LScript
日本語ユーザーガイド

著者： Brian Marshall、Scott Wheeler

日本語訳・マニュアル制作：株式会社ディ・ストーム

Manual version: 1.0J - beta

July, 2002

LScript 日本語ユーザーガイド 目次

第 3 章：関数

第 4 章：条件文

第 5 章：ループ文

第 6 章：モデラー LScript 概論

第 7 章：モデラースクリプトの基礎

第 8 章：モデラー／ポイントとポリゴン

第 9 章： VMAP（頂点マップ）

第１０章：サーフェイスとレイヤー

第１１章：変位マップ

第１２章：オブジェクト置き換え

第１３章：カスタムオブジェクト

第１４章：プロシージャルテクスチャ

第１５章：イメージフィルタスクリプト

第１６章：アイテムアニメーション

第１７章：チャンネルフィルタ

第１８章：ジェネリック（総括）スクリプト

第１９章：マスタークラス

第２０章：インターフェイス概論

第２１章：モデラークラスと Generic（総括）クラスのインターフェイス

マニュアルの表記方法について

書体と色

このマニュアルは、LightWave 3D [7] 日本語マニュアルの表記方法に準じて記述されていますが、プログラミングの解説という性質上、以下の違いがあります。

機能や用語

LScript や LightWave 3D の機能、メニュー名、および一般的なプログラミング用語が初めて出てくる箇所は、**黒の太字**で書かれています。

例：

このような場合、**変数**を使えばスクリプト実行中に、このデータを…

コマンドや構文

コマンドや構文は、青で書かれています。

例：

```
main
{
  // このスクリプトは info ボックスを画面上に表示します
  // 変数テキストの値は "Hello World!"です
  …
}
```

コメント行

本文のスクリプト中に、"//"や"/*"で囲まれたコメント行があります。このマニュアルでは、分かりやすく解説するためにコメント行を日本語で記述していますが、実際の LScript では英数字しか使用できません。

第 1 章：スクリプト概論

スクリプトの作成法を学びたいのですね？素晴らしいことです！独自のスクリプトを作成出来るということは、アーティストにとって有力な技能です。望みどおりのカスタムツールが作成できれば、組織やLightWaveコミュニティにとってあなたは貴重な人材となるでしょう。またアーティストとプログラマという両方の側面を持つ非常に稀な人材となります。素晴らしいツールを設計し、特定の人々つまりアーティストのために特別なツールを作ることが出来る人材になるのです。

NewTekのようなソフトウェア開発者にとって、ユーザーが必要とする機能を、必要な時に正確に処理出来るLightWaveツールを設計するなんてことは、とても出来るものではありません。LightWaveのユーザーは広範囲にわたっており、フィルムやビデオ、印刷やゲームなどのユーザーはカスタムツールをますます必要とするようになるでしょう。しかしNewTekはLightWaveコミュニティが全体として満足できるようにツールセットを設計しました。このツールセットはLightWaveの核から派生したものであり、独自のツールが作成できます。それが**LScript**です。

LightWaveのスクリプト言語を学び始める前に、一つはっきりさせておかなければなりません。スクリプトはプログラム形式です。こんなことを聞くと芸術家体質の体中の骨はきしんでしまうことでしょう。でもLScriptでは通常のプログラミングから苦痛と悩みが取り去られていると保証します。CやC++言語での機能性の大半を保ちながらも、それに伴う頭痛の種は取り除きました。

簡単だとは言い切れません。ここからは今までアーティストにとっては無縁だった話題や手続きなどについてカバーしていきます。章を読み進み、順番どおりにサンプルを実行しながら一度に一つずつ段階を重ね全てを根気良く続けていけば、きっと楽しくなってくるはずです。LScriptの素晴らしいプログラミングの世界をぜひ楽しんでください。

この本を読むべきは人は誰？

LightWaveのスクリプト言語でプログラミング方法を学ぶためには、まずLightWaveがどのように動作しているのかについて理解しておかなければなりません。基本的なモデリングやアニメーションについて話題に上りますし、サーフェイスやポリゴンも関係してきます。たとえ他にアプリケーションを十分に理解しており、LightWaveを学び始めたばかりだとしても、LScriptを学び始める前にまずLightWaveの基本的な技術を習得しておいてください。

LScriptのツールはLightWaveにあるツールのサブセットとなっていますから、これから解説し実践していく内容の大半は、LightWaveとLScriptがどのようにして相互に対話しているのかということと関わります。

このことを踏まえておかないと、わけもなくテーブルに頭を打ち付けたくなるかもしれません。

このマニュアルはLScriptでプログラミングを学ぼうとしている人を意図して設計されています。主に初心者や趣味で使用する人を養成するのが目的です。もちろん専門家でも大歓迎ですが、専門家が望むよりもはるかにゆっくりとしたスピードで進めていくことになります。また専門家には必要とされる、より難しいテクニックについてここでは取り組みません。経験を積んだプログラマにはLScriptドキュメントやリリースノートを読むことをお勧めします。

ですからLightWaveの動作については良く理解しており、なおかつ未知なるプログラミング領域に足を踏み入れたい人に、最適なマニュアルといえるでしょう。

何を学ぶの？

プログラミング習得において最も大変な個所といえば、初めてのスクリプトを作る前にまず、プログラミングのあらゆる面について少しずつ学んでおかなければならない点にあります。プログラミングについてより理解を深め知識が増えてくるにつれて、既知の知識を広げていくことが出来るのです。ここでは学生時代に教師から教わったようにきちんとした基礎に基づいて教えていくという方式を取っていきます。学ぶべき基礎知識は非常に膨大ですが、その見返りは十分にあります。

スクリプトを学びやすくするために、いくつか実世界の例を用いて難しい概念を説明していきます。ある問題に対し異なった視点から見つめることで、日常の概念と関連付けられればと考えています。

マニュアルはいくつかのセクションに分かれています。

第1部 プログラミング概論（第1章～第5章）

コマンドや変数、コントロールの構造、プログラムの流れ、関数といったプログラミングの概念について紹介していきます。

第2部 例（第6章～第8章）

サンプルを使用してモデラーでのスクリプトの作成法、またモデラーとLScript相互の動作法について学んでいきます。

第3部 LScriptとレイアウト（第9章～第19章）

実例を挙げながらLScriptとレイアウトについての関係を解説していきます。

第4部 LScriptプログラミング環境（第20章～第24章）

LScript Editor、**Interface Designer**などを含むLScriptのプログラミング環境についてカバーしていきます。

全てを終えると何が起きる？

全て終了する頃には、夢中になってスクリプトを書くようになっているでしょう！ですが、このマニュアルを読み終えた時点でプログラミング教育が終了したわけではありません。このマニュアルは実際の教育である実践と経験についての入門書としてのみ提供されるものなのです。机に向かい様々なスクリプトを書くことで、独自のプログラミング形態を発見し、本当に必要とされる便利なスクリプトを作成出来るようになるでしょう。また幾千通りも誤った方法を取ってしまうこともあるでしょうが、この分野にはつきものなのです。

幸いなことに、プログラミングのための様々な方法をカバーした何千もの本やオンライン記事があります。LScriptの動作法において全てが適用可能であるとは限りませんが、一般的なプログラミング、また3Dに特化したプログラミングに適用できるテクニックを教えてくれます。また討議グループやLScript愛好者達を探してみてください。彼らはプログラミングに関する情報を持ち、初心者に喜んで手を貸してくれるでしょう。

第２章：変数

スクリプト実行時において、後々使用するために情報を保存しておく必要が出てくることでしょう。記憶しておく必要がある情報が、押されていたボタンの状態なのか、それともオブジェクトの位置なのか、いずれにせよ後でアクセスしたいデータを持っているとします。このような場合、**変数**を使えばスクリプト実行中に、このデータを一時的に保存することが出来ます。変数を使用せずに多目的のスクリプトを作ることはまず不可能です。そのため、変数がどのように動作するのかを理解することは、プログラミングの不可欠な要素となります。

ここに幾つか、スクリプト内にある変数の例を挙げてみましょう。

```
myVariable
control01
frameNumber
frame_step
```

これらの変数には何も入っていません。変数に対して何の値も割り当てていないからです。LScript により確保された保存領域を示しているに過ぎません。値を保存したい場合、この変数の名称をリファレンスとして、情報を記憶領域内に挿入します。

その言葉からも連想できるとおり、変数内の値は動的なものであり、スクリプト実行中に修正することが可能です。変数内にどの値を保存するのか、いつ値を変更するのかを完全にコントロールします。しかしエラーが発生した場合、責任はあなたにあります。値が変わる場合、値を修正するコードを書いたのはあなたなのですから。変数内の値が独力で変更することはありえません。

トラッキングとデバッグ

スクリプトを書いていくうちに、変数内の値が想定していた値とは全く異なる値になってしまう場合もあるかもしれません。これは変数の責任ではなく、コードが間違っています。コードのあるどこかの部分で、誤って値を変更しているのです。でも心配しないで下さい。よくある間違いなのです。こういうエラーを見つけ出し修正する方法を把握していることが、変数を扱う際に常に正しさを保つ鍵となります。値がおかしくなっている箇所を追跡するのは必要悪でもあり、プログラミング初心者にとっては予想される事態なのです。

バグを見つけ出し修正するのに長い時間がかかる場合もあります。この処理を**デバッグ**といいます。最初は時間もかかり非常にいらだたしい思いもするでしょう。でも時間がたてば、より経験を積んできて、間違いなく過ちが少なくなっていきます。

変数名

変数の名前付けにはいくつか規則があります。

1. 名称は唯一無二のものにして下さい。コマンドや文、関数、他の変数と同じ名称を持たせることは出来ません。スクリプト実行時に名称で混乱を生じさせないためです。
2. 名称にはスペースやシンボル（アンダースコア（ `_` ）を除く）を含んではいけません。
3. 名称内部に数字を挟むことは出来ますが、冒頭に数字をいれることは出来ません。

例：

・変数名 `cricket` は有効
・変数名 `cricket3` は有効
・変数名 `3cricket` は無効

大文字/小文字の区別

LScriptの要素は全て大文字/小文字を区別しますが、変数も同様です。つまり大文字と小文字の違いで異なる二つの変数名称をつけることが出来るのです。コードを書く際、このことをしっかりと頭に入れておいてください。大文字/小文字の区別による単純な間違いが重大なバグになりえるのです。以下の変数名称は全て異なる変数となります。

```
cricket
Cricket
CriCKet
Cricket
```

変数の宣言

宣言は変数に対し単に名称と値を設定します。通常、変数の宣言は名称の後に等記号（`=`）、その後に特定の値を書きこみます。以下の段落では変数の種類を示し、その使い方について説明していきます。それではまず非常に簡単な値、整数から始めていきましょう。

注意
変数タイプには高等な機能や使用法を持つものもありますが、ここでは記しません。最初から混乱させたくないからです。残りの変数タイプについてはサンプルコードで使用する場合に、たっぷりと説明します。

整数

整数はスクリプトで使用する最も一般的なデータタイプの一つです。このデータタイプは正、負どちらも 4,294,967,294 未満の値を表現します。中学校の数学を思いおこしてもらうと、整数とは小数点無しの値をさします。

有効な整数値：`0, 1, 1481, -253, 823523`

無効な整数値：`1.0, 3532.99, -25.38, 21.001, 4294967295`

例 宣言：

```
myVariable = 0;
myVariable = 1481;
```

注意
セミコロンは行の終端を表しており、プログラム内ではこのようにコードを表示します。例を示す必要もないのですが、実際にコードがどのような外観になるのかを見慣れておいて欲しいのです。

整数は非三次元数学的処理の大半で使用されるデータタイプです。整数はプログラミングにおいて広くは使用されないというわけではなく、非少数値全体が三次元数学的処理において無効になる部分が多いのです。整数値だけでアニメーターがアイテムを移動させたり回転させたりスケールさせることは非常にまれだからです。非少数値は数値変数タイプへと変換します。

数値

数値データは LScript で広く使用されるデータタイプの一つです。整数データタイプがカバーしきれない小数値を、この数値タイプがカバーしているからです。値の小数部分は小数点もしくは指数表記を使用して表されます。

有効な数値: `2.45, -1.5, 22.1, 9e-10`

無効な数値: `(none)`

例 宣言：

```
myVariable = 2.45;
myVariable = -1.5;
```

文字列

文字列は変数内に保存される言葉またはフレーズです。文字列の決まりごととして、変数用の値は二重引用符（"）で囲んでおかなければならず、また、全て一行に収まるようにしなければなりません。LScript は、引用符の外側の値は他の変数もしくはコマンド/関数であるとみなします。

有効な文字列値：`"apples", "oranges", "Hello!", "1", "true"`
無効な文字列値：`apples, oranges, hello!, 1, true`

例 宣言：

```
myVariable = "apples";
myVariable = "oranges";
```

ベクトル

ベクトルはカンマで区切った三つの数値データタイプの値を、小なり記号（<）と大なり記号（>）で囲み、一つの型として表します。このタイプは座標値や RGB 値を保存する場合に便利です。

注意
ベクトルはオブジェクトメッセージもサポートしており、この章の後の方で解説しています。メッセージのリスト全体については、リファレンスマニュアル（12 章：変数）を参照してください。

有効なベクトル値：`<1.9, -2.4, 0.23>, <124.1, -2.35, 1>`
無効なベクトル値：`<"one", 5.0, 102.4>`

例 宣言：

```
myVariable = <1.8, -2.5, 0.23>;
myVariable = <124.1, -2.35, 1>;
```

ブール値（ブーリアン）

最後のデータタイプは**ブール値**です。ブール値はバイナリと整数値双方を表します。ブール値として保存可能な値は`0`、`1`、`true` それに `false` の 4 個だけです。

有効なブール値：`0, 1, true,` それに `false`
無効なブール値：`4, -2`

例 宣言：

```
myVariable = true;
myVariable = false;
```

配列

配列は、解説する変数タイプのうち最も複雑なタイプかもしれません。配列は一つの変数名称のもとに多くのデータの断片を寄せ集めたものです。配列はどのタイプの変数でも持つことが出来ます。実際、LScript における配列はタイプが異なる複数の変数をあわせることさえ可能です。

例えば、一つの変数に信号機の色を保存してみましょう。

最初の色は`"red"`
次の色は`"yellow"`
3 番目の色は`"green"`

これを書くもう一つの方法が

```
1 "red"
2 "yellow"
3 "green"
```

色を表す文字列の前に、それがリストのどの位置にあるのかを表す数値を付け加えてみました。"2 番目の色は？"と問われたら、自然と"yellow"と答えるでしょう。これが配列の動作なのです。

文字列名称を用いて色を書いてきましたが、配列にはどのタイプを用いることも出来ますから、このように簡単に表すことも出来ます。

```
<255, 0, 0>
<255, 255, 0>
<0, 100, 0 >
```

標準の変数名に括弧（`[]`）と、インデックスと呼ばれる連続する整数値を用いることで、配列に値を保存したり取り出すことが出来るようになります。インデックスは正の整数値、もしくは正の整数値を含んだ変数でなくてはなりません。ゼロ、負の値、小数などは使用できません。信号機の例では変数名称は以下のようになります。

```
trafficColor[1]
trafficColor[2]
trafficColor[3]
```

有効な配列インデックス:

```
trafficColor[1]
studentNumber[114]
trafficColor[lightNumber]
```

無効な配列インデックス:

```
trafficColor[0]
accountNumber[-4]
trafficColor[1.5]
pointNumber[<9,1.0, 12>]
```

例 宣言：

```
trafficColor[1] = "red";
trafficColor[2] = "yellow";
trafficColor[3] = "green";
```

配列のある部分をより細かく見てみると、インデックスとして変数を使用していますね。これを利用して少し複雑に設定してみましょう。

```
trafficColor[1] = "red";
trafficColor[2] = "yellow";
trafficColor[3] = "green";
lightNumber = 1;
```

`trafficColor[lightNumber]`には文字列値"red"が入っています。

少し説明を加えましょう。`lightNumber`は整数値1を持つ変数であることを宣言していますから、配列へのアクセス時にLScriptはこの値を読み込みます。つまりこの行は下記のように読み込まれていることになります。

```
trafficColor[1];
```

宣言リストから`trafficColor[1]`の値を探してみると、文字列"`red`"が入っているのがわかりますね。

今はまだ少し混乱してしまうかもしれませんが、このメソッドはプログラミングにおいて一般的な実践法であり、これについてはさらに解説する例が他にももっとたくさんあります。今の時点では基本コンセプトを理解しておくようにしましょう。

文字演算

変数を扱う一方で、少し**文字演算**についても勉強してみましょう。この宣言グループは何が行われると思いますか？

```
text1 = "Hello";
text2 = " World!";
text3 = text1 + text2;
```

`text3`の結果は"`Hello World!`"となります。`text1`は文字列 "`Hello`"、`text2`は文字列 "`World!`"と宣言しているため、`text3`はこの値 "Hello"+" World!"をともに取得し、"`Hello World!`"となるのです。これが文字演算です。

変数演算

文字演算と同様、**変数演算**も数学を使って他の変数を新たに作成したり、既存の値を修正したりします。ここにコードの一部を紹介します。

```
val1 = 1000;
val2 = 10;
val3 = 5;
val4 = val1/val2 * val3;
```

`val4`の結果は`500`となります。1000 を 10 で割り導き出された結果 100 に対し 5 を掛け合わせます。

```
val[1] = 1;
val[2] = 10;
val[3] = val[1] + val[2];
```

`val[3]`の結果は`11`となります。変数演算はプログラミングにおいて頻繁に使用されます。幸運なことにたくさんの実習例があります。

定数

忘れることの出来ない重要な変数タイプがもう一つあります。**定数**は他のタイプとはかなり異なりますので、この章の最後で解説することにしました。定数はかなり違いがあるものの、非常に簡単なタイプです。

定数は名前どおり一定の値として処理されます。値を変更することは出来ません。定数はLScriptが内部で使用する変数を定義していますので、変更するのは不可能なのです。例えばライト種を見てください。

ライト種：

平行ライトはライトタイプ`0`
点ライトはライトタイプ`1`
スポットライトはライトタイプ`2`

これらの値を事前定義することで覚えやすく、また使いやすくしています。

```
LWLIGHT_DISTANT = 0
LWLIGHT_POINT = 1
LWLIGHT_SPOT = 2
```

これでコード内に変数`LWLIGHT_DISTANT`を書き込めるようになります。平行ライトはタイプの値`0`をもっていると覚えるよりもはるかに簡単ですね！

定数は覚えやすく読み込むのが簡単ですので、LightWaveはこれらの値を保護しています。前述したとおり、値は変更することは出来ません。

注意
定数は他の変数とは著しく異なっており、また厳しい制限を持っているため、大文字で宣言されています。定数値を大文字化することでコード内部において即座に認識出来やすいようにしています。

他にも定数の利点として、LScriptにおける内部変更に対しコードを保護する役目もあります。仮にNewTekのプログラマが、コードの修正が必要な点を見つけ内部変数をいくつか修正した場合でも、`LIGHT_DISTANT`はスクリプトに対し常に同じものを指すようにしています。

第 3 章：関数

関数とはごく小規模でありながら、反復的なプログラミングのタスク（処理）を扱う自己内蔵型のコードセクションです。関数はメインプログラムとは別に設定し、独自の変数やメインスクリプトとは独立したコードを含んでいます。プログラムは関数で使用する特定のデータを関数側へと送信することで関数を使用することが出来ます。この処理を"関数呼び出し"といいます。

一度関数を呼び出すと、そのデータとコード行が通常のスクリプトのように処理されます。関数は呼び出し側のコードからは完全に切り離されます。特別にコードに送信されたデータだけがメインスクリプトから使用可能です。関数には"フォーカス"があります。

関数内のコードが実行終了すると、処理した値を、関数を呼び出したスクリプトへと送信することが可能です。処理した値が関数内に残されたままであれば、これらの値は何の意味もなくなってしまいます。その後フォーカスは関数を呼び出したメインスクリプトの場所へと戻り、中断されていた箇所から処理を続行します。

LScript 内部にある関数だけに限りません。必要に応じてカスタマイズされた独自の**ユーザー定義関数（UDF）**を簡単に作成することが出来るのです。実際、使用頻度の高い関数を幾つか一緒に集めてグループ化し、たくさんのスクリプトから呼び出せる共有可能なツールライブラリを作成することも出来ます。スクリプトを組んでいるとすぐに、非常に強力な関数群をいくつも持てるのだとわかってくるでしょう。

現実世界からの例

以下、実世界からの例を使用して関数がどのように動作するのかをよりわかりやすく説明していきます。

スクリプトを大きな自動車製造工場だと考えてみてください。工場は月に何千台もの車を生産しなければならないのですが、同じ工場で車の全部品を作ることはしません。例えば、タイヤは別の国で作られ、ガラスはまた別の国、点火プラグはまた別のという形になります。これらの部品全てが集まったときに、車を作ることが出来るのです。

