株式会社セガ

SoftimageSDK 入門

COLLADA エクスポーターを作ってみよう

AM開発技術部
立福　寛
Tatefuku_Hiroshi@sega.co.jp

2010/08/26

内容

1　概要

このドキュメントでは Softimage 上で一通り動作するゲーム開発用のエクスポーターを作成する過程を説明します。ドキュメントに従ってエクスポーターを作成していくと、SoftimageSDK の基本的な使い方、ゲーム開発用のエクスポーターの作り方に関する知識を得ることができます。

このドキュメントは発表用のパワーポイントとエクスポーターのソースコードとセットになっています。エクスポーターの詳細はソースコードを参照してください。

2　SoftimageSDK 入門

2.1　API の種類

SoftimageSDK にはスクリプトコマンド API、Softimage オブジェクトモデル、および C++API の３つの API があります。各 API の詳細は SoftimageSDK のマニュアルを参照してください。今回作成するエクスポーターは本体を C++API、インターフェースを Python スクリプトで作成します。

2.2　Softimage 上の Python について

Softimage2010 で Python を使うためには Python と PyWin32 のインストールが必要です。こちらは別途入手してインストールしてください。インストール後にスクリプトエディタのファイル＞設定メニューを開いて、スクリプト言語に Python が選べれば、Softimage 上で Python が使用できます。

2.3　Softimage の環境変数と VisualStudio の設定方法について

Softimage のプラグイン作成において最初に躓きやすいのが環境設定です。作成したプラグインを VisualStudio からデバッグするためには、VisualStudio から Softimage を起動する必要があります。しかし VisualStudio から直接アプリケーション本体の xsi.exe を実行しても Softimage を起動することはできません。Softimage を起動するために必要な環境変数が設定されていないからです。

通常、Softimage は特定のバッチファイル経由で起動します。そのバッチファイルの中で Softimage が使用する環境変数を設定してから、アプリケーション本体である xsi.exe を起動しています。VisualStudio から Softimage を起動する場合は、最初に Softimage の環境変数を設定してから VisualStudio を起動して、そこから Softimage を実行します。

一番簡単な方法は Softimage の SIShell.bat を使う方法です。Windows のスタートメニューから Softimage の SIShell.bat を起動すると、必要な環境変数の設定が終わった状態でコマンドプロンプトが立ち上がります。ここから VisualStudio を起動すれば、設定された環境変数を使うことができます。

この手順で VisualStudio を起動すると以下の環境変数を参照することができます。

```
SDK のルート
XSISDK_ROOT=d:¥Softimage¥Softimage_2010¥XSISDK

xsi.exe の置いてあるフォルダ
XSI_BINDIR=d:¥Softimage¥Softimage_2010¥Application¥bin

ユーザーホーム
XSI_USERHOME=C:¥Users¥TatefukuH¥Autodesk¥Softimage_2010
```

これらの環境変数を使うと VisualStudio のパスの設定を簡単に行えます。Softimage のバージョンが変わってもプロジェクトファイルを変更する必要がないので便利です。

毎回 SIShell.exe から VisualStudio を起動するのが面倒な方は以下の方法を使ってください。これは筆者が普段使用している方法です。

Application¥bin の setenv.bat をコピーしてリネームします。そのファイルを開いて最後の行に VisualStudio の起動コマンドを追加します。このバッチファイルを実行すると環境変数設定済みの VisualStudio が起動してソリューションを開くようになります。

```
"C:¥Program Files¥Microsoft Visual Studio 9.0¥Common7¥IDE¥devenv.exe" "D:¥sample.sln"
```

また Softimage 用のプラグインを作るためには VisualStudio のプロジェクトに対して以下の設定が必要です。

- コード生成＞ランタイムライブラリをマルチスレッドデバッグ DLL もしくはマルチスレッド DLL に設定
- 環境変数を使ってインクルードパス、ライブラリパスを設定
  - インクルードパスに$(XSISDK_ROOT)¥include を追加
  - 追加の依存ファイルに shader.lib sicoresdk.lib sicppsdk.lib を追加
  - ３２ビット環境の場合、追加のライブラリディレクトリに$(XSISDK_ROOT)¥lib¥nt-x86 を追加
  - ６４ビット環境の場合、追加のライブラリディレクトリに$(XSISDK_ROOT)¥lib¥nt-x86-64 を追加
- Debug ビルドのときはデバッグ設定のコマンドに$(XSI_BINDIR)¥xsi.exe を設定
- ビルド後のイベントに copy（プラグイン名）$(XSI_USERHOME)¥Application¥Plugins を追加（プラグインをユーザールートにコピーするため）

以上で VisualStudio の設定は終わりです。この設定を行っておけば作成したプラグインを VisualStudio でデバッグすることができます。

2.4 HelloWorld プラグイン

では実際に簡単なプラグインを作成してみましょう。ここで作成するのは Softimage 上で"Hello World"と表示する HelloWorld プラグインです。

Softimage には自動インストール型のプラグインというものがあります。この形式のプラグインは特定のフォルダに入れておくと Softimage が起動時に読み込んでくれます。今回作成するプラグインはこちらのタイプです。

自動インストール型のプラグインを作るためには、特定の名前の関数を作る必要があります。コマンド名が HelloWorld の場合は以下の関数が必要です。

```
ロード：XSILoadPlugin()
アンロード：XSIUnloadPlugin()
初期化：HelloWorld_Init( )
実行：HelloWorld_Execute()
```

関数の詳細は HelloWorld プラグインのソースコードを参照してください。このソースが Softimage のコマンドプラグインの基本形となります。エクスポーター本体の構造もこちらと同じです。

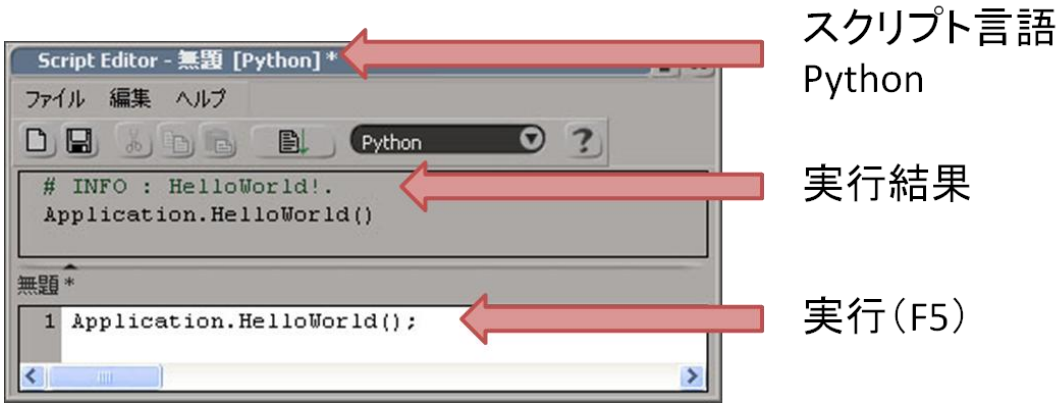
プラグインをビルドして dll が完成したら Softimage 上で実行してみましょう。ビルドされた dll を$(XSI_USERHOME)¥Application¥Plugins にコピーし、VisualStudio からデバッグを開始して Softimage を起動します。

作成したプラグインのロード状態はプラグインマネージャーで調べられます。プラグインマネージャーはメニューのファイル＞プラグインマネージャーから開きます。

プラグインが正常にロードされていれば、こちらの画像のように表示されます。プラグインのロードに失敗した場合は赤いマークが付きます。

PluginManager は便利なツールなので積極的に使っていきましょう。自分の環境では動いているのに、デザイナーの環境に持って行ったら動かなくなった、というケースはよくあります。そういうときは PluginManager を起動してプラグインが正しく読み込まれているかを確認しましょう。

今回作成したのは GUI を持たないコマンドのプラグインなので、スクリプトからこのコマンドを実行します。まず Softimage 上でスクリプトエディタを起動しましょう。スクリプトエディタはメニューの表示＞スクリプト＞ScriptEditor から開きます。



今回は Python を使いたいのでスクリプトエディタのファイル＞設定＞スクリプト言語から Python を選びます。次にスクリプトエディタの下のスクリプト部分に実行したいコマン

ドを入力します。HelloWorld コマンドを実行する場合、以下のスクリプトになります。

```
Application.HelloWorld();
```

F5 キーで実行するとスクリプトエディタの上のログ画面に"Hello World!"と表示されます。これで Hello World プラグインができました。VisualStudio から起動しているのでブレークポイントを設定することができます。適当な行にブレークポイントを設定してみてください。エクスポーターを作るときの手順も基本的には今回と同じです。

今回はコマンドの起動にスクリプトエディタを使用しました。スクリプトエディタは変数の値を調べるときや、サンプルプログラムを実行するときによく使うので操作方法を覚えておくと便利です。

3　エクスポーターの概要

3.1　サポートする要素

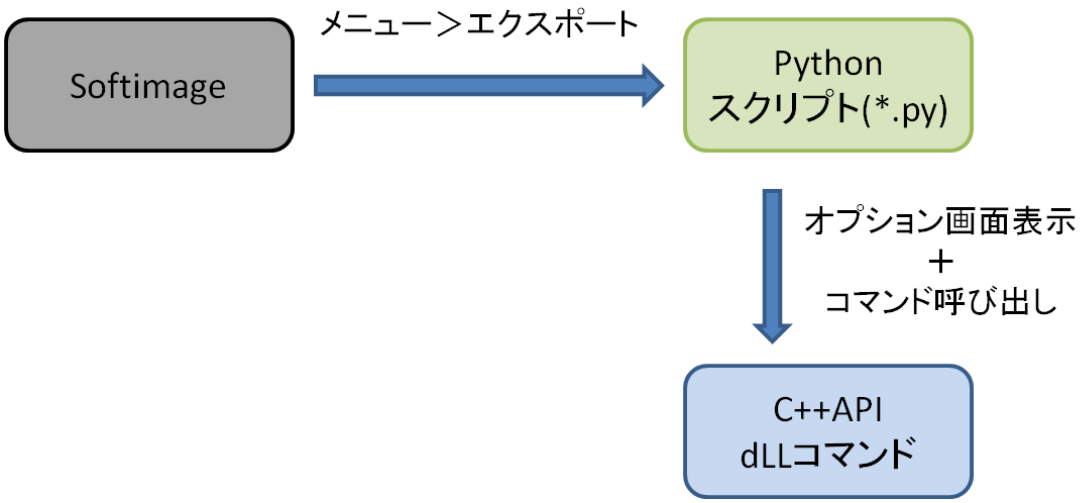
今回のエクスポーターでは以下の要素を出力します。

- ノード（ライト、カメラ含む）
- ジオメトリ（接線、従法線含む）
- マテリアル（レンダーツリー、テクスチャ）
- アニメーション
- スキニング
- シェイプ
- カスタムパラメータ

細かい対応状況は各項目を参照してください。

3.2　エクスポーターの構成

次に今回作成するエクスポーターの構成などを説明します。今回作成するエクスポーターは C++API の dll と Python スクリプトの組み合わせになっています。C++API の dll は引数で指定されたファイル名と設定を使って、シーン情報を取得して COLLADA ファイルを出力するコマンドプラグインです。一方の Python のスクリプトは Softimage のエクスポートメニューに項目を追加して、オプションダイアログを表示し、ユーザーに出力ファイル名やオプションを設定させてから dll のコマンドを呼び出します。



こちらの Python スクリプトの細かい説明はここでは行いませんので、興味のある方はスクリプトのソースを参照してください。

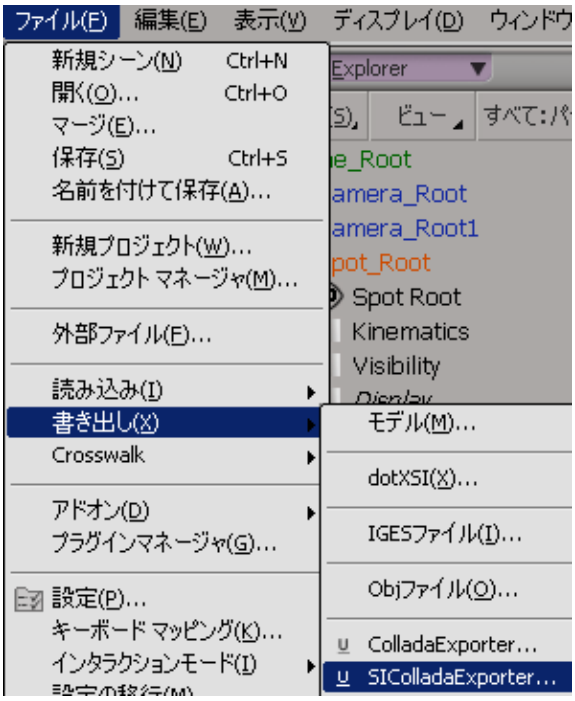
エクスポーター全体を一つの dll にすることもできますが、今回は２つに分けています。このような構成にしている理由は２つあります。最初の理由は Python スクリプトのほうが修正が容易だからです。エクスポーター全体を Python で書きたいところですが、速度上の問題があるのでエクスポーター本体は C++API にしています。２つ目の理由はユーザーがエクスポーターをスクリプトから呼び出せるようにするためです。大量のシーンを自動でエクスポートする場合は毎回手作業でエクスポートすると大変です。エクスポーター本体のコマンドが独立していると、ユーザーが作成したスクリプトからエクスポーターを呼び出すことができるようになり利便性が向上します。またユーザーがエクスポーターのインターフェースを変更することも可能になります。エクスポーターを作成する場合はこれらの点を考慮しておくとよいでしょう。

3.3　エクスポーターの使い方

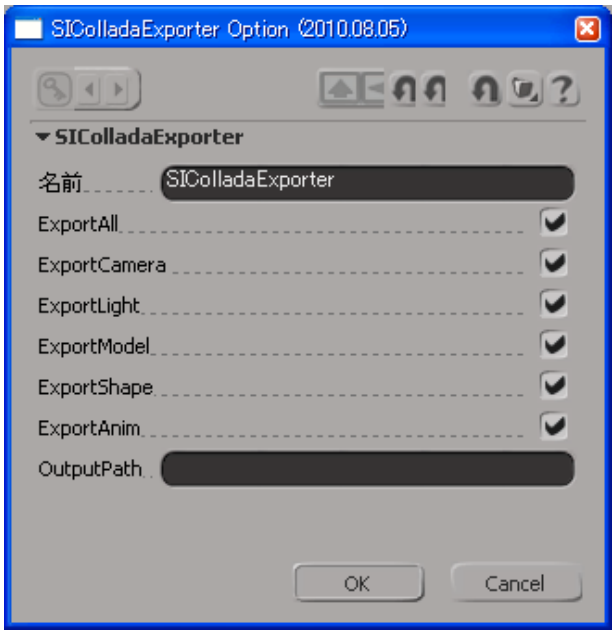
ここでは今回作成したエクスポーターの使い方を説明します。Softimage の SIShell.bat 経由で VisualStudio を立ち上げて、サンプルのソリューションファイルを開いてビルドすると dll が作られます。この dll と python スクリプトをユーザールートの Plugin フォルダにコピーします。

プラグインフォルダの例：
C:¥Users¥Tatefuku¥Autodesk¥Softimage_2010¥Application¥Plugins

Softimage を立ち上げるとメニューのファイル＞書き出しに SIColladaExporter という項目が追加されます。追加されなかった場合は Python のインストール状況を確認してください。



この SIColladaExporter を実行するとエクスポーターのオプション設定ダイアログが開きます。

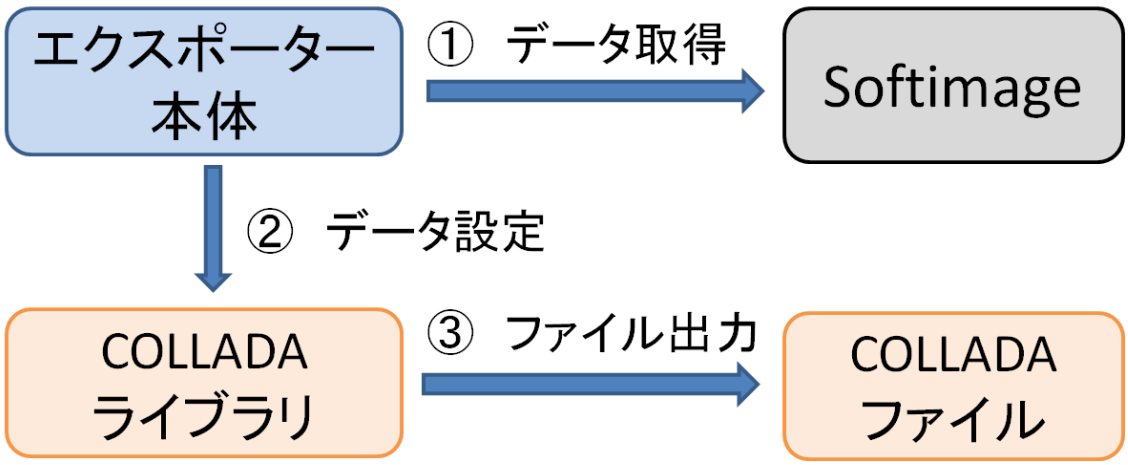


各オプションを設定してＯＫボタンを押し、出力先のファイル名を指定すると Collada(*.dae)ファイルが出力されます。各オプションの機能は以下のとおりです。

- ExportAll：シーン全体をエクスポートします。無効の場合は選択したものだけエクスポートします。
- ExportCamera：カメラをエクスポートします。
- ExportLight：ライトをエクスポートします。
- ExportModel：モデル（ポリゴンメッシュ）をエクスポートします。
- ExportShape：シェイプをエクスポートします。
- ExportAnim：アニメーションをエクスポートします。
- OutputPath：出力先を格納するためのパラメータです。特に指定する必要はありません。

3.4　C++API dll プラグイン

今回作成するエクスポーターの dll プラグインはエクスポーター本体と COLLADA ライブラリの２つから構成されています。また Dll プラグインは以下の３つの処理を行います。



最初に SoftimageSDK 経由でシーン情報にアクセスして必要な情報を取得します。その取得した情報を COLLADA ライブラリに設定して、COLLADA 形式でファイルを出力します。

このようにファイルフォーマット部分と情報を取得する部分を分けておくと、ソースコードがシンプルになるので後々のメンテナンスが楽になります。また他の DCC ツール用のエクスポーターを作るときに、このファイルフォーマットの部分が再利用できるので便利です。

3.5　COLLADA ライブラリ

今回作成するエクスポーターでは独自に作成した COLLADA ライブラリを使用しています。公式サイトで公開されている COLLADA DOM は今回の用途には大きすぎるので採用しま

せんでした。COLLADA ライブラリの詳細はソースコードを参照してください。このドキュメントでは細かい説明は行いません。

3.6　コマンドの設定

C++API で作成するコマンドプラグインは、自分自身のコマンドの引数を設定する必要があります。引数の設定は(CommandName)_Init()関数で行います。今回は SIColladaExporter という名前のコマンドなので、SIColladaExporter_Init()関数となります。

まず SIColladaExporter_Init()関数の引数を CRef から Context クラスにキャストします。Context::GetSource()関数で Command クラスが取得できます。Command クラスは文字通りコマンドの設定を行うためのクラスです。

```
SIColladaExporter_Init( CRef& in_ctxt ) {
        Context ctxt( in_ctxt );
        Command cmd = ctxt.GetSource();
}
```

次に Command::GetArguments()関数で引数の配列を ArgumentArray クラスとして取得します。このクラスの Add()関数で引数を追加することができます。ここでは引数の名前を文字列で追加します。SoftimageSDK に渡す文字列はワイド文字なので L"（文字列）"という形式になります。

```
ArgumentArray args = cmd.GetArguments();
args.Add( L"OutputFilename" );
args.Add( L"ExportAll" );
```

これでコマンドの引数が設定できました。次にこのコマンドが呼ばれたときの引数の取得方法について説明します。

3.7　dll から引数を取得する方法

今回のエクスポーターは Python スクリプトでエクスポーターのオプション画面を表示して、そこで設定された値を引数にして dll のプラグインを呼び出します。

```
# Python スクリプトからの呼び出し
Application.SIColladaExporter( “d:/hoge.dae”, true, … );
```

ここでは dll のプラグインが呼び出されたときの引数の取得方法を説明します。dll のコマンドから呼び出し時の引数を取得するときは Context クラスを使います。Context::GetAttribute()関数で CValueArray クラスの"Arguments"を取得します。

CValueArray::GetCount()で引数の数が取得できるので最初にチェックします。CValueArray クラスは[]によって内部要素にアクセスできるので、順番に引数の値を取得します。

CValueArray クラスの中身は CValue クラスです。CValue は任意の型の値を格納できるクラスなので、適切な型へ代入して中の値を取り出します。

```
// CRef の in_ctxt は関数の引数
Context ctxt( in_ctxt );

// Context から引数を取得
CValueArray args = ctxt.GetAttribute( L"Arguments" );

// 引数の個数チェック
if( args.GetCount() != 8 ) {
        app.LogMessage( "err: arguments number is wrong.¥n" );
        return   CStatus::Fail;
}

int index = 0;

// ファイル名（文字列）
CString filename( args[ index++ ] );

 // ExportAll フラグ（Bool 値）
const bool exportAll = ((bool)args[index++]) ? true : false;
```

これで dll プラグインから呼び出し時の引数を取得することができました。

3.8　SoftimageSDK の文字コードについて

SoftimageSDK では文字列をワイド文字（２バイト）で扱うので、SDK に渡す文字列は先頭に L を付ける必要があります。

```
app.LogMessage( L"hogehoge" );
```

SoftimageSDK には文字列操作クラスとして CString クラスが用意されています。CString クラスにはワイド文字と ASCII の両方を格納できます。今回作成するエクスポーターは文字列を ASCII で扱うので、Softimage から取得した文字列は一度 ASCII に変換する必要があります。ASCII⇔ワイド文字の相互変換は CString クラスで行えます。

```
std::string str( "HOGE" );
CString cstr( str.c_str() ); // std::string から CString に変換
str = cstr.GetAsciiString(); // CString から std::string に変換
```

4　COLLADA

COLLADA とはクロノス・グループが管理している３次元グラフィックスアプリケーション間の交換用ファイルフォーマットです。内部は XML になっており、通常 dae という拡張子が使われます。比較的最近作られたフォーマットなので完成度が比較的高く、多数のゲームエンジンで採用されています。

COLLADA フォーマットに関するドキュメントは公式ページから多数手に入ります（日本語バージョンもあります)。また COLLADA に関する書籍も出版されています。どちらも有益な情報が多いので一度目を通しておいてください。

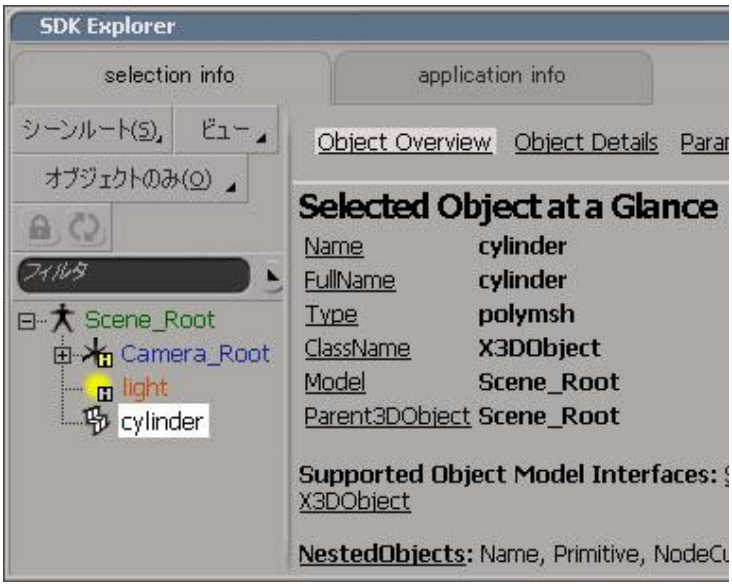
Softimage の CrossWalk による COLLADA ファイルの出力結果も大変参考になるので使ってみるとよいでしょう。COLLADA の規約として定められていない部分の実装を確認するときに便利です。今回はすでに存在するものを作っているわけですが、これは SDK の説明のためです。この点はご了承ください。

5　ノード情報の取得

ここからエクスポーター本体の説明に入ります。

5.1　SDK Explorer

まず一番基本となるノードの構造を取得するのですが、その前にデバッグ用に便利な SDK Explorer というツール紹介します。



SDK Explorer はメニューの表示＞スクリプト＞SDK Explorer から開きます。このツールを使うと現在選択しているノードやプロパティなどの属性、パラメータの値などを簡単に調べることができます。データの中身をのぞきたいときはこのツールを使ってください。プログラムを書いて確認するよりも便利です。

5.2　選択/非選択

それではノードの情報を取得していきましょう。通常、エクスポーターは「シーンの中身を全部出力する」「現在選択されているものを出力する」という二つのモードを持っています。今回作成したエクスポーターではオプションの ExportAll で切り替えることができます。

