定の daeElement を指すように設定したい場合には、該当する要素を daeURI::element に設定して、daeURI::resolveURI() を呼び出し、テキスト版の URI を書き込みます。このメソッドでは、daeElement のベース URI と ID の情報を利用して、daeURI データ・メンバーを設定します。
特定の daeURI をコピーする必要がある場合には、daeURI::copyFrom() メソッドを使います。代入演算子を使ってコピーすると、元 daeURI が削除されたときに、問題が発生する可能性があります。

DOM オブジェクト、および、複数のコレクションでの作業

特定のデータベースに、複数のコレクションが同時にロードされている場合には、DOM 関数呼出しを使って、コレクション間の要素の移動を行うことができます。daeElement::placeElement() メソッドは、該当する要素を現在の親およびコレクションから切り離し、新しい親に結びつけ、その daeElement と子要素のすべてを、新しい親と同一のコレクションに配置します。
たとえば、以下のように、単一の COLLADA インスタンス文書をロードして、新しいコレクションを生成してから、placeElement() を使って、シーン・ノードおよびその子要素のすべてを、新しいコレクションに移動することができます。

```
DAE *daeObject = new DAE;
// シーン、および、ライブラリを含むインスタンス文書をロードする
int error = daeObject->load("file:///full_url.xml");

// データベースを取得する
daeDatabase *database = NULL;
database = daeObject->getDatabase();

// シーンの移動先として、新しいコレクションを生成し、
// データベースに挿入する
daeCollection *collection;
int error = database->insertCollection(
                                        "file:///new_collection.xml",
                                        &collection);

// 移動するシーン・ノードを取得する。コレクションにはシーン・ノードが 1 つしかないので、
// インデックス 0 を使って、「full_url.xml」の最初のシーンを取得するだけでよい。
// 戻り値は、thisScene に返る。
error = db->getElement((daeElement**)&myLib, 0,
                                        NULL,
                                        COLLADA_ELEMENT_SCENE,
                                        "file:///full_url.xml");

// シーン・ノード、および、その子要素を、新しいコレクションに移動する
domCOLLADA *domRoot = collection->getDomRoot();
domRoot->placeElement(thisScene);
```

コレクション間で要素を移動する際には、その移動の結果変更される URL に対する参照を、すべて、手作業で解決する必要があります。上記の例では、シーン・ノードおよびその子要素のすべてを、新しいコレクションに移動したあと、シーン中の全ノードについて繰り返し処理を行い、同じコレクション内に存在しなくなった domInstance_* 要素の URL を、すべて変更する必要があります。
また、すでに要素が存在しているコレクションに要素を移動した場合には、ID の衝突も解決する必要があります。COLLADA 要素の多くには ID 属性が含まれているので、その ID は、そのインスタンス文書の中で重複していないことが必要です。上記の例では、そもそも転送先のコレクションが空なので、ID の解決は不要です。
特定の daeElement を複数のコレクションに追加する必要がある場合には、その要素を複製して、その複製をコレクションに追加する必要があります。要素を複製するには、daeElement::clone() を使います。
メソッド removeFromParent() や removeChildElement() は、daeElement を親、および、親の所属するコレクションから取り除きます。一般に、その要素を参照するポインタが他に存在しない限り、

除去された子要素は、自動的に削除されます。除去したのに自動的に削除されなかった要素およびその子要素のすべては、どのコレクションにも所属しませんが、データベース上には存在しています。

注意：daeElement ツリーがどのコレクションにも所属していない場合でも、検索キーにそのコレクション名を指定しない限り、getElement()や getElementCount()メソッドは、そのツリーの要素を返します。ただし、標準の saveAs()メソッドは、どのコレクションにも所属していない daeElements を出力しません。