この考えをもっと細かくみてみましょう。車製造工場は車を組み立てはじめると、タイヤが必要であることがわかってきます。タイヤ会社を呼び出し、価格10,000ドルで1000個のタイヤを注文します。タイヤ会社はお金を受け取って、タイヤを製造し、それからタイヤを車工場へ納品します。タイヤを製造するということが、車組み立てという大きな処理の中の一つの関数です。この考えをプログラムと関数に置き換えてみましょう。

プログラム：“車の製造”

```
フレームを製作
エンジンを製作
タイヤ会社から 10,000 ドルで 1000 個のタイヤを買う
...他のアクション...
車が完成

関数: "タイヤを買う"
10,000 ドルで 1000 個のタイヤを買う
タイヤを製造するための原料を買う
タイヤを集める
1000 個のタイヤを工場へ送る
```

関数はLScript内にある関数とは違ったように見えます（上記は"擬似コード"と呼ばれます）が、必要なコンポーネントは持っています。より詳細に見ていきましょう。

まずメインのプログラムでは、"車の製造" はフレームを製作し始めてから、エンジンを製作し、その後で10,000ドルで1000個のタイヤを買う必要が出てきました。この時点でプログラムは "タイヤを買う" と言う名前の関数があることを理解しており、これを呼び出します。タイヤ会社にとってタイヤを納品するためには、何個のタイヤを送るのか、いくら支払いを受け取るのかを把握しておく必要があります。

関数のこの部分を**引数**といいます。引数は、仕事（この場合はタイヤの製造）を完了するために関数が必要とするデータです。タイヤ会社は適切な情報を受け取り、タイヤを製造してから車工場へとタイヤを送ります。メインプログラム"車の製造"では、"タイヤを買う" 関数を呼び出し、情報（1000個のタイヤと10,000ドル）を送ります。このデータ送信の処理を、関数へ引数を渡すといいます。

関数の最後の部分は**戻り値**といいます。一旦関数の処理が終われば、メインプログラムへと返すことを同意したものが戻り値となり、単にそこに置くだけでよいのです。この場合、戻り値はタイヤです。

プログラムと関数の作成

今回の例は実際の LScript 関数ではありそうもない話なのですが、どのように関数が動くのかをはっきりと示しています。実際の LScript 用語を使わずに、この例を LScript 用にフォーマットしてみましょう。

メインプログラム：車の製造

```
main
{
  buildFrame();
  buildEngine();
  tires = getTires(1000,10000);
  ...車の完成...
}
```

そして関数部分：

```
getTires: numberOfTires, payment
{
  buyMaterials();
  assembleTires();
  ...車製造のコード...
  return(tires);
}
```

関数のパーツ

上記の例は正確ではありませんが、その前の例とは異なる点がたくさんあります。この相違点について見ていきましょう。

最初の違いは'`main`'という語句にあります。最初のスクリプトはモデラー用のLScriptでしたので、モデラースクリプト独自のルールに焦点を当てています。言葉が示す通り、`main()`関数はLScriptがコードを実行を開始し始める箇所です。どのモデラーLScriptも`main()`を持たねばなりません。

ボディマーカー

次に、この波括弧は一体何でしょうか？これは**ボディマーカー**というもので、LScriptにコードの始端（`{`）と終端（`}`）を表しています。スクリプト内にある他のコード全てを除去すると、`main()`関数はこのようになります。

```
main
{
}
```

このボディマーカー内部にあるコード行、またはボディマーカーでくくられている行は`main()`関数に属しています。関数名称の後にくる開括弧（`{`）は、閉括弧（`}`）に到達するまでの存在する全てのコードを所有していると関数に対して示しているからです。関数用の閉括弧が偶然にも残ってしまった場合、LScriptは`main()`関数がどこで終わるのかがわからなくなってしまい、嫌なエラーを引き起こしてしまいます。

注意
二つのマーカーの間にあるコードもコードブロックとして参照されます。

セミコロン

この例ではコードは`main()`関数に属しています。コードの各行の終端にセミコロン（`;`）がついているのに注目してください。このセミコロンは各文の終端を示しています。文の終端にあるピリオドと同じような動作をするのです。

テキストエディタではENTERキーを押さないと新規行には移れませんよね。LScriptはセミコロン（`;`）に到達するまでコード行の終了だとはみなされないのです。不適当な句読点は、エラーの原因となります。例えば次のコードを見てください。

```
main
{
  buildFrame();
  buildEngine();
  tires = getTires(1000,10000);
}
```

これは以下と同義です：

```
main
{
  buildFrame();
  buildEngine();
  tires
  =
  getTires
  (1000,10000);
}
```

大変不恰好でこのように書くべきではないのですが、コードとしては正当です。スクリプト開発者はコードの終端にセミコロン（`;`）を使用する限り、どのように整列させフォーマットしようとも構わないのです。最後に、以下のコードもまた同義であり正当なコードです。

```
main
{
  buildFrame(); buildEngine(); tires = getTires(1000,10000);
}
```

注意
最もよくありがちな誤りは、セミコロンを忘れてしまうことです。ここはイージーミスとなる箇所です。

例へと戻りましょう・

```
tires = getTires(1000,10000);
```

さて、等記号の後のデータのみを見ていくことにしましょう。

```
getTires(1000,10000);
```

LScriptはコマンドや文のデータベースを通して`getTires()`という語句を内部的に探しにいきます。`getTires()`という名称のコマンドや命令文が見つからなかった場合は、**ユーザー定義関数（UDF）**であると仮定し、スクリプト全体を検索して関数を探しに行きます。`getTires()`関数は宣言していますので、LScriptは遠くまで探しに行く必要はありません。しかし関数を見つけられなかった場合にはエラーが発生します。

二個のデータ、1000と10000（コンマで区切られている）が`getTires()`関数へと渡されます。これは丸括弧内に含まれる二つの値を示します。LScriptでは関数内で定義したのと同数の引数を渡さなくてはなりません。`getTires()`関数の中では、`numberOfTires`と`payment`という二つの引数を宣言しました。

思い出してください：

```
getTires: numberOfTires, payment
{
  ...
}
```

この宣言文において、コロン（`:`）の後に続くものが、関数で必要となる引数です。`getTires()`関数本体の中のどこかで、引数`numberOfTires`と`payment`を使用して、この処理を実行しています。

`getTires()`関数の最終部分はリターン文です。

```
...
return(tires);
...
```

リターン文は`getTires()`関数に、`main()`関数にデータを送り戻すように指示します。この場合、タイヤの個数を返します。戻り値は括弧の中に記述します。後で戻り値の取得法について学んでいきましょう（ヒント：等記号の左側に変数があるということを覚えておいてください）。これで戻されたデータが`main()`関数で使用出来るようになります。

関数への引数渡し

それではモデラー上で実際に動作する例を見ていきます。このサンプルはコードの断片でたいした処理は行いませんが、関数呼び出しの概念をよりよく理解できるように解説しています。

```
main
{
  a = 10;
  b = 15;
  c = addAB(a, b);
  info(c);
}

addAB: a, b
{
  tmpVal = a + b;
  return(tmpVal);
}
```

ここには二つの関数、`main()`と`addAB()`があります。まず最初に、`main()`関数はそれぞれ整数値`10`と`15`を持つ変数`a`と`b`を宣言します。この変数`a`と`b`を渡すことで、フォーカスは`main()`関数から`addAB()`関数へと移動します。

`addAB()`では、変数`tmpVal`を宣言しています。ここにはa+bの値、すなわち10+15の値を入れます。このため変数`tmpVal`の値は`25`となります。

次の行で

```
return(tmpVal);
```

関数に対し変数の値（25）を`main()`関数へと送り返すように指示します。メイン関数では変数`c`でこの値を受け取ります。これを検証するため、`info()`コマンドで画面上に変数`c`の値を表示するようにします。勿論`25`と表示されるはずです。

新しい関数へ値を送り、その関数で足し算を行い、`main()`関数へと値を戻しました。これが全ての関数の基本的な手続きなのです。

第 4 章：条件文

理想の世界では、スクリプトは常にシーン内またはモデルに対し何が行われているのかを正確に把握していることでしょう。ユーザーが加える修正や調整も認識し、偶然出くわしたタスクの処理法についても既に把握しているかもしれませんね。一つのシーンやオブジェクトに対してだけ実行するような一度限りのスクリプトを作るのであれば、状況を全て把握した上でプログラムを組むことが可能です。しかし勿論このようなやり方では、多様なシーンやオブジェクトに対して動作する洗練されたスクリプトは期待出来ません。

オブジェクトやシーンにいくつかの条件が存在している場合、その条件を比較し、状況によってスクリプトに異なる動作をさせることがあります。この判定を**条件文**といいます。

例えば、モデラースクリプトを使用して二つの選択したポイントを統合するとしましょう。統合する前に、ユーザーがポイントを二つ選択しているかどうか確認したいですよね。もしポイントが選択されていれば、スクリプトは二つのポイントを統合できます。しかし選択されていない場合には、まず統合する二つのポイントを選択するよう、ユーザーに通知する必要があります。

このような状況はプログラミングの最中には頻繁に起こるもので、様々な文が条件に基づいて処理されます。条件文の役割は他の文を検証することにあります。文が正しければ（`true`）、LScript はコードのある部分を実行し、正しくなければ（`false`）無視します。これらの条件文で最も一般的なものとして、`if-then` 文があります。

If-then 条件文

`if-then` 条件文は式が正しいかどうかをチェックします。式が正しければ（`true`）スクリプトは条件文のすぐ後にあるボディマーカーに囲まれたコードブロックを実行します。式が正しくない（`false`）場合にはコードブロックは無視され、閉括弧（`}`）の後に続く次のコード行を実行します。

構文：

```
if(…式…)
{
  …  式が true の場合に実行されるコード…
}
```

例：

```
if(pointsSelected == 2)
{
  mergePoints();
}
…次の文…
```

この例を一通り眺めてください。`if-then` 条件文は `pointsSelected` 変数の値が2であるかどうかを評価しています。2の場合には `mergePoints()` コマンドが実行されます。`pointsSelected` の値が2ではない場合、`mergePoitns()` コマンドは飛ばされ、次の行が実行されます。

この例では、コードブロックは実行用のコードを1行（`mergePoints` 文）だけ含んでいます。このような場合、`if-then` 条件文は実際には以下のコードで示す通り、ボディマーカー無しで書くことが出来ます。

```
if(pointsSelected == 2)
  mergePoints();
…次の文…
```

式が正しい場合に単一行のコードだけが実行されるのであれば、このようにコードを書くことが可能なのです。

演算子

if-then 条件文の中に pointsSelected==2 という式があるのがわかりますね。これは誤植ではなく、この式の中で二つの等記号は**演算子**となっています。LScript 内で使用可能な様々な演算子について見ていきましょう。

以下に基本的な演算子を記します。

演算子	言葉で表すと
`=i`	値 `i` の代入
`!`	非
`==`	等しい
`>`	大なり
`<`	小なり
`>=`	大なり　もしくは　等しい
`<=`	小なり　もしくは　等しい
`!=`	等しくない

前述の `if-then` 条件文を使用して、各演算子の適切な使用法を解説していきます。

```
a = b          a に値 b を代入
if(a == b) a が b と等しい場合
if(a != b) a が b と等しくない場合
if(a > b)    a が b より大きい場合
if(a >= b) a が b より大きい、または等しい場合
if(a < b)    a が b より小さい場合
if(a <= b) a が b より小さい、または等しい場合
```

If-then-else 文

`if-then-else` 文は、先ほどの `if-then` 文とほとんど同じですが、一つだけ大きな違いがあります。式が正しくない（`false`）場合にコードを完全に飛ばすのではなく、`else` ブロックへと移り、そこにあるコードを実行するのです。

構文：

```
if(…式…)
{
  …式が true の場合に実行されるコード…
}
else
{
  …式が false の場合に実行されるコード…
}
…  次の文…
```

`else` 文は `true` 文の閉括弧（}）の後に直接続いており、その後にはセミコロンはありません。これは `else` 文も `if-then` 文の一部だからです。

例：

```
if(pointsSelected == 2)
{
  mergePoints();
}
else
{
  error( "2 or more points must be selected." );
}
```

前述の文は以下のように書くことも可能です。

```
if(pointsSelected == 2)
  mergePoints();
else
  error( "2 or more points must be selected." );
```

実際にはこちらの記述法が前述の形式よりも正しいのです。条件のボディマーカーは文が単一行のコードを使用している場合には不必要となります。見てわかるとおり、このフォーマットでは長いスクリプトよりも空間を節約できるのです。

`error()`文は単に画面上にエラーメッセージを表示し、スクリプトの実行を中断します。

ブール値と if-then-else

条件式の中で `if-then-else` 文とブール値の変数を組み合わせると、面白い状況が生まれます。変数のブール値が `true` の場合には条件式が `true` であるとみなされます。同様に変数が `false` の場合には、式が `false` であるとみなされます。

例 1 :

```
objSaved = false;
if(objSaved)
  info( "Object was saved." );
else
  saveObject();
```

… 次の文…

例 2 :

```
done = false;
if(!done)
  info( "The procedure IS NOT done." );
else
  info( "The procedure IS done." );
```

…次の文…

Switch 条件文

switch()条件文は非常に強力な機能です。if-then条件と同様、式を解くのですが、その結果をリストされている値と比較します。値がリストアイテムと一致したら、case文の後ろに直接書かれているコードを実行します。breakコマンドはswitch()文を全てそのまま残します。

構文：

```
switch(…式…)
{
  case /値1/:
    ...文...
    break;

  case /値2/:
    ...文...
    break;

  default:
    ...文...
    break;
}
… 次の文…
```

例：

```
valueA = 3;
valueB = 1;
switch(valueA - valueB)
{
  case 1:
    info( "A - B = 1" );
    break;

  case 2:
    info( "A - B = 2" );
    break;

  default:
```

```
        info( "No Matches found!" );
        break;
    }
    …次の文…
```

上記例では、変数 `valueA` と `valueB` には整数値が与えられます。式の結果 `valueA-valueB`（この値は2）が `switch()` コマンドへと渡されます。`switch` コマンドはこの値を `case` の値リストと比較します。一致すれば `case` コマンドにつけられているコードが実行されます。

最初の `case` 文の値は `1` です。式の値はこの値と一致していないので、LScript はコードを飛ばして次の `case` 文と比較します。次の `case` 値は式の値と一致するので、LScript は `info()` コマンドを実行します。このコマンドは `info()` ボックスの中に "`A - B = 2`"という文字を表示します。次の文は `break` 文です。この `break` 文により、LScript は `switch()` 文の閉マーカー（`}`）を直接実行するように指示されます。

`switch()` 文を使用すれば、数種類の異なる値に対し式を比較する場合などに `if-then` や `if-then-else` 文をいくつも使用することなく実行できます。

第 5 章：ループ文

LScript の最も簡単なものとして、レイアウトコマンド、またはモデラーコマンドがあります。これらのコマンドはスクリプトの終端まで一度に一行ずつを順に実行していきます。

しかし x 回もコマンド文を繰り返したいと思うでしょうか？何度も何度も正確に文を入力することは可能かもしれません。でもそれは馬鹿げたことですし、非常に非効率的です。そればかりではなく、何回繰り返せばよいのか把握出来るでしょうか？値はユーザー入力値に基づく値かもしれませんし、他のコマンドから設定される変数かもしれません。ループ文を使用せずにこれらの状況を処理する LScript を作成するのは、まず不可能です。

ループ文は、条件が `true` である限りコードブロックを実行します。コードブロック内部の最初の行が実行され、閉括弧（`}`）に遭遇した時点で、開括弧（`{`）文まで戻りコードを繰り返します。式が `false` になるまで処理を繰り返した後、次の文へと移動します。最も一般的な文は `for-loop` 文です。

for-loop 文

for-loop 文は複雑ではありますが、ループ文の中では最も一般的なものです。この文はループを繰り返します。つまりループ内のコードブロックが処理された時点で、変数を変更し、式内の値を比較してもう一度処理を反復するかを決定します。式の値が false になるまで処理を繰り返します。

構文：

```
for(変数宣言;  …式…;  反復)
{
  …ループする文…
}
```

例：

```
for(i = 0; i < 10; i++)
{
  info(i);
}
```

LScript が文を実行している時、4 つの事柄が起きています。一つは変数が 0 で宣言されます。

```
for(i = 0; i < 10; i++)
```

次にLScriptは、二番目の文で定義されている式が`true`か否かをチェックします。この場合、LScriptは変数`i`が10よりも小さいかどうかをチェックします（現在`i`は`0`であり、0は10よりも小さな値ですから、式は`true`となります）。

```
for(i = 0; i < 10; i++)
```

式が`true`の場合、LScriptは次に続くコードブロック内の文を直接実行します。実行するように設定しておいた文は、簡単な`info()`ボックスだけです。

```
for(i = 0; i < 10; i++)
{
  info(i);
}
```

しかしこの`info()`は表示テキストではありません。この`info()`文は、反復変数である変数`i`の現在の値を表示しています。これにより実際に変数`i`がどのように変化しているのかがわかるようになります。

`for`文はまた終わっていません。ブロック内にある全ての文を実行したら反復変数が修正されます。どのように変数が修正されるかは、`for`文の3番目の部分に定義されています。

```
for(i = 0; i < 10; i++)
```

この場合`i++`が使用されていますが、これは`i=i+1`を手っ取り早く表しています。この文はコードがループされるたびに変数`i`がどのように変更するのかを定義しています。ですから`i`は`0`の値から始まり、コードブロック内の文がすべて実行された後には値が`1`、`2`、`3`と変化していきます。つまり`i`の値が10より小さい間は、`info()`ボックスで表示し続けるということです。

見てわかるとおり、`for`文を使用すると、ループが可能なだけではなくカウントも出来ます。配列内の値を扱う場合にもこれはとても便利です。以下の配列を見てください。

```
objects[1] = "Null";
objects[2] = "cow.lwo";
objects[3] = "ball.lwo";
```

このコードは何をしているのか想像してみてください。

```
for(i = 1; i <= 3; i++)
{
  info(objects[i]);
}
```

`for`文の反復が`false`になったとき、LScriptは次のコードへと移動します。

while loop 文

`for` 文は既知の量数だけ文を反復したい場合には最適です。しかしスクリプトが広範囲の状況においても使用出来るように設計されている場合など、ループを中断する時期を必ず把握できるとは限りません。スクリプトから何らかのアクションがあるまでループを実行させておきたいというような場合、どうすればループを中断することが出来るのでしょうか？そんな場合には、`while` ループを使うとうまくいきます。

注意
このコードは厄介な側面も持っています。while ループを使うプログラミングでは、かなり慎重に、集中して記述していかないと、恐ろしい事態に見舞われる可能性があります。

構文：

```
while(…式…)
{
  …文…
}
…次の文…
```

`while` ループは `if-then` 文と似ているように見えますね。これが基本的な `while` 文の処理です。ただしコードの次の行に移動する代わりに、式が `true` ではない限り `while` ループの実行文は反復します。

注意
このやり方を見てみると、while ループは前述の for 文にも似ていますね。

例：

```
done = false;
while(!done)
{
  if(x ==5)
    done = true;
  else
    x++;
}
```

`while` ループを入力する前にブール値を `false` に宣言しておかなくてはなりません。この変数は `while` ループが反復を続行するか否かをチェックするためのものです。

```
done = false;
```

この変数には、while 文につけられているコードブロックに対し、ライトスイッチのような役割だけが与えられています。変数 done が false であればループが実行され、true であればループは中断します。ですから while のブロックコードを一度でも実行させるためには、変数 done には初期値として false が設定されていなくてはなりません。

まず while() 文は、ブール値の変数が true ではない値（つまり false）かをチェックします。そうであれば設定されているコードを実行します。

```
while(!done)
{
  if(x ==5)
  ...
```

if-then 文は整数変数値 x が 5 と等しいかどうかをチェックします。この場合、変数 x には 0 の値が入っています。ですから if-then 文の else ブロックが実行されます。else ブロックはこの while() ループ用の反復処理です。変数 x は 1 となり、ループが繰り返されます。

最終的に x は値が 5 となり、if-then 文は true となります。ブール変数 done には true が代入されます。

```
done = true;
```

while() の式は done が false であるかをチェックし、ループは終了します。それからプログラムのフォーカスは次に設定されている文へと移ります。

では、どこが危険な箇所なんでしょうか？この例をしっかりと見て、どこが悪いのかを見つけてみましょう。

```
done = false;
while(!done)
{
  if(x ==5)
    done = true;
}
```

これは**無限ループ**と呼ばれるものです。while() コマンドはブール値 done の値が false か否かをチェックしますが、while() 文に対するコードブロックでは反復が行われていません。x の値は変化することがなく、決して 5 の値になることがないため、done は永遠に true にはならないのです。このため、スクリプトも終了することが出来ません。無限ループに陥ったときには手動でレイアウトまたはモデラーを強制終了し、LightWave を再起動させなくてはなりません。