この機能を実装するためには「現在選択されているノード」を取得する必要があります。Softimage 上で選択されているものを取得するときは Application クラスを使います。Application クラスは現在実行中の Softimage のインスタンスに相当します。このクラスは必要な時にインスタンスを作成してアクセスします。

```
Application app;
```

Application::GetSelection()関数は現在選択されているオブジェクトの一覧を返します。戻り値は CRefArray というクラスで、これは SoftimageSDK で非常によく使われるコンテナです。CRefArray クラスは CRef という参照用のクラスの配列です。

```
Application app;
CRefArray selectionArray = app.GetSelection();
```

CRef クラスは今後もよく出てくるのでここで説明します。SoftimageSDK の多くの関数は結果を CRef クラスの形で返します。ユーザーは受け取った CRef クラスのオブジェクトを適切な型へキャストして使用します。キャストできるかどうかは CRef::IsA()関数か、CBase::IsValid()関数を使います。

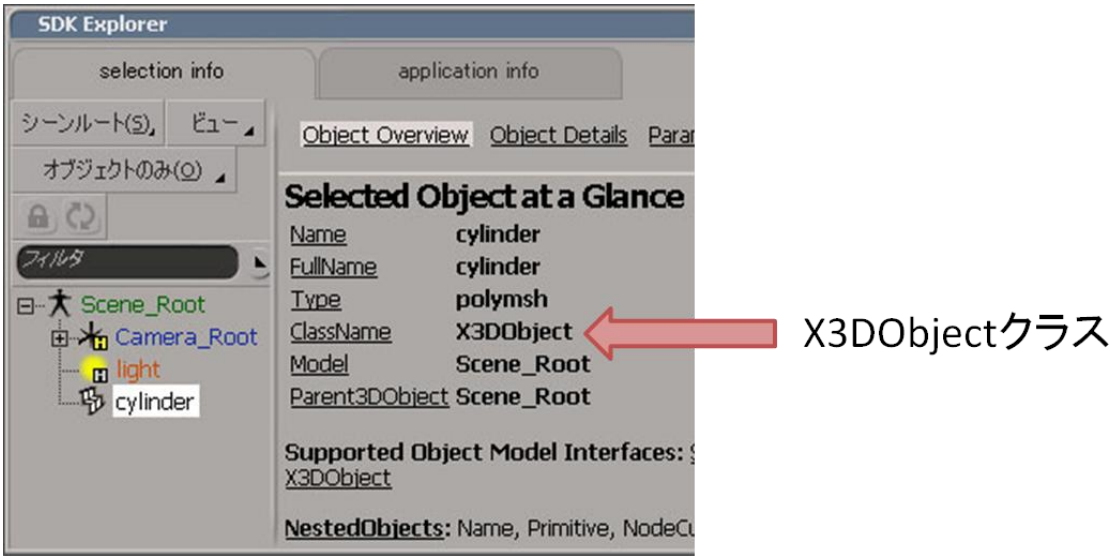
IsA()関数を使うと、CRef クラスから他のクラスへキャストできるかどうか調べることができます。

```
if( selection[ i ].IsA( siX3DObjectID ) ) {
        X3DObject x3DObject( selection[ i ] );
}
```

CBase::IsValid()関数はキャストした後でそのキャストが有効かどうか調べるときに使います。

```
X3DObject x3DObject( selection[ i ] );
        if( x3DObject.IsValid() ) {
                /* x3DObject への処理 */
}
```

Softimage のプラグイン開発では、このようにあるオブジェクトが特定のクラスにキャストできるかどうかを調べながら情報を取得していきます。キャストできるクラスは SDK Explorer で確認することができます。



元に戻って Application::GetSelection()関数の戻り値を調べてみましょう。このコードでは CRef クラスから SIObject クラスへ変換して、変換が有効ならば名前を取得して表示しています。

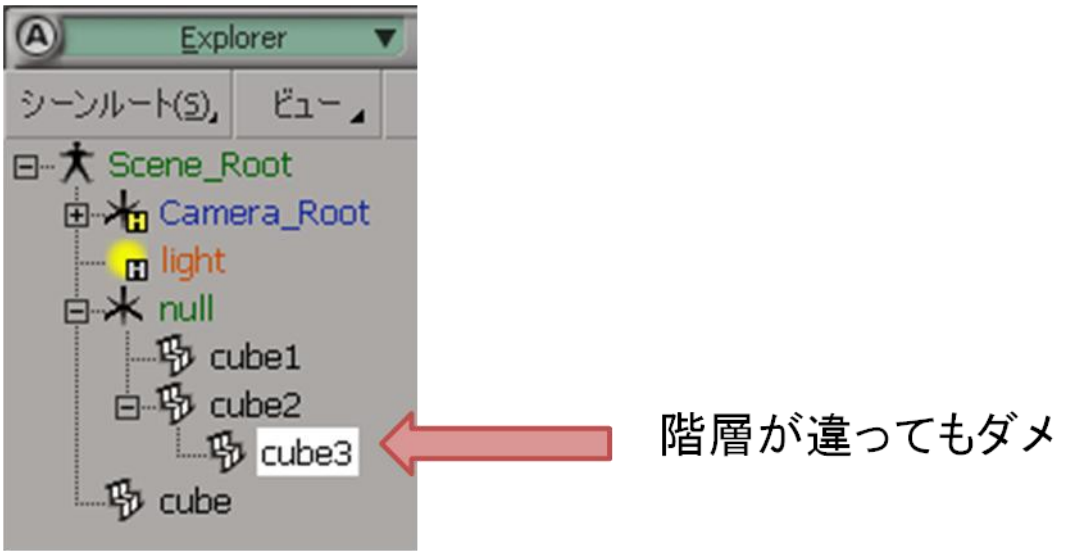
```
Application app;
CRefArray selection = app.GetSelection();
for( unsigned i=0; i<selection.GetCount(); i++ ) {
        SIObject siObject( selection[ i ] );
        if( siObject.IsValid() ) {
                app.LogMessage( siObject.GetName() );
        }
}
```

SIObject は SoftimageSDK のオブジェクトモデルのクラスの一種で、オブジェクトの名前情報を持っています。SIObject::GetName()関数で名前が CString 型で返ってくるので、これを Application::LogMessage()関数で表示します。キャスト後の IsValid()関数によるチェックが不必要な場合は省いても構いません。以後の説明では簡略化のために必要がない限り省きます。

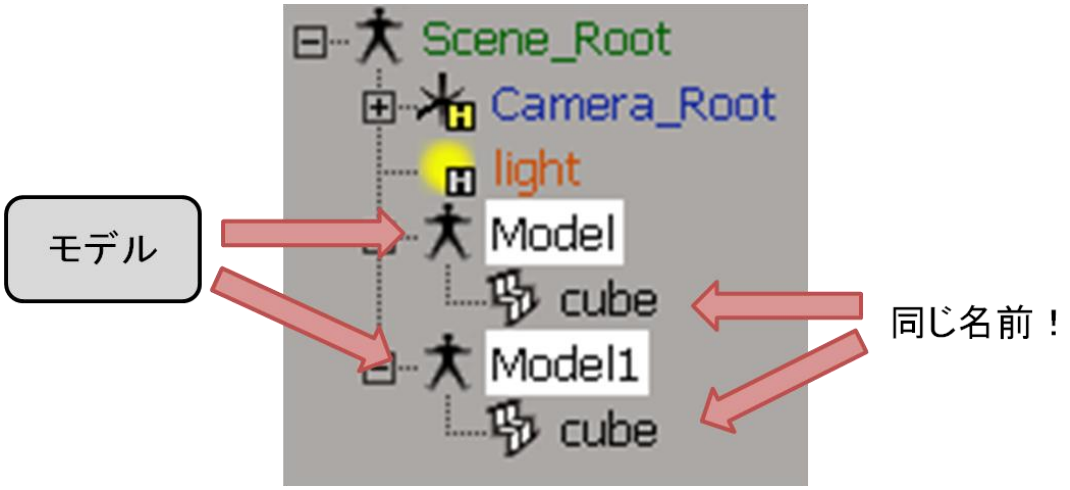
5.3　ユニークな名前

先ほども説明したようにノードの名前情報は SIObject::GetName()で取得できます。一方、COLLADA ではノードにユニークな名前を付ける必要があります。GetName()で取得した名前はそのまま使えるでしょうか？

通常、Softimage のノードの名前は重複しません。階層構造が違っていても同じ名前を付けることができません。つまりノードの名前はすべてユニークになります。

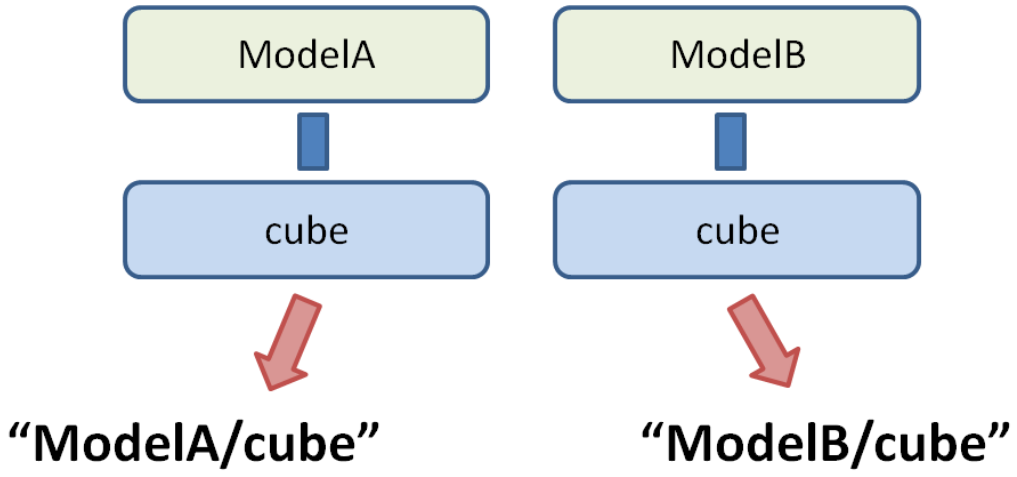


ところが Softimage にはモデルと呼ばれる特殊なノードがあり、この下のノードは名前の重複が許されます。



この画像の場合"cube"という名前はユニークではありません。以上のことから Softimage のノードの名前は「重複する」可能性があります。GetName()で取得した名前をそのまま COLLADA のノードの名前に使えないということです。

エクスポーターでユニークな名前を作る場合、通常は親から子供までの名前を連結した文字列を使います。この方法は同一階層で同じ名前がない限り重複しないので便利です。今回はこちらの方法を使ってユニークな名前を作成します。



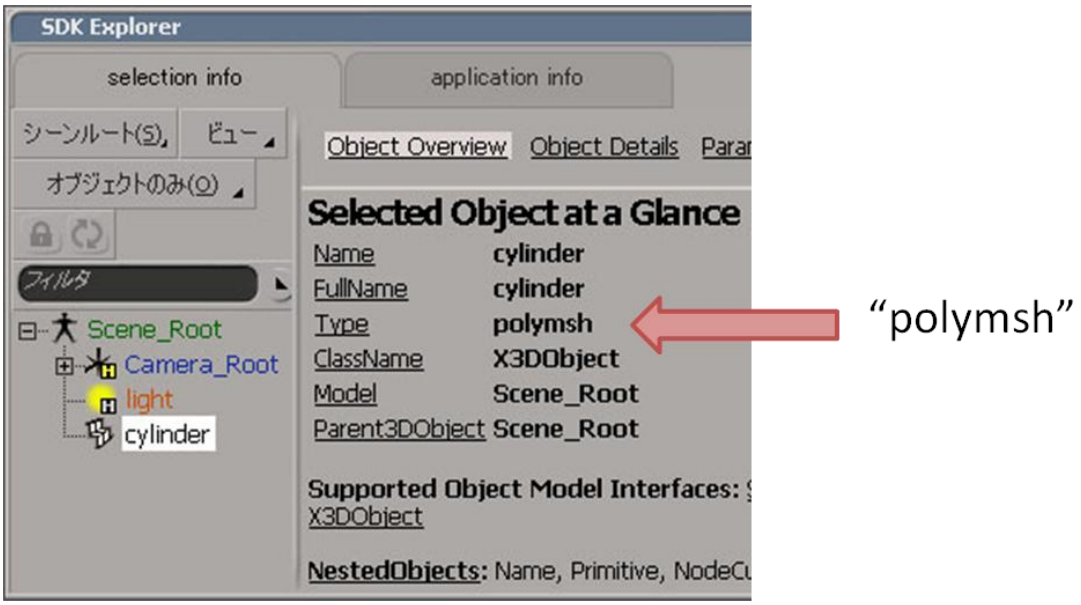
これでユニークな名前が手に入りました。COLLADA のノードの名前にはこちらを使うことにします。

5.4　ノードの種類

ノードにはライト、カメラ、ポリゴンメッシュなどの様々な種類があります。ノードの種類は SIObject::GetType()か GetClassID()で調べます。今回の実装ではデバッグしやすさを優先して GetType()を使っています。

```
// カメラ？
CString type = siObject.GetType();
if( type == L"camera" || type == L"CameraRoot" || type == L"CameraInterest" ) {
        /* カメラを出力 */
}
```

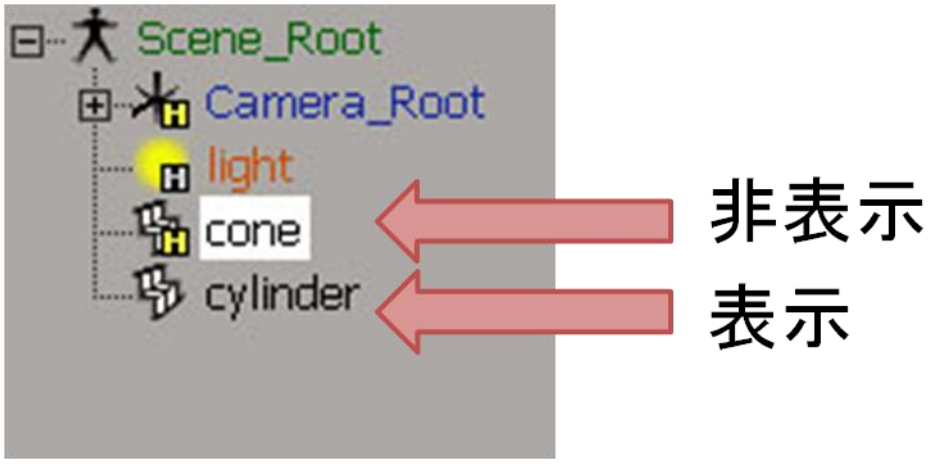
ここで取得するノードの種類は SoftimageSDK のオブジェクトモデルとは別の概念です。混同しないようにしてください。GetType()で取得できる値は SDK Explorer から調べることができます。



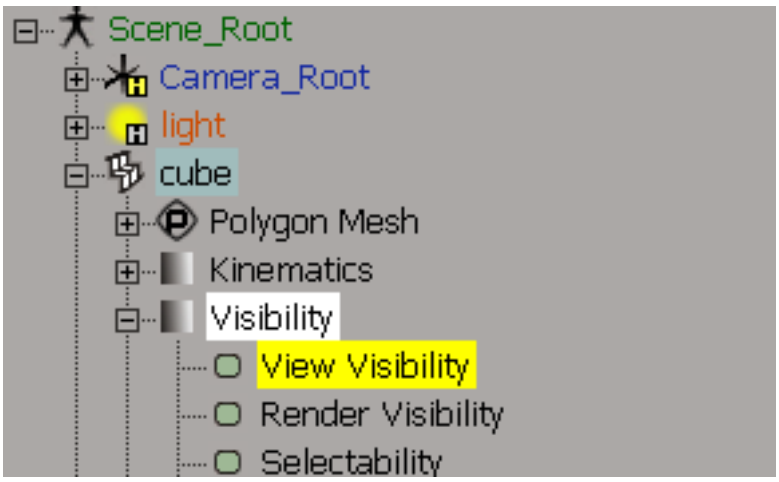
5.5　表示/非表示

今度はノードの表示/非表示の状態を取得してみましょう。デザイナーがデータを作る場合、エクスポートするデータ以外にも作業用のデータを沢山作成します。作業用のデータを残したままエクスポートするためには何らかの方法でエクスポートするデータとしないデータを区別する必要があります。今回は画面に表示されているものはエクスポートして、非表示のものはエクスポートしないという仕様にしました。この仕様にしておけば出力結果が見た目通りになるのでわかりやすくミスも減ります。

Softimage 上で表示/非表示の切り替えを行うには「ハイド」か「レイヤー」を使います。まずハイドについて説明します。Softimage 上でノードを選んで H キーを押すと、そのノードが非表示になります。Explorer 上で H マークが付いているノードは非表示になっています。H キーをもう一度押すと表示状態に戻ります。



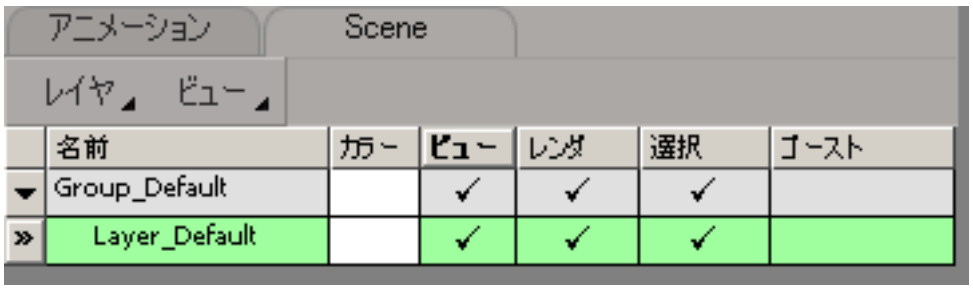
このハイドの状態をノードから取得してみましょう。ハイドの状態は"visibility"プロパティの"viewvis"パラメータに格納されています。プロパティはパラメータを複数格納するための入れ物と思っておけばよいでしょう。Softimage の GUI 上では"View Visibility"となっていますが、SDK からアクセスする場合は"viewvis"という名前になります。このように表示されている名前と SDK で使用する名前が違うことがあるので注意してください。これらの名前の対応関係は SDK Explorer の Parameter の項目で調べることができます。SDK で使用するのは ScriptName と呼ばれる名前です。



あるオブジェクトの“visibility”プロパティは SceneItem::GetProperties()関数から取得できます。プロパティが取得できたら Parameter::GetParameterValue()関数で”viewvis”パラメータの値を取得します。

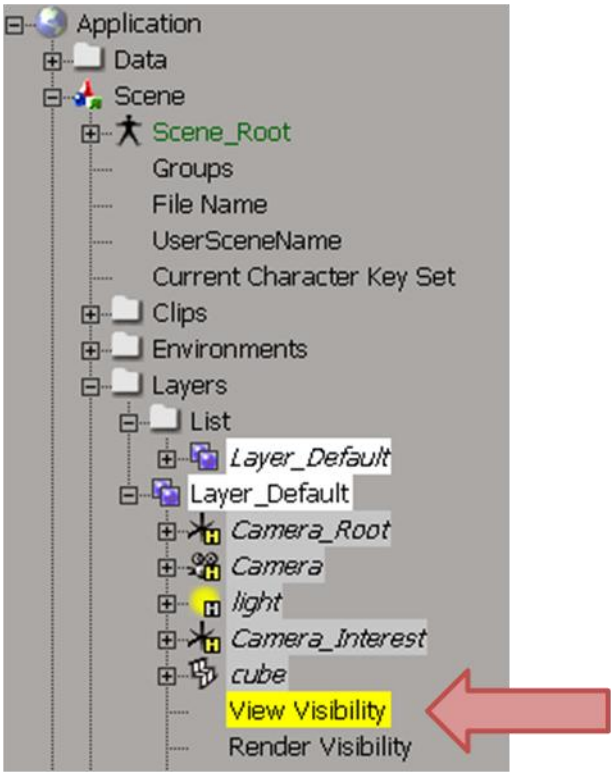
```
CRefArray props = sceneItem.GetProperties();
Property prop  = props.GetItem( L”visibility” );
bool v = prop.GetParameterValue( L”viewvis” );
```

次にレイヤーの表示状態を取得しますが、その前に Softimage のレイヤーについて説明します。



Softimage のレイヤーは Photoshop などのレイヤーと同じようなもので、レイヤー単位で表示/非表示を切り替えられます。各ノードは任意のレイヤーに登録することができ、レイヤーの表示/非表示の切り替えによって、登録しているノードの表示状態を一括で変更できます。通常、デザイナーはエクスポート用のレイヤーを作っておいて、出力するノードをそのレイヤーに登録しておきます。エクスポートするときにはエクスポート用のレイヤーだけ表示して、エクスポートを実行します。

それではレイヤーの表示状態を取得してみましょう。レイヤーの表示状態はレイヤーの”View Visibility”パラメータから取得できます。レイヤーはアプリケーションの下に入っています。



まずノードが属しているレイヤーを取得します。ノードが属しているレイヤーは ProjectItem::GetOwners()から取得できます。GetOwners()の結果から ClassID が siLayerID のものを選びます。レイヤーは Layer クラスなのでキャストして、それから "viewvis"パラメータの値を取得します。

```
CRefArray owners = projectItem.GetOwners();
for( int i=0; i<owners.GetCount(); i++ ) {
        if( owners[ i ].IsA( siLayerID ) ) {
                Layer layer( owners[ i ] );
                bool v = layer.GetParameterValue( "viewvis" );
                ...
        }
}
```

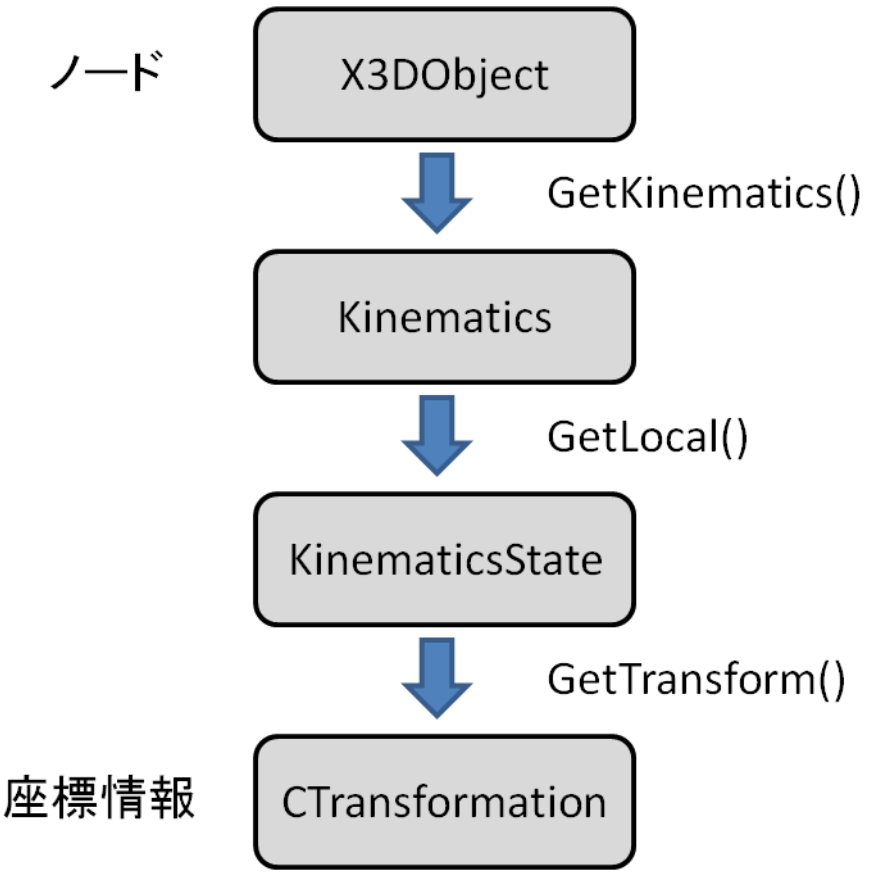
5.6　親子情報

次は親の情報を取得してみましょう。SIObject::GetParent()という関数でノードの親の CRef クラスを取得することができます。オブジェクトが親を持たない場合、GetParent() 関数は自分自身を返します。シーンの一番上のノードは親がいないので自分自身が返ってきます。

```
SIObject siObjectParent( siObject.GetParent() );
if( siObjectParent == siObject ) {
        app.LogMessage( L"No Parent." );
}
else {
        app.LogMessage( siObjectParent.GetName() );
}
```

5.7　座標情報

今度はノードの座標情報を取得します。SIObject は名前しか持っていないので、SIObject の派生先の X3DObject クラスを使います。X3DObject はシーン中の 3D オブジェクトを表現するクラスで、3D 座標の情報を持っています。この X3DObject クラスから座標情報の格納された CTransformation クラスを取得します。



X3DObject::GetKinematics()関数で Kinematics クラスが取得できます。この Kinematics クラスから位置情報へアクセスできるのですが、Kinematics クラスは GetGlobal()、GetLocal()という二つの関数を持っています。名前の通り、それぞれがグローバル（ワールド）座標、ローカル座標の位置情報を返します。今回はローカルの座標情報を取得したい