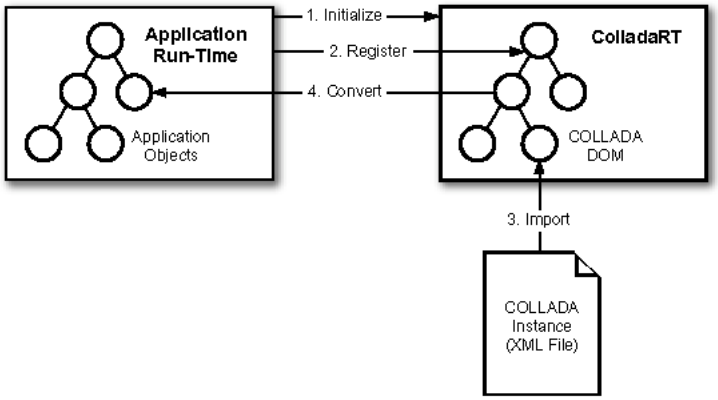
6 統合用テンプレートの使用

この章では、統合用テンプレートを使って、COLLADA DAE オブジェクトをアプリケーション独自のオブジェクトに変換する方法、およびその逆を行う方法を説明します。ここでは、アプリケーションのデータ構造体を、COLLADA ランタイムインフラストラクチャ（COLLADA DOM）と統合する例を、順を追って示します。この例を理解したり、ユーザ独自のアプリケーション固有のデータ構造体に統合したりするのに、COLLADA DOM アーキテクチャについての詳細な知識は必要ありません。

概要

COLLADA オブジェクトモデルとは、COLLADA XML インスタンス文書内の各要素に対応する、ランタイム COLLADA DAE オブジェクトのセットを指します。インポートを行う際には、COLLADA XML インスタンス文書の解析時に構築されます。エクスポートを行うには、COLLADA オブジェクトモデルを構築するコードが必要となる場合があります。COLLADA DAE オブジェクトに含まれるデータをアプリケーション独自のデータ構造へ変換する、またはその逆を行えるようにするために、DOM によってすべてのオブジェクトに対して統合用テンプレートが提供されています。統合用テンプレートには、変換コードを挿入できるプラグインポイントが用意されています。挿入された変換コードは、DOM が文書を COLLADA DAE オブジェクトへインポートした時点で、またはエクスポートのリクエストを行った時点で呼び出されて変換が行われます。

図4　インポート時の COLLADA DOM 統合法のモデル



統合の基本的な手順を以下に示します。

(1) 新しい DAE オブジェクトを作成することによって、COLLADA DOM を初期化する。
(2) アプリケーション独自の統合用ライブラリを登録する。
(3) インポート後に変換するには：
(a) COLLADA XML ファイルをロードしてインポートする。これによりデータはランタイムの COLLADA DAE オブジェクトに配置される。登録された統合用ライブラリを持つ COLLADA DAE オブジェクトによって、対応するアプリケーションデータ構造体が自動的に作成される。
(b) COLLADA ランタイムによって統合用ライブラリ内の変換メソッドが呼び出され、COLLADA DAE オブジェクトの内容が対応するデータ構造体に変換される。
(4) エクスポート前に変換するには：
(a) 対応する COLLADA DAE オブジェクトがまだ存在しない場合には、`createTo` を呼び出して作成する。

(b) COLLADA DAE オブジェクトを保存することによって、COLLADA DAE 構造体をエクスポートする。COLLADA ランタイムによって統合用ライブラリの変換メソッドが呼び出され、ユーザのデータ構造体は COLLADA DAE オブジェクトに変換されてエクスポートされる。

COLLADA DOM 統合用テンプレート

COLLADA インスタンス文書の要素は、インポート時に、ランタイム COLLADA オブジェクトモデルにロードされます。COLLADA DOM では、統合用テンプレートを使用することによって、COLLADA データを、COLLADA オブジェクトモデルとユーザ独自のランタイム構造体との相互変換を可能にしています。
COLLADA オブジェクトモデルの要素は、それぞれが、ヘッダファイル（.h）およびコードファイル（.cpp）によって構成される、固有の統合用テンプレートを持っています。DOM 構造体からアプリケーション独自の構造体に、またはその逆に変換する要素について、対応するテンプレートファイルをアプリケーションディレクトリにコピーします。このコピーすることが、**統合用ライブラリ**をカスタマイズするための開始時点となります。これで、対応する DOM オブジェクトに保存されたデータを独自アプリケーションのオブジェクトへ、またはその逆方向へ変換するためのコードを、統合用ライブラリに追加することができるようになります。このコードのためのプラグイン・ポイントは、テンプレートのソース内のコメントに示されています。
COLLADA DOM には、次の 2 組の統合用テンプレートが用意されています。