注意
for 文の中でも、無限ループが作成されることがあります。

`for` または `while()` ループで作業する際には、かなり慎重に行ってください。スクリプトが終了できるように変数が反復処理を行っているか、値を変更しているのかなどを確かめてください。スクリプトがハングしてLightWaveを強制終了し、これまでの全ての作業を失わなければならない破目に陥ったときには、ユーザーは本当にうんざりするものです。

第 6 章：モデラー LScript 概論

ここまでは実際の LightWave のリファレンスや三次元アニメーションまたは LScript を使わずにコンピュータプログラミングの基礎を学んできました。さてここからは、学んできた知識をお馴染みの概念と理論、つまり三次元モデリングやアニメーションに適用していきましょう。

このセクション（第 6 章～第 8 章）では、前セクションでカバーした理論と実例を使用しながら、モデラースクリプトの作成方法を見ていきます。つまり、既に以下の概念については基本的な理解がなされているものと仮定します。

・変数

・条件式

・関数

よくわからなくなってきたら、もう一度時間を取ってこれらの章を読み返すようにして下さい。上記の概念について、よく理解しておく必要があるからです。この時点でプログラマー上級者になっているとは期待していませんが、この研究を続けていきながら前回の資料に戻るようなことはしたくありません。その代わり、これらのプログラミング概念を基に補強を重ね、既に学んだ知識を広げていきましょう。

思い出してください。LScript はプログラミング言語であり、他の言語と同様、基本から始めていかなければならないのです。名詞や動詞については学びましたが、今からは文や節をどのように組み立てていくのかを学んでいきましょう。最終的には短い物語が作れるようになり、ついにはベストセラーも夢ではありません。でもピューリッツァー賞を取る前にまず、第 2 段階を生き残らねばなりません！

このセクションでは、今までとは違ったアプローチで教えていきます。前セクションで行ったように概論や理論を検討することで学んでゆくのではなく、解剖実習と呼ばれる形式をとりましょう。解剖実習では、コードを行ごとに解析することでプログラミング法を学び、どのようにスクリプトの作業に影響を与えているのかを学んでいきます。ここから先のプログラミングはこの方法で解説していきますので、慣れていってくださいね。

このセクションでの目標は、モデラー用LScriptの基本を授けるということにあります。モデラー用とレイアウト用のプログラミング言語は似てはいますが、構造とプログラムの流れが根本的に異なります。レイアウト実行には必要となるプログラミングオーバーヘッドの大半が、モデラーのスクリプトにはありません。ただし、モデラースクリプトで技術的に必要とすることが少ないからといって、強力さに欠けるということではありません。

モデラーをLightWaveという複雑なマシンの中の巨大な回転歯車だと考えてみてください。実際スクリプトは一個の歯車を持つ小さなマシンで巨大なモデラー歯車にかみ合わせようとしています。ご想像どおり、既に回転しているギアに対して小さなギアをかみ合わせようとするのは容易な仕事ではありません。しかし全てが正しく行われれば二つのギアは一致してともに回転し、モデラーのギアはより小さなLScriptマシンが実行するための力を与えてくれるようになります。

レイアウトで実行するスクリプトは、それよりちょっと複雑です。モデラーのように一個のギアをかみ合わせるのではなく、スクリプトのギアは半ダースものレイアウトのギアに組み合わせなくてはなりません。注意が必要ですが、このマニュアルのレイアウトの章を読めばわかるとおり、絶対に大丈夫です。レイアウトと格闘する前に、まずモデラーのLScriptからはじめるのが良いでしょう。

LScriptプログラミング言語はモデラー内にあるコマンドや関数に完全にアクセスできるようにしてあります。LightWaveに付属している利用可能な多くの強力なスクリプトからも、この可能性をおぼろげながらも感じ取ることが出来るでしょう。モデラーで出来ることはLScriptでも実現可能なのです。

つまりモデラーのセクションとの対話は他よりもはるかに簡単なのです。このマニュアルの目標である「スクリプトを組み始めるのに十分な知識を授ける」ために、モデラー関数のほんの少しのサブセットをカバーしていきます。それらサブセットをどのように使用すればよいのかを示し、残りの部分を独力で学べるような知識を与えることが目標なのです。

では始めましょう！

第 7 章：モデラースクリプトの基礎

では手始めに、モデラー用の簡単なLScriptを作ってみましょう。このコードの断片で、最初のスクリプト作成処理とモデラーへどのようにしてスクリプトを入れ込むのかなどについて解説していきます。それ以上のことは行いません（どこかで必ず始めなければなりませんけどね）。

この処理ではスクリプト記述に使用可能な2～3のツールについて注釈を入れていきます。これらのツールの一つが **LScript Editor** と呼ばれるNewTekから提供されているカスタムメイドのエディターです。LightWaveのProgramsディレクトリの中に **LSED.exe** というエディタープログラムがあります。LScript Editorはスクリプトの記述とデバッグを行うプログラミング環境です。このマニュアルの章全体がLScriptプログラミング環境を使用することに捧げられていますので、ここで紹介されたスクリプトを自由に拡張していってください。

Hello World

今までにプログラミング言語を習得しようとした経験があるなら、最初に記述するプログラムの一つに"**Hello World!**"というものがあるのをご存知でしょう。これはスクリプトの使い方、またこの場合モデラーでの実行の仕方を解説することだけを目的としたプログラムです。

では始めましょう。

1. **LScript Editor** を開きます。
2. **File > New** を選択して新規スクリプトを作成します。インターフェイスの大部分で新規LScriptファイルを表示します。ここにコードを打ち込みます。
3. 以下のコードをワークスペースに入力してください。

```
main
{
  info( "Hello World" );
}
```

4. **File > Save As...** を選択してスクリプトを"hello.ls"という名称で保存します。忘れずにファイルを保存してください！綺麗にまとめるためスクリプト用の新規フォルダを作成するようにして下さい。そうすれば手早く簡単にスクリプトを保存したり探し出せるようになります。

注意
初期段階のスクリプトでは、全てLScriptのファイルであることを示す拡張子(.ls)がつきます。

5 モデラーで **Construct（構造）> LScript** を選択し、"hello.ls"スクリプトを検索して下さい。スクリプトを選択するとモデラーはスクリプトを実行します。コードが全て正しく入力されていれば、info ボックスに"`Hello World!`"という文字が表示されているはずですね。エラーメッセージが出てきたならエディターに戻り、コードをチェックしてみてください。

注意
info や error メッセージの表示箇所は警告レベル設定によります。初心者レベルであればメッセージは画面中央にパネル形式で表示されます。上級者レベルであればメッセージはモデラーのツールチップセクションに表示されます。

これで終了です！今やあなたはプログラマなのです！

では、今作成したスクリプトをもっと詳しく見ていくことにしましょう。

main()関数

モデラースクリプトには全てこの`main()`関数が存在します。名前が示す通り、この関数がスクリプトのメインの関数となります。モデラーがスクリプトを読み込むと、まずLScript は即座に`main()`関数を検索しに行きます。それからボディマーカー内にあるコードを全て実行します。実際に以下のスクリプトを動かして見ましょう。

```
main
{
}
```

このスクリプトは何の動作も起こしませんが、このわずかなコードだけがモデラーでのスクリプト実行に必要となるのです。

関数の章で学んだとおり、関数で変数を使用出来るように引数として渡すことが出来ます。ですが`main()`関数はこの規則には当てはまりません。引数として変数を渡すことは出来ないのです。引数を渡そうとするとエラーメッセージが出てきます。

ボディマーカー内部には1行だけコードが記述されています。

```
info( "Hello World!" );
```

見てわかるとおり、`info()`関数を使用してスクリーン上に様々なメッセージを送信します。通常は、ユーザーに対して必要な処理や警告のメッセージなどを伝えます。この関数を使い、スクリーンに送りたいテキストや値を括弧内に挿入することで、メッセージを表示します。文字メッセージや文字メッセージを含んだ変数、もしくは直接値を使用することも可能です。実際に何が起きているのかをよりよく理解するために、ここではプログラミングメッセージを使用しましょう。

"`Hello World!`"文字列が`info()`関数へ渡されました。

しかし、この行に以下のように入力してみると

```
info(Hello World!);
```

LScript から以下のようなエラーメッセージが表示されてしまうでしょう。

```
"In main, line 3, found 'W', expecting ")" or "~
(Line: info(Hello World!);)"
```

このメッセージはかなり説明的な記述になっていますね。このメッセージでは 3 行目の文字"W"の個所でエラーが発生し、ここには文字")"または"~"が来ると想定されていましたと伝えています。基本的に二重引用符がない場合、info()関数は変数が送信されたのだと想定します。これは全く正しいのですが、変数の間にスペースが入っていると正しくありません。そうなんです、この習性をすっかり見落としていました。変数を送っているつもりなど全くなかったのです。しかし簡単なメッセージでもどれほど LScript を迷わせてしまうかがわかりましたね。このようなエラーはこれからも起きるでしょうし、先に進むことにしましょう。

この変数の考えを続けるため、以下のように入力してください。

```
info(HelloWorld);
```

スクリプトは、今度はエラーメッセージ無しに実行します。しかし実行しても変数 HelloWorld には何も入っていないということを示す"(nil)"という値が表示されます。

ではどのようにして HelloWorld 変数を取得すればよいのでしょうか？次のコードを試してみましょう。

```
main
{
  HelloWorld = "Hello World!";
  info(HelloWorld);
}
```

3 行目で HelloWorld 変数が文字列値"Hello World!"を持つ変数として定義しています。従ってこの変数 info()関数へ渡したときには、info()はこの値"Hello World!"を表示します。

文字列を定義するためには引用符("")を忘れずにつけるようにして下さい。この括弧をつけないと、LScript は変数だとみなしてしまいます。

もう少し見栄えの良いプログラムにしてみましょう。

```
main
{
  text = "Hello World!";
  info(text);
}
```

変数名をより一般的な名称にすることで、より広い範囲で値を記述することが可能です。今おこなった修正は、コードをよりよく見せるためのものです。変数名には何をつけても構いません。以下のような名称をつけることも可能です。

```
tfaGxCw = "Hello World!";
```

名称は事実上有効ではありますが、ちっとも説明的ではありませんね。他の人があなたのコードを見なくてはならなくなった場合、もっとまずいことにあなた自身が何年か後にこのコードを見る必要が出てきた場合のことを想像してみて下さい。変数が何の動作をしているのかが即座に理解できるように、より説明的で読みやすい変数名が良いでしょう。

ここではスクリプトをより読みやすくするための話をしていますので、コメントについても解説していきましょう。

コメント

コメントはスクリプト内に注釈を差し込む方法です。注釈はスクリプトで何をしようとしているのかを、読む側に対し気づかせてくれるものです。ですからコメントにより情報を入力することで、読む側はより高くあなたの努力を評価することでしょう。コメントはLScriptからは完全に無視されます。コメントはモデラーがスクリプトを実行させる前に内部的に全て除去されるのです。構文という点から言えば、コメントには何を差し挟んでも間違いになることはないのです。

新しいHello.lsスクリプトにコメント文を挿入してみましょう。

```
main
{
  // ここはコメントです
  text = "Hello World!";
  info(text);
}
```

注意
このマニュアルでは、分かりやすく説明するためにコメントに日本語を使用していますが、実際のLScriptでは英数字のみしか使用できません。

ダブルスラッシュ文字（//）を打つことでコメントが作成されます。この文字（//）の後に続くテキストがコメントであるとみなされ、LScript からは無視されます。

次のコメント設定は悪い例です。

```
main
{
  コメント  //
}
```

何行にもわたってコメントしたい場合にはどうすればよいのでしょうか？二つの方法が挙げられます。まずは既に学習した方法です。

```
main
{
  //  このスクリプトは info ボックスを画面上に表示します
  //  変数テキストの値は  "Hello  World!"です
  …
}
```

二番目の方法では新しい文字を使用します。

```
main
{
  /*  このスクリプトは info ボックスを画面上に表示します
      変数テキストの値は  "Hello  World!"です。  */
  …
}
```

/* と */文字に囲まれた文字は、コメントとなります。注釈が長くなる場合、または不具合が発生するコードの大部分をコメント化したい場合には、この方法を使用して下さい。より効果的で見た目も良いです。

注意
/* と */を使用してコメントを作っても、//を使用してコメントを作ったコードには影響を及ぼしません。どのコメントもコメントアウトされたままとなります。

コメントは完全にオプションですが、出来る限りたくさん差し挟むようにして下さい。後でこの忠告が役に立つときが来ます。

さて、ここまでが基本編です。次の章では、ほんの少し中身について、ポイントやポリゴンの動作法について触れていきます。

第 8 章：モデラー：ポイントとポリゴン

モデラーにおける素晴らしさのまさに核となるのが、簡単にポイントやポリゴンの作成が出来るという点です。モデラーでは一歩先に進めて、このジオメトリを編集するために信じられないほど強力なツールをいくつも提供してくれています。どの 3D アーティストも知っているとおり、ポイントとポリゴンは三次元モデルで使用される基本となるジオメトリのコンポーネントです。ですからLScriptでジオメトリをどのように作成し、編集するのかを知っておくことは、きわめて重要なことなのです。

ポイントとポリゴンの作成

始める前に、まだ触れていないLScriptの領域である操作モードについてカバーする必要があります。LScriptコマンドは二つのカテゴリに系統立てられています、一つは **Command Sequence(CS)**モード、もう一つは **Mesh Data Edit(MD)**モードです。既にモデリングとスクリプト双方において **Command Sequence** モードで操作を行ってきました。このモードのコマンドはジオメトリのグループを集合体として処理するコマンド、例えば `move()`や `rotate()`、`bevel()` などや `extrude()`などです。

Mesh Data Edit モードは、使用するには少し注意が必要ですが、もっと泥臭いコマンドがたくさん用意されています。これらのコマンドを使用するとポイントやポリゴンを作成したり、量数などの情報を取得したり、サーフェイス名の割り当てなどが可能になります。これらのコマンドを実行するためには、`editbegin()`コマンドを使用して、モデラーを **Mesh Data Edit** モードへと切り替える必要があります。モデラーは **Mesh Data Edit** モードへと切り替わりましたから、エラーを出さずに **Command Sequence** モードでのコマンド実行は不可能です。これらのコマンドには `addpoint()`や `addpolygon()`、`pointinfo()`、`polysurface` などがあります。

再び **Command Sequence** モードへと切り替えるには、`editend()`コマンドを実行しなくてはなりません。このコマンドは二つの処理を行います。**Commmand Sequence** コマンドを走らせ **Mesh Data Edit** 操作モードにより送られた編集を実際に実行することです。この時点までMesh Data Editモードで実行されたコードは全てバッファに溜め込まれています。`editend()`関数を呼び出すまでジオメトリに対し実際の操作は処理されません。`editend()`はバッファに溜め込まれた修正をキャンセルし、バッファ内の操作を無視するブール値を受け取ります。

この概念をわかりやすく説明するため、三つのポイントを作成するスクリプトを作ってみましょう。モデラーのスクリプトですから、例のごとく `main()`関数から始めます。

```
main
{
}
```

では`editbegin()`と`editend()`コマンドを追加してみましょう。

```
main
{
  // Command Sequence モード
  editbegin(); // Mesh Data Edit モード
  editend();   // Command Sequence モード
}
```

このスクリプトを実行してみても何も起こりません。モデラーはモードを変更する以外何の指示も出していないからです。全く何も実行されません。

`addpoint()`コマンドは、引数として渡された三つの座標値を使用してポイントを1個作成します。このコマンドを`editbegin()`と`editend()`コマンドの間に差し込んでください。

```
main
{
  // Command Sequence モード
  editbegin();  // Mesh Data Edit モード
  addpoint(0,0,0);// point 1 の作成
  addpoint(0,1,0);// point 2 の作成
  addpoint(1,0,0);// point 3 の作成
  editend();    // Command Sequence モード
}
```

スクリプトを実行してみると、モデラーの正面ウィンドウに三つのポイントが作成されているのがわかりますね。作成したポイントを基に`addpolygon()`を使用してポリゴンを一つ作成してみましょう。まずはこの新しい機能を扱えるようにオリジナルのスクリプトを変更しなくてはなりません。

もともと以下のコードを使用してポイントを作成していました。

```
addpoint(0,0,0);
```

このコードでは`addpoint()`関数へ値`0,0,0`を送りました。`addpoint()`はポイントを作成し、LScriptは自動的に次の行へと移っていました。しかしポイントが作成された後に起こる処理を見逃していました。関数について解説した内容を覚えていますか？関数の中には処理が終了すると呼び出したスクリプトへと値を戻すものがあると説明しましたね。

今回の場合でも値が返されました。値を取得していなかっただけです。`addpoint()`コマンドから返される値は作成したポイントのポイント ID です。この ID は新規ポイントやポリゴンが作成されたときに作られる内部的な値です。ポイント ID を使用してどのポイントを選択しているのか、またどのポイントを編集したいのかを識別します。ではスクリプトを修正してみましょう。

```
main
{
  //Command Sequence モード
  editbegin();      // Mesh Data Edit モード
  point[1] = addpoint(0,0,0); // point 1 の作成
  point[2] = addpoint(0,1,0); // point 2 の作成
  point[3] = addpoint(1,0,0); // point 3 の作成
  editend();        // Command Sequence モード
}
```

このスクリプトを実行してみると、前と全く同じように実行されているのがわかりますね。差といえば、`addpoint()`関数が返す値にあります。`point[1]`から始まる変数配列には、`addpoint()`コマンドから返される値が入ることになります。これを証明するために、変数配列を使用してもう一つの新規コマンド`addpolygon()`を使用してみます。

三角形を作りましょう。このスクリプトでは上記スクリプトと全く同じですが、コメントが取り除かれ MeshDataEdit ブロックに 1 行追加されています。

```
...
main
{
  editbegin();
  point[1] = addpoint(0,0,0);
  point[2] = addpoint(0,1,0);
  point[3] = addpoint(1,0,0);
  polygon = addpolygon(point);
  editend();
}
...
```

見てわかるとおり、`addpolygon()`コマンドは作成されたポイントからポリゴンを作成します。変数配列`point[1], point[2]`そして`point[3]`に保存されたポイント ID を渡し、ポリゴンを作成するのです。モデラーでポリゴンを作成するときと同様、`addpolygon()`コマンドではポイントの順番が非常に大切になります。作成した順に値を渡すことでポリゴンの法線の向きを決定します。配列を逆順に並べると、ポリゴンの法線は反転します。

ポイント作成時における実例で示したとおり、`addpolygon()`コマンドも同様の関数ですから、保存用の変数へとポリゴンのIDを返します。この値を他のLScriptコマンドで使用してみると、三角形ジオメトリの編集が可能になります。ここでは`polypointcount()`関数へとポリゴンIDを渡してみましょう。この関数はポリゴン内のポイント数を返すだけの簡単な関数です。既にポイント数が何個あるかは知っていますが、ポリゴンIDを簡単にテストできます。

でも一つ見落としていることがあります。`polypointcount()`関数はMeshDataEditモードでのみ動作するという点です。MeshDataEditモードでは全ての編集はバッファに溜め込まれ、`editend()`コマンドが呼び出されるまで編集は処理されないんでしたね。これは難問です。ポリゴンは`editend()`関数が呼び出されるまで作成されないわけですから、当然ポリゴンも作成されておらず、このMeshDataEditセッション内ではポリゴンIDを取得できなくなります。

`editend()`を呼び出しポリゴンIDを作成してから、もう一度`editbegin()`関数を使用しMeshDataEditモードへと切り替える必要があります。ですから、関数を呼び出して戻り値を保存しておいた上でCommand Sequenceモードへと切り替えるようにしましょう。

```
main
{
  editbegin();
    point[1] = addpoint(0,0,0);
    point[2] = addpoint(0,1,0);
    point[3] = addpoint(1,0,0);
    polygon = addpolygon(point);
  editend();
  editbegin();
    pntCnt = polypointcount(polygon);
  editend();
  info(pntCnt);
}
```

注意
このコードは他のスクリプトには使用しないほうが良いでしょう。ポイント数の情報は最初のeditbegin()コマンドを呼び出すよりも前に既に把握しているからです。

ポイントとポリゴンの編集

ここまではポイントやポリゴンの作成法について解説してきましたが、既に存在しているポイントやポリゴンを編集したい場合には、どうすればよいのでしょうか？通常はカレントレイヤー上にあるジオメトリを調整したいでしょうから、このデータを取得できなければなりません。処理を行うジオメトリがポイントであろうとポリゴンであろうと、データへのアクセスや修正はLScriptで簡単に行えます。これを実演するために、ポイントを統合するスクリプトを作っていくことにしましょう。