ので Kinematics::GetLocal()関数を使います。

Kinematics::GetLocal()関数は KinematicState クラスを返します。KinematicState クラスは X3DObject クラスの現在のポーズ情報を格納しています。KinematicState::GetTransform()関数から MATH::CTransformation クラスが取得できます。これでようやく座標情報の格納された CTransformation クラスが取得できました。

ここまでは一つずつ説明してきましたが、実際には X3DObject から一気に CTransformation クラスを取得できます。

```
X3DObject x3DObject( selection[ i ] );
MATH::CTransformation  transform;
transform  = x3DObject.GetKinematics().GetLocal().GetTransform();
```

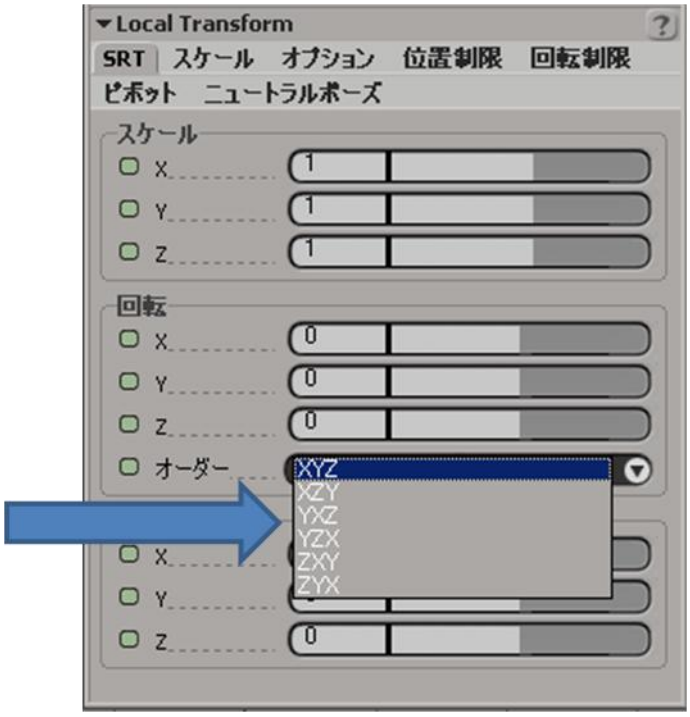
CTransformation クラスが取得できたので、その中の座標情報を取得します。各関数の詳細は CTransformation クラスのマニュアルを参照してください。

```
MATH::CVector3   scale = transform.GetScaling();
MATH::CVector3   rotation = transformation.GetRotationXYZAngles();
MATH::CVector3   translation = transformation.GetTranslation();
MATH::CQuaternion quaternion = transformation.GetRotationQuaternion();
```

rotation の値はラジアンです。度に変換したい場合は MATH::RadiansToDegrees()関数を使います。

```
const float rx = static_cast< float >( MATH::RadiansToDegrees( rotation.GetX() ) );
const float ry = static_cast< float >( MATH::RadiansToDegrees( rotation.GetY() ) );
const float rz = static_cast< float >( MATH::RadiansToDegrees( rotation.GetZ() ) );
```

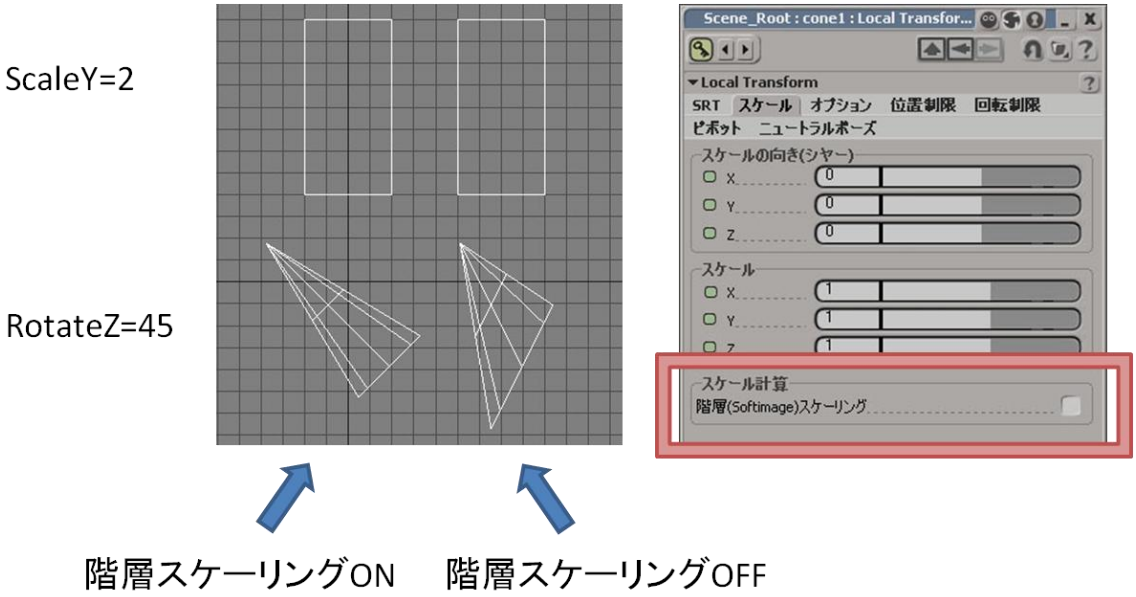
Softimage などの DCC ツールでは XYZ 回転の適用順をノード単位で変更できるので、注意する必要があります。



これを描画エンジンで対応するのは結構大変です。事前にデザイナーと相談して対応するかどうかを決めておきましょう。今回はデフォルト設定の XYZ 順で固定しています。なお、Softimage 上での回転の適応順は CTransformation::GetRotationOrder()関数で取得できます。

5.8　階層(Softimage)スケーリング

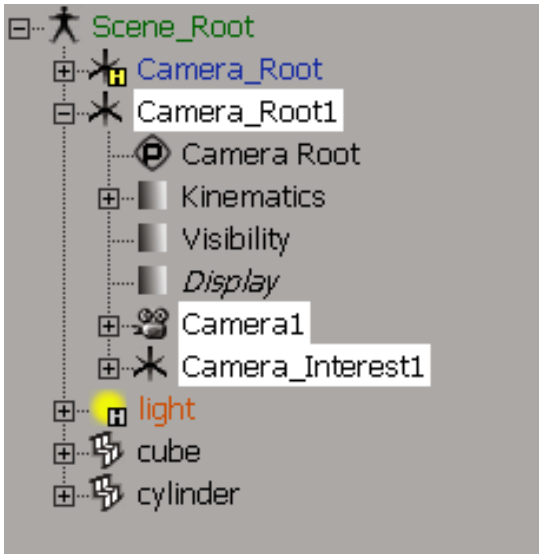
Softimage には階層(Softimage)スケーリングという特殊なモードがあります。このモードを使うとマトリックスの計算が変わってくるのでここで説明します。通常、マトリックスの変換は親から子へ演算の結果が伝わっていきますが、この階層(Softimage)スケーリングが有効な場合はスケール成分だけ別扱いで計算します。



この図ではコーンをキューブの子供にして、キューブに ScaleY=2, コーンに RotateZ=45 を適用しています。通常のマトリックス演算では子供を回転してから親のスケールを適用するので、子供のコーンはゆがんでいます（右側の図形）。階層(Softimage)スケーリングが有効な場合はスケールを別に計算するので、子供のコーンがゆがんでいません。これはこれで便利そうな機能ですが、この機能を実現する場合は描画エンジン側の対応も必要になります。対応するかどうかはデザイナーと相談して決めてください。Maya や MAX など他の DCC ツールには無い機能なので、複数のツールが混在する場合は使わないほうがよいでしょう。

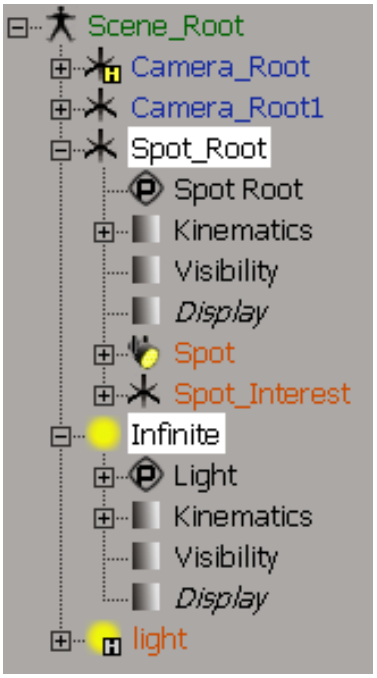
5.9　カメラの取得

Softimage のカメラはカメラルート、カメラの注視点、カメラ自体から構成されます。SDK Explorer で調べると、対応するクラスはそれぞれ CameraRig,Camera,Null、タイプは”CameraRoot”,”Camera”,”CameraInterest”になっていることがわかります。カメラルートとカメラの注視点は Null の一種と考えればよいので、通常のノードと同じ扱いにします。カメラの各種パラメータはカメラ自体から取得できます。Camera クラスは ProjectItem の派生クラスなので、ProjectItem::GetParameter()関数を使います。



5.10　ライトの取得

今度はライトの情報を取得します。ライトの構造はライトの種類によって多少変わってきます。並行光源の場合はライト本体のみですが、スポットライトの場合はカメラのようにライトのルートと注視点のノードから構成されています。カメラと同じように SDK Explorer で調べてみると、ライト本体のクラスは Light クラス、ルートは LightRig クラス、注視点は Null クラスになっています。タイプはそれぞれ”light”,”SpotRoot”,”SpotInterest”です。ルートと注視点は Null と同じ扱いにして、ライト本体からライトのパラメータを取得します。



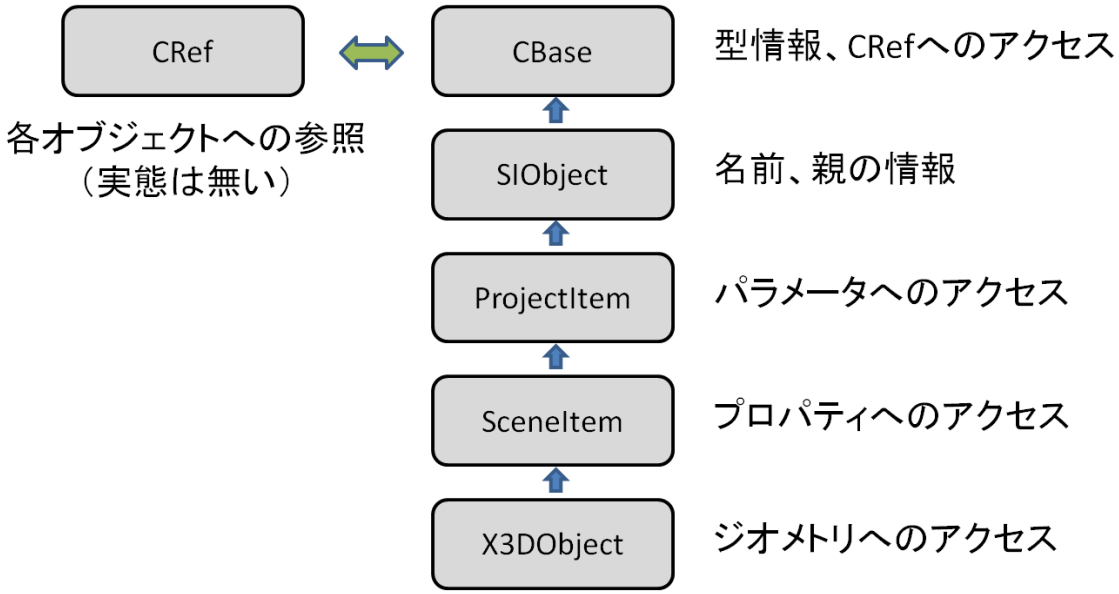
ライト本体は Light クラスです。ライトの種類は Light クラスの”Type”パラメータから取得できます。ここで取得するのは”Type”という名前のパラメータで、SIObject::GetType() 関数で取得するタイプとは別物です。代表的なライトの”Type”パラメータの値を以下にまとめました。他のライトの値は SDK Explorer で確認してください。

- ポイントライト：0
- 並行光源：1
- スポットライト：2

Softimage のライトには多数のパラメータがありますが、今回作成したエクスポーターではその中の”soft_light”のカラーを取得しています。他のパラメータを取得したい場合は SDK Explorer で詳細を確認してください。

5.11 オブジェクトモデルの基本

ここまでで CRef,CBase,SIObject,X3DObject と沢山のクラスが出てきました。SoftimageSDK のクラス階層は深くてわかりにくいので、代表的なクラスとその構造をまとめてみました。

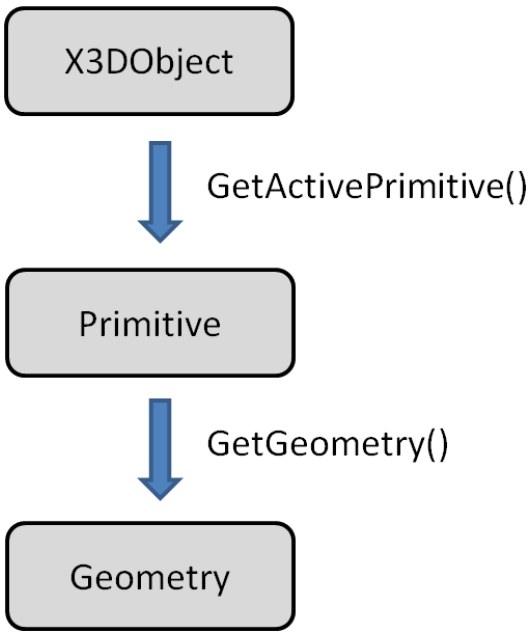


Softimage のオブジェクトモデルの一番基本となるのが CBase から X3DObject までの一連のクラスで、各クラスが段階的にそれぞれの役割を持っています。CRef クラスは各オブジェクトへの参照で、実体はありません。

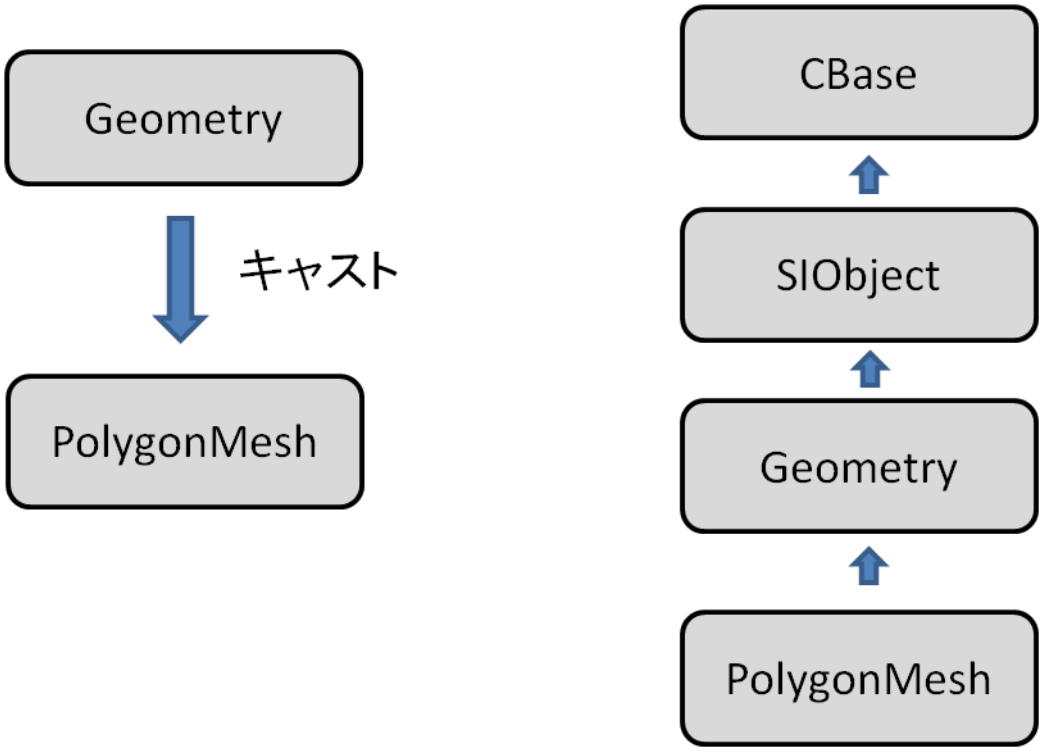
6　ジオメトリ

6.1　Geometry, PolygonMesh クラス

ノードの情報が取得できたので次はポリゴンメッシュの情報を取得してみましょう。まず X3DObject::GetActivePrimitive()関数を使って Primitive クラスを取得します。Primitive クラスは X3DObject クラスの形状情報を表現するクラスです。今回 Primitive クラスで使用する関数は Primitive::GetGeometry()関数だけです。Primitive::GetGeometry()関数を使って Geometry クラスを取得します。Geometry クラスは X3DObject の内部のジオメトリ情報にアクセスするためのクラスです。



ノードがポリゴンメッシュの場合は、Geometry クラスは PolygonMesh クラスに変換できます。最初のところでポリゴンメッシュであることを確認しているので、今回は無条件に PolygonMesh クラスに変換します。



PolygonMesh クラスから頂点情報を取得することもできますが、SoftimageSDK にはエクスポーター用の CGeometryAccessor というクラスが用意されています。こちらを使ったほうが簡単なので、以降の説明ではこのクラスを使用します。CGeometryAccessor は PolygonMesh クラスの GetGeometryAccessor()関数で取得できます。

GetGeometryAccessor()関数の引数には siConstructionMode を渡します。この引数によって取得するジオメトリの種類を切り替えることができます。ここでは siConstructionModeModeling を渡して、シェイプやスキニングによる変形の影響を受けないオリジナルのジオメトリ情報を取得しています。

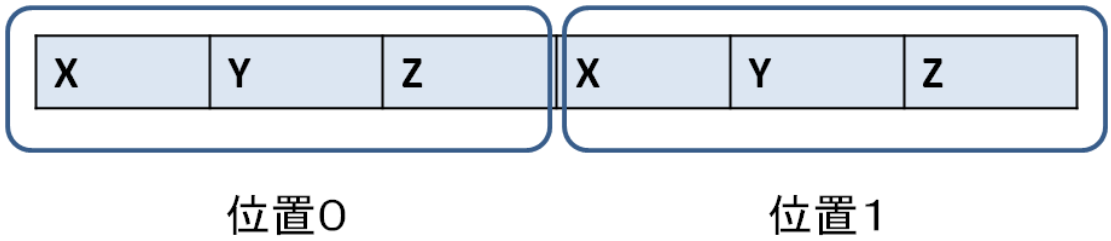
```
// X3DObject から Primitive を取得
Primitive primitive( x3DObject.GetActivePrimitive() );

// Primitive から Geometry を取得
Geometry geometry( primitive.GetGeometry() );

// Geometry から PolygonMesh を取得
PolygonMesh polygonMesh( geometry );

// PolygonMesh から CGeometryAccessor を取得
CGeometryAccessor ga = polygonMesh.GetGeometryAccessor(siConstructionMode);
```

以上をまとめて書くとこうなります。

```
// X3DObject から CGeometryAccessor を取得（引数は省略）
CGeometryAccessor ga;
ga = PolygonMesh( x3DObject.GetActivePrimitive().GetGeometry() ).GetGeometryAccessor()
```

6.2　位置情報

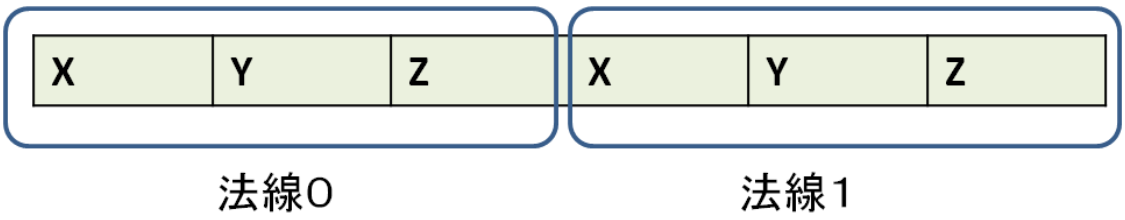
それでは CGeometryAccessor クラスを使って情報を取得していきましょう。まずは位置情報を取得します。位置情報は CGeometryAccessor::GetVertexPositions()関数で取得できます。その結果は CDoubleArray の中に double の配列として格納されています。位置情報は３つの float 値で構成されているので、float の数は位置情報の数 x ３になります。




```
// 位置の配列を取得
CDoubleArray positions;
ga.GetVertexPositions( positions );
```

法線情報

次に法線情報を取得します。法線情報は CGeometryAccessor::GetNodeNormals()関数で取得できます。こちらも float の配列として格納されています。法線情報も３つの float 値で構成されており、float の数は法線情報の数 x ３となります。



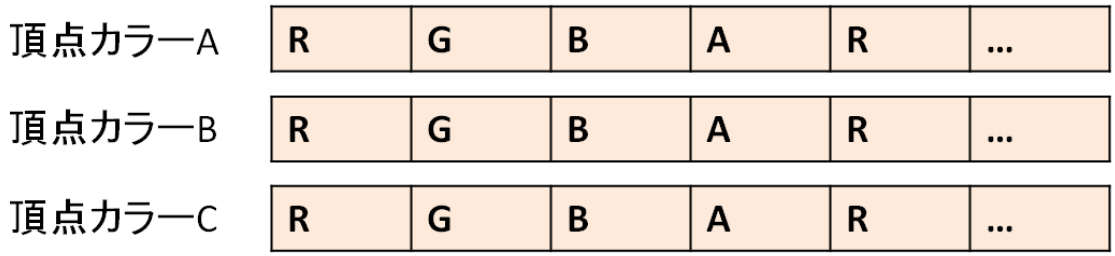

```
// 法線の配列を取得
CFloatArray normals;
ga.GetNodeNormals( normals );
```

位置情報の数と法線情報の数は違うことがあります。この点は注意してください。詳しい説明はインデックスの取得のところで行います。

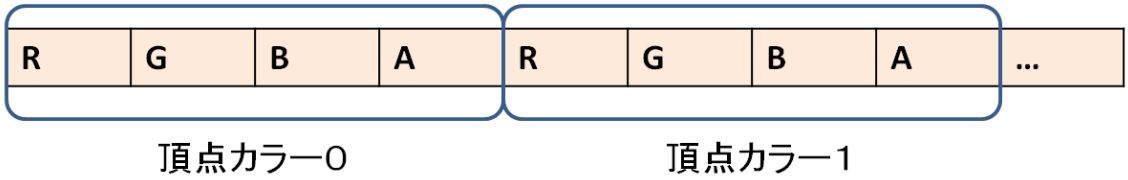
6.3　頂点カラー情報

今度は頂点カラーを取得します。DCC ツールによっては頂点カラーをデフォルトで持っているものもありますが、Softimage のポリゴンメッシュは初期状態では頂点カラーを持っていません。ユーザーが追加した段階で頂点カラーが使えるようになります。またポリゴンメッシュは頂点カラーを複数持つことができます。頂点カラーを複数持っている場合は、全部の頂点カラーを出力するのか、現在有効な頂点カラーだけ出力するのか選ぶ必要があります。今回の実装では全部の頂点カラーを出力する仕様にしました。

頂点カラーは CGeometryAccessor::GetVertexColors()関数で取得します。頂点カラーが複数ある場合は、その数だけ取得できるので注意してください。頂点カラー自体の名前は ClusterProperty から GetName()関数で取得できます。



頂点カラーの ClusterProperty から値を取得するときは ClusterProperty::GetValues()関数を使います。結果の CFloatArray にはカラーの値が float のフラットな配列で入っています。カラーの値は 4 つの float 値で構成されるので、float の個数はカラーの個数 x 4 となります。順番は R,G,B,A です。

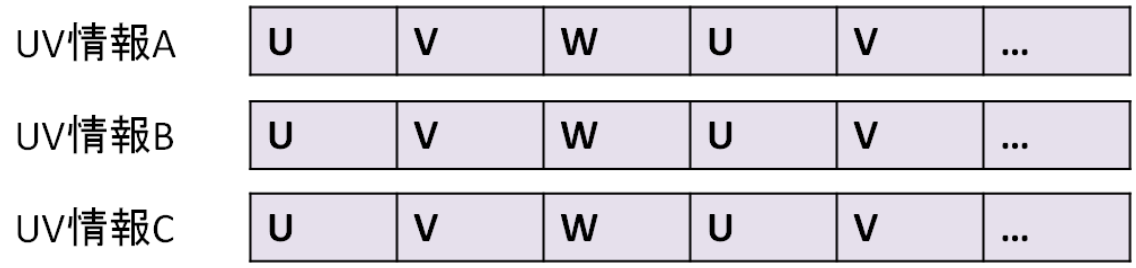


頂点カラーには 1 以上の値を設定することができます。今回の実装では取得した頂点カラーの値の範囲チェックを行って 0 ～ 1 に収るようにしています。今回はすべての頂点カラーを出力していますが、現在有効な頂点カラーだけ出力したい場合は PolygonMesh::GetCurrentVertexColor()関数を使ってください。