- **シンプル統合用テンプレート**: このテンプレートは templates/integrationSimple ディレクトリに入っており、最も頻繁に変換される COLLADA DAE オブジェクトに対してプラグイン・ポイントを提供します。このテンプレートでは、たとえば domAuthor クラスによって定義される、<asset> の中にある<author>要素といった、階層化された XML 要素に対しては、プラグイン・ポイントを提供していません。
- **完全統合用テンプレート**: このテンプレートは templates/integrationFull ディレクトリに入っており、すべての COLLADA DAE オブジェクトに対してプラグイン・ポイントを提供します。

DOM は基本的な COLLADA DAE オブジェクトそれぞれに対して、1 組のテンプレートファイルを提供します。たとえば、domNode オブジェクトを変換するテンプレートファイルの場合は、intNode.cpp と intNode.h となります。

統合用オブジェクト

COLLADA オブジェクトモデルの各要素に対応する統合用ライブラリは、その要素のための統合用オブジェクトを定義するコードを提供します。統合用オブジェクトは、変換に関するあらゆる情報の中核として働きます。このオブジェクトには、変換を実行するためのメソッドが定義され、COLLADA オブジェクトモデルの要素を、変換対象のアプリケーション独自のデータ構造体に割り当てるためのデータ・メンバーが用意されています。
統合用オブジェクトは、daeIntegrationObject クラスによって表現されます。daeElement から派生したクラスはすべて、daeElement::_intoObject というデータ・メンバーによって、統合用オブジェクトを提供しています。
各要素の統合用オブジェクト・クラスは、接頭辞「int」のついた名前で定義されています。たとえば、domGeometry クラスには、intGeometry 統合用オブジェクトが用意されています。
要素のための統合用コードを定義する際に、統合用オブジェクトの_element、および_object というデータ・メンバーを介して、その要素をアプリケーション独自のオブジェクトに割り当てることになります。これらのデータ・メンバーにアクセスするには、daeIntegrationObject::getElement()および getObject()メソッドを使います。

COLLADA インスタンス文書を新しいコレクションにロードすると、DOM によって適切な統合用オブジェクトが自動的に作成されます。特定の要素の統合用オブジェクトを取得するには、daeElement::getIntObject()メソッドを使います。このメソッドを呼び出すと、未変換のオブジェクトに対する変換処理も同時に開始されます。

注意： COLLADA DOM のベータ版では、データベースを保存する前のアプリケーション・オブジェクトから、COLLADA オブジェクトモデルのデータ構造体を更新するためのプラグイン・コードを、自動的に呼び出しません。変更したアプリケーションデータを新しい COLLADA インスタンス文書に書き出したい場合には、アプリケーション・コード中の「to」（つまり、アプリケーションから COLLADA への変換用）プラグイン・ポイントを呼び出して、COLLADA オブジェクトモデル構造体を更新しておく必要があります。これは、通常、daeElement::getIntObject()メソッドを呼び出すことによって可能になります。なぜなら、このメソッドは、対応する COLLADA DAE オブジェクトとアプリケーション・オブジェクトとの相互変換を行うからです。

統合用テンプレートのプラグイン・ポイント

COLLADA オブジェクトモデル要素の統合用クラスには、変換コードを追加することのできるプラグイン・ポイントが、それぞれ 6 つずつ用意されています。プラグイン・ポイントはメソッドとして実装されます。このメソッド本体のコードは、ディベロッパが用意する必要があります。ディベロッパが実装する必要があるのは、自分のアプリケーションに関連するプラグイン・ポイント用のメソッドの本体だけです。