例： weldfast

問題：統合後に現れる`info()`ボックスをどうにかして取り除くことにより、統合処理を効率的に行わなければなりません。モデラーのポイントの統合ツールはこの目的にぴったりなのですが、何千回も一緒にポイントを統合するとなれば、話は別です。

調査：まずはドキュメントとリリースノートにさっと目を通して、この問題にアプローチする最良の手段を考えてみて下さい。またLightWaveのCD-ROMやオンライン上にある動作可能な既存のスクリプトから研究してみることも出来ますね。これらのスクリプトは私たちの求めているものそのものではありませんが、どのように問題へとアプローチしていけばよいのかについて洞察力が養われるでしょう。注目すべきは以下の点です。

1. ポイントの位置情報を取得する方法
2. 三次元空間でポイントを動かす方法
3. それらを統合する方法

解決法：モデラーの統合ツールを作り直し、こうるさい`info()`ボックスを省略します。

さらに調査を重ねると、スクリプト作成の際に従う全体的な攻略法が作成できます。

1. 選択されているポイントのリストを集めます
2. 最後に選択されたポイントの座標値を確定します
3. 各ポイントを最終ポイントの座標位置へと移動します
4. ポイントを統合します

コード：

いつもどおり、`main()`関数から始めていきましょう。

```
main
{
  // 選択されているポイントのリストを集めます
  // 最後に選択されたポイントの座標値を確定します
  // 各ポイントを最終ポイントの座標位置へと移動します
  // ポイントを統合します
}
```

攻略法のテキストをそのままスクリプト内部へコメントとして入力しているのがわかりますね。こうすることで計画に沿って予定通りにスクリプトを組んでいくことが出来ます。またどこまで進んだのか、どのくらい作業が残っているのかも一目でわかります。

weldfast.lsとしてコードを保存します。

さて、調査したいポイントやポリゴンはどれなのかを決定しましょう。全てのジオメトリに対してなのか、それとも現在選択されているジオメトリに対してだけ処理を行うのかが選択できます。今回の目的では権限をユーザーの手に委ねたいので、レイヤーで現在選択されているポイントに対してのみ処理を行うようにします。`selmode()`コマンドを使用すると、モデラーに対し選択しているジオメトリだけを編集するように指示できます。

```
main
{
  // 選択されているコンポーネントに対してのみ編集します
  selmode(DIRECT);
  // 選択されているポイントのリストを集めます
  // 最後に選択されたポイントの座標値を確定します
  // 各ポイントを最終ポイントの座標位置へと移動します
  // ポイントを統合します
}
```

このコマンドには三つのオプション値があります。選択しているジオメトリに対してだけなら`DIRECT`、暗黙の選択（例えばInclude/Exclude選択を仮定）に対してなら`USER`、レイヤー内の全てのジオメトリに対してなら`GLOBAL`という値を用意しています。このコード行では`selmode()`関数へ実際に値を送信していますが、大文字変数が定数を表しているということを思い出してください。LScriptの中心部ではハードコード化された変数`GLOBAL=1`, `USER=2`,そして`DIRECT=3`という値を保持しています。各アイテムがどの数値で表されているのかを覚えるよりも、これらの変数名称の方がはるかに覚えやすいので、この定数を使うことにしましょう。

次に実際にポイントが選択されているのかを確認し、エラー発生の可能性を取り除きたいと思います。`pointcount()`コマンドを使用して、現在選択されているポイント数を確定します。

```
main
{
  // 選択されているコンポーネントに対してのみ編集します
  selmode(DIRECT);
  // 現在選択されているポイント数を取得します
  pntCnt = pointcount();
  // 選択されているポイントのリストを集めます
  // 最後に選択されたポイントの座標値を確定します
  // 各ポイントを最終ポイントの座標位置へと移動します
  // ポイントを統合します
}
```

コマンドをテストするため、`info()`ボックスに値を入れ、`pntCnt`の値をチェックします。次にモデラーにいくつかジオメトリを作成し、ポイントを選択してからスクリプトを実行します。選択されているポイント数が正確に出力されているはずです。

またポイントが何も選択されていない場合、`pntCnt`変数の値がゼロになるかもテストしてみたいですね。値がゼロの場合、ユーザーに対しエラーメッセージを表示させましょう。

```
if(pntCnt == 0)
  error( "No points selected." );
```

`error()`関数は`info()`とよく似た動作を行います。`info()`関数がダイアログボックスを開いてスクリプトを実行し続けるのに対して、`error()`はエラーメッセージを表示しスクリプトの実行を中断します。ユーザーエラーの可能性に十分注意しながら、スクリプトの中身を作成していきましょう。

現在選択されているポイントのリストが欲しいですね。幸運なことに、今までに使用したことのあるコマンドがここでも使用出来ます。`editbegin()`関数はMeshDataEdit操作モードに切り替えると同時に、`points[]`と`polygons[]`変数配列も作成します。スクリプトがMeshDataEdit操作モードに切り替える時に、LScriptはこれらの配列を自動的に作成し、その配列に現在選択されているコンポーネントを入れるので、スクリプトで大変役に立つのです。

```
...
// 選択されているポイントのリストを集めます
editbegin();
  // 最後に選択されたポイントの座標値を確定します
  // 各ポイントを最終ポイントの座標位置へと移動します
```

```
editend();
//  ポイントを統合します
...
```

このコードでは`points[]`と呼ばれる変数配列を作成し構成します。三つのポイントを選択している場合には、変数は以下のようになります。

```
points[1] = ポイントID #1
points[2] = ポイントID #2
points[3] = ポイントID #3
```

都合の良いことに、配列のインデックス（括弧内の番号）は各ポイント番号を表しています。ですから`points[1]`は実際にポイント1を表します。これら値の扱い方がうまく処理されます。

次のステップでは最後に選択されたポイントの座標値を取得しましょう。この座標値は選択されている他のポイント全てを移動させる座標値となります。ポイントの座標値データを取得するためにMeshDataEditコマンドである`pointinfo()`を使わなくてはなりません。

```
...
//  最後に選択されたポイントの座標値を確定します
lastPntPos = pointinfo(points[pntCnt]);
...
```

引数としてポイントIDを指定すると、`pointinfo()`はx, y, z座標値を表すベクトルの値を返します。今回の場合、関数には`point[]`配列のポイントIDを渡します。3点が選択されているため、変数`pntCnt`に保存されている値のインデックス番号を選択し、変数`pntCnt`には選択されたポイント数が設定されます。配列内の最後のインデックスがサイゴン選択されたポイントであると仮定しても差し支えなくなります。その結果、`pntCnt`も最後に選択されたポイントと等しくなります。

最後に選択されたポイントのベクトル座標値を見るため、`pointinfo()`行の後に`info()`ボックスを挿入します。以下のような表示が出てくるでしょう。

```
<0.12, 4.53, 2.3>
```

次に、選択された各ポイントを通し値を取得して新しい座標位置へと移動できるようにしたいですね。"各ポイントを通して"という部分からはループ処理が必要になることがわかります。ループは中断処理がかけられるまでコード行を繰り返すコントロール構造でしたね。このコマンドをMeshDataEditブロックに挿入する必要があります。

ではこのループを作成するにはどうすればよいのでしょうか？ループ構造を正しく動作させるためには、データの 4 つの部分が必要です。

1. 始点。ループがカウント処理を始める起点である変数を初期化する必要があります。今回の場合には配列の最初のポイントが起点となります。
2. 終点。ループ処理を終了する終点を把握しておかねばなりません。これは選択されているポイント数となりますのでループを制御する条件式を構築することになります。
3. 変数修正。ループはどのようにカウントしていくのでしょうか？大半のループでは一度に変数を 1, 2, 3 といった具合にカウントしていきます。
4. 実行させるためのコード。この作業は後で行います。今はこのループ処理のことが心配ですね。`info()`式を使用して、正しくカウントされているかを監視することにしましょう。

`for` 文の構造は以下のようになっています。

```
for(  変数の初期化  ;  条件式  ;  変数の修正  )
```

値を当てはめてみましょう。開始点はポイント 1 であることがわかっていますから、カウント変数 `currPnt` を 1 からはじめます。

```
for(  currPnt  =  1;  条件式  ;  変数の修正  )
```

`pointinfo()`文における最終ポイントは変数 `pntCnt` を使用して表していますので、この変数 `pntCnt` を使用して選択されたポイント総数を表すことが出来るといえるでしょう。そこでこの値を"ループの回数"用の値として使用することにします。

```
for(currPnt  =  1;  currPnt  <=  pntCnt;  変数の修正  )
```

この `for` ループ文は、"変数 `currPnt` の値が変数 `pntCnt` の値以下であれば、`for` ループ文につけられているコードを実行する"と解釈されます。ではなぜ"`<`"ではなく"`<=`"なのでしょうか？条件式で何が行われているのかを正確に説明する例を以下に挙げてみましょう。

`currPnt  =  1` かつ `pntCnt  =  3` の場合には、`currPnt  <  pntCnt` が `true` となります。ですからコードを実行します。

`currPnt  =  2` かつ `pntCnt  =  3` の場合には、`currPnt  <  pntCnt` が `true` となります。ですからコードを実行します。

しかし、`currPnt  =  3` かつ `pntCnt  =  3` の場合には `currPnt  <  pntCnt` が `false` になります。ですからコードは実行しません。

おわかりのとおり、このコードでは3番目のポイントに対して処理が行われませんが、処理を実行したいのです。`pntCnt`は`currPnt`より大きくなく等しくなるため、条件式では`false`になります。しかし`<`の代わりに`<=`を使用すれば、`for`ループ文は実際2回ではなく3回実行されることになります。これが今回やりたいことなのです。最初になぜループの回数だけ`info()`コマンドを作成するかがわかりましたね。こうすれば何が行われているのか正確に把握することが出来るのです。

各ループ内の処理が終了した後に変数`currPnt`の値を1ずつ追加するように指定し、`for`ループ文は終了です。この増加処理がないと、変数`currPnt`に保存されている値は決して変更されません。そうすると無限ループを作成することになってしまい、モデラーはフリーズしてしまいます。増加処理を追加しましょう。

```
for(currPnt = 1; currPnt <= pntCnt; currPnt++)
```

`currPnt++`は、実際には`currPnt=cuttPnt+1`という処理に解釈されます。つまりコードブロックが処理されるたびにカウントが1ずつ増えるようになっているのです。これで関数は完成です。

```
...
editbegin();
  // 最後に選択されたポイントの座標値を確定します
  lastPntPos = pointinfo(points[pntCnt]);
  for(currPnt = 1; currPnt <= pntCnt;
  currPnt++)
  {
    // 各ポイントを最終ポイントの座標位置へと移動します
    info(currPnt);
  }
editend();
...
```

スクリプトを実行してみると計画していた通り、スクリプトは選択しているポイント番号を数え上げていますね。これで後2～3行を残して大部分が完成です！

`for`ループのカウント処理は正しく動いていますので、ポイントの位置を移動させましょう。`for`ループ文の中の`info()`行を、以下のコード行に書き換えてください。

```
pointmove(points[currPnt],lastPntPos);
```

`pointmove()`コマンドは二つの引数を受け取ります。まず最初に移動させたいポイントのIDを`pointmove()`へ渡します。`points[]`配列内のIDを使用することも出来ますが、ここで少し、`for`ループマジックといえるような技術を使ってみましょう。配列のインデックス値は整数値を使用する代わりに変数`currPnt`を参照しているのがわかりますね。この戦術は`for`ループ文の中では常時使用できます。ご紹介しましょう。

```
currPnt = 1;
point[currPnt]; // 1番目のポイントID
currPnt = 2;
point[currPnt]; // 2番目のポイントID
currPnt = 3;
point[currPnt]; // 3番目のポイントID
```

と続きます。

`for`ループは実行し続けると変数`currPnt`の値が増加し、1行のコードで配列内の全ポイントが扱えるようになります。

`pointmove()`へ渡す2番目のパラメータはポイントの新規座標位置です。今回の場合は、最後に選択されたポイントから保存された座標位置となります。

スクリプトを実行して全てうまくいっているか確認してください。一つだけやり残したことがあります。現在、ポイントは新規座標位置に移動しただけに過ぎません。このポイントを統合しなくてはいけません。最後の行では実際にこの処理を行います。

```
// ポイントを統合します
mergepoints();
```

`mergepoints()`コマンドで何の引数も指定しない場合には、このスクリプトでも設定している選択モードで定義されたポイント全てを統合します。統合されるポイントはお互いがぴったりと重なり合っている状態にあります。

これで全て完了です！最終的なスクリプトを見てみましょう。

weldfast.ls

```
main
{
  // 選択されているコンポーネントのみを編集します
  selmode(DIRECT);
  // 現在選択されているポイント数を取得します。
  pntCnt = pointcount();
  if(pntCnt == 0)
  {
    error( "No points selected" );
  }
  // 選択されているポイントのリストを集めます
  editbegin();
    // 最後に選択されたポイントの座標地を確定します
    lastPntPos = pointinfo(points[pntCnt]);
    for(currPnt = 1; currPnt <= pntCnt; currPnt++)
    {
      // 最終ポイントの座標位置に移動します
      pointmove(points[currPnt],lastPntPos);
    }
  editend();
  // ポイントを統合します
  mergepoints();
}
```

weldfast スクリプトに基づき、他にも二つ簡単なスクリプトを作成することが出来ます。オリジナルの weldfast.ls と異なる箇所にはコメントを追加しました。

WeldAvg.ls

```
main
{
  //スコープ外で変数を宣言します
  totPntVal = < 0, 0, 0>;
  selmode(USER);
  pntCnt = pointcount();
  if(pntCnt == 0)
  {
    error( "No points selected" );
  }
  editbegin();
    /* 座標値を加算し選択されているポイント数で
        除算を行うことでポイント座標値の平均値を計算します */
    for(currPnt = 1; currPnt <= pntCnt; currPnt++)
    {
      currPntVal = pointinfo(points[currPnt]);
      totPntVal = totPntVal + currPntVal;
    }
    pntAvgPos = totPntVal / pntCnt;
    // 選択されている全ポイントを移動
    for(currPnt = 1; currPnt <= pntCnt; currPnt++)
    {
      pointmove(points[currPnt],pntAvgPos);
    }
  editend();
  mergepoints();
}
```

また、次のスクリプトではポイントのグループを統合しています。

```
main
{
  selmode(USER);
  pntCnt = pointcount();
  if (pntCnt == 0)
  {
    error( "No points selected" );
  }
  moveId = floor(pntCnt/2);
  editbegin();
    for(currPnt = 1; currPnt < (moveId + moveId); currPnt++)
    {
      info(currPnt);
      pntPos = pointinfo(points[currPnt+1]);
      pointmove(points[currPnt],pntPos);
      /* ポイントのグループを扱うので
          変数currPntは二度値を上げないといけません */
      currPnt++;
    }
  editend();
  mergepoints();
}
```

第 9 章：VMAP（頂点マップ）

この章では**VMap（頂点マップ）**を扱うための基本的な関数をいくつかカバーしています。サブパッチサーフェイスをコントロールするためウェイトマップを作成するにしろ、UV マッピングでテクスチャマップを使用するにしろ、頂点マップは巨大な力を与えてくれます。LScript も同様に、これらの頂点マップを修正するための同等の強力なツールセットを提供します。ですがこの新しいコードを探求していく前に、LScript のいくつか新しい領域について解説しなければなりません。

今回作成する例は、単にウェイトマップの値を 50% にするだけのスクリプトですが、途中で新しいコマンドや関数などを広範囲にわたってカバーしていきます。それでは始めましょう。

通常どおり、`main()`関数から始めますが、以下のコードではいくつかひねりを加えています。

```
// プリプロセッサコンパイラ指示子
@script modeler
main
{
  // VMap Object Agent を作成します
  // 選択されているポイント数を取得します
  // 選択されている VMap の値を 50% の重みにします
}
```

プリプロセッサとは？

プリプロセッサという言葉を解析してみると、"処理の前に起こる"ということを意味しており、これはそのまま、プリプロセッサコマンドの処理行動を表しています。`main()`関数のコンテンツが実行される前に、コンパイラはプリプロセッサ指示子を検索します。この場合、`@script modeler` はLScriptコンパイラに対し、これはモデラーで実行するスクリプトであると指示しています。スクリプトがプラグインとして追加されるときに、LScriptはプリプロセッサでスクリプトの種類を確定します。では実際に、プリプロセッサ指示子をいくつか追加してみましょう。

```
@name  "tmp"
@version 2.3
```

推測どおり、`@name` プラグマを使用してスクリプトに名称をつけることが出来ます。この名称はモデラープラグインのドロップリストとコンフィグ画面に表示されます。`@version` プラグマはスクリプトが動作するよう設計されたLScriptの最小バージョンを指定します。

バージョンは重要です。なぜならLightWaveのシステムは拡張を続けており、LScriptもまた同じだからです。現在動作するスクリプトが、前バージョンのLScriptのでは動かない可能性があります。`@version` プラグマはユーザーがどのバージョンを持っているかをチェックし、そのバージョンでスクリプトが動作するかを検証します。古いバージョンのLScriptではサポートしていないコマンドを使用したスクリプトを実行させようとすると、警告が発せられます。

Object Agents とは？

Object Agents はスーパー変数であるとみなされます。Object Agent はデータメンバとメソッドと呼ばれる二種類の異なるツールを持っています。このツールで自身の変数のようなコンテナ内部に保存されているデータにアクセスします。Object Agent データメンバには保存されているデータに関する適切な情報が入っています。この情報は名称であったりタイプやフラグ、値などの情報です。また Object Agent にはタスクを処理したり値を返すためのメソッドと呼ばれる関数があります。例としてメソッドには `count()`、`getvalue()`、`setvalue()` や `setcolor()` などがあります。

`main()` 関数内で試してみましょう。

```
// VMap Object Agent を作成します
vmapObj = VMap(VMWEIGHT);
// マップが存在しない場合はユーザーに通知します
if(vmapObj == nil)
  error( "No weight maps in the mesh!" );
```

前述のコードには二つの異なる Object Agents を適用しています。最初の例では特定の VMap のタイプを参照するための Object Agent を作成しています。このコードをもう少し詳しく見ていきましょう。

```
vmapObj = VMap(VMWEIGHT);
```

`vmapObj` と呼ばれる Object Agent は、`VMap()` コンストラクタから作成されます。コンストラクタは単に関数的なコードの一部であり、Object Agent を作成して使用出来るようにします。この特殊なコンストラクタが、オブジェクト内の VMap を参照する Object Agent を作成します。`VMap()` は引数を一つ受け取ることが出来ます。この引数は参照する VMap の種類です。引数は文字列もしくは定数どちらを渡しても構いません。以下のリストでは `VMap()` に渡せる定数値とその値を示しています。

```
VMSELECT  "select"
VMWEIGHT  "weight"
VMSUBPATCH  "subpatch"
VMTEXTURE  "texture"
VMMORPH  "morph"
VMSPOT  "spot"
VMRGB  "rgb"
VMRGBA  "rgba"
```

定数値 `VMWEIGHT` を渡してジオメトリ内のウェイトマップを参照するように指示します。

以下の例のように、このObject Agent内にあるデータメンバにアクセスするには等記号を使用出来ます。

```
nameOfVMap = vmapObj.name;
numOfValues = vmapObj.dimensions;
typeOfVMap = vmapObj.type;
```

ピリオド（.）はObject Agentの名称とそのデータメンバおよびメソッドを区切ります。現在、作成されたObject Agent `vmapObj` にはオブジェクト内で作成されている1番目のウェイトマップ用データが保存されています。

では以下のコードを追加することで、オブジェクト内に少なくとも一つのウェイトマップが存在しているかどうかをテストしてみましょう。

```
if(vmapObj == nil)
  error( "No weight maps in the mesh!" );
```

`vmapObj` が`(nil)`の場合、オブジェクト内にはVMapが存在しないということなので、その状況を通知するエラーメッセージをユーザーに送信します。しかし `vmapObj` が少なくとも一つのウェイトマップを探知した場合には、次のコード行を続行します。1番目のVMapを参照したい場合には、`if` ブロックの後に以下のコードを続けます。

```
info(vmapObj);
```

しかし、スクリプトをテストする前に、実験用のテストオブジェクトを作成しなくてはなりません。

テストオブジェクトの設定

LScript 作成時には、非常に基本的なオブジェクトでコードをテストしてみたくなる場合が頻繁に出てくることでしょう。コードが動いているかどうかを示すという、必要最低限の要求に応じられるオブジェクトでなくてはなりません。スクリプトが大きくなり、その処理に対する信頼度も増すにつれ、テストはさらに複雑なものとなり、ついには本物のオブジェクトでスクリプトをテストすることになります。

今回の場合、必要となるのはポイントに対し"testMap"という名称の 1 個のウェイトマップが割り当てられている一枚のポリゴンだけです。作成するスクリプトはそれほど動的なものではありませんので、一回限りのコードであると考えられます。つまりこのテストオブジェクト以外では決して適用されないスクリプトなのです。この章の後の方ではインターフェイスが解説してあります。そこでも今回のように一回限りのシンプルなスクリプトを作りますが、複数使用が可能となる強力なツールへと変換できます。