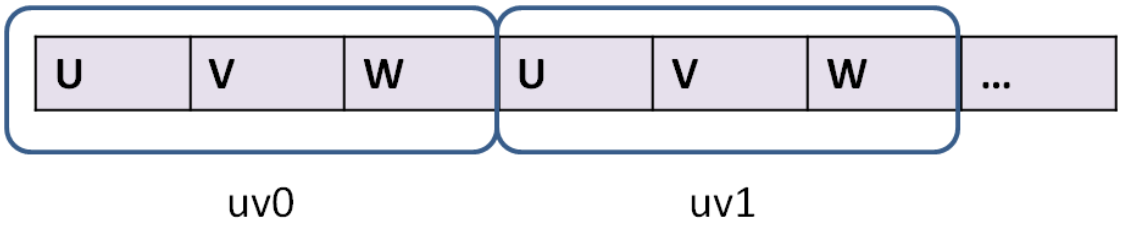
```
// 頂点カラーセットの配列を取得
CRefArray vertexColors = ga.GetVertexColors();

// 頂点カラー（複数）
const int vertexColorsNumber = vertexColors.GetCount();
for( int vc=0; vc<vertexColorsNumber; vc++ ) {
        // 頂点カラープロパティ
        ClusterProperty vertexColorProperty = vertexColors[ vc ];

        // 頂点カラーの値の配列
        CFloatArray values;
        vertexColorProperty.GetValues( values );
        ...
}
```

6.4　ＵＶ情報

今度は UV 情報を取得します。UV 情報も頂点カラーと同じく初期状態では存在しません。またポリゴンメッシュは UV 情報を複数持つことができます。UV 情報は CGeometryAccessor::GetUVs()で取得します。UV の名前は GetName()関数で取得することができます。



取得方法は頂点カラーと同じです。UV 情報は３つの float 値で構成されているので、float の個数は UV の数ｘ３となります。ただし今回の実装では w 要素を使わないので無視しています。

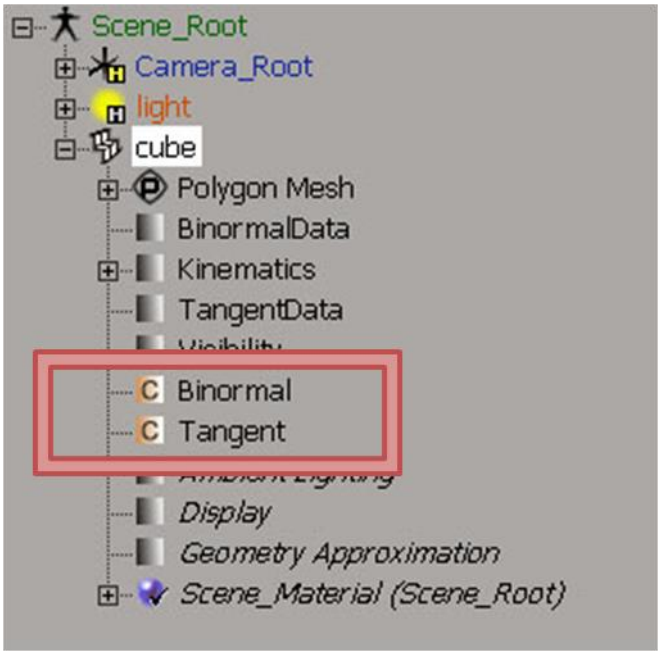


```
// UV セットの配列を取得
CRefArray uvs = ga.GetUVs();

const int uvsNumber = uvs.GetCount();
for( int uvsIndex = 0; uvsIndex<uvsNumber; uvsIndex++ ) {
        // UV
        ClusterProperty uvProperty = uvs[ uvsIndex ];

        // UV の値の配列
        CFloatArray values;
        uvProperty.GetValues( values );
        ...
}
```

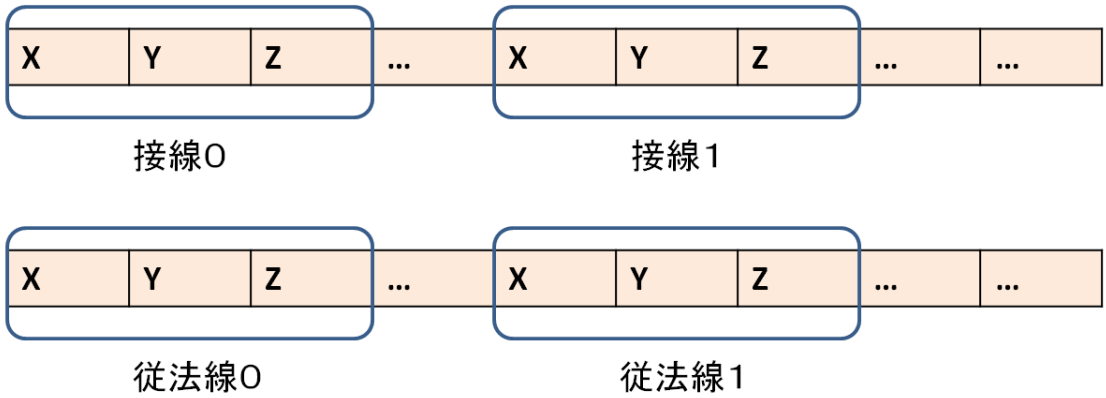
6.5　接線、従法線情報

今度は接線、従法線の情報を取得します。Softimage のポリゴンメッシュはデフォルト状態では接線・従法線の情報を持っていません。ユーザーがプロパティ＞タンジェント（もしくはプロパティ＞従法線）を実行すると接線（従法線）情報の格納された ClusterProperty が追加されます。接線、従法線の情報の取得方法は Softimage のバージョンによって多少変わってきます。今回は Softimage2010 用の取得方法を説明しますので、必要な場合は修正してください。

それでは接線・従法線情報を取得してみましょう。以下の説明では PolygonMesh が接線・従法線の ClusterProperty を持っていると仮定しています。



Softimage 上では接線・従法線情報は頂点カラーの一種として格納されています。どの頂点カラーが接線・従法線情報なのか調べるためには RenderTree 側の情報が必要ですが、かなり面倒なので今回は"Tangents"、"Binormals"という名前の頂点カラーを接線・従法線情報として取得することにしました。



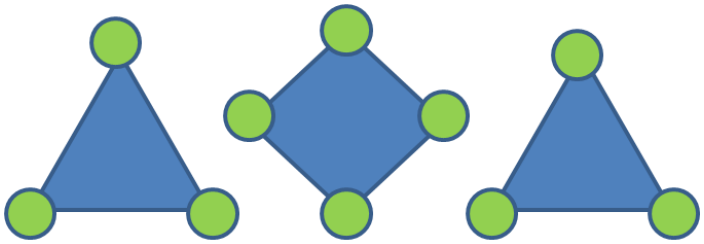
接線・従法線は頂点カラーとして格納されているので４つの値から構成されています。接線と従法線はベクトルなので最初の XYZ の値を使用します。Softimage2010 では接線・従法線情報を float と short の２つの型で持つことができます。どちらで格納されているか調べるときはTangentsプロパティの"vertexcolorchangedatatype.desireddatatype"というパラメータの値を調べます。short のときは-1 から+1 が 0 から 1 にマッピングされているので変換する必要があります。

```
// "Tangents"という名前のプロパティを取得している。
ClusterProperty tangentProperty = ga.GetVertexColors().GetItem( L"Tangents" );
CFloatArray values;
tangentProperty.GetValues( values );

// float か short か
bool floatDataType = false;
{
        CRef ref;
        CString name = tangent.GetFullName();
        name += L".vertexcolorchangedatatype.desireddatatype" );
        ref.Set( name );
        if( ref.IsValid() ) {
                Parameter paramDesireDataType( ref );
                if( paramDesireDataType.IsValid() ) {
                        const int DESIRE_DATA_TYPE_SHORT = 0;
                        const int DESIRE_DATA_TYPE_FLOAT = 1;
                        const int value = paramDesireDataType.GetValue();
                        if( value == DESIRE_DATA_TYPE_SHORT ) {
                                floatDataType = false;
                        }
                        else if( value == DESIRE_DATA_TYPE_FLOAT ) {
                                floatDataType = true;
                        }
                }
        }
}
...
int     count = 0;
if( floatDataType ) {
        for( int c=0; c<num; c++ ) {
                Collada::Float3 tan;
                tan.x = static_cast< float >( array[ count++ ] );
                tan.y = static_cast< float >( array[ count++ ] );
                tan.z = static_cast< float >( array[ count++ ] );
                count++;
                …
        }
}
else {
        for( int c=0; c<num; c++ ) {
                float x = static_cast< float >( array[ count++ ] );
                float y = static_cast< float >( array[ count++ ] );
                float z = static_cast< float >( array[ count++ ] );
                count++;
                x       = x*2.0f - 1.0f;
                y       = y*2.0f - 1.0f;
                z       = z*2.0f - 1.0f;
                …
        }
}
```

6.6　インデックス情報

次にポリゴンのインデックスの情報を取得しましょう。CGeometryAccessor クラスからポリゴンと三角形、両方のインデックスを取得することができますが、今回はポリゴンのインデックスのみ取得します。

まずポリゴン単位の頂点数を取得します。CGeometryAccessor クラスの GetPolygonVerticesCount()関数を使うとポリゴンごとの頂点数が取得できます。各ポリゴンが以下のような構造になっている場合、GetPolygonVerticesCount()関数の結果はこのようになります。

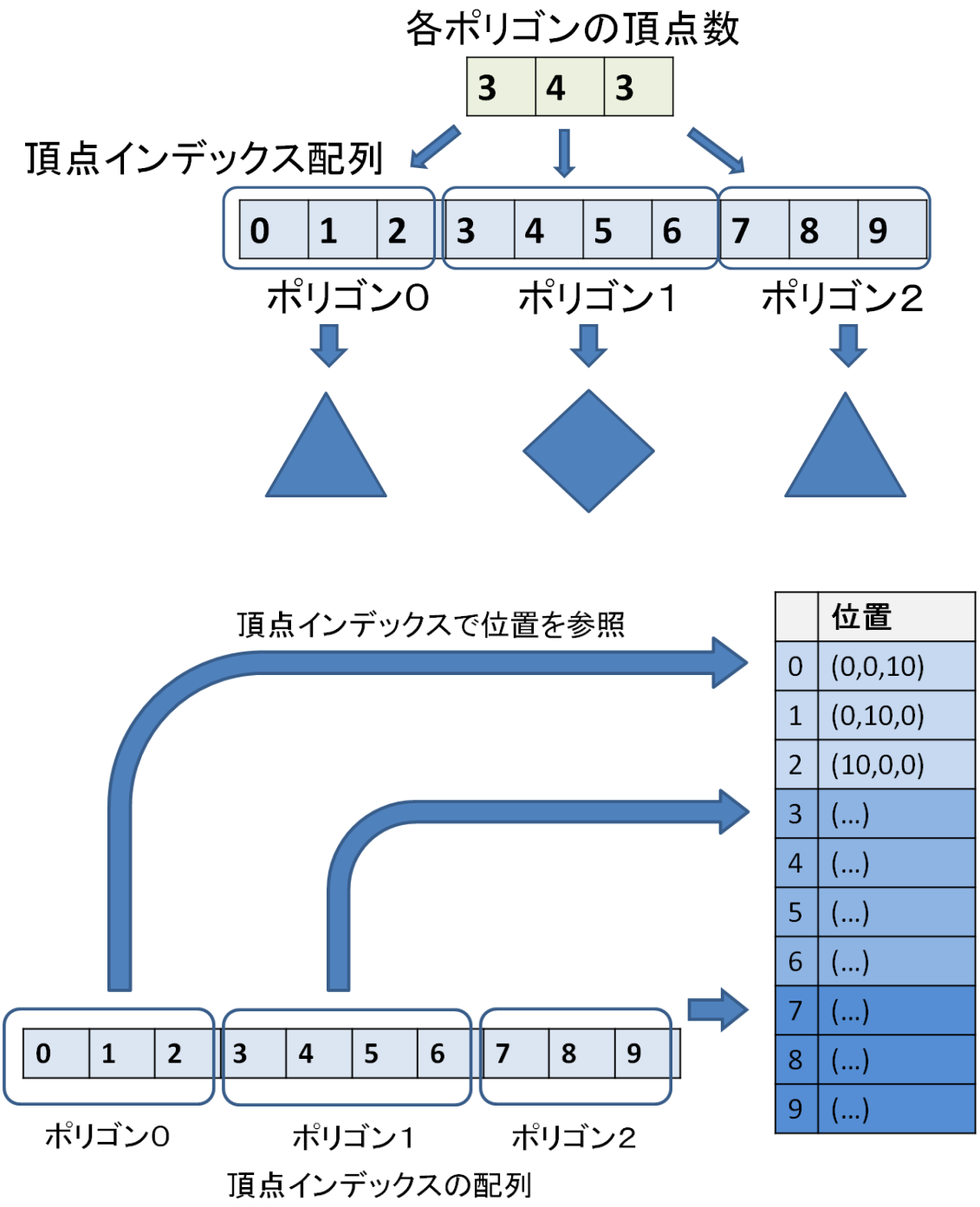


各ポリゴンの頂点数

3	4	3

次に CGeometryAccessor::GetVertexIndices()関数で各ポリゴンの頂点インデックス配列を取得します。この頂点インデックス配列は先ほどの「各ポリゴンの頂点数」と一緒に使います。頂点インデックスは位置情報と頂点ウェイトへのインデックスとして使用します。

0	1	2	3	4	5	6	7	8	…

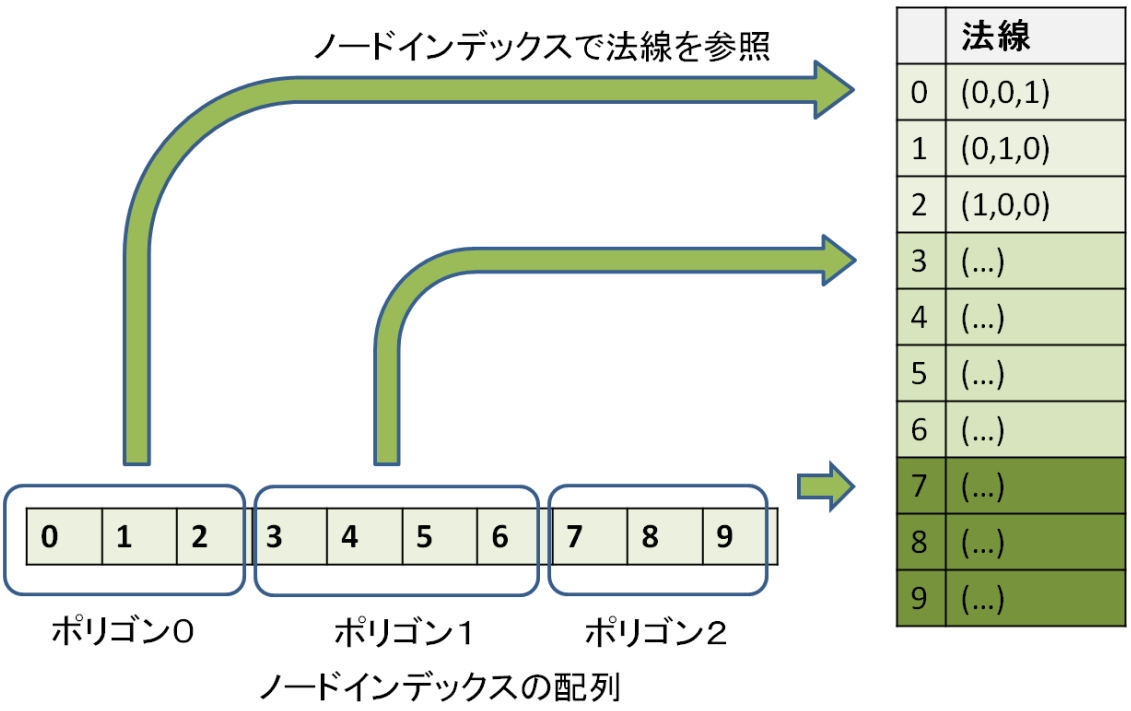



```
// ポリゴンの頂点インデックス配列を取得
CLongArray vertexIndices;
ga.GetVertexIndices( vertexIndices );
```

今度は CGeometryAccessor::GetNodeIndices()関数でノードインデックスを取得します。
ノードインデックスは法線、頂点カラー、ＵＶ、接線、従法線のインデックスとして使用

します。

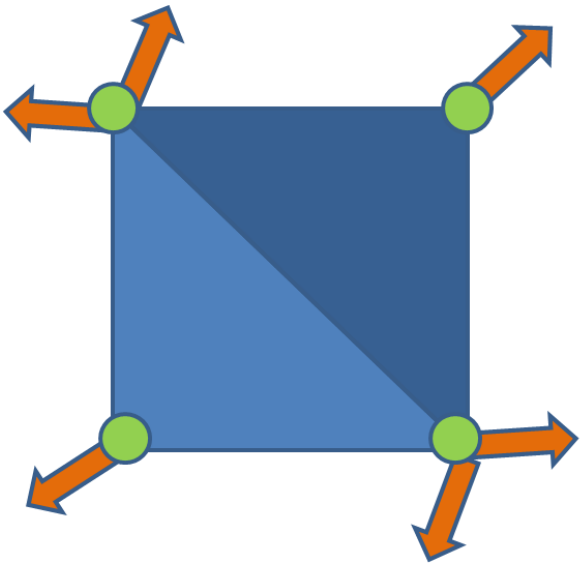
0	1	2	3	4	5	6	7	8	…



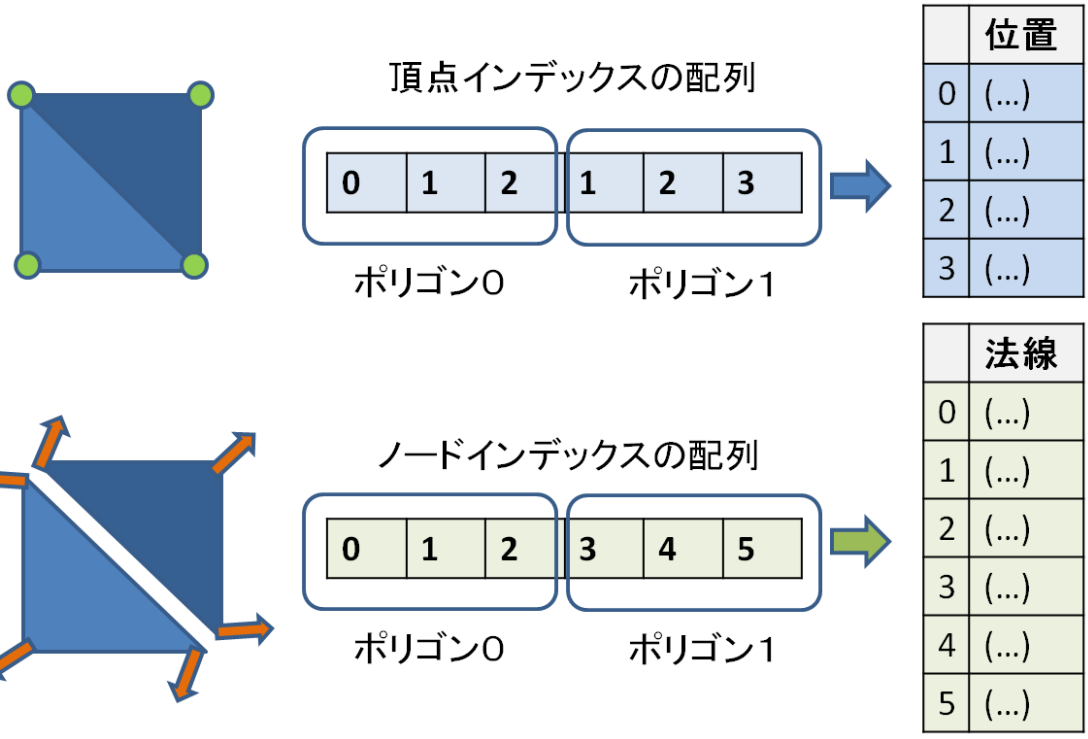
	法線
0	(0,0,1)
1	(0,1,0)
2	(1,0,0)
3	(…)
4	(…)
5	(…)
6	(…)
7	(…)
8	(…)
9	(…)

```
// ポリゴンのノードインデックス配列を取得
CLongArray nodeIndices;
ga.GetNodeIndices ( nodeIndices );
```

頂点インデックスとノードインデックスの違いに注意しましょう。



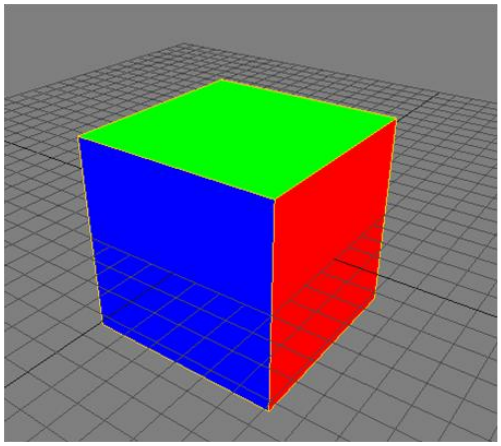
- 3頂点x2ポリゴン
- 位置x4
- 法線x6
- 頂点インデックスx6
- ノードインデックスx6



この図のように位置情報と法線情報の数は一致しません。それぞれの情報へアクセスするためには頂点インデックスとノードインデックスを使う必要があります。

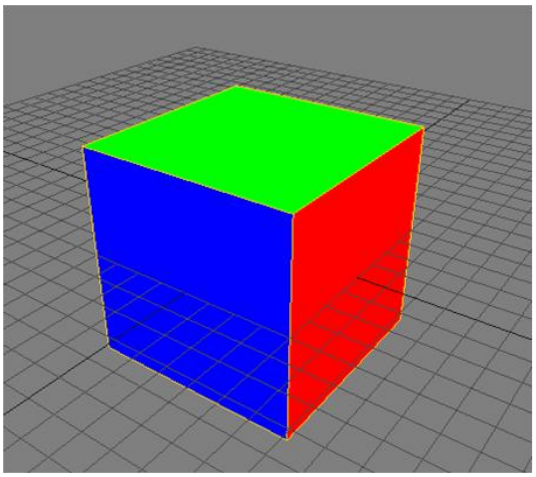
6.7　ポリゴンのマテリアル情報

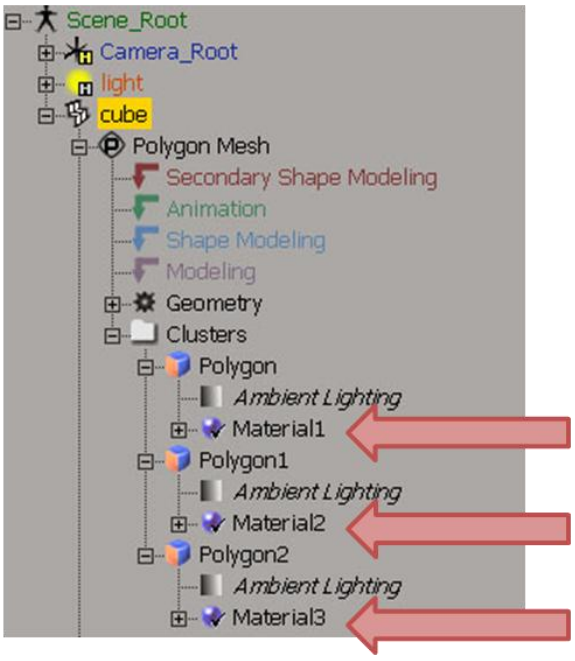
最後にポリゴンメッシュが使用しているマテリアルを取得します。



キューブ（マテリアルx3）

この画像のようにポリゴンメッシュには複数のマテリアルを割り当てることができます。Explorer からこのポリゴンメッシュが使用しているマテリアルの一覧を見ることができます。

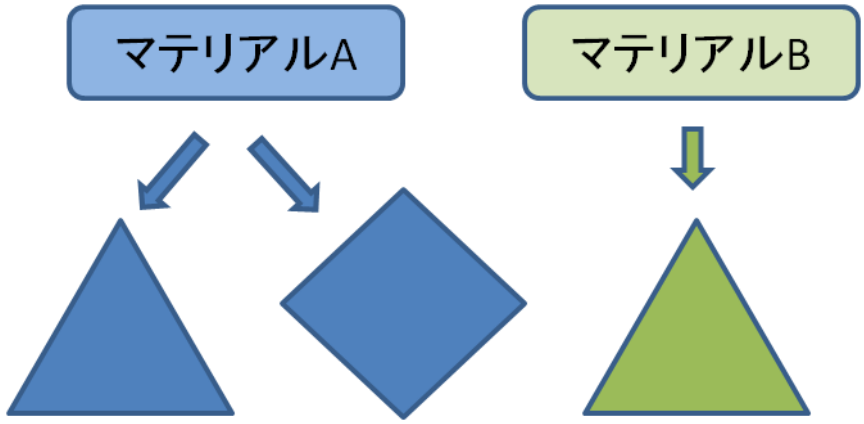




まずポリゴンメッシュが使用しているマテリアルを CGeometryAccessor::GetMaterials() 関数で取得します。ポリゴンメッシュの構成によっては重複した結果が返ってくるので注意してください。

```
CRefArray materials = ga.GetMaterials();
```