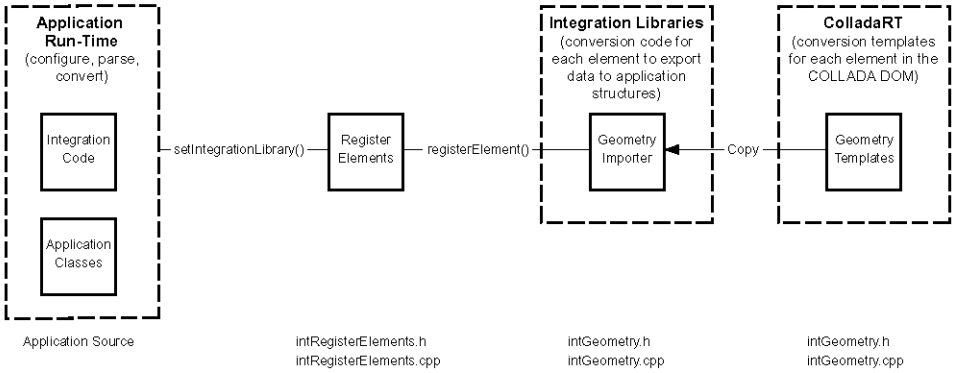
- createFrom()：このテンプレートの DOM クラスに対応するアプリケーション独自のデータ構造体を生成するためのコードを定義します。このメソッドは、DOM クラス用の統合用オブジェクトを設定します。
- fromCOLLADA()： COLLADA オブジェクトモデル・データ構造体を、アプリケーション独自のデータ構造体に変換するためのコードを定義します。
- fromCOLLADAPostProcess()： アプリケーション独自のデータ構造体への基本的な変換を行った後にかならず実行する、任意の後処理コードを定義します。
- createTo()：このテンプレートの DOM クラスに対応する COLLADA オブジェクトモデル・データ構造体がまだ存在しない場合に、データ構造体を作成するコードを定義します（たとえば、インポート中に作成されなかった場合など）。
- toCOLLADA()： アプリケーションのデータ構造体を COLLADA オブジェクトモデルのデータ構造体に変換するコードを定義します。
- toCOLLADAPostProcess()： アプリケーション独自のデータ構造体からの基本的な変換を行った後にかならず実行する、任意の後処理コードを定義します。

図形の統合例

この例では、<geometry>要素の統合の方法を示します。

例では、3 つの基本的な手順があります。まず、対応する統合用テンプレートを、アプリケーション独自のバージョンにコピーします。次に、これらのクラスを COLLADA DOM に登録します。アプリケーションの実行時には、COLLADA DOM の初期化、ファイルのロードの呼び出し、および統合用クラスからのアプリケーション・オブジェクト要求を行うコードを追加します。

図 5　この例で使用されるファイル



関連する統合用テンプレートのコピー

変換コード作成の第一段階は、統合用ファイルのサブディレクトリから、関連する統合用テンプレート・ファイルを、アプリケーションのディレクトリにコピーすることです。この例では、関連する統合用テンプレートファイルは、intGeometry.cpp および intGeometry.h です。コピーしたファイルを変更して、変換コードを組み込みます。

統合用ライブラリを DOM に登録する

DOM が解析時またはエクスポート前に統合用オブジェクトを生成するためには、インフラストラクチャに統合用オブジェクトを登録しておく必要があります。そのためには、使用している各統合用ライブラリに registerElement() メソッドを呼び出す必要があります。これらのクラスを COLLADA DOM に登録する簡易関数は、テンプレートディレクトリに格納されている intRegisterElements.cpp ファイルに定義されています。この関数を、アプリケーション・ディレクトリにコピーします。この例に関連するコードを以下に示します。

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

統合用ライブラリ・ファイルには、intGeometry クラスの定義が用意されています。この内容については後の手順で説明します。

後の「統合用ライブラリの登録を行う」セクションで示すように、intRegisterElements() 関数の定義が済んだら、この関数へのハンドルを DAE::setIntegrationLibrary() メソッドに渡します。

統合用ライブラリのヘッダ・ファイルを設定する

今度は、変換したいオブジェクト用の、新しい統合用ライブラリを編集する手順を開始します。ここでは、intGeometry.h テンプレートからコピーした importGeometry.h 統合用ライブラリを雛形にします。このテンプレートは、修正して、COLLADA オブジェクトモデル構造体の変換元または変換先となるアプリケーション・オブジェクトに関する情報を追加する必要があります。このヘッダでは、変換元または変換先となるクラスを宣言する必要があります。

```
// クラス myGeometry の本体は、アプリケーションのヘッダ・ファイルで定義される。
class myGeometry;
```