基本となるオブジェクトを作成しディスクに保存します（後で再読み込みしたくなるかもしれませんから）。例を続けて、以下のコードを入力します。

```
// VMap のスケールを 50% に設定します
scaleAmt = .5;
// 選択されているジオメトリのみに対し処理を行います
selmode(DIRECT);
// MeshDataEdit モードに切り替えます
editbegin();
editend();
```

ここでは新しいことは何一つ行っていません。変数を作成し、VMap の値をスケールするために使用するパーセンテージ（小数）を代入しました。それから選択モードを DIRECT に設定し MeshDataEdit モードに入りました。この編集ブロックに以下のコードを挿入します。

```
// 選択されているポイント数を取得します
pntCnt = pointcount();
// 選択されているポイントを通して反復処理を行うために
// for ループ文を設定します
for(pnt = 1; pnt < pntCnt + 1; pnt++)
{
  // ループをテストします
  info( "point" + pnt);
}
```

今回は`info()`関数を少し違った形式で使用しました。大規模なスクリプトにおいては多数の値を出力させたいでしょうが、これらの区別がつけにくくなる場合があります。そこで、文字演算を使用して、多少なりとも意味のある形式で出力するようにしました。テストオブジェクトを読み込み、ポリゴンの4点を選択し、スクリプトを実行してください。

ポイントを数え上げる`(point1、point2...)`4個の`info()`ボックスが出てきますね。

次に`info()`関数を次のコードで置き換えてください。

```
pntCnt = pointcount();
// カレントのポイントがVMapの中でマッピングされて
// いるかどうかをチェックします
if(vmapObj.isMapped(points[pnt]))
{
  // マッピングされていればVMapの値を取得します
  values = vmapObj.getValue(points[pnt]);
  // 出力をテストします
  info(values);
}
```

`vmapObj` Object Agentを使用してウェイトマップがカレントポイント上にマッピングされているかをチェックします。ポイントIDを`isMapped()`メソッドに渡してチェックします。メソッドは`if`条件式を解くブール値`true`または`false`を返します。

次の行では`getValue()`メソッドにポイントIDを渡します。このメソッドはカレントポイントに対しVMapで保存されている値を返します。それから`info()`関数で値をチェックします。スクリプトを実行すると、0.0～1.0の範囲内にある値が4個表示されます。

あとはVMapの値を現在の値の50%に設定するだけです。`info()`行を次のコードで置き換えてください。

```
// 値に対する次元数全てにループします
for(x = 1;x <= vmapObj.dimensions; x++)
  values[x] *= scaleAmt;
```

`for`ループは値に対する全次元数に反復処理を行います。今回の場合、ウェイトマップには一つの値だけが割り当てられています。ループブロックは値の各要素（ここでは一つだけ）に対し0.5（50パーセント）の値を持つ`scaleAmt`を掛け合わせます。

```
values[x] *= scaleAmt;
```

これは、実際には以下の処理を行っています。

```
values[x] = values[x] * scaleAmt;
```

変数values[x]の値にscaleAmtが掛け合わされ、その結果が再びvalues[x]へと保存されることになります。これはforループ内にある式と同様の簡単な増分です。

最後の行はこうなります。

```
// 選択したVMapの値に50%掛け合わせます
vmapObj.setValue(points[pnt],values);
```

この行では、setValue()メソッドに対し二つの引数points[pnt]とvaluesが渡されます。valuesという変数の値が、points[pnt]で表されるポイントのVMapの値と置き換わります。

それでは、処理がよく見えるようにモデラーのOpenGLシェードモードをウェイトマップに変えてスクリプトを実行してください。値が半分になってのがわかりますね。素晴らしいです。

これが最終コードとなります。

```
// プリプロセッサコンパイラ指示子
@script modeler
@name "tmp"
@version 2.3
main
{
  // VMap Object Agent を作成します
  vmapObj = VMap(VMWEIGHT);
  // マップが存在しない場合はユーザーに通知します
  if(vmapObj == nil)
    error( "No weight maps in the mesh!" );
  // VMapの値に50%を掛け合わせます
  scaleAmt = .5;
  // 選択されているジオメトリのみを扱います
  selmode(DIRECT);
  // MeshDataEdit モードに切り替えます
  editbegin();
    // 選択されているポイント数を取得します
    pntCnt = pointcount();
    // 選択しているポイントを通して反復処理を行う
```

```
    // for ループを設定します
    for(pnt = 1; pnt <= pntCnt; pnt++)
    {
      // VMap がカレントのポイントにマッピング
      // されているかどうかをチェックします
      if(vmapObj.isMapped(points[pnt]))
      {
        // マッピングされていれば VMap の値を取得します
        values = vmapObj.getValue(points[pnt]);
      }
      // 値の次元数全てに反復処理を行います
      for(x = 1;x <= vmapObj.dimensions; x++)
      values[x] *= scaleAmt;
      // 選択している VMap の値を 50% にします
      vmapObj.setValue(points[pnt],values);
    }
  editend();
}
```

あとほんの少しのコードとインターフェイスを追加して、何が出来るか見てみましょう。

```
// プリプロセッサコンパイラ指示子
@script modeler
@name "tmp"
@version 2.3
main
{
  // VMap Object Agent を作成します
  vmapObj = VMap(VMWEIGHT);
  // マップが存在しない場合はユーザーに通知します
  if(vmapObj == nil)
    error("No weight maps in the mesh!");
  /* while 文を使用することで、VMap の値が存在し、
     VMap のタイプが VMWEIGHT である場合には
     変数に名称を付け加えます。これはオブジェクト内の
     複数の VMap を扱います */
  while(vmapObj && vmapObj.type == VMWEIGHT)
```

```
    {
      vmapnames += vmapObj.name;
      // .next()メソッドを使用して次の VMap に行きます
      vmapObj = vmapObj.next();
    }
    /* これはインターフェイス部分のコードであり
        後ほど詳細を説明します */
    reqbegin( "Scale Weight VMap" );
      c1 = ctlpopup( "VMap", 1, vmapnames);
      c2 = ctlnumber( "Scale by (%)",50.0);
      return if !reqpost();
      vndx = getvalue(c1);
      scaleAmt = getvalue(c2)/100;
    reqend();
    /* ドロップリストからユーザーが選択した VMap の名称で
        再び Object Agent を作成します */
    vmapObj = VMap(vmapnames[vndx]);
    // この場合は Object Agent は作れません
    if(vmapObj == nil)
      error( "Could not instance_VMap`", vmapnames[vndx],
"`!" );
    // 選択されているジオメトリのみを扱います
    selmode(DIRECT);
    // MeshDataEdit モードに切り替えます
    editbegin();
      // 選択されているポイント数を取得します
      pntCnt = pointcount();
      // 選択されているポイントを通して反復処理を行う
      // for ループを設定します
      for(pnt = 1; pnt <= pntCnt; pnt++)
      {
        // カレントのポイントに対し VMap がマッピングされているかを
        // チェックします
        if(vmapObj.isMapped(points[pnt]))
        {
          // マッピングされていれば VMap の値を取得します
```

```
        values = vmapObj.getValue(points[pnt]);
        // 値の次元数全てに対し反復処理を行います
        for(x = 1; x <= vmapObj.dimensions; x++)
          values[x] *= scaleAmt;
        // VMap の値に 50% を掛け合わせます
        vmapObj.setValue(points[pnt],values);
      }
    }
  editend();
}
```

第 10 章：サーフェイスとレイヤー

モデラーにおけるサーフェイスの処理はLScriptにとっては簡単な仕事です。サーフェイスとは基本的に名称を表す文字列であり、ポリゴンはサーフェイスに属しています。この情報を編集するためのたくさんのツールが使用できます。また少数ではありますが、比較的苦労せずにサーフェイスを作成したり編集するための、自由に使える強力なコマンドが用意されています。

レイヤーはLightWaveのモデリングにおける重要な側面です。レイヤーのコントロール法を把握していれば、モデラーやLScriptの能力を十分に活用することが出来るのです。一般的にLScriptにおけるレイヤーはモデラーにおいての動作とほぼ同じです。どのレイヤーが前景レイヤーなのか、背景レイヤーなのか、ジオメトリはどのレイヤーに属しているのかなどを確定します。

この章ではポリゴンのサーフェイス名称に基づいてオブジェクトをばらばらにするスクリプトを作成し、LScriptの二つの側面を解説していきます。ポリゴンを小さな断片に分離させた後、新規オブジェクトを別個のレイヤーへと配置します。テストオブジェクトとして、NewTekのお気に入りである牛のオブジェクトを使っていきましょう。牛のオブジェクトには程よい量のジオメトリがあり、サーフェイスも数種類用意されているからです。

では始めましょう。

```
@script modeler
@version 2.3
@name  "FissureLite"

main
{
  // サーフェイス名称を取得します
  // サーフェイスが付けられているポリゴンを選択します
  // 選択しているジオメトリをカットします
  // 空レイヤーを検索します
  // 新しいレイヤーにジオメトリを貼り付けます
}
```

まずは予備データをいくつか取得しましょう。`main()`関数に次のコードを入力します。

```
// USERモードに切り替えます
selmode(USER);
/* 現在の前景レイヤーを取得します
    これがオブジェクトが入っているレイヤーとなります */
workLayer = lyrfg();
// サーフェイスリストから1番目のサーフェイスを取得します
currSurf = nextsurface();
```

lyrfg()関数は、どのレイヤーがカレントの前景レイヤーかを示す単数もしくは複数の整数値を返します。この値を変数workLayerに保存しておき、オリジナルのオブジェクトが入っていたレイヤーを保存しておきます。

サーフェイスは、モデラーで管理されている内部リスト内に整列化されています。nextsurface()コマンドを使用することで、このリストを通して見ることが出来ます。コマンドに対し引数が何も与えられない場合は、リスト内の1番目のサーフェイスを返します。しかしコマンドにサーフェイス名称が与えられると、この引数を参照点として使用し、リスト内にある次のサーフェイスを返します。上記のコードでは1番目のサーフェイス名称を検索しています。

while()コマンドは、スクリプト組み立て時に使用するコマンドの中で最も危険なコマンドの一つです。不正に使用すると無限ループに落ちいってしまい、手動でモデラーを強制終了する破目になり、データを全て失ってしまいます。このような結果を避けるため、while()コマンドは非常に慎重に設定します。**重要：必要となるコードを全て配置し安全に使用出来るまで、このコードはテストしません。**

```
// サーフェイスの最後までループを繰り返します
while(currSurf != nil)
{
}
```

ループ処理が終了した段階で、モデラーの内部リストにある全サーフェイスを通して実行したことになります。変数currSurf内の文字列値がnilになった段階でループを終了します。ですから変数currSurfに対し、ある時点でnilを代入しないと無限ループとなってしまいます。while()コマンドの中でfalseとなるような条件式を一時的に設定することにより、この問題を解決し、スクリプトを終了させます。

```
// サーフェイスの最後までループを繰り返します
while(currSurf != nil)
{
  // ループを終了させます
  currSurf = nil;
}
```

ループ処理は強制的に終了させましたから、スクリプトを安全にテストすることが出来ます。スクリプト実行時には何の処理も行いませんが、スクリプトは無事終了し、モデラーがコントロールできるようになります。ループから抜け出すために`break`コマンドを使用することも出来ますが、どんな値が条件式を`false`にするかをテストしたかったのです。`currSurf`の値に`nil`を設定出来ることがわかりましたので、ループ処理は終了します。

この条件式はそのように動作するよう設計されています。サーフェイス名称に対するリスト内において、検索を中断する時期を把握する方法が必要でした。幸運なことに、リスト内の最後のサーフェイス名は名称に対し常に`nil`の値を取ります。ですからリストを通し最終的に終端まで検索すると、`nil`の値が示されることが保証されています。ここで終端まで来たかどうかを確認する必要があります。

```
// サーフェイスの最後までループを繰り返します
while(currSurf != nil)
{
  // サーフェイスに属するポリゴンを選択します
  selpolygon(SET, SURFACE, currSurf);
  // ループを終了させます
  currSurf = nil;
}
```

`selpolygon()`関数は、非常に複雑なコマンドです。使用時に最もわかりにくいコマンドの一つです。今回の簡単な例においてこのコマンドには三つの値を渡しています。最初の二つの値はメジャーまたはマイナーモードを表す定数値です。`SET`はポリゴンを選択し、`CLEAR`は選択を解除します。`SURFACE`はマイナーモードを表す引数で、選択ポリゴンに対するパラメータとしてサーフェイス名を使用することを指定しています。3番目の引数は、検索する実際のサーフェイス名称です。

cowオブジェクトにスクリプトを実行してみましょう。スクリプトが終了した時点で、目が選択されているはずです。リストの中において`CowEyes`サーフェイスが1番目のサーフェイスだからであり、`selpolygon()`コマンドを使用して目のポリゴンが選択されたのです。

リストには、ジオメトリを何も保持していないサーフェイス名称が含まれている場合があります。そのためリストをスキャンするときに、実際にポリゴンが選択されているのかを確認しなくてはなりません。このチェック処理を行わないと、スクリプトはジオメトリを何も選択せず`cut()`コマンドを使用してしまい、オブジェクト全体が除去される場合があるのです。これは避けたいので`selpolygon()`コマンドの後に以下のコードを入力します。

```
...
pntCnt = pointcount();
if(pntCnt)
{
  // 条件式をテストします
  info( "Got Some!" );
}
...
```

edit ブロックは前章で何度も処理してきましたから、もうお馴染みですね。pointcount()を使用して、選択されているジオメトリのポイント数を取得します。その後、if 文で値が実際に変数 pntCnt に保存されているのかを判定します。この条件文はあまりに簡単すぎて何も出来ないのではないかと思われるかもしれませんが、変数や条件式に関する規則を思い出してください。

まず、条件式は if ブロック内にある値が true か false かにより、コードを実行するのかしないのかを依存しています。内部ブール値 true と false はそれぞれ値 1 と 0 を持っています。実際には true の値はゼロ以外の値ならどの値でも true となります。ですから pntCnt が 0 でない限りは true であるとみなされ、if ブロック内のコードを実行します。これはまさに今回機能してほしいことですね。テストしてみると、info()ボックスに " Got Some! " という文字が表示されるでしょう。つまりスクリプトは予測どおりに動いたことになります。

次に info()行を以下のコードで置き換えてください。

```
// 選択されているジオメトリをクリップボードに移します
cut();
// 空レイヤーを検索します
emptyLyr = lyrempty();
// そのレイヤーへと切り替えます
lyrsetfg(emptyLyr[1]);
// クリップボードからレイヤーへとジオメトリを貼り付けます
paste();
```

まずは cut()コマンドでオブジェクトからポリゴンを削除し、モデラーのクリップボードへと配置します。それから次に利用できる空レイヤーを検索します。これは lyrempty()関数を使用して処理します。この関数（と lyrfg()）は、安全に使用出来る空レイヤーを表す整数配列を返します。空レイヤーに切り替えるため lyrsetfg()コマンドを使用し、引数として整数値を指定します。emptyLyr 配列変数における 1 番目のインデックスを使用すると、使用可能な 1 番目のレイヤーが表示されます。最後に paste()コマンドを使用し、クリップボードからジオメトリを取得して、空レイヤーへと挿入します。

このスクリプトを実行すると、牛の目が切り取られ、他のレイヤーへと貼り付けられましたね。

さてループ処理を終了させるためのコードを、より論理的なものへと置き換えなくてはなりません。条件式に`false`を設定しているコード行を削除し、`while()`ループの一番下の箇所に以下のコードを挿入します。

```
// リファレンスとしてカレントのサーフェイスを使用し
// 次のサーフェイスを取得します
currSurf = nextsurface(currSurf);
/* ループが続行するようであればジオメトリの残りの部分がある
    オリジナルのレイヤーへと戻ります
    オリジナルのレイヤーは変数workLayerに記憶しておきましたね */
lyrsetfg(workLayer);
```

現在、変数`currSurf`に保存されているサーフェイスを`nextsurface()`関数のリファレンスとして使用することが可能です。この2行のコードは`while`ループ文と結びつけられており、サーフェイスの名称が`nil`になるまで反復処理を行います。`nil`になった時点でスクリプトは終了します。

これで完成です！以下が最終的なコードとなります。

```
@script modeler
@version 2.3
@name "FissureLite"
main
{
  // USER モードに切り替えます
  selmode(USER);
  /* 現在の前景レイヤーを取得します
      これがオブジェクトが入っているレイヤーとなります */
  workLayer = lyrfg();
  // サーフェイスリストから１番目のサーフェイスを取得します
  currSurf = nextsurface();
  // サーフェイスの最後までループを繰り返します
  while(currSurf != nil)
  {
    // サーフェイスに属しているポリゴンを選択します
    selpolygon(SET, SURFACE, currSurf);
    pntCnt = pointcount();
```

```
    if(pntCnt)
    {
      // 選択されているジオメトリをクリップボードに移します
      cut();
      // 空レイヤーを検索します
      emptyLyr = lyrempty();
      // そのレイヤーへと切り替えます
      lyrsetfg(emptyLyr[1]);
      // クリップボードからレイヤーへとジオメトリを貼り付けます
      into layer.
      paste();
    }
    // リファレンスとしてカレントのサーフェイスを使用し
    // 次のサーフェイスを取得します
    currSurf = nextsurface(currSurf);
    /* ループが続行するようであればジオメトリの残りの部分が
         おいてあるオリジナルのレイヤーへと戻ります
         オリジナルのレイヤーは変数 workLayer に記憶しておきましたね */
    lyrsetfg(workLayer);
  }
}
```

第 11 章：変位マップ

Displacement Map スクリプトでは時間が経過するとともにオブジェクトのポイント座標位置を修正することが出来ます。Displacement Map Object Agent 用のメソッドを使用すると、オブジェクトの外観を修正するため、ポイント座標値データの読み書きが可能になります。Displacement Map スクリプトはオブジェクトが溶け出しているような外観を作ったり、慣性の力に従ったり、または他の複雑なアニメーションを操作することが出来ます。

LScript にある Layout スクリプト構造の大半と同様、Displacement Map もこのスクリプトクラスで使用可能なオプションを利用するため事前定義されている関数群を使用します。スクリプトが適切に動作するためにこれらの関数がいくつか必要となる一方で、その他は単にオプションの機能を提供するだけとなります。

create() と destroy()

`create()` と `destroy()` 関数は、変数の初期化とスクリプトに対する関数のクリーンアップを行います。`create()` 関数はオプション引数として Mesh Object Agent を受け取ります。Object Agent はスクリプトが適用されているオブジェクトから作成されます。

newtime()

`newtime()` 関数は、シーンのカレントフレームが修正されるたびに呼び出されます。この関数はレイアウトから三つの引数、Mesh Object Agent（`id`）とカレントフレーム（`frame`）、それにカレントタイム（`time`）を受け取ります。Mesh Object Agent（`id`）はスクリプトが適用されているオブジェクトから作成されます。

一般的に、これらの引数は複数の大域変数を通して他の関数でも利用可能になります。

flags()

`flags()` 関数は、レイアウトに対しスクリプトがポイントデータをどのように処理するのかを指示します。このスクリプトが関数に正しく要求するのは `return()` 文を含む単一行だけというのが一般的です。`return()` 文から渡される二つの定数のうち一つは、スクリプトが `WORLD` 座標のポイントを処理するのか、それとも `LOCAL` 座標のポイントを処理するのかを指示します。定数が省略されている場合、ポイントは `WORLD` 座標で処理されます。

process()

`process()`関数は、ポイント修正コードが記述されている最も重要な関数です。この関数はレイアウトから呼び出されたときにDisplacement Map Object Agent（`da`）を引数として受け取ります。`process()`関数はフレームが変化するたびにレイアウトにより呼び出されます。

Displacement Map Object Agentには、二つのデータメンバ`oPos[]`と`source[]`があります。どちらのデータメンバもポイント座標値x, y, zにアクセスします。しかしこれらのデータメンバには大きな二つの違いがあります。一つは`oPos[]`は読取専用であり、`source[]`はそうでないという点です。もう一つは、`oPos[]`はオブジェクトの起点（ローカル）からの位置を返し、`source[]`はワールド座標における座標値を返すという点です。

load() と save()

`load()`と`save()`関数を使用すると、レイアウトはこのスクリプト操作に付属する特定のデータの読み書きが可能になります。選択されたオプションや設定はスクリプトをどのように実行するのかを決定するパラメータであり、シーンファイル内部に保存されます。シーンが保存または読み込まれるときに、スクリプトが適用されているアイテムのチャンネル情報が確実に破棄されないようにします。

レイアウトがシーンを読み込みスクリプト用のエントリが現れると、`load()`関数が呼び出されます。この関数内では、想定どおりのスクリプト処理を可能にする設定や値を読み込み、割り当てます。同様に`save()`関数はユーザーがシーンを保存するときに呼び出されます。この関数には、シーンファイルにスクリプトのデータを系統立てて保存するときに必要なコードが全て含まれています。

options()

`options()`関数は、プラグインリストでスクリプトのプロパティをダブルクリックしたり、編集を選択するたびに呼び出されます。そこにはスクリプトのインターフェイスを設定し実行するために必要なコードが全て含まれています。

例: Splat!