次にポリゴンとマテリアルの対応関係を調べます。CGeometryAccessor クラスの GetPolygonMaterialIndicesByMaterial()関数の引数に調べたいマテリアルを渡すと、結果が Bool の配列で返ってきます。Bool の配列の要素数はポリゴン数と等しく、引数のマテリアルを使っている場合は true、そうでない場合は false になります。



マテリアルA	true	true	false

マテリアルB	false	false	true

```
// このマテリアルを使用しているポリゴンのインデックスを取得
CBitArray polygonIndices;
ga.GetPolygonMaterialIndicesByMaterial( material, polygonIndices );
```

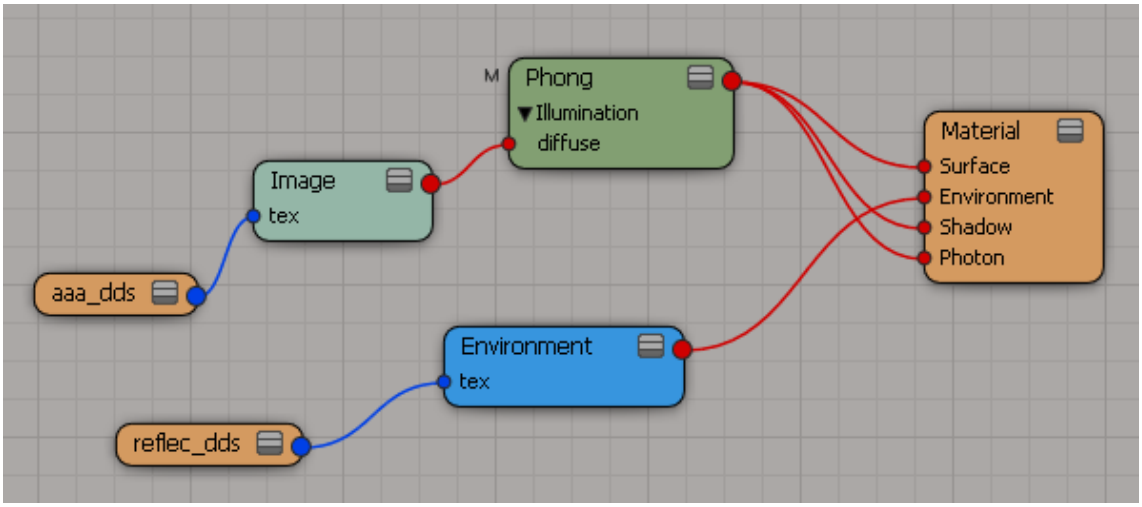
7　マテリアル

ジオメトリ情報の取得が終わったので次はマテリアル情報を取得しましょう。

7.1　質感情報の設定方法

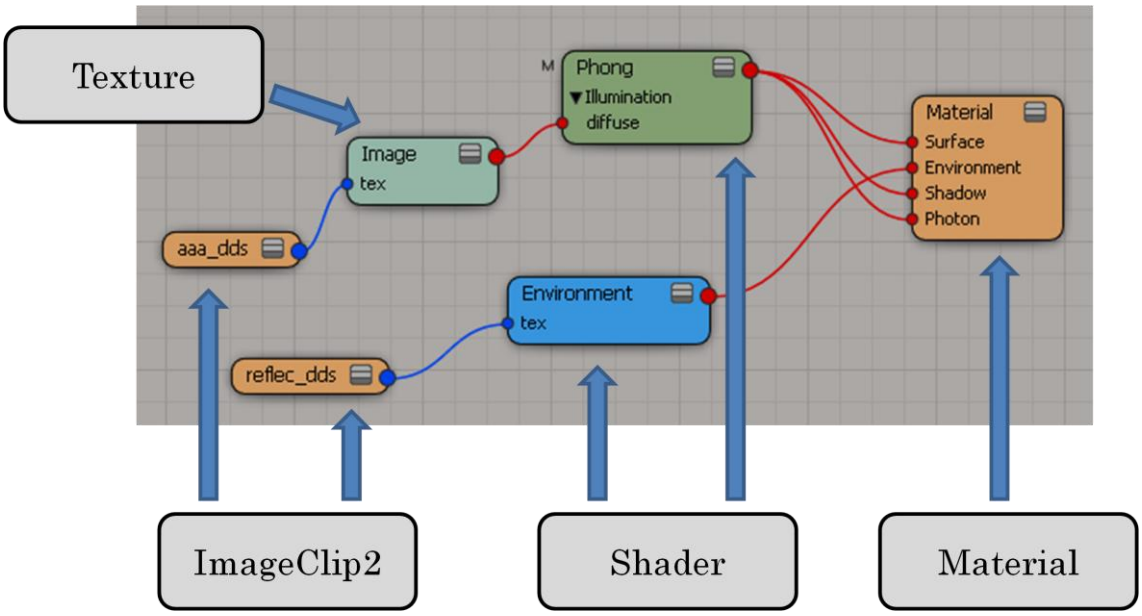
ゲーム用の質感設定を行う方法は大きく２つに分けられます。外部エンジン/ツールで設定する方法と、DCC ツール上で設定する方法です。DCC ツール上で設定する方法もいろいろあるのですが、今回はレンダーツリーを使って設定する方法を採用しています。

これからレンダーツリーを使う方法について説明します。レンダーツリーは質感設定をツリー構造で構築します。今回はこのツリー構造をたどってテクスチャと各種パラメータを取得します。



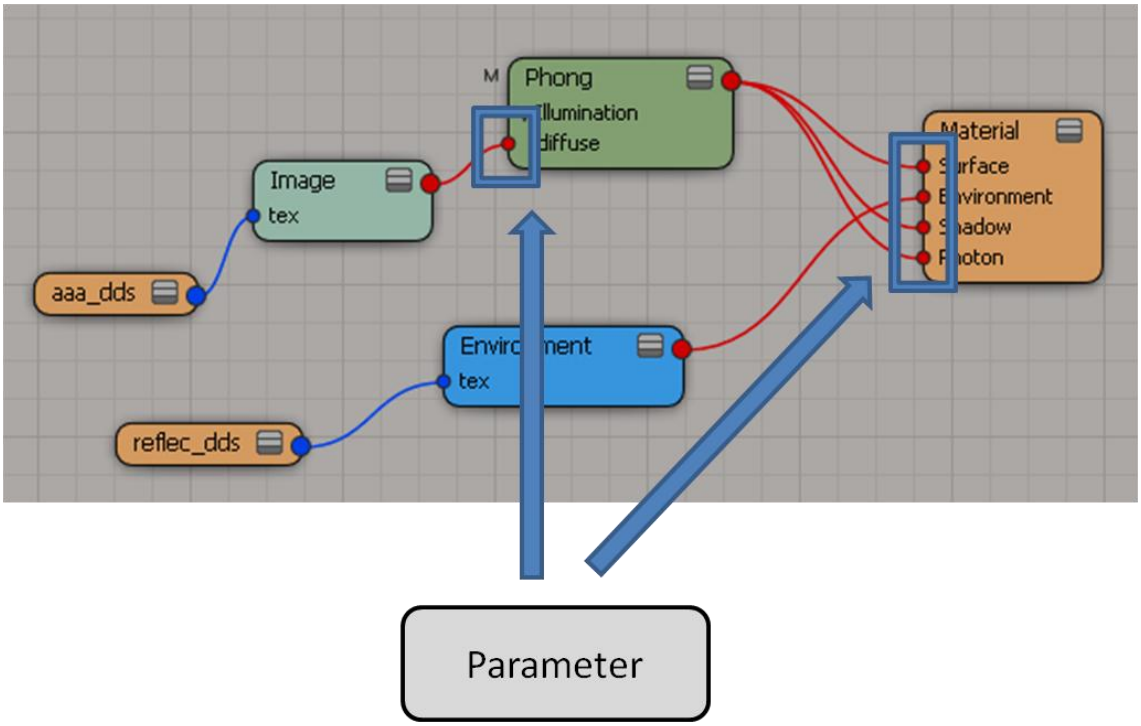
7.2　レンダーツリーからの情報取得

まずレンダーツリーと SoftimageSDK のクラスの対応を理解しましょう。

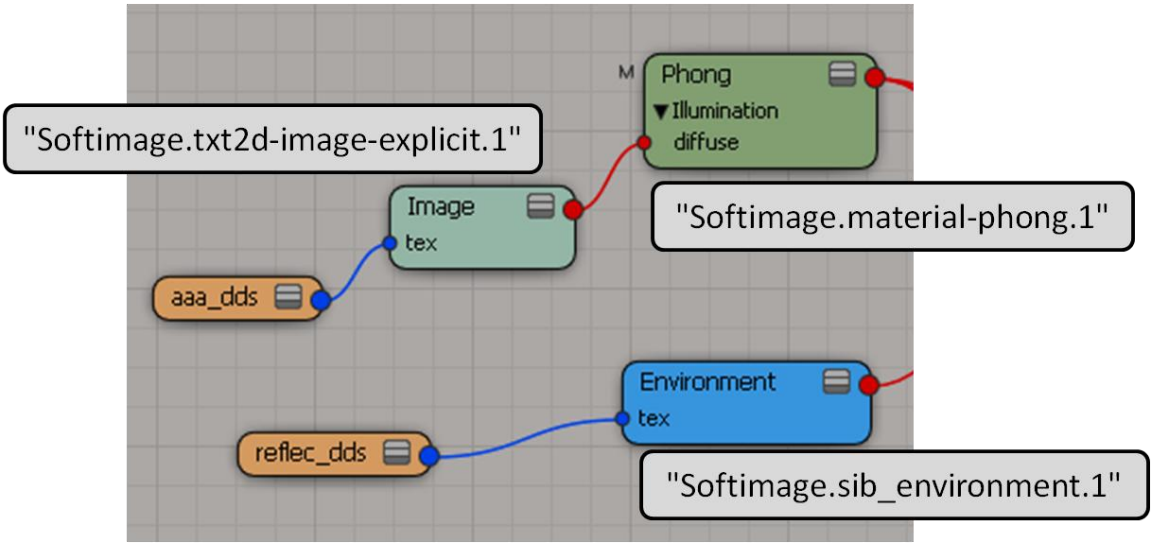


一番右側のノードが Material クラスです。ここから左側のノードをたどりながら必要な情報を取得していきます。レンダーツリーの大半のノードは Shader クラスです。一部の Shader クラスに Texture クラスが繋がっており、その Texture クラスの先に ImageClip2 クラスが繋がっています。

レンダーツリーの各ノードはパラメータを持っています。ノードの左側の丸い部分がパラメータで Parameter クラスに相当します。



レンダーツリーの各ノードの識別には ProgID を使います。Shader::GetProgID()関数を使うと ProgID が文字列で取得できます。



7.3　レンダーツリーのノードの取得

それでは実際にレンダーツリーの中身を調べていきましょう。スタート地点は一番右側のマテリアルです。ここから始めて一番左側のイメージクリップを取得します。

まず Material クラスのパラメータを取得します。Material クラスは ProjectItem クラスの派生クラスなので ProjectItem::GetParameters()関数が使えます。

```
CParameterRefArray parameters = material.GetParameters();
```

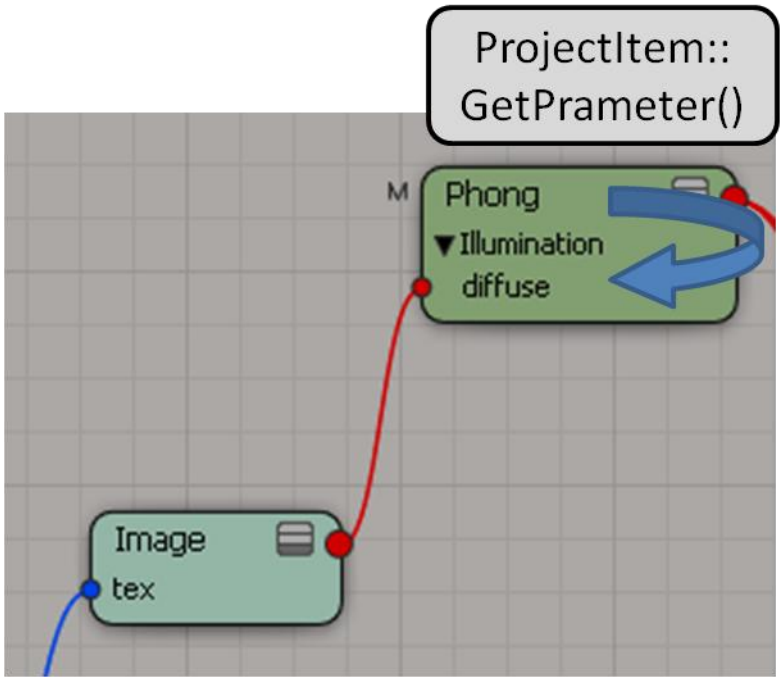
次にこのパラメータに繋がっているノードを Parameter::GetSource 関数で取得します。パラメータにノードが繋がっていない場合、GetSource()関数は空の参照を返します。

```
for( i=0; i<parameters.GetCount(); i++ ) {
        Parameter param = parameters[ i ];
        SIObject source( param.GetSource() );
        if( source.IsValid() ) {
                /* ノードが繋がっていた場合 */
        }
}
```

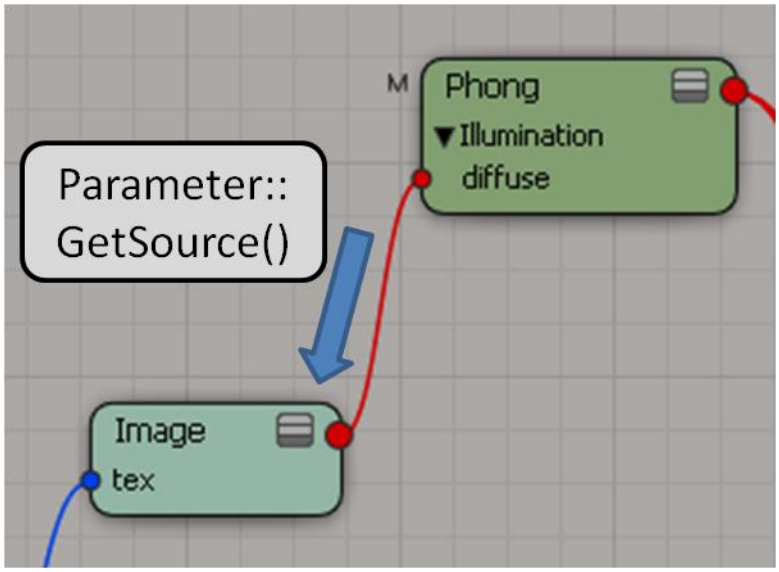
マテリアルのパラメータに繋がっているノードは Shader クラスか Texture クラスです。このコードではキャスト後に IsValid()関数で判定しています。

```
SIObject source( param.GetSource() );
if( source.IsValid() ) {
        Texture texture( source );
        if( texture.IsValid() ) {
                /* テクスチャ情報を取得 */
        }
        else {
                Shader shader( source );
                /* シェーダー情報を取得 */
        }
}
```

Shader クラスもパラメータを持っています。Shader クラスは ProjectItem クラスの派生クラスなので、ProjectItem::GetParameters()関数でパラメータを取得します。

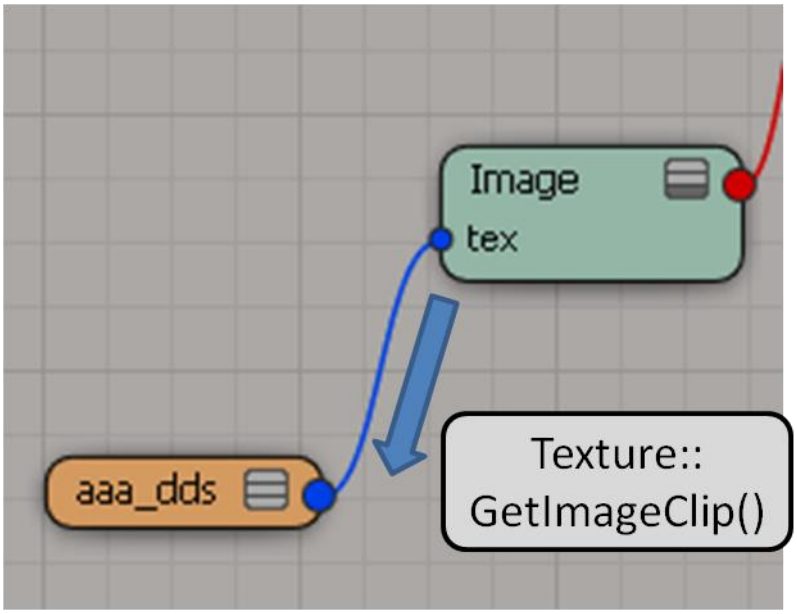


このパラメータにはさらにノードが繋がっているので、先ほどと同じように調べます。



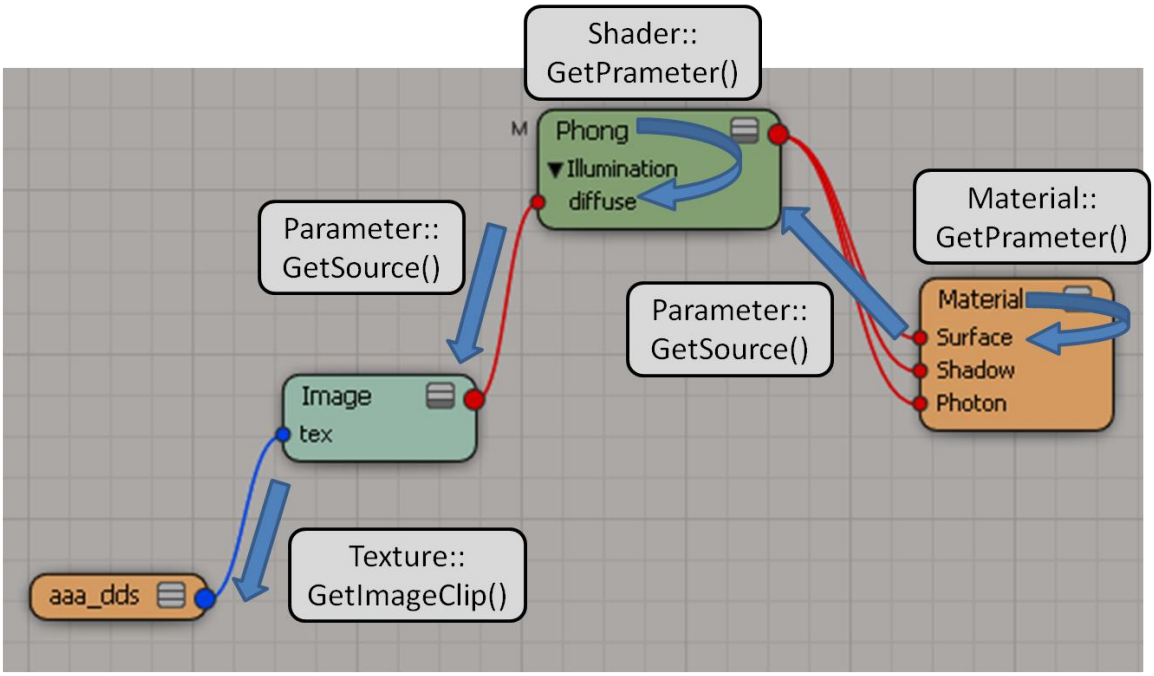
```
Shader shader( source );
CParameterRefArray parameters = shader.GetParameters();
const int num = parameters.GetCount();
for( i=0; i<num; i++ ) {
        Parameter param = parameters[ i ];
        SIObject source( param.GetSource() );
        if( source.IsValid() ) {
                ...
}
```

最後は Texture クラスからイメージクリップの情報を取得しましょう。テクスチャノードに繋がっているイメージクリップは Texture::GetImageClip()で取得します。

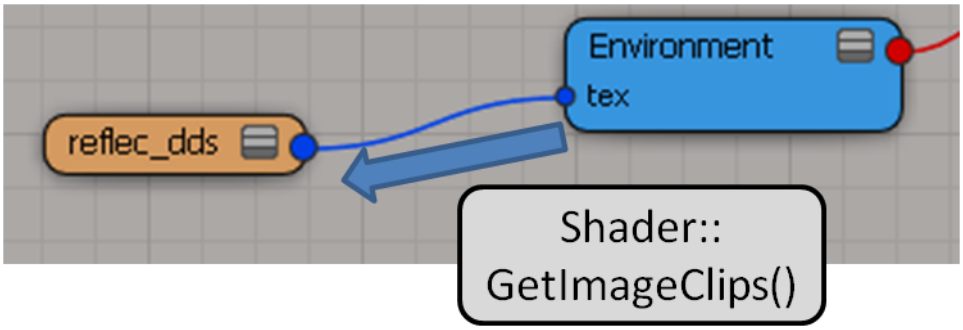



```
Texture texture( source );
ImageClip2 imageClip = texture.GetImageClip();
```

これでレンダーツリーのノードを一通り取得することができました。



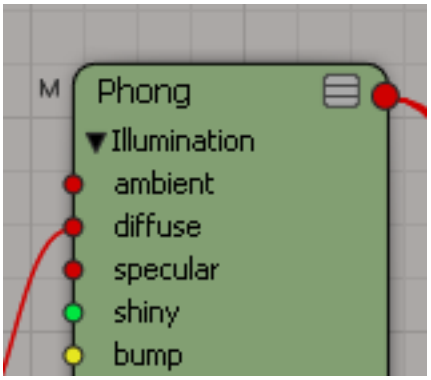
環境マップの場合は、シェーダーノードに直接イメージクリップに繋がっているので、Shader クラスから ImageClip を取得します。このときは Shader::GetImageClips()関数を使います。




```
Shader shader( source );
ImageClip2 imageClip = shader.GetImageClips()[ 0 ];
```

7.4　パラメータの値の取得

次にシェーダーノードが持っているパラメータの値を取得してみましょう。シェーダーノードの種類によって取得できるパラメータは変わってきます。たとえばイルミネーション用の Phong ノードは ambient、diffuse、specular、transparency、incandescence などのパラメータを持っています。該当するパラメータが存在しない場合は GetParameter()関数が無効なオブジェクトを返します。




```
Parameter paramDiffuse = shader.GetParameter( L"diffuse" );
Parameter paramSpecular = shader.GetParameter( L"specular" );
Parameter paramTransparency = shader.GetParameter( L"transparency" );
Parameter paramIncandescence = shader.GetParameter( L"incandescence" );
```

Parameter クラスから値を取得する場合は、格納されている値の種類を Parameter::GetValueType()を使って判別する必要があります。今回はカラーが格納されていることがわかっているので、値の種類の判定は行わずにカラーの値を直接取得します。

Parameter クラスがカラーの値を持っている場合、RGB の各成分は子供の Parameter に入っています（Parameter クラスは他の Parameter クラスを子供として持つことができます)。パラメータがカラーの場合、"red","green","blue"という名前のパラメータを子供に持っているので、そちらから値を取得します。

```
if( paramDiffuse.IsValid() ) {
        const float r = paramDiffuse.GetParameterValue( L"red" );
        const float g = paramDiffuse.GetParameterValue( L"green" );
        const float b = paramDiffuse.GetParameterValue( L"blue" );
}
```

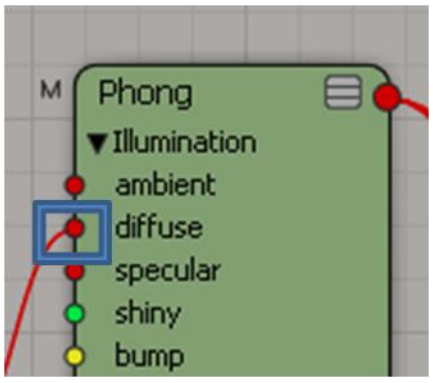
Softimage の transparency はカラー型なので注意が必要です。また diffuse や specular のカラーと違い、transparency のカラーは alpha を含んだ 4 要素となっています。


```
if( paramTransparency.IsValid() ) {
        const float r = paramTransparency.GetParameterValue( L"red" );
        const float g = paramTransparency.GetParameterValue( L"green" );
        const float b = paramTransparency.GetParameterValue( L"blue" );
        const float a = paramTransparency.GetParameterValue( L"alpha" );
}
```