さらに、関連する構造体を定義するためのコードを追加して、その構造体を返すメソッドを用意します。これらの定義は、intGeometry 統合用オブジェクト宣言本体の中で行います。

```
class intGeometry : public daeIntegrationObject
{
// ここに intGeometry テンプレート宣言を用意する
・・・

public: // ユーザコード
     virtual ~intGeometry();
     // myGeometry オブジェクトのアクセッサを定義する
     myGeometry *getGeometry() { return _object; }

private: // ユーザコード
     // 統合用オブジェクトのデータ・メンバーの型を宣言する
     myGeometry *_object;
     daeElement *_element;
};
```

アプリケーション・データ構造体を定義する

統合用ライブラリ・ファイルでは、変換する COLLADA DAE オブジェクトに対応する、アプリケーション独自のデータ構造体を作成するコードを提供する必要があります。

この例では、図形を表すアプリケーション独自のデータ構造体をアプリケーション・ヘッダファイルの myGeometry.h に定義します。

```
//  アプリケーションの myPolygon クラスの定義もここに含まれる
class myGeometry
{
public:
   unsigned int _iVertexCount;
   float *_pVertices;
   std::vector<myPolygon> _vPolygons;
};
```

インポート用のアプリケーション・オブジェクトを生成するためのプラグイン・コードを用意する

インポートに関して、この手順に対応するプラグイン・メソッドは createFrom()メソッドです。createFrom()メソッドには、新しい myGeometry オブジェクトを作成し、初期化し、intGeometry 統合用オブジェクトのデータ・メンバを初期化するためのコードを追加します。intGeometry.cpp の中に、以下のコードを追加します。

```
void intGeometry::createFrom(daeElementRef element)
{
   // ジオメトリ情報を格納するクラスを作成し、
   // オブジェクトを空に初期化する
   _object = new myGeometry();
   _object->pVertices = NULL;

   // 統合用オブジェクトの_element データ・メンバーを設定する
   _element = element;
}
```

これで intGeometry 統合用ライブラリが DOM に登録されたので、COLLADA インスタンス文書が COLLADA ランタイム・データベースにロードされ、domGeometry オブジェクトが検出された時には、createFrom()メソッドによって新しい myGeometry オブジェクトが生成されます。

インポート用の変換コードを作成する

今度は、DOM オブジェクトに保存されたデータを、対応するアプリケーション・オブジェクトに変換するためのコードを、統合用ライブラリに追加する必要があります。fromCOLLADA()メソッドは、アプリケーション独自の基本的な変換コードのプラグイン・ポイントです。また、fromCOLLADAPostProcess()メソッドは、変換処理にさらなる柔軟性を与えます。
このコードは、COLLADA DAE オブジェクトやアプリケーション・オブジェクトによっては、かなり異なることがあります。この例では、intGeometry::fromCOLLADA()メソッドを使って、COLLADA DAE 図形オブジェクトから新しい頂点バッファを生成します。

以下のコードは、COLLADA domGeometry オブジェクトから mesh オブジェクトを取得します。

```
// この統合用オブジェクトから図形要素を取得する
domGeometry* geomElement = (domGeometry*)(domElement*)getElement();
domMesh *meshEl = geomElement->getMesh();
```