まず最初にヘッダー情報を取得しましょう。このスクリプトは最新バージョンから特別なことを要求してはいないので、`version2.3`にしておきましょう。Displacement Mapスクリプトであることを指定し、"`Splat`"という名称にします。このスクリプトをプラグインとして追加すると、リストに現れます。

```
@warnings
@version 2.3
@script displace
@name Splat
```

このスクリプトに対して4つの関数を使用します。ここで設定を行います。

```
create
{
}

newtime: id, frame, time
{
}

process: da
{
}
```

それに

```
options
{
}
```

後でインターフェイスも扱いますが、いくつかテストできるように初期値を設定しておく必要があります。複数の関数にまたがり、これらの値を共有できるよう大域変数として設定しましょう。

```
@script displace
@name Splat
// 大域変数
splatValue = 0;
create
{
```

変数 `splatValue=0` と設定しました。後々、インターフェイスを設定してユーザーから `splat` の割合を設定してもらうようにしますが、今はこうしておきましょう。次に、プラグインリスト内で名称を表示するためのスクリプトの解説文を簡単に設定することにします。

```
create
{
  setdesc( "Splat!" );
}
```

さて次に、`newtime()` や `process()` 関数が呼び出されるときにレイアウトから受け取る値を学ぶため、これらの関数の中に `info()` 関数をいくつか挿入してみましょう。

```
newtime: id, frame, time
{
  info( "newTime: id: ", id, " frame: ",frame, " time: ",
time);
}

process: da
{
  // da はレイアウトから渡される Displacement Map Object Agent
  info( "process: ", da.oPos[1]);
}
```

そろそろスクリプトを保存しても良いでしょう。splat.ls として保存してください。でもスクリプトを実行する前にテストオブジェクトを作成する必要があります。Displacement Map はオブジェクトにある多くの（あるいは全ての）ポイントの移動処理を行うため、テストオブジェクトはジオメトリ数が出来るだけ少ないほうが良いですね。そうすればスクリプトに何か正しくないことが発生しても、何百回も発生するわけではないですから。

モデラーで、中心点をワールド座標（0, 0, 0）に持つ簡単なボックスを作成してください。アクセスするポイントデータは、三次元空間におけるボックスの位置によって劇的に変化します。

オブジェクトを Box.lwo として保存してください。いろいろなモデリング状況をカバーするために、利用可能なテストオブジェクトをいくつも用意しておくほうが良いでしょう。スクリプトをテストする状況が増えるほど、設計ミスやバグを見逃してしまう可能性は少なくなるのです。

レイアウトに Box.lwo を読みこむ

オブジェクト Box.lwo を選択し、オブジェクトの **Item Properties（アイテムプロパティ）**パネル > **Deform（変形）**タブ > **Add Displacement Plugin（変位プラグイン追加）**ポップアップメニューから splat.ls を適用します。

すぐに `info()` ボックスリクエスタが開き、データを表示します。これはエラーメッセージではなく、`newtime()` 関数内に設定した `info()` 関数の結果です。うっかりリクエスタを閉じてしまったら、タイムスライダを前後に動かすだけで同じような結果を表示させることが出来ます。最初の `info()` パネルには、スクリプトが適用されているオブジェクト名称、現在のフレーム番号、タイムが表示されます。

二番目に表示される `info()` パネルは、`process()` 関数に挿入した `info()` 関数の結果です。画面上に送られるデータはオブジェクトの最初のポイントの x 位置を表しています。その後に続く `info()` パネルは、残りのポイントそれぞれの x 座標位置を表示します。ボックスは 8 個のポイントを持っていますから合計 8 回パネルが表示されます。このためにも多くのポイントを持つオブジェクトを作成するのは避けたのです。

現在のテストの要求に対し各ポイントの x 座標値を見るのはちょっとやり過ぎています。本当に必要としているのは、スクリプトをテストするためにポイントデータを一つだけ見ることです。現在オブジェクトのどのポイントに対して処理を行っているのかを把握するため、カウンター変数を設定しましょう。まずはカウンターを大域変数として設定します。

```
// 大域変数
splatValue = 0;
pntCounter = 0;
newtime: id, frame, time
```

もう画面上にポイント情報を送る必要はなくなりましたので、`newtime()` と `process()` 関数から `info()` 関数を削除しましょう。オブジェクトのポイントデータを個別にハンドルするための新しい `info()` 関数を追加します。以下のコードを追加してください。

```
process: da
{
  pntCounter++;
  if(pntCounter == 1)
  info("process: point: ", pntCounter, " ", da.oPos[1]);
}
```

このコードを追加すると、処理されているカレントのポイント番号を追いかけることが出来ます。スクリプトを実行してみると、最初のポイントだけが表示されているのがわかりますね。しかしこの理論の抜け穴を調べるために、フレームを変更してみてください。今までに得たDisplacement Mapスクリプトについての知識から、イベント全体において全ポイントを通し最初のポイントのX座標値だけを表示していくだろうと想定することでしょう。しかしそのようには動作しません。どこが悪いのかわかりますか？

エラー箇所はここです。カウンターは正しく処理されているものの、スクリプトに対しフレームが変更されたときにカウンターをリセットするようスクリプトに指示していないために、エラーが発生するのです。最初にスクリプトが実行されたときには、1から8までカウントされます。しかし、二回目に実行されたときには9から16までカウントされます。この処理が続いていくため`pntCounter`は再び1になることが二度とないのです。

この問題を修正するためには、フレームが変更された時点でカウンターをリセットするように、スクリプトに指示してあげるだけでよいのです。`newtime()`関数に次のコード行を挿入してください。

```
newtime: id, frame, time
{
  pntCounter = 0;
}
```

スクリプトを保存しオブジェクトから古いスクリプトを除去した上で、もう一度新しいスクリプトを試してみてください。何度かテストしてみると、タイムスライダが変更されるたびにカウンターがゼロへ戻っているのがわかるでしょう。この結果、フレームが更新されるたびに一個の`info()`パネルが表示されることになります。

では次に、アニメーションがポイントのX位置にどのような影響を及ぼすのかを見ていきましょう。まず始めにこの例を通して使用する簡単なテストアニメーションを設定していきましょう。アニメーションをより簡単に作成するため、プラグインリスト内にあるスクリプトをオフにします。そうすればスクリプトはスクリーンにメッセージを送信するのを一時的に中断します。次に二つのアニメーションキーフレームを作成します。0フレームでは0, 0, 0の位置に、フレーム10では2, 0, 0の位置にキーを作成します。これでシーンを保存すれば、シーンがおかしくなってしまってもシーンを簡単にリセット出来るようになります。

シーンに対しアニメーションをつけたら、スクリプトをもう一度オンにし、毎フレームのアニメーションを通して見てみましょう。シーンではポイントのX座標値は変わっているにもかかわらず、メッセージに表示されるX座標値は変化していませんね。これはオブジェクトのローカル座標値を使用した例です。Object Agentの`oPos[1]`データメンバを使用することで、ポイント座標値はオブジェクトがシーン内のどこに位置しているかなどを気にせずに、ローカルに決定することが出来るのです。レイアウトでピボットポイント位置を変更した場合でも同じです。

ポイントのワールド座標へアクセスするにはどうすればよいのでしょうか？Displacement Map Object Agetの`oPos`データメンバにアクセスする代わりに、`source`データメンバを使用してみてください。`info()`コードの行を以下のように修正します。

```
info("process: point: ", pntCounter, " ", da.source[1]);
```

スクリプトを更新し再実行しましょう。タイムスライダを変更してみると、`source`データメンバとの違いがわかりますね。オブジェクトのポイントをX軸を横切るように移動すると、ワールド座標も変更されます。必要となるため、オブジェクトのワールド座標値を使用し、カレントの`process()`関数を以下のコードへと置き換えます。

```
process: da
{
  pntCounter++;
  if(pntCounter == 1)
  {
    // ポイントのWORLD座標値を取得します
    xPos = da.source[1];
    yPos = da.source[2];
    zPos = da.source[3];
    info(xPos, " ", yPos, " ", zPos);
  }
}
```

新しく追加された`process()`関数でスクリプトを実行してみると、まだ`info()`関数は表示されるものの、ポイントのx, y, z座標値が表示されていますね。この値が私たちが処理を加える値となります。

`oPos[]`と`source[]`データメンバには、もう一つ大きな違いがあります。`oPos[]`を通してポイントの座標値は取得できるものの、値を設定することは出来ません。例えば以下の式はエラーが発生します。

```
da.oPos[1] = 1;
da.oPos[2] = 1;
da.oPos[3] = 1;
```

ところが、以下のコードは正当です。

```
da.source[1] = 1;
da.source[2] = 1;
da.source[3] = 1;
```

source[]データメンバはデータの読み書きどちらも可能ですから、手動でポイントのワールド座標値を設定することが出来ます。LScriptではオブジェクトの元のポイント座標値を修正することは出来ません。

このSplatスクリプトでは、単にオブジェクトジオメトリの最小のY座標値を強制的に配置します。このためにprocess()関数にもう少しラインを追加します。

```
info(xPos, " ", yPos, " ", zPos);
// ポイントの値が最低許容値よりも
// 小さいかどうかを判定します
if(yPos <= splatValue)
  da.source[2] = splatValue;
```

技術的にはこれで完了です。しかし、最初のポイント値だけをテストしてきましたので、pntCounterリファレンス、info()関数や条件式を省いていきましょう。また、コード内でもはや意味をなさない関数は除去しましょう。

最終的なスクリプトは以下のようになります。

```
@warnings
@script displace
@name splat
// 大域変数
splatValue = 0;

create
{
  setdesc("Splat!");
}

process: da
{
  // ポイントのWorld座標値を取得します
  xPos = da.source[1];
  yPos = da.source[2];
  zPos = da.source[3];
  // ポイントの値が最低許容値よりも
  // 小さいかどうかを判定します
  if(yPos <= splatValue)
    da.source[2] = splatValue;
}
```

このスクリプトの全ての効果をさらに解説するために、フレーム0の値が（0, 2, 0）、10フレームの値が（0, 0, 0）となるようにキーフレームを変更しましょう。XZ平面にボックスのオブジェクトが衝突するさまを見てください。

約束どおり、ここにスクリプト用のインターフェイスコードがあります。`process()`関数のボディマーカーが閉じられた後にこのコードの断片を追加するだけです。

```
options
{
  reqbegin("Splat!");
    c1 = ctlnumber("minimum y value: ", splatValue);
    return if !reqpost();
    splatValue = c1.value;
  reqend();
}
```

変数値やその変数がどこから来るのかなどをうまく説明するため、長たらしく不必要なコードをいくつか`process()`関数内に残しておくことにします。どうすれば`process()`関数をより効果的に出来るのかを考えてみましょう。以下は極めて効率的な`process()`関数です。

```
process: da
{
  //  ポイントの値が最低許容値よりも
  //  小さいかどうかを判定します
  if(da.source[2] <= splatValue)
    da.source[2] = splatValue;
}
```

例: dynSplat!

インターフェイスでデータを取得するのではなく、他のオブジェクトのワールド座標値を用いて前回の例でも使用した衝突面を定義していきます。こうすればスクリプトはより動的なものとなり、アニメーション効果のオプションを提供するようになります。新規スクリプト用に、前回のsplat.lsスクリプトを起点として使用しましょう。

まずスクリプトの名前を変更します。

```
@name pragma to read:
@script displace
@name dynSplat
// 大域変数
```

次に、使用する大域変数をいくつか変更する必要があります。変数`splatValue`を削除し、以下のコードを追加します。

```
@name dynSplat
obj;
objName = "Null";
currTime;
create
{
```

これらの値には参照しているオブジェクトから作成されたObject Agentやオブジェクトの名称、さらにシーンの現在時刻が入ります。スクリプトの新しい名称を考慮して、スクリプトの`create()`関数を変更しましょう。

```
create
{
  setdesc("dynSplat!");
}
```

前回のスクリプトでは`newtime()`関数を全く使用していませんでしたので省いていました。しかし今回は`newtime()`関数を使用してシーンの現在時刻用の新しい変数`currTime`を設定します。`currTime`は大域変数ですので、スクリプト内にある関数は全てその値にアクセス可能です。

```
newtime: id, frame, time
{
  currTime = time;
}
```

また、参照オブジェクトの名称から作成されたObject Agentを大域変数objへと代入します。参照オブジェクトの名称は変数objNameに保存されています。現在この値は"Null"となっていますが、このスクリプトがシーン内で正しく動作するためには"Null"という名称のオブジェクトが必要となります。最終的にはこれをカバーするためのインターフェイスを追加しますが、今は手動で"Null"を設定するようにしましょう。

```
newtime: id, frame, time
{
  currTime = time;
  if(objName)
    obj = Mesh(objName);
}
```

お気づきの通り、Object Agentを作成する前に変数objNameに実際に値が入っているのかを確認する時点でエラー判定を作成しました。速度の関係上、process()関数内ではなく、ここで代入を行うようにしてあります。process()関数内でObject Agentを作成すると、絶えず呼び出されることになってしまい、ユーザーにうるさくなってしまいます。newtime()関数はカレントフレームが変更した場合にのみ呼び出されますから、操作はそれほど目立たなくなります。

最後にprocess()関数を作成しましょう。修正の大半はここで行われます。実際に一から始めるのが多分良いでしょう。

```
process: da
{
}
```

アニメーションしているオブジェクトの位置データにアクセスし、そのY座標値をポイントの最小値として使用したいのです。まずはMesh Object Agentであるobjが作成されているかどうかを確認しましょう。

```
process: da
{
  if(obj)
  {
  }
}
```

Object Agentの作成に成功していれば、その位置情報を取得できます。

```
if(obj)
{
  pos = obj.getPosition(currTime);
}
```

`getPosition()`メソッドは、シーン内の任意の時刻`currTime`（この大域変数は`newtime()`関数内で設定されています）におけるオブジェクトの位置情報をベクトルで返します。さて、あとはsplat.lsと同様、値を比較するだけです。

```
{
  pos = obj.getPosition(currTime);
  if(da.source[2] < pos.y)
    da.source[2] = pos.y;
}
```

シーン内に"Null"という名称のオブジェクトがある限り、このオブジェクトのY座標値を使用して衝突面をどこに配置するのかを決定することが出来ます。以下、新規インターフェイスコードに適応するように二つ変更点を加えた最終スクリプトを記します。

```
@warnings
@script displace
@name dynSplat
// 大域変数
obj;
objName;
currTime;

create
{
  setdesc("dynSplat!");
}

newtime: id, frame, time
{
  currTime = time;
  if(objName)
    obj = Mesh(objName);
```

```
}

process: da
{
  if(obj)
  {
    pos = obj.getPosition(currTime);
    if(da.source[2] < pos.y)
      da.source[2] = pos.y;
  }
}

options
{
  reqbegin("dynSplat!");
    c1 = ctlmeshitems("objects: ", "Null");
    if(!reqpost())
    return;
    obj = c1.value;
    if(obj)
      objName = obj.name;
  reqend();
}
```

第 12 章：オブジェクト置き換え

Object Replacement（LS-OR）用のLScriptでは、アニメーション中にどのオブジェクトをいつ使用するのかを制御することが出来ます。オブジェクトは特定のフレームや時刻に基づいて置き換えることも可能ですし、位置によって置き換えることも出来ます。実際、ほとんど全てのものをオブジェクト置き換えの引き金として使用することが出来ます。プログラマに全て委ねられているのです。

LS-OR メソッドとメンバ

LScript Object Replacementのコマンドの中心となるのは、`process()`関数です。`process()`関数では引数**Replacement Object Agent**を一つだけ受け取ります。Replacement Object Agentはアニメーションに関する重要な情報をスクリプトに与えるデータメンバ群を提供します。このデータメンバの大半は読取専用であるため、情報を取得することは出来ても変更することは出来ません。

以下に利用可能なメンバを記します。

読取専用

`objID`は置き換えるジオメトリのオブジェクトを表すObject Agentです。`objID`によって共通のObject Agentメソッド全てにアクセスできるようになります。

`curFrame`はカレントジオメトリに対するフレーム番号を表す整数値です。

`curTime`はカレントジオメトリに対するタイムインデックスを表す整数値です。

`newFrame`は次のフレーム番号を表す整数値です。オブジェクトがこの新規フレームにおいて異なる外観とならねばならない場合に、新規ジオメトリが読み込まれます。

`newTime`は次のタイムインデックスを表す数値です。オブジェクトがこの新規フレームにおいて異なる外観とならねばならない場合に新規ジオメトリが読み込まれます。ネットワークレンダリングなどでは非連続の時刻間を移動する場合もありますので、`curTime`と`newTime`が連続していない場合もあります。

タイムインデックスは各フレームで生成される浮動小数点数値です。フレーム番号とは異なり、タイムインデックスはシーン中のフレーム/秒の設定に依存します。

例：

シーン中で24フレーム/秒に設定されている場合、フレームに対するタイムインデックスは以下のようになります。

```
フレームタイムインデックス(24フレーム/秒)
1/24 もしくは .042
2/24 もしくは .083
3/24 もしくは .125
```

シーン中で30フレーム/秒に設定されている場合、フレームに対するタイムインデックスは以下のようになります。

```
フレームタイムインデックス(30フレーム/秒)
1/30 もしくは .033
2/30 もしくは .066
3/30 もしくは .1
```

`curType`は現在のレンダリングの種類を指示する定数値です。スクリプトはこの値を調べ、インタラクティブなプレビューや実際のレンダリング用として異なるジオメトリを提供することが可能です。`curType`には以下のタイプがあります。

`NONE`はカレントのタイムインデックスに対しジオメトリを何も読み込まないことを示しています。

`PREVIEW`はレイアウトプレビューが生成されていることを示しています。

`RENDER`は完全なレンダリングが処理されていることを示しています。

`newType`は次のフレーム/タイムインデックスのタイプを示す定数値です。`NONE`, `PREVIEW`もしくは`RENDER`です。

`curFilename`はカレントのオブジェクトジオメトリファイルを表す文字列であり、ジオメトリが読み込まれていない場合には'`nil`'になります。ディスク上に存在するオブジェクトのフルパス付き名称となります。

書き込み可能

`newFilename`

`newFilename` は、カレントオブジェクトに置き換えるオブジェクトの名称です。LScript はこの変数を使用して新規ジオメトリをレイアウトへ読み込むので、オブジェクトのフルパス付き名称を指定してください。

注意
カレントジオメトリを置き換えるときにのみ newFilename に値を代入してください。カレントオブジェクトを代入しないようにして下さい。

では Object Replacement のサンプルスクリプトを見てみましょう。このスクリプトではアニメーション中にある特定したフレームにおいて二つの異なるオブジェクトを入れ替えるようにします。スクリプトを管理しやすいように複数のセクションに分けてどのように処理しているのかを見ていきましょう。

```
CHANGER.LS
//----------------------------------------
// Changer
//
```

どのスクリプトタイプでも、'//'文字はユーザーコメントの領域を表します。コメントに関しての詳細は107 ページ 14 章をご覧下さい。

```
@version 2.3
@warnings
@script replace
@name Changer
```

プラグマ指示子はスクリプトラベルの後に記述します。まず最初にこれらの指示子が処理されますので、スクリプト内のどの場所に置いても自由です。しかしスクリプトをより読みやすくするためにはスクリプトの冒頭にこの指示子を置くのが良いでしょう。

次に大域変数の宣言領域となります。LScript には二種類の変数宣言、グローバル（大域）とローカルがあります。ローカル変数はスクリプトのある特定の領域に属する変数です。関数内で宣言されている変数はローカルであると解釈され、変数が終了すると同時に破棄されます。カウンター用に変数を 2 ～ 3 使用したい場合には大変便利で、スクリプト内のどこかにある変数を参照する必要もありません。

しかし大域変数は全ての関数から利用可能であり、この変数を使用した関数が終了した時点でも破棄されることはありません。

```
changeobj1 = "";
changeobj2 = "";
prog = "Changer v1.01";
swapframe1 = 1;
swapframe2 = 30;
```