これでパラメータの値を取得することができました。

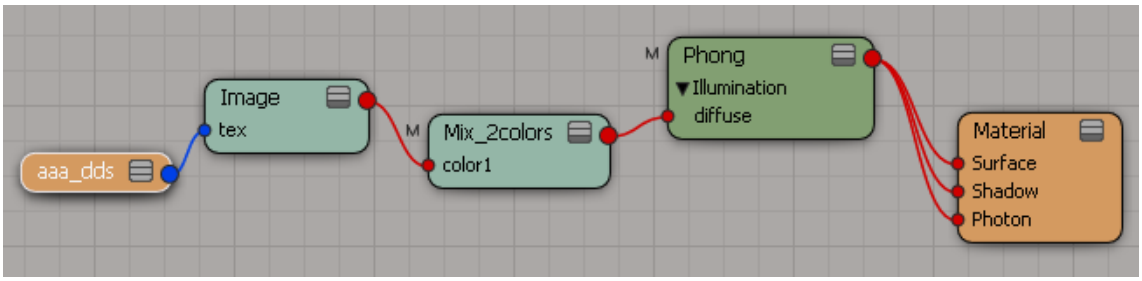
7.5　パラメータにテクスチャが繋がった場合

今度はパラメータにテクスチャが繋がっている場合を考えてみましょう。



このように diffuse にテクスチャを繋いでしまうと、diffuse のカラーが設定できません。プログラム上では diffuse の Parameter が取得できなくなります。diffuse にテクスチャを繋ぎつつ、カラーも設定したい場合はミキサーノードを使います。ミキサーノードは複数

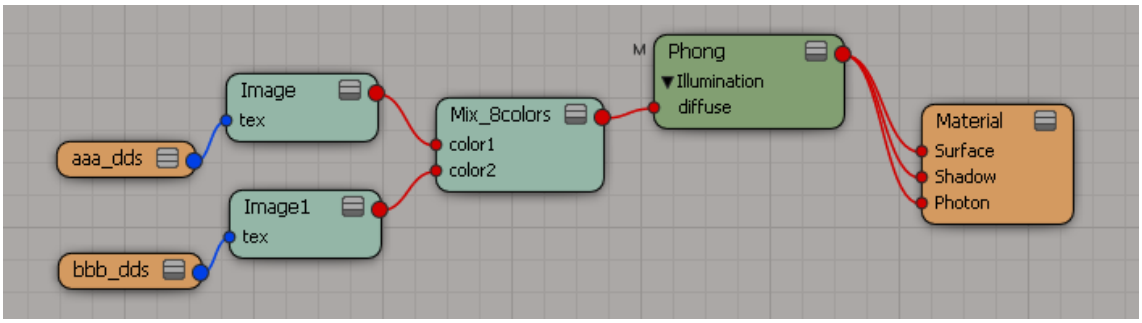
のカラーを混ぜ合わせるためのノードです。テクスチャノードとイルミネーションのノードとの間にミキサーノードを挟むと、テクスチャを繋ぎつつ、カラーを設定できるようになります。



ミキサーノードは Shader クラスです。識別には ProgID を使います。ミキサーノードで設定するカラーは"base_color"という名前で取得できます。

```
if( shader.GetProgID() == L"Softimage.sib_color_2mix.1" ||
        shader.GetProgID() == L"Softimage.sib_color_8mix.1" ) {
        Parameter paramBaseColor = shader.GetParameter( L"base_color" );
        float r = paramBaseColor.GetParameterValue( "red" );
        float g = paramBaseColor.GetParameterValue( "green" );
        float b = paramBaseColor.GetParameterValue( "blue" );
}
```

ちなみに、同じパラメータにテクスチャを２枚以上接続する場合もミキサーノードを使います。プログラムからの取得方法は先ほどと同じです。下の画像ではミキサーノードをつかって diffuse にテクスチャを２枚接続しています。



7.6　テクスチャ

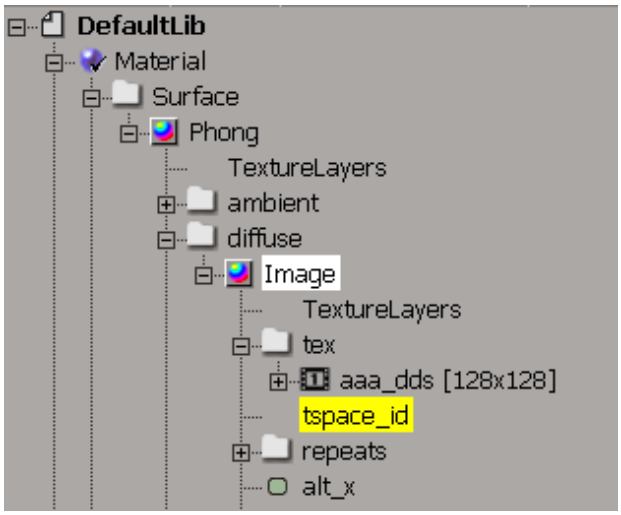
ここではテクスチャの情報を取得します。まずイメージクリップから画像ファイル名を取得してみましょう。イメージクリップの画像ファイル名は ImageClip2::GetFileName()で取得できます。戻り値は画像ファイル名のフルパスです。画像ファイルが設定されていな

い場合は空文字が返ります。

```
CString filename = imageClip.GetFileName();
```

今度はテクスチャが使用しているＵＶ座標の名前を取得します。テクスチャが使用するＵＶ座標はこの画面のテクスチャプロジェクションの部分で指定します。

このテクスチャプロジェクションの値は Texture クラスの”tspace_id”パラメータから取得できます。Explorer 上では以下の場所に入っています。




```
Parameter paramTSpaceID;
paramTSpaceID = texture.GetParameter( “tspace_id” );
CString uvName = paramTSpaceID.GetValue();
```

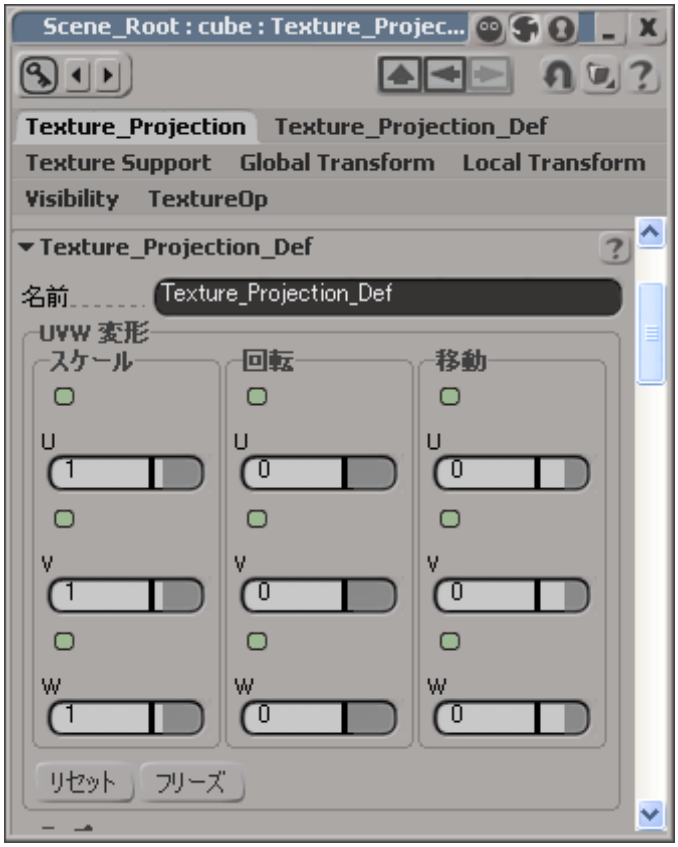
これでＵＶ座標の名前が取得できました。今回作成したエクスポーターのＵＶ座標を取得する部分はもう少し複雑になっています。たとえば環境マップの場合ＵＶ座標を使用しないので、テクスチャプロジェクションの値を取得できません。またテクスチャプロジェクションで指定されているＵＶ座標が存在しないこともあります。テクスチャプロジェクションを設定した後でポリゴンメッシュのＵＶ座標が削除されてしまうと、そのテクスチャは存在しないＵＶ座標を参照していることになります。今回のエクスポーターの実装ではポリゴンメッシュ側のＵＶ座標とテクスチャプロジェクションの名前をチェックして、適切なＵＶ座標を設定するようになっています。詳細はソースコードを参照してください。

次にテクスチャのリピート数を取得してみましょう。Softimage のリピート数は下の画像のテクスチャリピートで設定します。

テクスチャのリピート数は Texture クラスの"repeats"パラメータの"x"と"y"に入っています。

```
Parameter paramRepeats = texture.GetParameter( L"repeats" );
const float repeatX = paramRepeats.GetParameterValue( "x" );
const float repeatY = paramRepeats.GetParameterValue( "y" );
```

テクスチャのリピート数はテクスチャ自体の繰り返し回数のことです。一方、ポリゴンメッシュの UV に対してスケールとオフセットをかけることができます。



リピート数はテクスチャの属性、UV のスケールとオフセットはジオメトリの属性です。Softimage では両方設定できるので、間違えないようにしてください。

7.7　レンダーツリーの設定を簡単に行う方法

レンダーツリーでノードを一つ一つ繋げていく作業は複雑なので大抵のデザイナーは間違えます。そして設定を間違えた部分を後から探すのは一苦労です。これを回避する方法は３つあります。

最初の一つは先ほども述べたレンダーツリーを使わない方法です。すべての設定をカスタムパラメータに移してしまえば、レンダーツリーで行うのはテクスチャの接続だけになります。Softimage 上でレンダリングする必要がない場合は便利な方法です。

二つめの方法は使用するレンダーツリーの構造をプリセットとして保存しておいて使いまわす方法です。最初にプリセットを作成しておいて、それを共有すればミスが減ります。レンダーツリーを自動的に作成するスクリプトを作ってもよいでしょう。

最後はシェーダーコンパウンドという機能を使う方法です。これはレンダーツリー上の複数のノードをまとめて、一つのノードにしてしまう機能です。複雑なノードをひと固まりにまとめてしまうのでミスが減ります。また外部に公開するパラメータを設定できるので、設定する必要のある部分だけを公開して、残りの余分なパラメータを隠すことができます。

8　アニメーション

これからアニメーション情報を取得していくわけですが、その前に DCC ツール上のアニメーションついて考えてみましょう。

8.1　アニメーションについて

DCC ツールのアニメーションは大きく２つに分けられます。ファンクションカーブが存在するものと、そうでないものです。

ファンクションカーブが存在するもの：

- 手付けモーション
- モーションキャプチャ

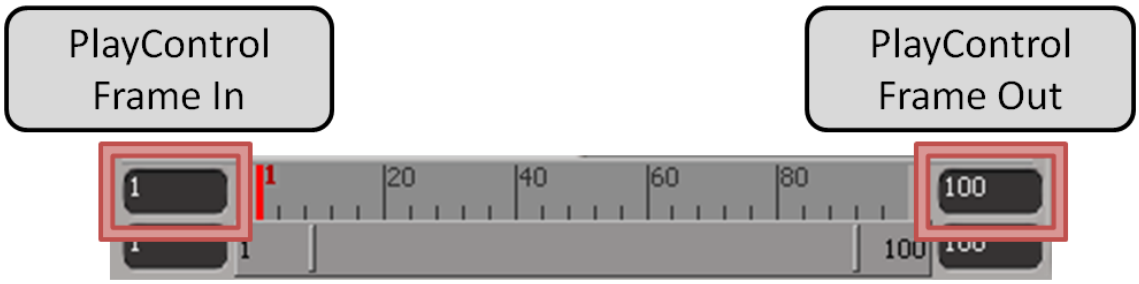
ファンクションカーブが存在しないもの：

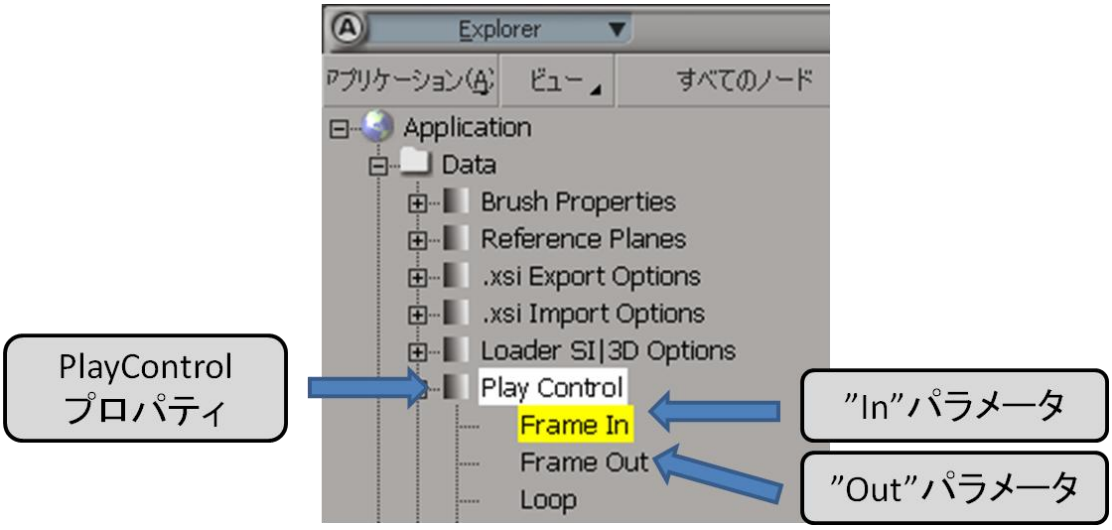
- エクスプレッション
- コンストレイン
- 物理シミュレーション
- その他

今回の実装ではファンクションカーブの有無に関わらず、アニメーションしているものはすべて出力するという仕様にしました。そのためアニメーションを出力する要素のすべてのフレームの値を取得しています。

8.2　アニメーションの範囲

それでは実際にアニメーションを出力する部分を説明します。最初にアニメーションが設定されているフレームの範囲を取得します。フレーム数の情報はプロジェクトの"Play Control"というプロパティが持っています。





Application クラスから Application::GetActiveProject()関数で現在有効なプロジェクトを取得します。それから Project::GetProperties()関数で"Play Control"プロパティを取得します。

```
Application app;
Property playControl;
playControl = app.GetActiveProject().GetProperties().GetItem( L"Play Control" );
```

アニメーションの設定されている範囲は"In"と"Out"というパラメータから取得できます。

```
const int inFrame = playControl.GetParameterValue( L"In" );
const int outFrame = playControl.GetParameterValue( L"Out" );
```

今回の実装では全フレームのアニメーションを出力しますが、一部の範囲のアニメーションを出力したいこともあります。その場合はエクスポーターのオプションで出力する範囲を指定することになるでしょう。

出力するフレームの範囲がわかったので、この間でループを回してアニメーションを出力します。ここではフレームの範囲に注意しましょう。たとえば１フレームから１００フレームまでアニメーションが定義されている場合、アニメーションは「１００フレーム」まで存在します。

```
for( int frame=inFrame; frame<=outFrame; frame++ ) {
        /* frame のアニメーションを出力 */
}
```

8.3　Transform アニメーション

それでは実際にノードの Transform アニメーション情報を取得してみましょう。最初の説明では省略しましたが、KinematicState::GetTransform()関数は取得したいフレームの番号を渡すと、そのフレームの Transform 情報を返します。

```
MATH::CTransformation transformation;
transformation = x3DObject.GetKinematics().GetLocal().GetTransform( frame );
```

CTransformation からスケールなどの各要素を取得する方法はノードのところで説明したのでここでは省略します。

8.4　パラメータのアニメーション

次にパラメータのアニメーションを出力してみましょう。マテリアルやジオメトリの情報を取得する過程でアニメーションしているパラメータを見つけたら、あらかじめリストに登録しておきます。パラメータがアニメーションしているかどうかは Parameter::IsAnimated()で調べます。

```
Parameter param = shader.GetParameter( "diffuse" );
if( param.IsAnimated() ) {
        /* アニメーションしているのでリストに登録 */
}
```

最初の説明では省略しましたが、Parameter::GetValue()関数の引数にフレーム数を与えると、そのフレームのパラメータの値が取得できます。

```
const float value = parameter.GetValue( frame );
```

GetParameterValue()関数の場合は第２引数にフレーム数を指定できます。

```
const float value = parameter.GetParameterValue( L"red", frame );
```

8.5　アニメーションの名前

COLLADA フォーマットではアニメーションに名前を付ける必要があります。この名前の付け方に厳密なルールはありませんが、アニメーションとその対象の対応が取れる名前にしておきましょう。今回のエクスポーターでは一般的に使われそうな名前にしています。

9　スキニング

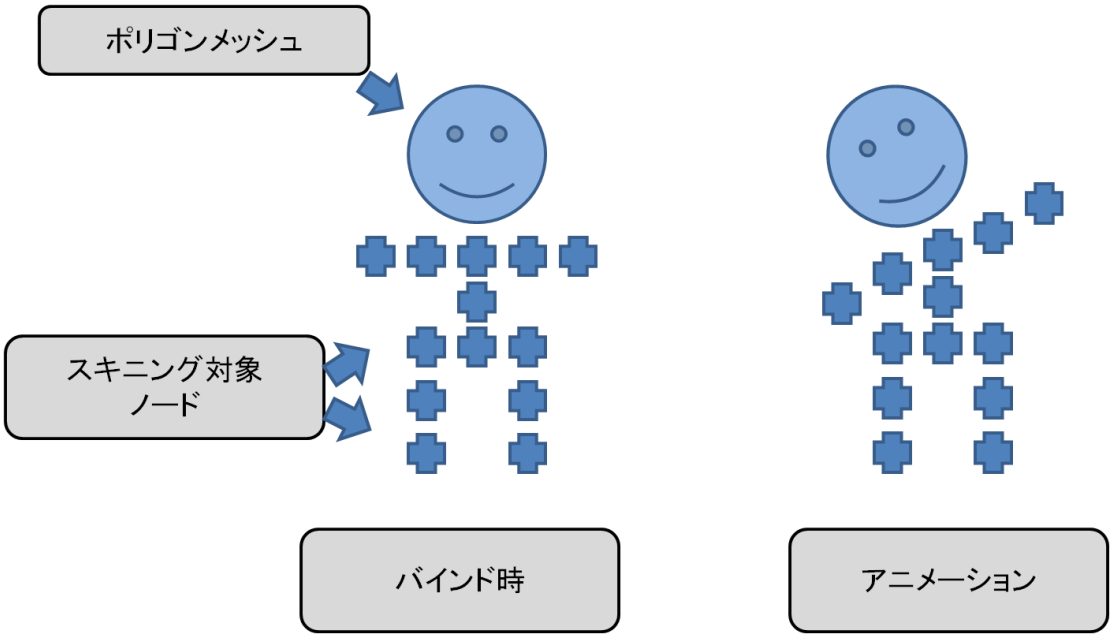
エクスポーター作成において、スキニング情報の取得は比較的難しい部分です。スキニングの正しい計算方法を理解するとデータの取得と値の検証が容易になるので、ここではスキニングの原理から見ていくことにしましょう。

9.1　用語説明

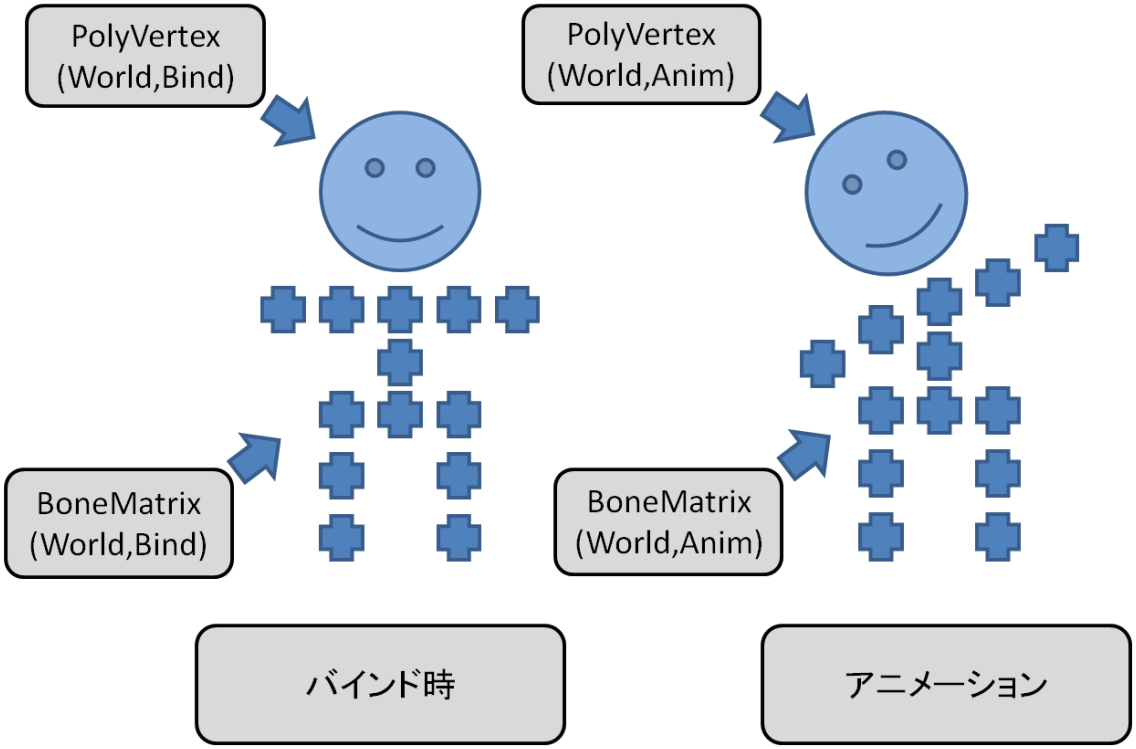
スキニングの説明の前に用語を２つ説明します。Softimage ではスキニングのことを「エンベロープ」と呼びます。意味はスキニングとほとんど同じです。またポリゴンメッシュとノードに対してスキニングの設定を行うことを「バインドする」と呼びます。これは他の DCC ツールでも使われる比較的一般的な用語です。

9.2　スキニングの原理

まずこちらの図をご覧ください。



左側はスキニングを設定した直後の状態です。左側の状態からボーンを動かしたのが右の状態になります。ここで左側のポリゴンメッシュのワールド座標での頂点位置を PolyVertex(World,Bind)，右側を PolyVertex(World,Anim)、左側のボーンのマトリックスを BoneMatrix(World,Bind)，右側を BoneMatrix(World,Anim)としてみましょう。



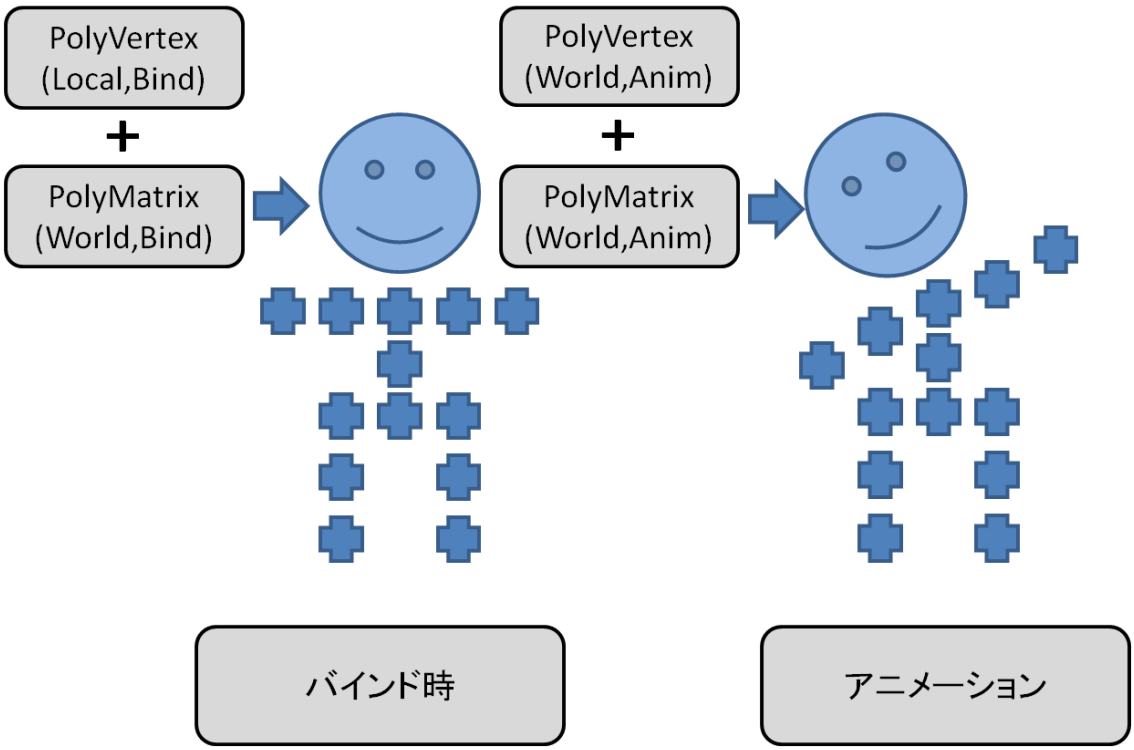
このときエクスポーターが出力する必要のあるポリゴンの位置情報はPolyVertex(World,Bind)と PolyVertex(World,Anim)のどちらでしょうか？　またマトリックスはBoneMatrix(World,Bind)とBoneMatrix(World,Anim)のどちらを出力するべきでしょうか？　そしてゲームエンジン上ではどのような計算を行うべきでしょうか？