mesh の取得が済んだら、myGeometry オブジェクト用のアプリケーション独自のデータ構造体 iVertices を構築することができます。下記のコードは、新しい頂点バッファを生成する方法を示しています。

```
// createFrom の呼び出しによって、ロード中に自動的に生成された
// アプリケーション定義の図形オブジェクトへのポインタを取得する。
myGeometry *local = (myGeometry *)_object;

// この domMesh の中にある domPolygons へのポインタを取得する。
// 例をシンプルにするため、1 つの domPolygons を持つ domMesh のみを扱うことにする。

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();

// 例をシンプルにするため、domPolygon は domInput を 1 つだけ保持するものと仮定する。

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

// domPolygons 要素の全ポリゴンに対してループする

for (int i=0;i<polygonCount;i++)
{
        myPolygon myPoly;
```

```
            // このポリゴン（domP）へのポインタを取得する
            domPolygons::domP *poly = polygons->getP_array()[i];
            // domP のインデックス数を取得して構造体に保存する
            myPoly._iIndexCount = poly->getValue().getCount();
            // COLLADA オブジェクトからアプリケーション・オブジェクトへのコピー時に
            // データを加工することができる。
            // ここでは、リストの最初のインデックスを最後のインデックスとして繰り返し、
            // 連続した線分として描画される閉じた図形を作成する。
            myPoly._iIndexCount++;
            myPoly._pIndexes = new unsigned short[myPoly._iIndexCount];
            // domP のすべてのインデックスを自分の構造体にコピーする。
            for (int j=0;j<myPoly._iIndexCount-1;j++)
                  myPoly._pIndexes[j] = poly->getValue()[j];
            // 最初のインデックスを最後のインデックスとして繰り返して、閉じた図形を作成する
            myPoly._pIndexes[j] = myPoly._pIndexes[0];
            // このポリゴンを自分の構造体のポリゴンのリストへプッシュする。
            local->_vPolygons.push_back(myPoly);
    }

    // 使用する頂点を myGeometry にコピーする。サンプルをシンプルにするため、
    // domMesh には domSource および domFloatArray が 1 つだけあるものと仮定し、
    // domMesh は X、Y、Z の形式で指定される頂点の配列であると仮定する。
    // 実際のアプリケーションでは、domPolygons から始めて、domInput、domVertices、
    // domSource、domFloat_array、domTechnique へのリンクを介することによって
    // 頂点を検出するのが通常である。

    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];

    // 1 つの頂点につき 3 つの値があり、ストライドが 3 であると仮定する。
    local->_iVertexCount = floatArray->getCount()/3;
    local->_pVertices = new float[local->_iVertexCount*3];

    // 頂点を自分の構造体へ 1 つずつコピーする。
    // （COLLADA の doubles を floats へ変換する）
    for ( unsigned int i = 0; i < local->_iVertexCount*3; i++ ) {
            local->_pVertices[i] = floatArray->getValue()[i];
    }
```

アプリケーションから COLLADA オブジェクトにアクセスする

これで、統合用ライブラリを登録して、COLLADA データをメモリ内の DOM 構造体にロードし、変換されたデータにアクセスするアプリケーション・コードを追加することができます。

統合用ライブラリの登録を行う

「統合用ライブラリを DOM に登録する」では、intRegisterElements()関数を定義しました。次は、この関数へのハンドルを DAE::setIntegrationLibrary()メソッドへ渡さなければなりません。
この手順では、変換したい要素を DOM に登録します。COLLADA インスタンス文書が DOM へロードされた時点で、変換したい要素が COLLADA オブジェクトモデル構造体から、アプリケーション独自の構造体に変換されます。また、オブジェクトの保存時には逆の処理が行われます。
たとえば、メインアプリケーションのコードでは、以下のようにして、統合用ライブラリの登録関数にハンドルを渡すことができます。

```
// 参照実装をインスタンス化する
daeObject = new DAE;
//統合用オブジェクトを登録する
daeObject->setIntegrationLibrary(&intRegisterElements);
```

COLLADA ファイルを解析する

これで、COLLADA インスタンスを解析して、ランタイム COLLADA オブジェクトモデルに変換する準備ができました。このロード手順では、統合用オブジェクトを生成して、createFrom()および fromCOLLADA()メソッドを呼び出し、データを COLLADA オブジェクトモデルの構造体からアプリケーション定義の構造体に変換します。

```
// COLLADA ファイルをロードする
int res = daeObject->load(filename);
```