大域変数をより詳しく見て、スクリプト実行中に各変数で何を行っているのかを解明していきましょう。

changeobj1には置き換える1番目のオブジェクト名称が入っています。スクリプトのポイントは既存のオブジェクトを異なるオブジェクトに置き換えるところにあるのですから、置き換えるオブジェクトの名称を保存しておく場所が必要なのです。

changeobj2は2番目に置き換えるオブジェクトのファイル名称を保存しておくもう一つの変数です。

progはスクリプトラベル用の便利なショートカットです。これを使えば一箇所でリクエスタ用のスクリプト名称を変更することが出来、名称を載せるたびにスクリプト全体を検索しなくてもすみます。

swapframe1はオブジェクトchangeobj1に対応するフレーム番号です。ユーザーからはフレーム番号を二つ取得します。これらはオブジェクトを切り替えるフレームです。

swapframe2はchangeobj2に対応する2番目の切り替えフレームです。

それではcreate()関数から始めましょう。

```
create
{
  setdesc(prog);
}
```

create()関数はスクリプトが読み込まれた最初の時点で呼び出されます。この関数はスクリプトが最初に追加された時、もしくはシーンファイルが読み込まれた時に呼び出されます。create()関数の中ではスクリプト作成時に処理したい変数やその他のアクションなどを準備します。

今回の場合、setdesc()関数を使用して、アクティブなスクリプトを識別させます。関数setdesc()は一つの文字列変数を受け取り、この文字列がプラグイン行に表示されることになります。変数progのコンテンツを渡すと以下のようになります。

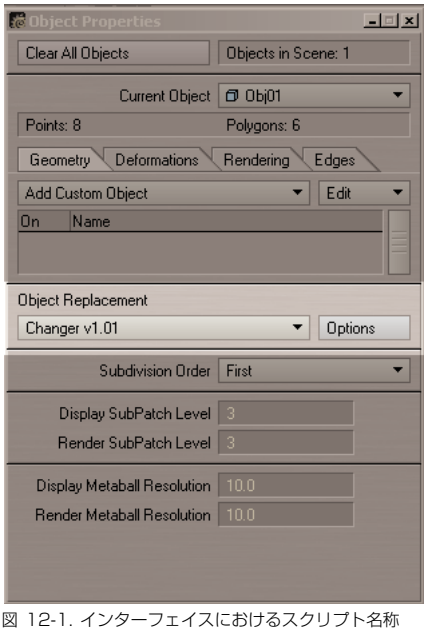

図 12-1. インターフェイスにおけるスクリプト名称

```
process: ra
{
  thisframe = ra.newFrame;
}
```

実行早々、`process()`関数でReplacement Object Agentとオブジェクトの全データメンバにアクセスできるようになります。Replacement Objectは変数`ra`を通してスクリプトに渡されます。変数`ra`は好きなように呼び出せますが、LScript Editorから提供されるテンプレートの一部ですので、ここで使用しましょう。

ローカルとグローバル変数の違いについて先ほど解説しました。`process()`関数の二行目に変数`thisframe`がありますが、この変数はここまで宣言されていません。`process()`内部以外では必要としませんので、関数内部において動的に作成します。この`thisframe`はスクリプトの他のセクションからは見えませんが、スクリプトを実行する箇所でフレームの跡を追うためだけに使用するので大丈夫なのです。

変数`thisframe`はReplacement Objectのデータメンバ`newFrame`を通して現在のフレームを取得します。Object Agentのデータメンバを取得するためには正しい構文を使用します。Object Agentを入れてある変数名称の後にピリオドをつけ、データメンバの名称を付け加えます。以下、サンプルを示します。

```
ObjectAgent.datamember
```

このスクリプトの正しい構文は以下のとおりです。

```
ra.newFrame
```

newFrame は読取専用のデータメンバですから、変数に記入するためだけに使用されます。以下、代入可能なデータメンバの扱い方を示します。

```
switch(thisframe)
{
  case swapframe1:
    ra.newFilename = changeobj1;
    break;

  case swapframe2:
    ra.newFilename = changeobj2;
    break;
}
```

ra.newFrame からの情報を使用して、オブジェクトを変更するべきかどうかをチェックします。

switch()文を使えば、読みやすく簡単に拡張可能なコードで選択を系統立てることが出来ます。switch()は変数（今回の場合thisframe）を受け取り、定義しているcaseの一連の値と比較します。今回は各フレームにおいて検討するため二つのcase文を用意しました。thisframeの値をswapframe1とswapframe2の両方に対して比較し、値が一致した場合にはそれぞれのコマンドを処理します。switch()関数はswap変数に対する一連のif()文と似ていますが、より論理的な形式でグループ化しています。

スクリプトを実行しているうちに、switch()で変数swapframeのどちらかの値と一致した場合、ra.newFilenameに変数changeobjのどちらか一方を代入します。これでレイアウトでは変数changeobjのどちらかのオブジェクトファイルが読み込まれることになります。

下記のoptions()関数内では、変数swapframe、changeobjのグループ双方に対し値を割り当てます。

```
load: what,io
{
  if(what == SCENEMODE)
  {
    changeobj1 = io.read();
```

```
    swapframe1 = integer(io.read());
    changeobj2 = io.read();
    swapframe2 = integer(io.read());
  }
  setdesc(prog);
}

save: what,io
{
  if(what == SCENEMODE)
  {
    io.writeln(changeobj1);
    io.writeln(swapframe1);
    io.writeln(changeobj2);
    io.writeln(swapframe2);
  }
}
```

load()とsave()関数を使用すると、スクリプト用の大切な情報を保存したり取得出来ます。双方の関数を走らせるためにOBJECTMODEとSCENEMODEという二つのモードがあります。置き換え情報は特定のオブジェクトに対してではなくシーンファイルのみに保存されるため、どちらの関数に対してもSCENEMODEを使用します。

SCENEMODEではシーンファイル保存時にsave()関数内の情報が.lwsファイルに書きこまれ以下になります。

```
Plugin ObjReplacementHandler 1 Changer
Script I:\PAPERWORK\LSCRIPTDOCS\LSOR\SCRIPTS\CHANGER.LS
X:\Obj02.lwo
1
X:\Obj03.lwo
30
EndPlugin
```

レイアウトはプラグイン再読み込みの際に必要となる情報を全て取得し、記憶用の値に記入します。保存した順番どおり情報をシーケンシャルに記憶させる方法に注意してください。

また、シーンファイルに書き込まれている情報は全てテキスト（文字）とみなされます。ですから整数変数`swapframe1`をコマンド`writeln()`でシーンファイルに書き込むと、テキストへと変換されます。シーンファイルを再び読み込む場合は文字から整数へと変換する必要がありますからよく覚えておいて下さい。`integer()`コマンドは`load()`関数内を見てもわかる通り、変換処理を行います。

```
options
{
  reqbegin(prog);
    c1 = ctlfilename("Swap Object",changeobj1,20,true);
    c2 = ctlinteger("Swap Frame",swapframe1);
    c3 = ctlfilename("Swap Object",changeobj2,20,true);
    c4 = ctlinteger("Swap Frame",swapframe2);
    return if !reqpost();
    changeobj1 = getvalue(c1);
    swapframe1 = getvalue(c2);
    changeobj2 = getvalue(c3);
    swapframe2 = getvalue(c4);
  reqend();
}
```

`options()`関数内でユーザー用のインターフェイスを作成します。レイアウトでオプションボタンを選択するとインターフェイスが表示されます。インターフェイス作成はこのマニュアルのあらゆる箇所でカバーされていますが、スクリプトを実行しレイアウトにてどのような動作をするのか見てみましょう。

スクリプトの実行

レイアウトを開く前に、使用する三つのオブジェクトを選択しなければなりません。一つはベースとなるオブジェクトであり、後の二つは置き換え用のオブジェクトです。手早くテストできるように、オブジェクトは小さくて簡単なものにして下さい。テスト用にお気に入りの何万ポリゴンもある代表作を選んでしまうと、切り替え時において読み込みに時間がかかってしまいます。

それではレイアウトを開き、**Edit Plug-ins（プラグイン編集）**パネルでスクリプトをLightWaveに読み込みます。スクリプトが読み込まれたら**Edit Plug-ins（プラグイン編集）**パネルを閉じ、最初のオブジェクトを読み込みます。

オブジェクトを読み込んだら、オブジェクトの**Item Properties（アイテムプロパティ）**パネルを開き、**Geometry（ジオメトリ）**タブの**Object Replacement（オブジェクト置き換え）**プルダウンメニューから**Changer**を選択します。

図 12-2. オブジェクト置き換えのプルダウンでハイライトになっている

Object Replacement（オブジェクト置き換え）プルダウンのとなりにあるオプションを選択し、**Changer**のインターフェイスを開きます。

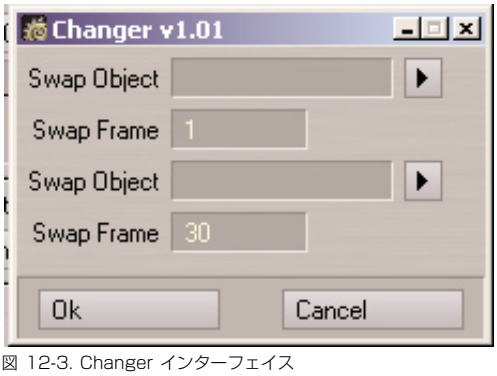

図 12-3. Changer インターフェイス

パネル右側にある小さな三角形でファイルリクエスタを開き、置き換え用オブジェクトを選択します。

図 12-4. 三角を押してファイルリクエスタを開く

選択したオブジェクトがシーンに挿入されるフレームを数値フィールドに入力します。

図 12-5. オブジェクトが挿入されるフレームを表示するフィールド

オブジェクトを選択し、フレーム番号を修正して'**OK**'ボタンをクリックします。アニメーションをプレビューすると、オブジェクトが置き換わっているが確認出来ます。

注意
Object Replacement はキーフレームをスライドさせただけでは処理されません。オブジェクト置き換えを確認するにはプレビューを作成するかシーンをレンダリングして下さい。

例に沿ってスクリプトを書いてきましたが、以下に完全なコードリストを用意しています。

フルリスト(REPLACER.LS)

```
@version 2.3
@warnings
@script replace
@name Changer
changeobj1 = "";
changeobj2 = "";
prog = "Changer v1.01";
swapframe1 = 1;
swapframe2 = 30;

create
{
  setdesc(prog);
}
process: ra

{
  thisframe = ra.newFrame;
  switch(thisframe)
  {
    case swapframe1:
      ra.newFilename = changeobj1;
      break;

    case swapframe2:
      ra.newFilename = changeobj2;
      break;
  }
}

load: what,io
{
  if(what == SCENEMODE)
  {
```

```
    changeobj1 = io.read();
    swapframe1 = integer(io.read());
    changeobj2 = io.read();
    swapframe2 = integer(io.read());
  }
  setdesc(prog);
}

save: what,io
{
  if(what == SCENEMODE)
  {
    io.writeln(changeobj1);
    io.writeln(swapframe1);
    io.writeln(changeobj2);
    io.writeln(swapframe2);
  }
}

options
{
  reqbegin(prog);
    c1 = ctlfilename("Swap Object",changeobj1,20,true);
    c2 = ctlinteger("Swap Frame",swapframe1);
    c3 = ctlfilename("Swap Object",changeobj2,20,true);
    c4 = ctlinteger("Swap Frame",swapframe2);
    return if !reqpost();
    changeobj1 = getvalue(c1);
    swapframe1 = getvalue(c2);
    changeobj2 = getvalue(c3);
    swapframe2 = getvalue(c4);
  reqend();
}
```

第 13 章：カスタムオブジェクト

LightWaveのCustom Objectでは、レイアウトに読み込まれている他のメッシュアイテムに対しレンダリングされない特別なオブジェクトをユーザーが割り当てることが出来ます。このオブジェクトはユーザーに対し情報を表示したり、ツール特有のデータを提供したり、LightWaveで事前定義されていない視覚効果を提供するために使用出来ます。この機能を使うと、簡単なヌルオブジェクトを文字列で表示したり、効果範囲を示したり、さらにはより複雑なヘッドアップディスプレイにすることも可能なのです。これらは全てユーザーにとっては信じられないほど強力な視覚化ツールですが、さらに重要なのは、LightWaveのレンダラーからは不可視の状態にあるという点です。

カスタムオブジェクトは、スクリプトにより提供されるポイント座標値とエッジのリストによって構築されます。このリストはカスタムオブジェクトになるワイヤーフレームメッシュを定義します。エッジは実線や点線、ダッシュ、ロングダッシュなど様々なメソッドを使用して描画することが出来ます。

Custom Objectスクリプトは、以下のレイアウト関数を使用します。

create() と destroy()

`create()`と`destroy()`関数はスクリプトに対する変数の初期化と関数のクリーンアップを扱います。Custom Objectスクリプトタイプ用として、`create()`関数はスクリプトが適用されるオブジェクトから作成されたMesh Object Agentをオプションとして受け取ることが可能です。

init() と cleanup()

`init()`と`cleanup()`関数は、レンダーパスの開始（`init()`）と終了（`end()`）時に呼び出されます。

newtime()

`newtime()`関数はシーン内でカレントのタイムインデックスが変更されるたびに呼び出されます。Custom Objectスクリプト内において、`newtime()`関数はレイアウトから二つの引数`frame`と`time`を受け取ります。これらの値は大域変数に割り当てるのが一般的で、サポートしている複数の関数内で使用出来るようにします。

flags()

`flag()`関数はレイアウトにフラグの値を返すことでCustom Objectスクリプトクラスに利用可能なオプションを設定します。現在フラグの値は、`SCHEMA`だけが定義されています。このフラグはレイアウトに対し、スケマティックビューにおける描画をサポートするか否かを指示します。スクリプトが描画をサポートしていない場合には、この関数は定義しない方が良いでしょう。一般的に一つのコマンドだけ、つまり`return()`文だけがこの関数には付加されます。

process()

`process()`関数には、重要な描画コードの大半が含まれます。オブジェクトに再描画の必要ありとレイアウトが判断するたびにこの関数が呼び出され、引数を一つ、Custom Object Agentである`ca`を受け取ります。このObject Agentには、開発者がカスタムオブジェクトの作成/描画を行うための多数のデータメンバとメソッドが含まれています。

注意
Custom Object Agentで利用可能なデータメンバとメソッドの完全なリストについては第2巻第29章239ページを参照してください。このマニュアルのリファレンスセクションにあるCustom Object Agentsを参照してください。

load() と save()

`load()`と`save()`関数を使用すると、このスクリプトの操作に関する特定のデータを読み込んだり保存することが出来ます。選択されている任意のオプションやスクリプトをどのように実行するのかを決定する設定は、シーンファイル内部に保存されなくてはなりません。シーンが保存または読み込まれるときに、スクリプトが適用されているアイテムのチャンネル情報が確実に破棄されないようにします。

レイアウトがシーンを読み込みスクリプト用のエントリが現れると、`load()`関数が呼び出されます。この関数内では、想定どおりのスクリプト処理を可能にする設定や値を読み込み割り当てます。同様に`save()`関数はユーザーがシーンを保存するときに呼び出されます。この関数はシーンファイルにスクリプトのデータを系統立て保存するときに必要なコードが全て含まれています。

options()

`options()`関数は、プラグインリストでスクリプトのプロパティをダブルクリックしたり、編集を選択するたびに呼び出されます。そこにはスクリプトのインターフェイスを設定し実行するために必要なコードが全て含まれています。

例： custText.ls

まず最初の例では、スクリプトが適用されているオブジェクトの名称を表示する簡単なカスタムオブジェクトを作成してみましょう。手始めに、ヘッダー情報を取得し、二つの大域変数を設定します。

```
@warnings
@version 2.4
@script custom
@name custText
// 大域変数
obj;
currTime = 0;
```

必要とする情報の一番目の部分は、スクリプトのオブジェクトの名称です。オブジェクトの名称を表示するために使用するラベルを作成するため、この情報が必要となります。`create()`関数から送られるMesh Object Agent引数からこのデータを取得することが可能です。

```
create: ma
{
  obj = ma;
  setdesc( "custText" );
}
```

作成しておいた大域変数`ma`にMesh Object Agentを割り当てることで、スクリプト内の関数全てからObject Agentのデータメンバとメソッドを利用することが出来ます。このヘッダー部分ではスクリプトの解説文も定義しておきます。

レイアウトにおける他のアニメーションオブジェクト同様、カスタムオブジェクトはXYZ座標値システムを使用して、三次元空間のどの箇所に描画すればよいのかを決定します。今回のスクリプトに対しては、メッシュオブジェクトの現在位置を文字描画の始点として使用します。こうすると名称テキストが探しやすく、属しているメッシュオブジェクトがわかりやすくなります。

以前使用したグローバルMesh Object Agent用のメソッドを通してメッシュオブジェクトの位置情報を取得することが可能です。しかし、`getPosition()`メソッドは関数がMesh Objectのポイント情報を取得する時刻を指示する引数を必要とします。シーンの現在時刻はスクリプトの`newtime()`関数へと渡されますので、この関数を追加して、引数`time`へ大域変数`currTime`を代入すればよいのです。

```
newtime: frame, time
{
  currTime = time;
}
```

大域変数を設定すると同時に、`process()`は動作し始めます。まずメッシュオブジェクトの位置を取得したいですね。

```
process: ca
{
  // Mesh Object から情報をいくつか取得します
  objPos = obj.getPostion(currTime);
}
```

これらの変数をなぜ大域変数にするのかはお分かりでしょう。`create()`から Mesh Object Agent である`obj`を使用するだけではなく、`process()`から`currTime`も使用します。次に実際に文字を描画しましょう。

```
process: ca
{
  // Mesh Object から情報をいくつか取得します
  objPos = obj.getPostion(currTime);
  // Object Agent である ca を使用してカスタムオブジェクトを作成します
  ca.setColor(<100,100,100>);
  ca.drawText(objPos, obj.name, X , CENTER);
}
```

まずはカスタムオブジェクトの描画色を灰色（<100,100,100>）に定義します。最終的には`drawtext()`メソッドを使用して、実際にオブジェクトの名称をスクリーン上に描画します。変数`objPos`を渡し、データメンバ`obj.name`を使用してラベルの位置と表示したいテキストを定義します。それから定数値`X`と`CENTER`を渡すことで、ラベル上のテキストの位置合わせを定義します。

スクリプトを保存しヌルオブジェクトに適用してください。オブジェクトのピボットポイントにメッシュオブジェクトの名称が表示され、ラベルが描画されているのがわかりますね。またメッシュオブジェクトの位置を移動させたりキーフレームをつけてみると、ヌルオブジェクトがアニメーション中にどのような動作を取るのかもわかるでしょう。

最終コードは以下のとおりです。

```
@warnings
@version 2.4
@script custom
@name custText
// 大域変数
obj;
currTime = 0;

create: ma
{
  obj = ma;
  setdesc( "custText" );
}

newtime: frame, time
{
  currTime = time;
}

process: ca
{
  // Mesh Object から情報をいくつか取得します
  objPos = obj.getPostion(currTime);
  // Object Agent である ca を使用してカスタムオブジェクトを作成します
  ca.setColor(<100,100,100>);
  ca.drawText(objPos, obj.name, X , CENTER);
}
```

例：Barn.ls

このサンプルでは非常に簡単な家の形をしたカスタムオブジェクトを作成してみましょう。まずはスクリプトのヘッダー情報を作成するところから始めましょう。

```
@warnings
@version 2.4
@script custom
@name custBarn
```

次にカスタムオブジェクトをどのように描画するのかを定義しなくてはなりません。二つの大域変数`vert`と`edge`を宣言することで定義します。

それぞれオブジェクトを描画するために使用する値のリストを保持します。

```
vert = @ <0.0, 0.0, 0.0>, <1.0, 0.0, 0.0>,
<1.0, 1.0, 0.0>, <0.5, 1.5, 0.0>,
<0.0, 1.0, 0.0>, <0.0, 0.0, -1.0>,
<1.0, 0.0, -1.0>, <1.0, 1.0, -1.0>,
<0.5, 1.5, -1.0>, <0.0, 1.0, -1.0> @;
edge = @ 1, 2, 2, 3, 3, 4, 4, 5, 5, 1, 6, 7,
7, 8, 8, 9, 9, 10, 10, 6, 1, 6, 2, 7, 3,
8, 4, 9, 5, 10, 2, 5, 1, 3 @;
```

変数`vert`には始点ポイントを`vert[1]`、終点ポイントを`vert[10]`としたオブジェクト内の各ポイントの位置を記しています。変数`edge`リストでは始点と終点ポイントのインデックスをリストすることでワイヤーフレームのエッジを定義しています。例えば（上記`vert`と`edge`リストを使用すると）

```
vert[1] = <0.0, 0.0, 0.0>
vert[2] = <1.0, 0.0, 0.0>
edge 1 = 1, 2
```

これはつまり

```
edge 1 = vert[1], vert[2]
```

もしくは

```
edge 1 = <0.0, 0.0, 0.0>, <1.0, 0.0, 0.0>
```

オブジェクトが定義されましたので、`process()`関数を設定し、正しく描画させなければなりません。総数17本のエッジが作成されます。注意深く`for`ループ文を設計すれば、1行だけで簡単に描画することが可能です。

```
process:
{
  // 直線を描画します
  for(x = 1;x < 35;x += 2)
  ca.drawLine(vert[edge[x]],vert[edge[x+1]]);
}
```