ここのポイントは、ポリゴンメッシュはスキニング情報を割り当てられた（バインドされた）状態を基準として、そこからボーンが動かされることによってポリゴンメッシュの変形が行われるということです。バインドされたときのポリゴンメッシュの形状を出力しておいて、これをスキニングが割り当てられたときのボーンと動かされたボーンとの差分だけ変形すれば、スキニングによって変形された結果のポリゴンメッシュの頂点位置が計算できます。

```
PolyVertex(World,Anim)　=　PolyVertex(World,Bind)　*　BoneMatrix(World,Anim)　*　Inverse( BoneMatrix(World,Anim) )
```

ここでもうひとつ考えておきたいのがポリゴンメッシュ自体のマトリックスです。スキニングの付いたポリゴンメッシュは変形すると同時に、自分自身のマトリックスによって移動します。このマトリックスを考慮しておかないと、エクスポートしたモデルがゲームエンジン上で違う場所に表示されてしまいます（これはよくある不具合です）。

ここではポリゴンメッシュのマトリックスをそれぞれ PolyMatrix(World,Bind)、PolyMatrix(World,Anim)とします。先ほどまではポリゴンメッシュの頂点位置をワールド座標で考えていましたが、これをポリゴンメッシュ自体のマトリックスとローカル座標に分解しておきます。



スキニングを割り当てられた状態のポリゴンメッシュのローカル座標での形状を PolyVertex(Local,Bind)、動いた後のものを PolyVertex(Local,Anim)とすると以下の式になります。

```
PolyVertex(World,Bind) = PolyVertex(Local,Bind) * PolyMatrix(World,Bind)
PolyVertex(World,Anim) = PolyVertex(Local,Anim) * PolyMatrix(World,Anim)
```

これを先ほどの式に当てはめると以下の式になります。これが最終的なスキニングの計算式です。

```
PolyVertex(Local,Anim) * PolyMatrix(World,Anim) =
PolyVertex(Local,Bind)  *  PolyMatrix(World,Bind)  *  BoneMatrix(World,Anim)  *
Inverse( BoneMatrix(World,Anim) )
```

この式が正しいか調べるために、ポリゴンメッシュの座標がスキニングした状態から一切動いていない場合を考えてみましょう。バインド時とアニメーション時のポリゴンメッシ

ュ自身のマトリックスは等しくなります。

```
PolyMatrix(World,Bind) = PolyMatrix(World,Anim)
```

式の両辺からポリゴンメッシュのマトリックスを取り除くと以下の式になります。

```
PolyVertex(Local,Anim) =
PolyVertex(Local,Bind) * BoneMatrix(World,Anim) * Inverse( BoneMatrix(World,Anim) )
```

さらにボーンがスキニングした状態から動いていない場合は以下の式が単位マトリックスになります。

```
BoneMatrix(World,Anim) * Inverse( BoneMatrix(World,Anim )
```

これを先ほどの式に代入すると以下の結果になります。

```
PolyVertex(Local,Anim) = PolyVertex(Local,Bind)
```

ボーンもポリゴンメッシュも動いていない場合、ポリゴンメッシュは変形していません。よってこの結果は正しいと言えそうです。

9.3 出力する情報

これでエクスポートする必要のある情報がわかりました。

- バインド時のポリゴンメッシュのローカル座標での位置情報
- バインド時のポリゴンメッシュのワールド座標
- バインド時のボーンのマトリックス（ワールド座標）

これに加えて以下の情報も必要です。

- 頂点単位のウェイト値
- スキニング対象のノードの名前

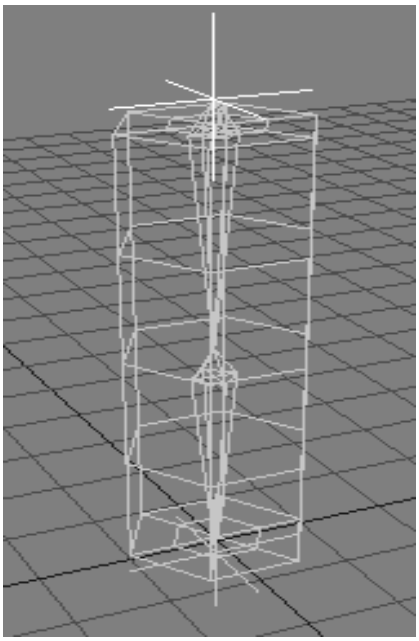
実際にはノードは親子構造になっているので、エクスポーターではノードの座標をローカル座標で出力して、ランタイム側でワールド座標に変換して使います。

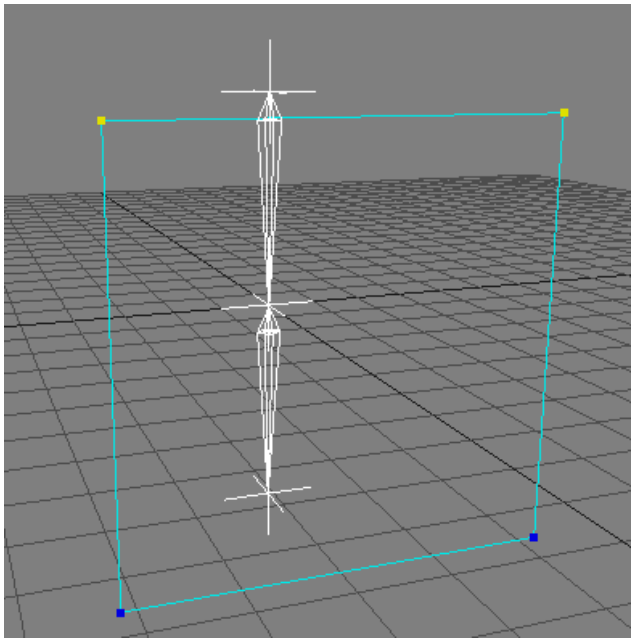
9.4 テストデータの作成

実際に Softimage から必要なデータを取得する前にテストデータを作成しましょう。まず

適当なポリゴンメッシュを作成して、スケルトン＞２Ｄチェインを描く、を選び骨構造を作成します。それからポリゴンメッシュを選んでエンベロープ＞エンベロープの設定を選びます。アイコンが変わるので、スキニングを設定したいノードを Explorer 上で左クリック選択して、右クリックで終了します。ノードを動かしてポリゴンメッシュが変形するのを確認してください。

最初のテストデータのポリゴンメッシュはできるだけシンプルにしてください。一番初めは４角形ポリゴン一枚だけで十分です。最初から円柱を作ってしまうとデバッグが大変です。ポリゴン数が多い上に、位置情報を見ただけではどこの部分かすぐにわからないからです。





テストデータができたので、実際にスキニングの情報を取得していきましょう。スキニングの情報は Envelope クラスに格納されています。まず X3DObject クラスから Envelope クラスを取得します。スキニングの設定が行われていない場合は Envelope クラスを取得することはできません。基本的には X3DObject にエンベロープは一つしか付かないので最初のエンベロープだけ調べればよいでしょう。

```
CRefArray envelopes = x3DObject.GetEnvelopes();
if( envelopes.GetCount() > 0 ) {
        Envelope envelope( envelopes[ 0 ] );
        ...
}
```

Envelope::GetDeformers()でウェイト対象が取得できます。戻り値は CRefArray なので適当なクラスにキャストします。今回は座標情報を取得したいので X3DObject にキャストしています。

```
CRefArray deformers = envelope.GetDeformers();
for( int i=0; i<deformers.GetCount(); i++ ) {
        X3DObject targetX3DObject( deformers[ i ] );
        CString targetName = targetX3DObject.GetName();
}
```

これでスキニングの対象となるノードの名前が取得できました。次に頂点ごとのウェイト値を取得しましょう。ウェイトの情報は Envelope::GetDeformerWeights()関数で取得できます。この関数は引数にウェイトを取得したいターゲットの X3DObject クラスを指定します。

```
CDoubleArray weights = envelope.GetDeformerWeights( targetX3DObject );
```

この関数の戻り値は頂点ごとのウェイト値（パーセント）の配列です。COLLADA ではウェイト情報を 0 から 1 の範囲で扱うので 0.01 を乗算します。

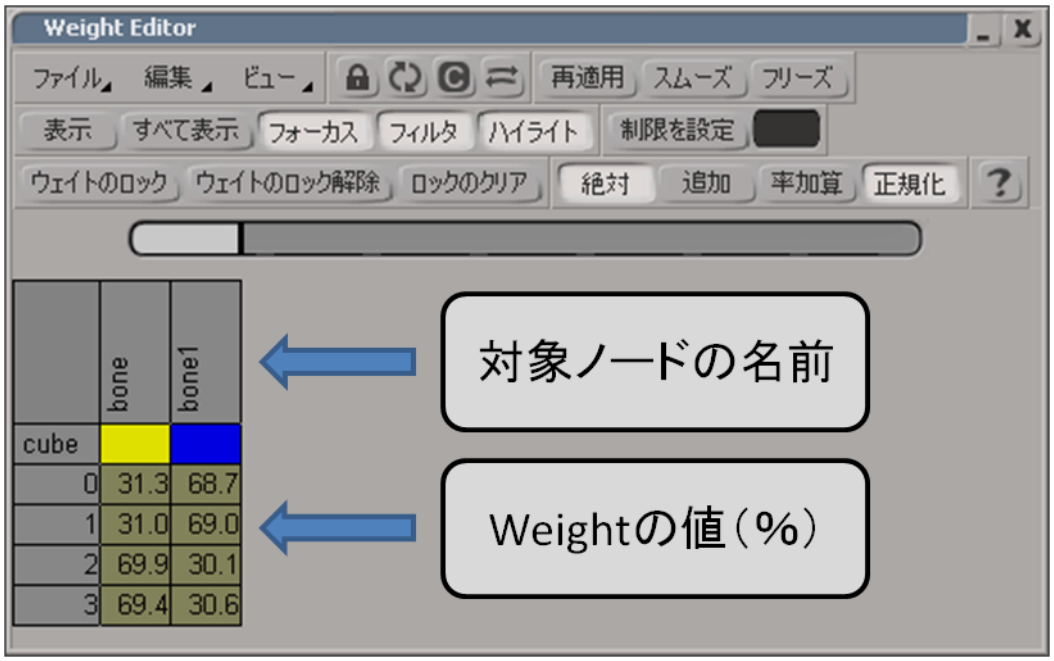
bone	31.3	31.0	69.9	69.4
bone1	68.7	69.0	30.1	30.6

Envelope::GetWeights()関数も使えますが、こちらは全ターゲットのウェイト値がまとめて返ってくるので面倒です。

頂点のウェイトの情報は頂点数と同じ数存在します。ターゲットとなっているノード数が N、頂点数を M とすると、頂点のウェイト情報は N x M 存在することになります。これはかなり大きな数ですが、実際には頂点ウェイトの対象として使われていないノードも沢山存在します。無駄なデータは出力しないほうがよいので、あらかじめウェイト値をすべて調べて、全部が 0 のノードは出力しないようにしましょう。今回のエクスポーターではその処理を行っています。

取得できたウェイトの値の確認は Weight Editor で行います。Weight Editor はエンベロー

プ＞ウェイトの編集で開きます。Weight Editor ではスキニング対象のノードの名前と頂点ごとのウェイトの値を表形式で見ることができます。取得した値が正しいかどうか一度確認しておいてください。



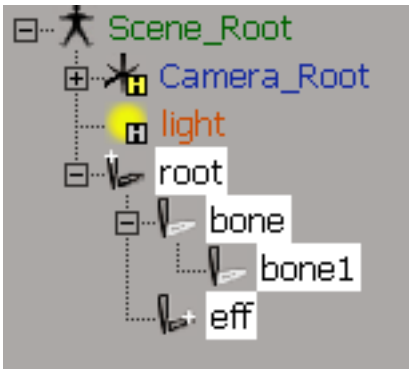
今度はスキニング対象のノードからバインド時の座標を取得してみましょう。スキニング対象となっている X3DObject クラスの GetStaticKinematicState()関数を呼ぶと、StaticKinematicState クラスを取得することができます。StaticKinematicState クラスはオブジェクトの基本ポーズを持っているクラスです。StaticKinematicState クラスも KinematicState クラスと同じように GetTransform()関数を持っているので、同様に座標情報を取得します。

```
CTransformation transform = x3DObject.GetStaticKinematicState().GetTransform();
```

ポリゴンメッシュのバインド時の座標も同様に取得することができます。

9.5 Softimage の骨構造

Softimage は独自の骨構造を持っています。ゲーム用のデータを作る上で、この骨構造が問題になることがあるので少し説明します。Softimage の骨構造はルート,ボーン,エフェクタの３つから構成されます。骨が一つだけの場合も必ずこの３種類のノードが作られてしまいます。この３種類のノードをすべてエクスポートするとランタイム側の負荷が大きくなり、スキニング計算の負荷も増えます。



Softimage でエクスポートするノードの数を減らす方法は２つあります。最初の方法はルート、ボーン、エフェクタのボーンだけを使う方法です。ルートとエフェクタにはアニメーションと頂点ウェイトを設定できなくなりますが、ノードの数は最低限に抑えられます。この方法の場合、モデルを作成するデザイナーが間違えてルートとエフェクタを使ってしまうと正常に動作しないデータになってしまうので注意が必要です。

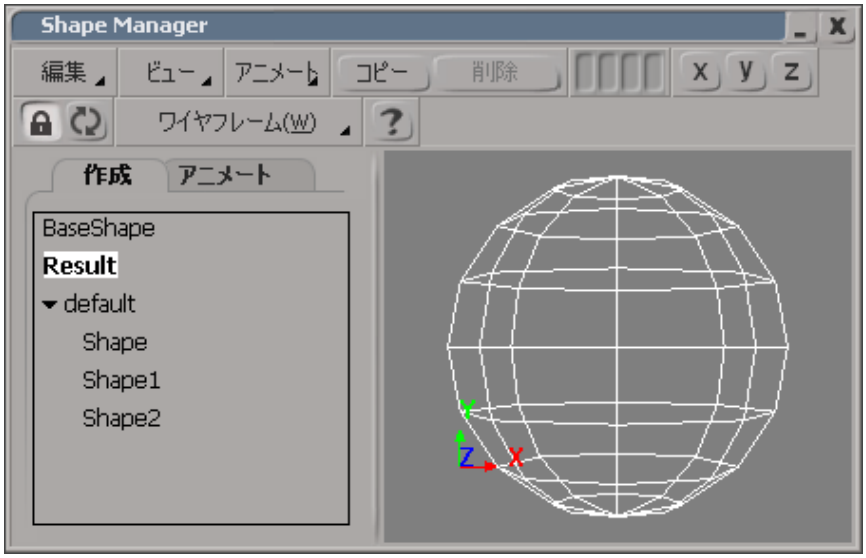
もう一つの方法はすべての骨構造を Null で置き換えてしまう方法です。Softimage では Null をスキニングの対象に使えるので、エクスポートする骨構造を Null で置き換えれば最低限のノードだけにすることができます。モーションを作成する場合には、別に Softimage の骨構造を作っておき、その骨構造と Null をコンストレインして使います。

10 シェイプ

次はシェイプの情報を取得します。

10.1 Softimage 上のシェイプ

Softimage 上でのシェイプは Shape Manager で見ることができます。メニューの表示＞アニメーション＞Shape Manager から表示します。



左側のシェイプを切り替えると、右側に表示される形状が切り替わります。アニメートタブに切り替えてウェイト時を変化させると、各形状をブレンドすることができます。このように複数の形状をブレンドさせて、ポリゴンメッシュの形状を変化させるのがシェイプと呼ばれる機能です。一般的にはモーフィングと呼ばれることがあります。このシェイプ機能は大半の DCC ツールに用意されています。

10.2 出力する情報

このシェイプ機能をランタイム側で実装するためには、各シェイプの名前と頂点情報が必要です。

シェイプの形状の計算に必要な頂点情報は位置と法線です。頂点カラーや UV 座標は必要ありません。そのため２つ目以降のシェイプは位置と法線だけ持っていればよいことになります。実際には接線と従法線もシェイプによって変形するのですが、ゲームエンジンによってはその計算を省略することがあります。今回のエクスポーターでは接線と従法線は最初の形状だけ出力することにしました。

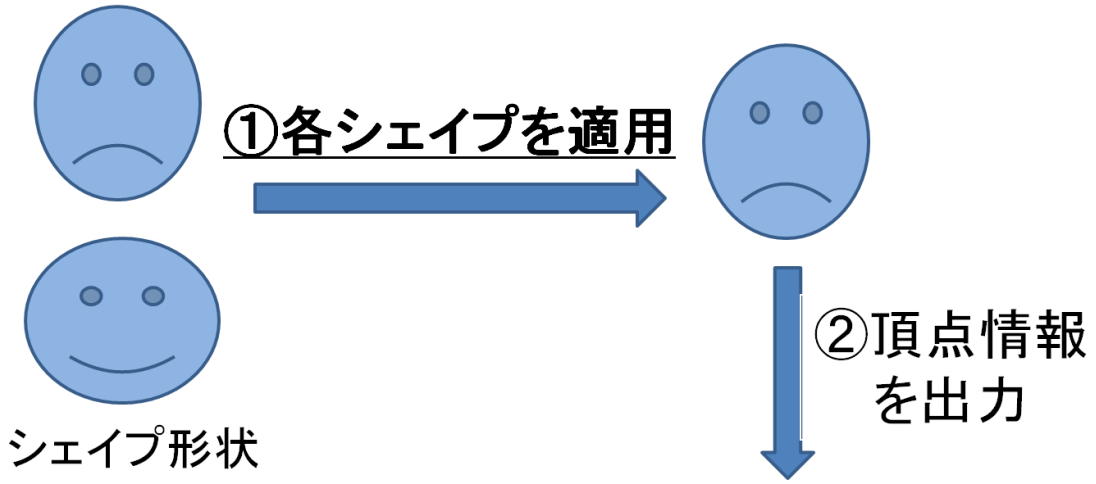
10.3 シェイプ形状の出力

シェイプの基本形状はジオメトリ情報のところで出力しました。ここではシェイプによって変形した形状を出力します。

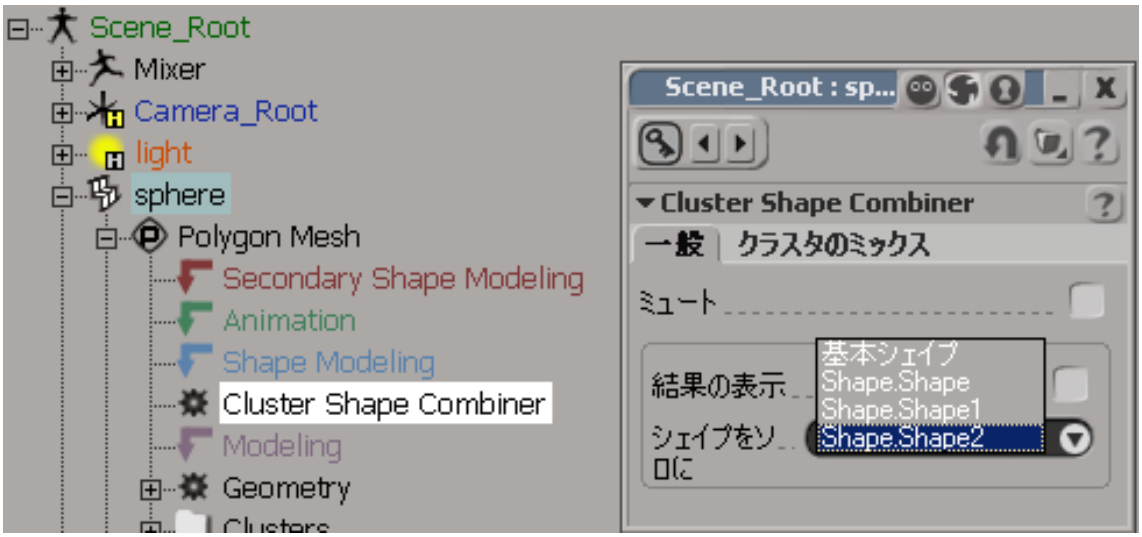
まず各シェイプの名前を取得しましょう。シェイプの名前はCGeometryAccessor::GetShapeKeys()から取得できます。ShapeKey の配列が返ってくるので、そこから名前を取得します。

```
CRefArray shapeKeys = ga.GetShapeKeys();
for( int i=0; i<shapeKeys.GetCount(); i++ ) {
        ShapeKey shapeKey( shapeKeys[ i ] );
        CString name = shapeKey.GetName();
}
```

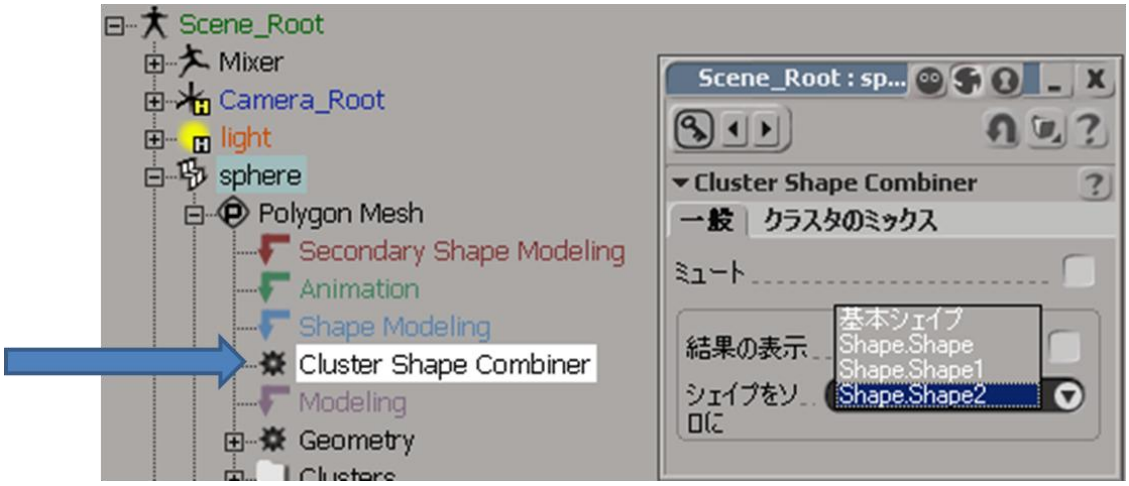
今度は各シェイプの頂点情報を取得します。今回の実装では Softimage 上のシェイプ形状を実際に切り替えてから頂点情報を出力しています。



まず Softimage の GUI 上でシェイプを手動で切り替えてみましょう。Explorer からシェイプを設定したポリゴンメッシュの中身を表示すると"Cluster Shape Combiner"というオペレータが見つかります。このオペレータを開いて"結果の表示"のチェックを外して、"シェイプをソロに"の項目を変更するとシェイプ形状が切り替わります。

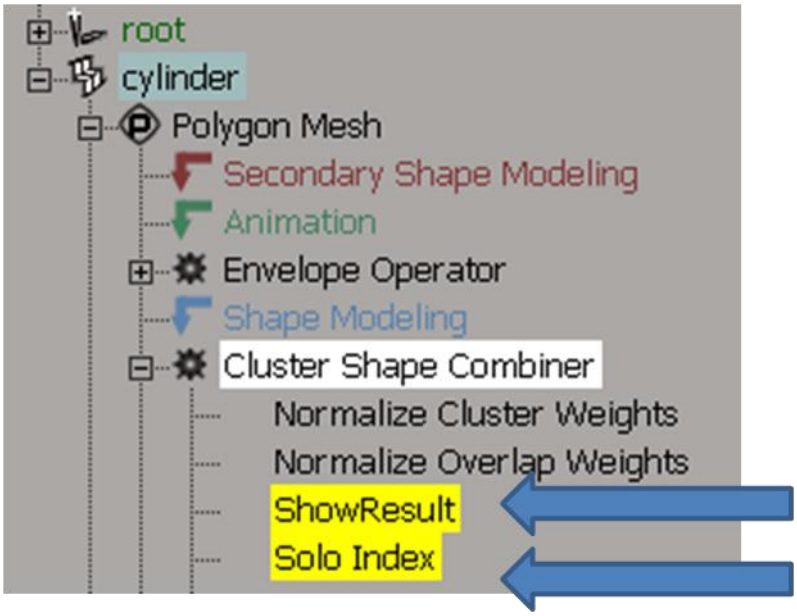


今度はこの一連の作業をプログラム側から行ってみましょう。まず X3DObject から先ほどの "Cluster Shape Combiner" オペレータを取得します。スクリプト用の名前は "clustershapecombiner" なので、そのフルネームを作成してから CRef::Set() を使って Operator クラスへの参照を取得しています。ここで出てきた Operator クラスは Softimage 上のオペレータに相当するクラスです。




```
CRef ref;
ref.Set( x3DObject.GetFullName() + L".polymsh.clustershapecombiner" );
Operator clusterShapeCombiner( ref );
```