COLLADA オブジェクトにアクセスする

COLLADA ファイルを解析してメモリ内のデータベース・オブジェクトにロードした後は、データベースにオブジェクトを要求することができます。ここでは、データベースに最初の図形要素を返すことを要求し、返ってきた要素を pElem に代入します。

```
// ランタイム・データベースに要素の取得を要求する
int res = daeObject->getDatabase()->getElement

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

変換後のアプリケーション・オブジェクトを取得する

下記のコード中の myGeometry クラスは、図形データを含むアプリケーション・オブジェクトを表しています。上で pElem に代入された COLLADA DAE オブジェクトは、（その前の登録のおかげで）対応する統合用オブジェクトへの参照を含んでいます。intGeometry クラスは、COLLADA データの変換を行い、importGeometry.h で定義した getGeometry()メソッドを介して、変換されたデータを myGeometry のインスタンスとして返します。

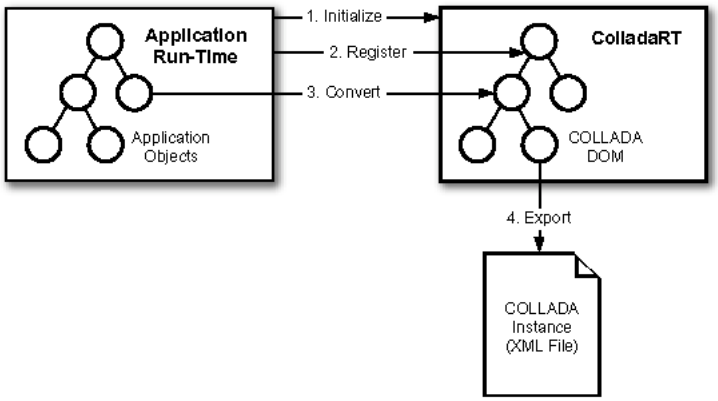
```
// 要素から統合用オブジェクトを取得する
daeIntegrationObject *pIntegrationObj = pElem->getIntObject();
intGeometry *importGeometry =(intGeometry *)pIntegrationObj;

// 統合用オブジェクトからユーザ・データを抽出する
myGeometry geom = importGeometry->getGeometry();
```

統合を用いたエクスポート

前節では、統合用オブジェクトを設定し、COLLADA インスタンス文書をインポートし、データをアプリケーション独自のデータ構造体へ変換する方法を説明しました。
データの値に変更を加え、その結果を COLLADA インスタンス文書に書き出したい場合、その方法はインポートして変換する方法を逆にしたものとよく似ていますが、若干の違いがあります。

図6　エクスポート時の COLLADA DOM 統合法のモデル



COLLADA DOM オブジェクト構造体を変更する

場合によっては、アプリケーションが COLLADA DOM オブジェクトからロードしたアプリケーション・オブジェクト構造体に変更を加えることもあります。データを COLLADA インスタンス文書へ再びエクスポートするには、アプリケーション独自の構造体を変換する前に、変更に対応させるように COLLADA DOM オブジェクトの構造体を変更しなければなりません。サンプルコードでは、この方法は扱いません。

createTo を使って新しい COLLADA DOM オブジェクトを作成する

既存の COLLADA インスタンス文書からインポートしたデータを扱うのではなく、自分のアプリケーションで作成したデータを扱う場合などのように、完全に新しい COLLADA ドキュメントにデータをエクスポートしなければならない場合もあります。このような場合に、統合用オブジェクトを用いてアプリケーション構造体を DOM オブジェクトへ変換してから新しい COLLADA インスタンス文書としてエクスポートするには、アプリケーション構造体からの変換を行う前に、対応する COLLADA DOM 構造体を作成する必要があります。インポートの手順と比較してみましょう。アプリケーションが COLLADA インスタンス文書をロードすると、COLLADA データのロード先となる DAE オブジェクトが、COLLADA DOM によって自動的に作成されます。続いて、カスタマイズした createFrom を、統合用クラスを持つ各要素に対して呼び出して、アプリケーション独自のオブジェクトを作成します。その後に fromCOLLADA を呼び出して DAE オブジェクトからアプリケーション・オブジェクトへデータをコピーします。
新しいアプリケーション・オブジェクトを作成する場合には、そのオブジェクトに対応する COLLADA 要素（および DAE オブジェクト）が存在していません。単にファイルへ保存するだけでは、新しいデータはファイルに書き込まれません。createTo の目的は、COLLADA 要素を作成して、アプリケーション・オブジェクトとの対応付けを行うことです。createTo は新しいアプリケーション・オブジェクト作成時に自動的に呼ばれることはなく、自分で明示的に呼び出さなければなりません。createTo が呼び出された後は、残りのセーブ処理は自動的に行われます。つまり、統合用ライブラリを登録しておけば、セーブ時には統合用クラスを持つ COLLADA 要素すべてについて toCOLLADA が呼び出されて、アプリケーション・オブジェクトのデータが COLLADA オブジェクトへコピーされます。

サンプルコードでは、この方法は扱っていません。createTo メソッドの例を次に示します。

```
void
intGeometry::createTo(void *userData)
{
        // この関数は、アプリケーション定義のオブジェクトから
        // 新しい COLLADA 要素を作成する。
        // COLLADA DOM によって自動的に呼び出されることはない。
        // 新しいアプリケーション独自のオブジェクトを作成する際には、ユーザが自分で
        // この関数を呼び出さなければならない。
}
```

データの値の変更をエクスポートする

COLLADA データの値のみを変更して､改訂した COLLADA インスタンス文書へエクスポートしたい場合には、次に示す例のようにデータの変換を行うことができます。
すでに統合用ライブラリの登録は済んでいるため、この例でのセーブ処理は自動的に行われます。保存時には、統合用クラスを持つ COLLADA 要素それぞれに対して toCOLLADA が呼び出されます。これにより、アプリケーション・オブジェクトのデータが COLLADA オブジェクトへコピーされます。
このコードは、本質的にはインポートの例で示した fromCOLLADA コードと逆の処理を行っていることに注目してください。

```
void
intGeometry::toCOLLADA()
{
        // ランタイムへの変換を行うコードをここに挿入する
        // 以下の行は、テンプレートからのサンプルコードである：
        // myRuntimeClassType* local = (myRuntimeClassType*)_object;
        // element->foo = local->foo;
        // element->subelem[0]->bar = local->bar;

        // このコードは、アプリケーション定義のオブジェクトからデータを取得し、
        // 適切な COLLADA オブジェクトへ戻す処理を行う。

        // COLLADA domGeometry 要素へのポインタを取得する
        // （これが、この関数を呼び出している要素である）
        domGeometry* geometryElement = (domGeometry*)(domElement*)_element;

        // domGeometry の domMesh 要素へのポインタを取得する
        domMesh *meshElement = geometryElement->getMesh();

        // この COLLADA オブジェクトに対応する自分のオブジェクトへのポインタを取得する
        myGeometry *local = (myGeometry *)_object;

        // この domMesh 内の domPolygons へのポインタを取得する。
        // このサンプルをシンプルにするため、
        // domMesh は domPolygons を 1 つだけ保持するものとする
        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();

        // このサンプルをシンプルにするため、
        // domMPolygons は domInput を 1 つだけ保持するものとする

        if(polygons->getInput_array().getCount() != 1)
        {
              fprintf(stderr,
                 "this example supports only one domInput per domPolygons\n");
              return;
        }
        // domPolygons 要素のすべての polygons に対してループをかけて、
        // myGeometry からのデータを domPolygons 要素に戻す。
        // この例では、ポリゴンとインデックスの数は変更されないものと仮定して、
        // 値だけを変更すればよいものとする。

        polygons->setCount(polygonCount);
        for (int i=0;i<polygonCount;i++)
        {
              // このポリゴン（domP）へのポインタを取得する
              domPolygons::domP *poly = polygons->getP_array()[i];
              // 自分の構造体のすべてのインデックスを domP へコピーし直す
              for (int j=0;j< local->_vPolygons[i]._iIndexCount-1;j++)
                     poly->getValue()[j] =
                     local->_vPolygons[i]._pIndexes[j] ;
        }

        // myGeometry の頂点をソースへコピーし直す。
        // この例では、頂点の数は変更されないものと仮定して、
        // 値だけを変更すればよいものとする。
        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];

        // myGeometry の頂点を、COLLADA の float 配列にコピーし直す
        //
        floatArray->setCount(local->_iVertexCount*3);
        for ( unsigned int i = 0; i < local->_iVertexCount*3; i++ ) {
              floatArray->getValue()[i] = local->_pVertices[i];
        }
}
```

更新履歴

本「COLLADA DOMプログラミングガイド」では、以前のバージョンに対して以下の変更を加えました。

第1章

- 「ベータリリースノート」のセクションを削除しました。

第2章　「リファレンス・マテリアル」セクション

- COLLADA ウェブサイトのURLアドレスを修正しました。

フッタ

- フッタの文字列を「PLAYSTATION®3 Programmer Tool Runtime Library Release 0.8.0」に変更しました。