"Cluster Shape Combiner" オペレータの "ShowResult" パラメータを false にしてから、"SoloIndex" パラメータを切り替えます。"SoloIndex" の値は 0 がオリジナルの形状で、シェイプの形状は 1 から始まります。




```
Parameter showResult = shapeCombiner.GetParameter( L"ShowResult" );
showResult.PutValue( false );

Parameter soloIndex = shapeCombiner.GetParameter( L"SoloIndex" );
for( int i=0; i<shapeCount; i++ ) {
        // シェイプ形状を切り替える
        soloIndex.PutValue( i+1 );

        /* 形状を出力 */
        ...
}

// 元に戻す
showResult.PutValue( true );
```

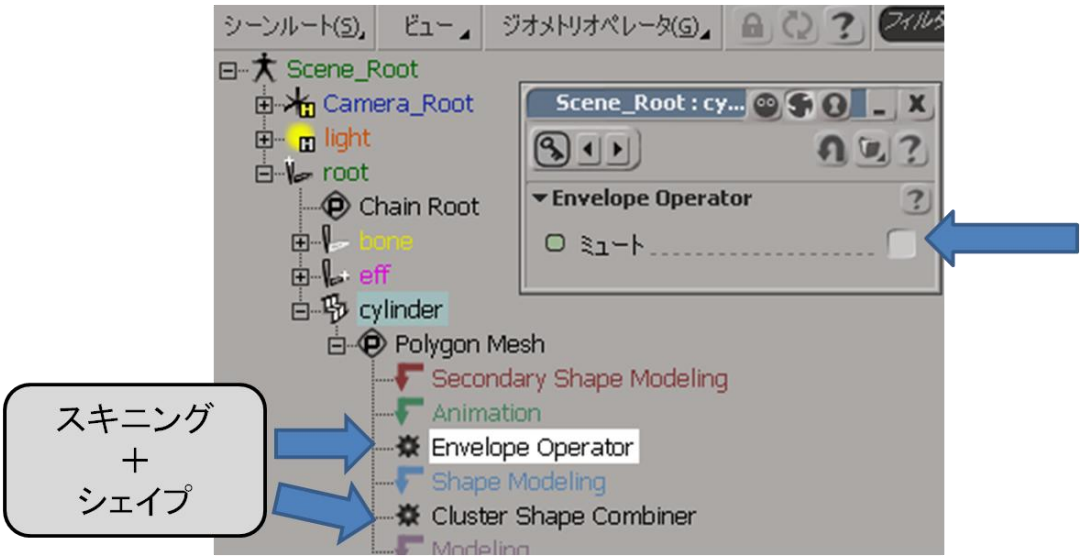
シェイプで変形されたジオメトリ情報を取得する場合は、PolygonMesh::GetGeometryAccessor()関数の引数に siConstructionModePrimaryShape を渡します。これを忘れるとシェイプで変形していないオリジナルの形状になってしまうので注意してください。

これでシェイプごとの頂点情報が取得できました。ここで使用した SoloIndex の番号は、CGeometryAccessor::GetShapeKeys()から取得したシェイプの順番と一致しています。ShapeManager でシェイプを切り替えるとスクリプトエディタにログが出るので確認してください。シェイプの名前を取得する方法はいくつかありますが、他の方法では SoloIndex と順番が一致しないので注意しましょう。

10.4 スキニング＋シェイプの場合

データによっては、スキニングとシェイプが両方適用されているポリゴンメッシュが存在します。この場合はスキニングを一度無効化してからシェイプの形状を取得します。

まず初めに Softimage 上でスキニング情報を無効化してみましょう。Explorer でスキニング適用済みのポリゴンメッシュの中身を表示すると"Envelope Operator"という名前の Operator が見つかります。この Operator を開いてミュートにチェックを入れるとスキニングが一時的に無効になります。



今度はこの作業をプログラム側から実行してみましょう。X3DObject::GetEnvelopes()で Envelope の一覧を取得して、"Envelope Operator"という名前の Envelope を探します。その Envelope の"mute"パラメータに Parameter::PutValue()で true を設定するとスキニング結果が無効になります。スキニングを「無効」にするときに「true」を指定するので間違えないようにしてください。シェイプ形状の取得が終わった後は元の値に戻します。

```
CRefArray envelopes = x3DObject.GetEnvelopes();
const int num = envelopes.GetCount();
for( int i=0; i<num; i++ ) {
        Envelope envelope = envelopes[ i ];
        if( envelope.GetName() == L"Envelope Operator" ) {
                Parameter paramMute = envelope.GetParameter( L"mute" );
                if( paramMute.IsValid() ) {
                        // スキニングを無効化する
                        paramMute.PutValue( true );
                }
        }
}
```

11 カスタムパラメータについて

ここではゲーム開発でDCCツールを使う上で避けては通れないユーザー定義パラメータについて説明します。

11.1 ゲーム用のパラメータを設定する方法

ゲーム開発用のコンテンツパイプラインではゲーム特有のパラメータ設定が必要になります。このパラメータを設定する方法は大きく分けて２つあります。

- ゲームエンジンで設定する方法
- DCC ツール上で設定する方法

最初の「ゲームエンジンで設定する方法」では DCC ツールとは別のゲームエンジン上でデータを読み込んでそちらでパラメータを設定します。大規模なゲーム開発や、ミドルウェアを使用した開発でよく使われます。この方法は DCC ツールに一切依存しないので、自由に GUI を作ることができます。しかし、この方法を使うためにはエディタとして使えるゲームエンジンが必要になります。

一方の「DCC ツール上で設定する方法」は DCC ツールの機能を使ってパラメータを追加します。既存の DCC ツールに対してパラメータを追加することになるので、DCC ツール側の仕様に激しく依存します。また DCC ツールによって実現方法が異なってくるので、個別の対応が必要です。ツール間のデータコンバートのときも問題になってくるでしょう。ですが DCC ツールだけで完結できるので、ゲームエンジンが存在しない開発環境や比較的小規模なゲーム開発においては便利な方法です。今回のエクスポーターでは後者の「DCC ツール上で設定する方法」を想定しています。

Softimage 上でユーザー定義のパラメータを設定する方法にはいくつか種類があります。

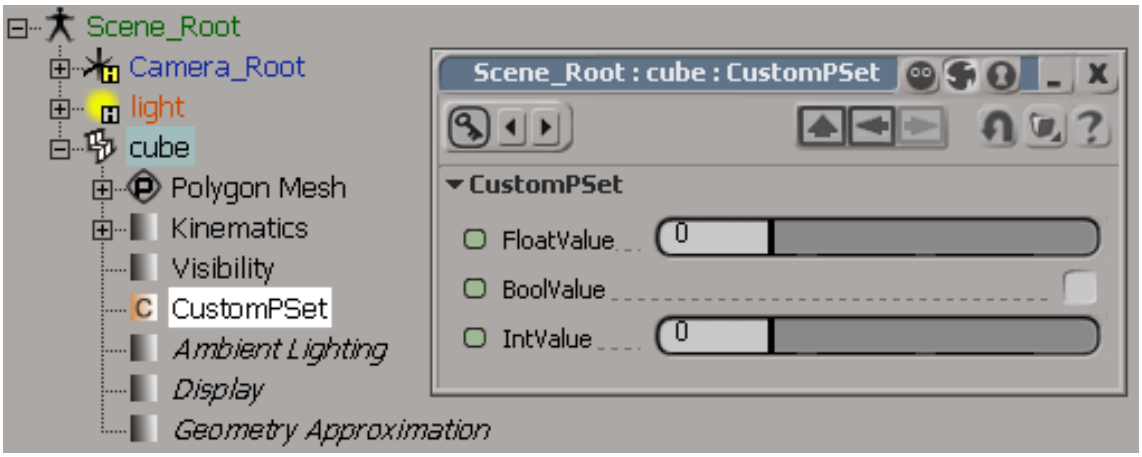
- 名前に埋め込む方法
- カスタムパラメータによる方法

「名前に埋め込む方法」ではユーザーが定義したいパラメータをノードやマテリアルの「名前」に文字列として追加します。設定されているパラメータが見た目ですぐにわかるというメリットがありますが、パラメータが増えてくると名前が長くなってしまうデメリットもあります。これは古い方法ですが、ツール間のデータコンバートでデータが失われにくいという隠れた長所もあります。

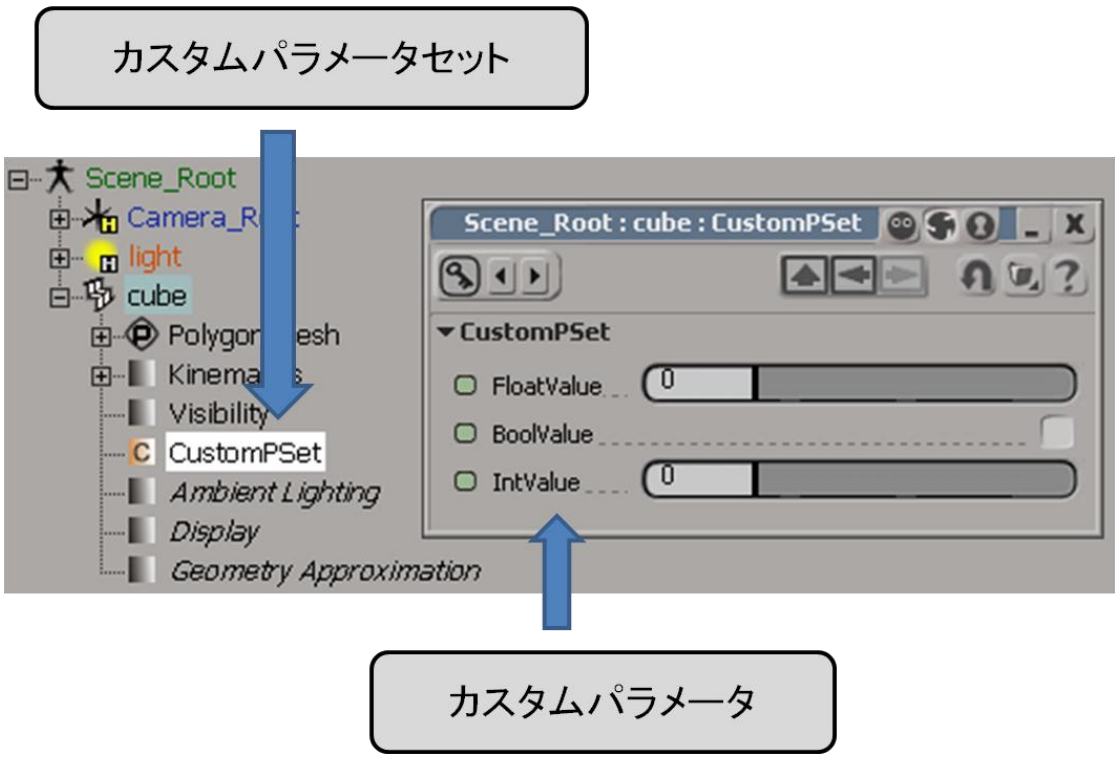
「カスタムパラメータによる方法」では Softimage のカスタムパラメータという機能を使ってユーザー定義のパラメータを設定します。カスタムパラメータは任意の個数追加できるので「名前に埋め込む方法」よりも便利です。ただ残念なことに Softimage には複数のカスタムパラメータの値を一括で編集する機能がありません。ゲーム用のパラメータは一度にまとめて変更したいことがあるので、この点は何かしらの対応をする必要があります（一括変換用のツールを用意するなど）。今回のエクスポーターではこの「カスタムパラメータによる方法」を採用しています。

11.2 カスタムパラメータとは？

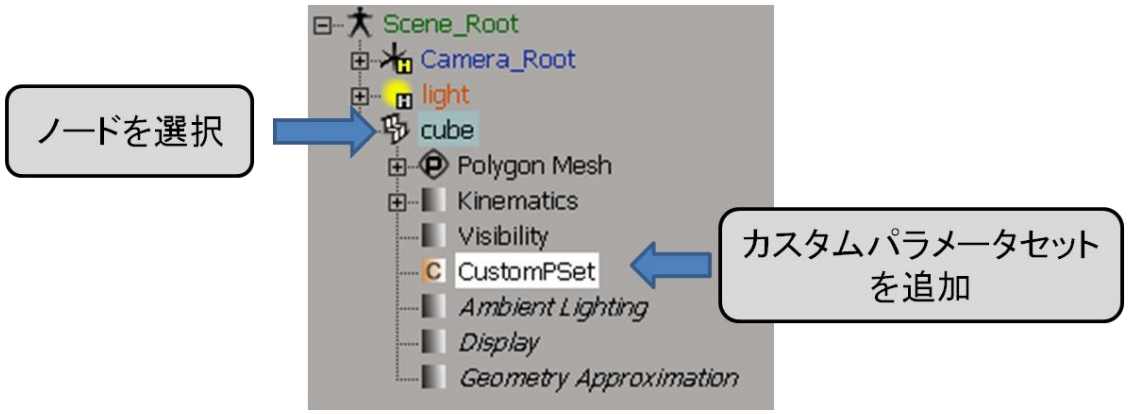
カスタムパラメータは Softimage 上でユーザーが任意に追加することのできるパラメータです。パラメータの型には任意の型が使用できます。またノード、マテリアル、テクスチャ、イメージソース（画像ファイル）などに追加することができます。



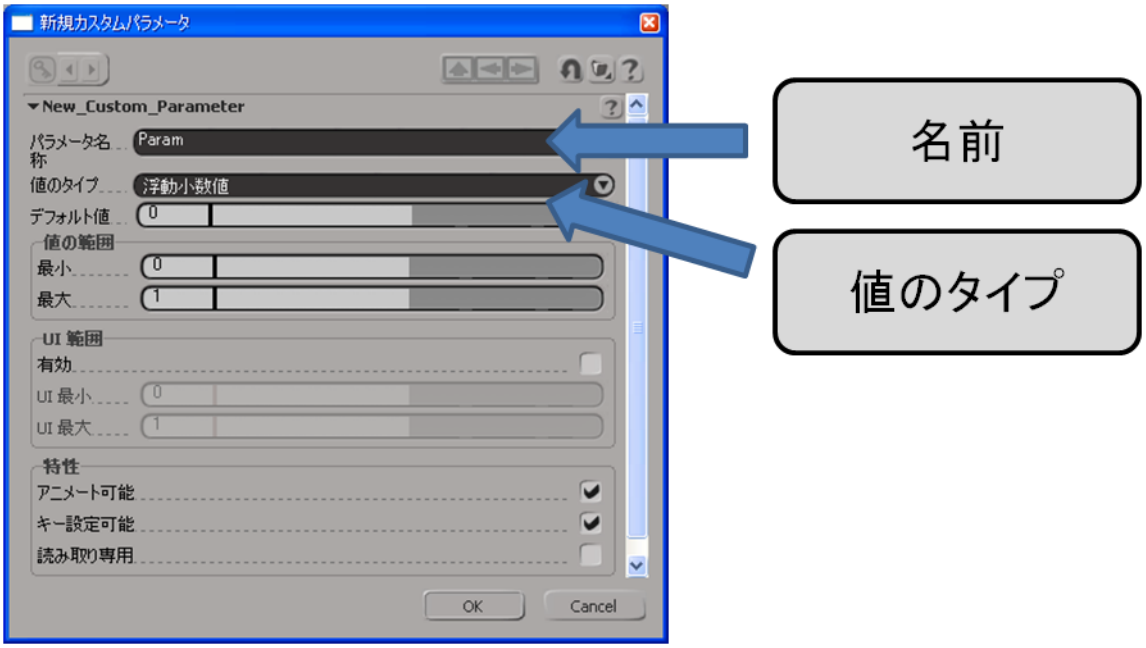
カスタムパラメータは「カスタムパラメータセット」と「カスタムパラメータ」から構成されます。カスタムパラメータが一つの値に相当し、複数のカスタムパラメータを格納するための入れ物がカスタムパラメータセットです。



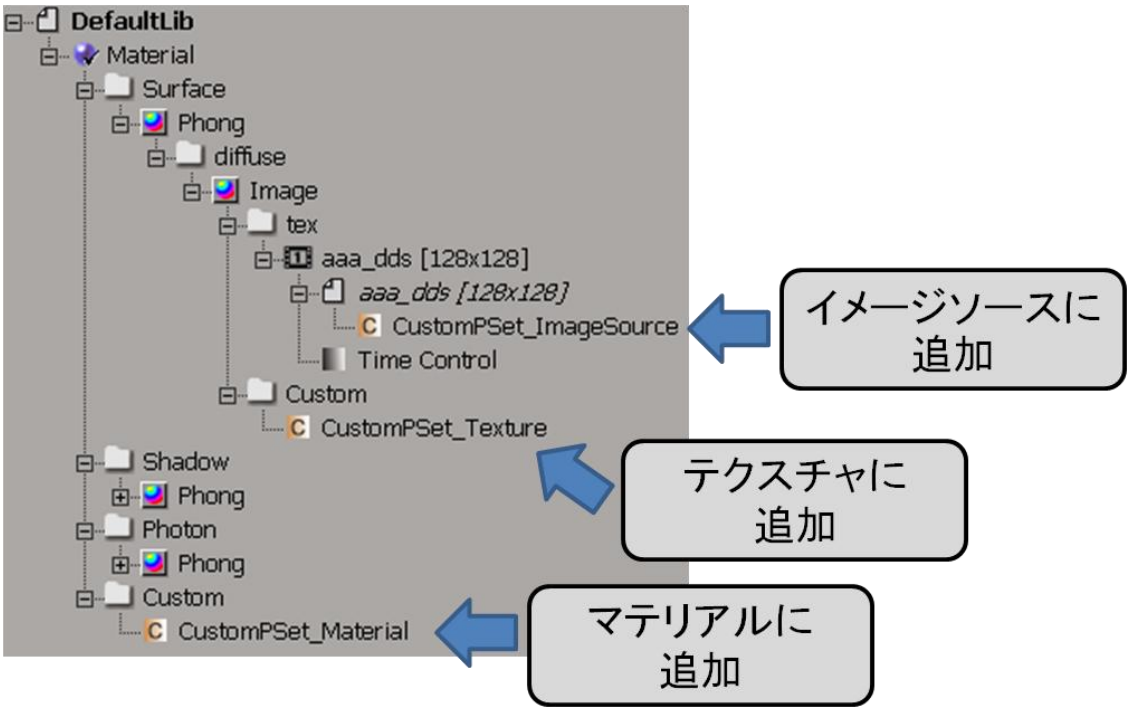
では Softimage 上で実際にカスタムパラメータを追加してみましょう。キューブを一つ作って、メニューのパラメータ＞新規カスタムパラメータセットを選ぶとキューブの下にカスタムパラメータセットが作られます。



今度は作成したカスタムパラメータセットを選択してメニューのパラメータ＞新規カスタムパラメータを選びます。新規カスタムパラメータのダイアログが開くので、名前や型を設定してＯＫボタンを押します。カスタムパラメータセットを開くとカスタムパラメータが追加されていることがわかります。同様に複数のカスタムパラメータをカスタムパラメータセットに追加することができます。



今回はノードに追加しましたが、マテリアル、テクスチャ、画像ファイル（イメージソース）にも同様に追加することができます。



このようにカスタムパラメータは Softimage のインターフェースから作成することができ

ますが、毎回手動で追加するのは大変なので、実際にはスクリプトを使って自動的に追加します。エクスポーターのオプション設定の部分は Python スクリプトからカスタムパラメータ（カスタムプロパティ）を作成しています。興味のある方はソースコードを参照してください。カスタムパラメータセットは同じ場所に複数追加できますが、この場合自動的にリネームされてしまいます。スクリプトで自動的に追加するときは重複に注意してください。

カスタムパラメータが追加できたので、今度はこの値をプラグインから取得してみましょう。最初はノードからカスタムパラメータセットを取得します。カスタムパラメータセットは Property の一種として格納されており、ClassID が siCustomPropertyID となっています。各クラスから ClassID が siCustomPropertyID となっている Property を取り出して情報を取得します。

SceneItem::GetProperties()関数を使うとプロパティ一覧を CRefArray クラスとして取得できます。

```
CRefArray properties = sceneItem.GetProperties();
for( int i=0; i<properties.GetCount(); i++ ) {
        Property prop = properties[ i ];
        if( prop.GetClassID() == siCustomPropertyID ) {
                /* カスタムパラメータセットの情報を取得 */
        }
}
```

今回の実装ではすべてのカスタムパラメータセットを取得する実装になっていますが、一部のカスタムパラメータセットだけ取得したいこともあります。この場合は識別用のカスタムパラメータを仕込んでおくか、カスタムパラメータセットの名前で識別します。

次にマテリアルのカスタムパラメータセットを取得します。Material クラスは SceneItem の派生クラスではありませんが、Material::GetProperties()関数を持っているので、こちらを使います。

```
CRefArray properties = material.GetProperties();
```

今度はテクスチャのカスタムパラメータセットを取得します。Texture クラスは Shader クラスの派生クラスなので、Shader::GetProperties()から取得できます。

```
CRefArray properties = texture.GetProperties();
```

イメージソースのカスタムパラメータセットは ImageClip2 クラスを経由して取得します。ImageClip2::GetSource()でイメージソースに相当する Source クラスが取得できるので、そのプロパティを取得します。

```
ImageClip2 imageClip;
...
Source source( imageClip.GetSource() );
CRefArray properties = source.GetProperties();
```

イメージソースは使用している画像ファイルと一対一対応しています。一方テクスチャはこのイメージソースを参照しているので、テクスチャの数はイメージソースの数より多くなることがあります。画像ファイル単位のパラメータはイメージソースに、テクスチャ単位のパラメータはテクスチャに設定するようにしましょう。

カスタムパラメータセットの Property クラスが取得できたので、今度は格納されているカスタムパラメータの情報を取得します。カスタムパラメータからは表示用の名前とスクリプト用の名前の両方が取得できますが、今回はスクリプト用の名前を使います。

```
CParameterRefArray parameters = property.GetParameters();
for( int p=0; p<parameters.GetCount(); p++ ) {
        Parameter parameter = parameters[ p ];
        CString scriptName = parameter.GetScriptName();
                /* カスタムパラメータの情報を取得 */
}
```

カスタムパラメータセットに入っているカスタムパラメータは値の型がわからないので、Parameter::GetValueType()関数を使って型情報を取得します。GetValueType()関数の戻り値は CValue::DataType です。DataType には沢山の型があるので、すべてに対応するのは大変です。今回のエクスポーターではよく使いそうな型だけ対応しています。

- カラー：siColor4f:
- ３２ビット符号あり整数：siInt4:
- ３２ビット符号なし整数：siUInt4:
- 実数(float)：siFloat:
- 倍精度実数(double)：siDouble:
- ブール型：siBool
- 文字列型：siString:

12 エクスポーター作成のコツ

最後にエクスポーター作成のコツについて説明します。

12.1 Softimage の操作を覚えよう

Softimage のプラグイン開発を行う場合、Softimage 自体の操作方法を習得しておくと開発効率が上がります。そもそも基本的な操作を覚えておかないと自分が作成したプラグインのテストすら行えません。プラグインのテストのたびにデザイナーにテストデータを作ってもらっていては時間の無駄です。エクスポーターを作るのならば以下の操作はできるようになっておきましょう。

- カメラの移動（基本的なショートカットを覚える）
- ポリゴンメッシュの作成
- マテリアルの設定（クラスターを使った複数マテリアルの割り当て）
- レンダーツリーを使った質感設定（環境マップ、法線マップ含む）
- ＵＶの設定（複数のＵＶ座標を別々のテクスチャに設定するなど）
- 頂点カラーのペイント
- スキニングの設定（ウェイトの編集含む）
- シェイプの作成
- 簡単なアニメーションの作成

基本操作の勉強には Softimage に付属しているチュートリアルが便利です。会社でデザイナー向けの研修を行っているのならば、そこに参加させてもらうのもよい方法です。

12.2 バグの調べ方のコツ

エクスポーターを作ってリリースすると大量のバグ報告が上がってきます。実際のゲーム開発の現場では、小さいデータでテストしているときは起こらなかった不具合が次々に発生します。またデザイナーはこちらの想像をはるかに超えたデータを作ってしまうことがあります。

エクスポーターの不具合が発見された場合は、該当するシーンのデータをもらって調べます。ツール関係に明るいデザイナーは問題が起こる最小限のシーンを用意してくれますが、大抵の場合はシーンを丸ごともらって調べることになります。

ここで大事になってくるのは原因の絞り込みです。シーンをもらったら最初に余計な要素を DCC ツール上で削ります。問題が起こる最小限のシーンを用意してからデバッグ作業に

入りましょう。DCC ツール上で要素を削っていく場合は、要素を半分ずつ削っていきます。1000 個のノードが入っているシーンの場合は、最初の 500 の要素を削除してエクスポートして問題が起こるか調べます。問題が起こった場合は残りの 500 の要素に原因があるので、今度はこちらを半分の 250 に削ります。このように半分、半分と削っていくと原因を楽に突き止めることができます。

エクスポーターなどのコンテンツパイプラインのデバッグ作業では、問題を起こしている要素を特定できれば作業は終了したも同然です。最初から目を皿にしてデバッガとにらめっこするのではなく、DCC ツール上での原因の絞り込みに大半の時間を使いましょう。