`drawline()`メソッドは二つの引数、始点ポイントと終端ポイントを受け取ります。これら二つの値を提供すると、LScriptはレイアウトに対し二点を結ぶ直線を描画するように指示します。`for`ループ文では単に`verts[]`とそれに関連している`edge[]`のリストを回していき、正しく描画されるようにフォーマットしています。ループするたびに変数`x`の値がどのように増加しているのかに注目してください。カウンター`x`を使用することで最初の頂点にアクセスし、`x+1`とすれば2番目の頂点にアクセスが可能です。ループの各回でこのような処理を行い、`drawline()`メソッドでエッジを一本描画します。

少し複雑になってきましたので、`process()`関数を修正します。

```
process:
{
  // 直線を描画します
  for(x = 1;x < 31;x += 2)
    ca.drawLine(vert[edge[x]],vert[edge[x+1]]);
  ca.setPattern("dot");
  for(x = 31; x < 35;x += 2)
    ca.drawLine(vert[edge[x]],vert[edge[x+1]]);
}
```

再びループを通して描画をコントロールしていますが、17本のエッジが15本しか描画していません。次に直線のパターンをドットに設定し、デフォルトである実線の代わりにドット状の直線を描画するように指定します。最後の二つのエッジはその前の15本のエッジと同じような`for`ループ分を使用して描画します。

スクリプトを保存しヌルオブジェクトにスクリプトを適用してください。17本で出来た簡単な家が描画されましたね。

サンプルコード:

以下のコードは、既存のLightWaveモデルを既成のカスタムオブジェクトスクリプトに変換してくれるモデラーのスクリプトです。コードを入力するのに多少時間がかかるかもしれませんが、このスクリプトはカスタムオブジェクトのスクリプト作成処理を非常に簡単にしてくれます。

```
//-------------------------------------------------
// Mesh2Custom by Bob Hood
//
// 現在可視状態にあるレイヤー(複数可)にあるメッシュを
// Layout Custom Object に変換する LScript
//
@warnings
@version 2.3

main
{
  vectors = nil;
  indices = nil;
  pointcount() || error("I need some mesh to work with!");
  reqbegin("Mesh2Custom by Bob Hood");
    c1 = ctlfilename("Script file","mycustobj.ls");
    return if !reqpost();
    filename = getvalue(c1);
  reqend();

  // 配列を構築します
  editbegin();
    foreach(poly,polygons)
    {
      pnts = polyinfo(poly);
      m = pnts.count();
      for(x = 2;x < m;x++)
      {
        vec1 = pointinfo(pnts[x]);
        vec2 = pointinfo(pnts[x + 1]);
        ndx1 = 0;
```

```
      ndx2 = 0;
      if(!vectors)
      {
        // 配列の初期化を行います
        vectors += vec1;
        vectors += vec2;
        ndx1 = 1;
        ndx2 = 2;
      }
      else
      {
        // 配列内における各ベクトル用のインデックスを検索します
        for(y = 1;y <= vectors.size();y++)
        {
          if(vectors[y] == vec1)
            ndx1 = y;
          if(vectors[y] == vec2)
            ndx2 = y;
        }
        if(!ndx1)
        {
          vectors += vec1;
          ndx1 = vectors.size();
        }
        if(!ndx2)
        {
          vectors += vec2;
          ndx2 = vectors.size();
        }
      }
      indices += ndx1;
      indices += ndx2;
    }
  }
editend();
// 重複インデックス値を最適化します
```

```
    lines
    x = 1;
    while(x <= indices.size())
    {
      if(indices[x])
      {
        y = x + 2;
        while(y <= indices.size())
        {
          if(indices[y])
          {
            if((indices[x] == indices[y] && indices[x+1] ==
indices[y+1]) || (indices[x] == indices[y+1] && indices[x+1]
== indices[y]))
            {
              indices[y] = nil;
              indices[y+1] = nil;
            }
          }
          y += 2;
        }
      }
      x += 2;
    }
    indices.pack();
    indices.trunc();
    //  配列データからカスタムスクリプトを生成します
    file = File(filename,"w") || error("Couldn't open output
file!");
    file.writeln("@warnings");
    file.writeln("@version 2.3");
    file.writeln("@script custom"); file.nl();
    file.write("vert = @ ");
    for(x = 1;x <= vectors.size();x++)
    {
      if(x > 1)
```

```
      {
        file.writeln(",");
        file.write(" ");
      }

file.write("<",vectors[x].x,",",vectors[x].y,",",vectors[x].z,"
>");
    }
    file.writeln(" @;"); file.nl();
    file.write("edge = @ ");
    for(x = 1;x <= indices.size();x++)
    {
      if(x > 1)
      {
        file.write(",");
        if(x & 1)
        {
          file.nl();
          file.write(" ");
        }
      }
      file.write(indices[x]);
    }
    file.writeln(" @;"); file.nl();
    file.writeln("process: ca");
    file.writeln("{");
    file.writeln(" for(x = 1;x < ",indices.size() + 1,";x +=
2)");

file.writeln("ca.drawLine(vert[edge[x]],vert[edge[x+1]]);");
    file.writeln("}");
    file.close();
  }
```

第 14 章：プロシージャルテクスチャ

プロシージャルテクスチャもしくはシェーダーは、オブジェクトのサーフェイス上に現れますが、実際にはオブジェクトに存在していません。LightWave はレンダリング時においてプロシージャルテクスチャを計算するため、最終画像において表示されます。

LScript の **Procedural Texture（LS-PT）** スクリプトを使うと、LightWave がサーフェイスのレンダリングをコントロールできるようになります。

シェーダー関数

他のレイアウトスクリプトと同様、LS-PT のために特別に設計されている関数が 2 ～ 3 あります。これらの関数は幾つかの使い道があります。シェーダーを初期化したりスクリプトが現在動作しているフレームを決定したり、また `flags()` UDF の場合には、スクリプトで必要となる特定のバッファだけをリクエストします。

init()

`init()` は各レンダーシーケンスの最初に一度だけ呼び出されます。レンダー処理が始まる前に設定が必要となるスクリプトの値は、ここで初期化しなくてはなりません。

cleanup()

`cleanup()` はレンダーシーケンスの終了時に呼び出されます。変数の破棄処理を行います。

newtime(frame, time)

`newtime()` はカレントのレンダーシーケンスにおいて各新規時刻の開始時に呼び出されます。整数値 `frame` はカレントのフレーム番号を、数値 `time` はカレントのタイムインデックスを表します。フレーム番号とは違い、タイムインデックスは各フレームから生成される浮動小数点数値です。タイムインデックスはシーンに対するフレーム / 秒設定に依存しています。

例：

シーン中で24フレーム/秒に設定されている場合、フレームに対するタイムインデックスは以下のようになります。

```
フレームタイムインデックス(24フレーム/秒)
1/24 もしくは .042
2/24 もしくは .083
3/24 もしくは .125
```

シーン中で30フレーム/秒に設定されている場合、フレームに対するタイムインデックスは以下のようになります。

```
フレームタイムインデックス(30フレーム/秒)
1/30 もしくは .033
2/30 もしくは .066
3/30 もしくは .1
```

flags()

LS-IFと同様、LS-PTスクリプトもレイアウトに対しサーフェイステクスチャのどの属性に対し修正を加えるのかを指定しなくてはなりません。スクリプトは`flags()`関数から以下の値を一つもしくは複数返します。

```
NORMAL
COLOR
LUMINOUS
DIFFUSE
SPECULAR
MIRROR
TRANSPARENT
ETA
ROUGHNESS
RAYTRACE
```

LS-PT 関数とデータメンバ

LS-PTの主な処理の核は`process()`関数です。プロシージャルテクスチャでは`process()`関数はピクセル単位に呼び出されます。

LS-PTは`process()`関数に対し引数を一つ提供します。この引数は`Shader`と呼ばれるもう一つのObject Agentです。Shader Object AgentにはレイアウトLScriptにおけるシーン用Object Agentのように、シェーダーに関する情報を教えてくれるメソッドやデータメンバが含まれています。

Shader Object Agentsは影響を与えるサーフェイスに関する必要な情報全てを提供します。これらの値のうち、制御したりサーフェイスを修正出来るものも幾つかあります。その他は情報を与えるだけで修正は出来ません。

まずは読み取り専用の情報を見ていきましょう。

`sx`は最終画像におけるピクセル座標値のスポットのX座標値を表す数値です。(0,0)は左上隅の座標値となります。

`sy`は最終画像におけるピクセル座標値のスポットのY座標値を表す数値です。(0,0)は左上隅の座標値となります。

`oPos[3]`はオブジェクト座標におけるスポットの座標位置を表す浮動小数点数値です。座標値ですからX,Y,Zを表す三つの数値を返します。

`wPos[3]`はワールド座標におけるスポットの座標位置を表す浮動小数点数値です。座標値ですからX,Y,Zを表す三つの数値を返します。

`gNorm[3]`はワールド座標におけるスポットのジオメトリ法線を表す浮動小数点数値です。これはスポットにおける無修正のポリゴンの法線であり、スムージングやバンプマッピングなどによって修正される前の法線情報です。座標値ですからX,Y,Zを表す三つの数値を返します。

`spotSize`はスポットの直径の近似値をあらわす数値です。エッジ上に見えるサーフェイス上のスポットは長く薄くなっているので近似値となっています。テクスチャアンチエリアシングを計算する場合などに使用可能です。

`raycast()`は、本来`raytrace()`と同じパラメータを受け取る速度向上版の関数ですが、レイの長さしか返しません。シェーディングは評価されることはなく、レイトレースの再起も行われません。

`raySource[3]`はワールド座標における視線レイが発生する起点を表す浮動小数点数値です。カメラになる場合が多いでしょうが、カメラである必要はありません。座標値ですからX,Y,Zを表す三つの数値を返します。

`rayLength`は視線レイが空間を通ってこのスポットに到達するまでの距離を表す値です。

`cosine` はスポットにおいて視線レイとサーフェイス法線との角度の余弦を表す値です。これはビューの跳ね返りを示し、スポットの近似サイズを測定します。

`oXfrm[9]` はオブジェクトからワールド座標変換行列を表す浮動小数点数値です。この行列は他の方法でも計算可能ですが、ここでは速度用にメソッドであり、主として方向ベクトル用に使用するために用意されています。座標値ですから X,Y,Z を表す三つの数値を返します。

`wXfrm[9]` はワールドからオブジェクト座標変換用の行列です。この行列は他の方法でも計算可能ですが、ここでは速度用にメソッドであり、主として方向ベクトル用に使用するために用意されています。

`objID` はシェーディングされるオブジェクトを表す Object Agent へのポインタです。

`polNum` はシェーディングされるオブジェクトのポリゴン数を表す整数値です。これは通常のメッシュオブジェクトの場合にはポリゴン数を指しますが、非メッシュオブジェクトの場合には他のサブオブジェクト情報を表現しています。

```
illuminate(light,position) -> array[6]
```

`illuminate` 関数は、カレントインスタンスにおいて指定したライトから指定した位置に当たるライトのレイを表す 6 個の数値の配列（色[1-3]と方向[4-6]）を返します。指定したワールド座標値にライトが全く照射されていない場合にはゼロが返されます。色には（もし存在するならば）陰やフォールオフ、スポットライトコーン、ライト、それにポイントの間にある透明なオブジェクトの効果が含まれます。

```
raytrace(position,direction) -> array[4]
```

`raytrace` 関数は指定した位置から指定した方向に向かう（ワールド座標で）レイを追跡するために呼び出されます。関数はレイの長さを表す要素[1]とその方向からくる色[2-4]を表す 4 つの要素を含む配列を返します。レイの長さが無限であれば-1.0 を返します。使用される `direction` は出て行くレイの方向であり、単位ベクトルに正規化されている必要があります。

これらの値を読み込み使用することは出来ますが、変更することは出来ません。読み込むだけでは面白くないので、LS-PT では好きなようにサーフェイスの値を修正することもできるようにしています。修正可能な値は下記に限定されています。アイテムの大半の名称はレイアウトにおけるコントロール名称に対応していますので、レイアウトにおいて色・質感編集になれていれば、ここでも名称をすぐに探し出せるでしょう。

`wNorm[3]` はワールド座標におけるスポットの新規ジオメトリ法線です。この値を修正することで、ジオメトリを修正することなくサーフェイスをバンプ状に見せることが出来ます（バンプマッピング）。シェーダーは修正後にベクトルを正規化しておく必要があります。

`color[3]` はサーフェイスカラーを表す赤[1]、緑[2]、青[3]のパーセンテージを表す数値です。値 1.0 は 100% に相当します。

`luminous` はサーフェイスの自己発光度を表す数値です（1.0 は 100%）。

`diffuse` はサーフェイスの拡散レベルを表す数値です（1.0 は 100%）。

`specular` はサーフェイスの反射光を表す数値です（1.0 は 100%）。

`mirror` はサーフェイスの鏡面反射率を表す数値です（1.0 は 100%）。

`transparency` はサーフェイスの透明度を表す数値です（1.0 は 100%）。

`eta` はサーフェイスの屈折率を表す数値です（1.0 は 100%）。

`roughness` はサーフェイスの粗さを表す数値です（1.0 は 100%）。

例：(PULSE2.LS)

LS-PT スクリプトを通してどのようにサーフェイスの値を修正するかを示した基本的な例を見ていきましょう。

これは Pulse2.ls というスクリプトです。サーフェイスの自己発光度と拡散レベルの値を受け取り、車のウィンカーのように規則的に明滅を起こします。わずかなインターフェイスで明滅の長さを調節し、ユーザーはサーフェイスの拡散レベルチャンネルを保護するかどうかを選択できます。

```
// Pulse2
//
// サーフェイスの自己発光度と拡散レベルを(交互に)反復することで
// サーフェイスを"波動"のように見せます
//
// 更新日 04.21.98 Bob Hood
// 更新日 07.20.01 Scott Wheeler
```

スクリプトのトップ部分はスクリプト名称や重要な情報を置いておくコメントの領域です。'//' 文字はその行がコメント行であると識別し、スクリプト実行時においてライン上のテキストは全て無視されます。これらの文字はスクリプトから2～3行を完全に削除するのではなく省きたい場合にとても便利です。その部分をコメント文にしてしまえばよいのですから。

```
@version 2.3
```

'@'文字で始まるコマンドはコンパイラ指示子です。コンパイラ指示子はコンパイラに対しスクリプトのパフォーマンスをあげたり、スクリプト実行に必要となるある特定の条件のコンパイラを修正する場合に指示します。

'@version'指示子はスクリプトを適正に実行させるために必要となる LScript の最低バージョンをコンパイラに指示します。新規コマンドセットを使用しているスクリプトにはこの指示子を必ず入れるようにして下さい。指示子はユーザに対しスクリプトが機能するためには LScript をアップグレードする必要があるかどうかをユーザに伝えるからです。

コンパイラ指示子はスクリプトのどの箇所でも配置することも可能なのですが、組織化し探しやすくするためにもスクリプトの一番上にコンパイラ指示子を全て配置しておくのが良いでしょう。

```
thisFrame;
pulserange = 2;
prog = "Pulse v1.01";
keepdiffuse = true;
```

次に大域変数とする変数を宣言します。大域変数はまるで関数の一部であるかのように全スクリプト関数から可視状態になります。この変数はローカル変数とは対照的であり、ローカル変数は関数内部においてのみ宣言されます。ローカル変数は宣言されている関数以外の関数からは見ることは出来ません。ループ処理はどのようにローカル変数が動作するのかを示す良い例です。関数内部において小さなループ処理を行っている場合、ループに使用される変数は関数外部から見える必要はありません。この目的はループ自身を越えて拡張することではないからです。

各大域変数の目的を理解する必要がありますから、それぞれ一つずつ見ていきましょう。

`thisFrame` にはカレントのフレーム番号が入っています。

`pulserange` は各パルスの長さを割り当てる初期値です。後ほど見ますが、スクリプト実行前に他の値を代入します。しかしスクリプトの最初に配置しておいて、スクリプトを通して検索し追跡しなくてもよいように中央に集めておけば修正が可能になります。

`prog` は現在のスクリプトのバージョンを保持する文字変数です。スクリプトにおいて特に重要な部分ではありませんが解説文の設定（後ほど確認）がより容易になります。

`keepdiffuse` はユーザによって設定された拡散レベルの値の情報を保護しておくかどうかを指定するブール値です。スクリプトの後の方で、サーフェイスの拡散レベルの値を保持または上書きすることになります。

```
create
{
  scene = Scene();
  pulserange = scene.fps;
  setdesc(prog+" is Pulsing on "+ string(pulserange)+" Frame
Intervals");
}
```

`create()` 関数はスクリプトが最初に読み込まれた時点で呼び出される関数なので、初期設定を行うには最適の場所です。`create()` 関数は `init()` 関数とは異なります。`init()` は各レンダーシーケンスの最初に呼び出されますが、`create()` 関数はスクリプトが読み込まれたとき、またはLightWaveが起動されたときに呼び出されます。

`create()` 関数の第１行は大切な行です。このたった一行の中にたくさんの情報が変数 '`scene`' に渡されます。`Scene()` 関数を使用することでScene Object Agentを作成しています。これでカレントシーンを処理するいくつものデータメンバやメソッドにアクセスできるようになります。

注意
Scene()の動作法や利用可能なタイプについては、リファレンスマニュアルのScene Object Agentsの章を参照してください。

Scene Object Agent の作成時には、カレントシーンに対し利用可能な情報全てが変数'scene'に保存されています。アクセス可能な情報の一つとしてシーンのフレーム/秒（fps）があります。ですからフレーム/秒（fps）データメンバを代入することで、変数 pulserange の設定が可能になります。

create()関数内の最後の行は setdesc()関数の呼び出しです。この関数ではシェーダーリスト内にスクリプトの値を表示した解説行を作成できます。ここではスクリプトの名称を記入してある変数'prog'を使用しましょう。オプションパネルを開かずにスクリプトがどのように動作するのかについてフィードバックを与えられるよう、setdesc()関数を使用することが頻繁にあります。スクリプトが始めて実行されると、シーンの fps が 24 に設定されている場合、解説文は以下のように表示されます。

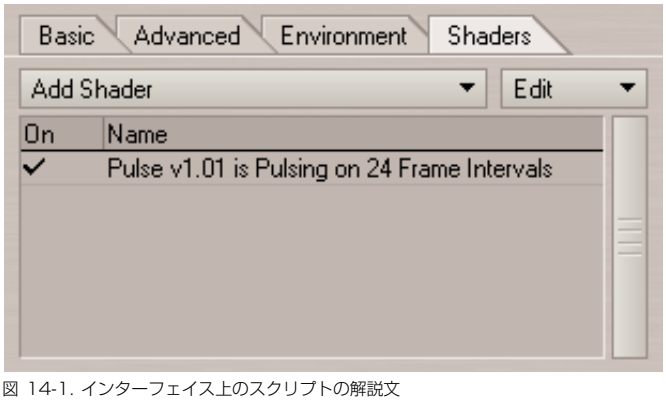

図 14-1. インターフェイス上のスクリプトの解説文

```
newtime: frame, time
{
  thisFrame = frame;
}
```

newtime()関数は新しいフレーム毎に一回ずつ呼び出され、開発者はフレームに基づき情報を取得したり変更することが出来ます。今回はカレントのフレーム情報を取得し、大域変数'thisFrame'に代入します。

```
flags
{
  return(LUMINOUS,DIFFUSE);
}
```

flags()関数では、LScript に対しサーフェイスのどの属性に対し影響を及ぼすのかを正確に指示します。flags()関数を追加しなくてもスクリプトは実行されますが、サーフェイスのどの属性の値を修正するのかコンパイラにきちんと伝える方がメモリにそのバッファだけを読み込みますので、はるかに効率的です。

今回の例ではサーフェイスに対しLUMINOUSとDIFFUSEの値を修正します。ですからこの二つのサーフェイスの属性だけが利用可能になるように指示します。

次の関数は`process()`関数です。`process()`関数はスクリプト実行を全て処理する場所です。LScriptのProcedural Textureでは、`process()`関数はイメージの各ピクセルに一回ずつ呼び出されます。`process()`関数は`newtime()`の後に呼び出されるため、フレームが変更されることにより、集められた情報を使用出来ます。

```
process: sa
{
  if(thisFrame == 0)
  return;
```

`process()`関数でShader Object Agentにアクセスすることが出来ます。このAgentは関数へと渡される変数'`sa`'です。ですからサーフェイスに関する情報を得たいとき、またサーフェイスの値を変更したいときには毎回、'`sa`'変数を通して処理することになります。これは関数内のより下のところで意味をなしてきます。

ゼロで除算してしまうなどのエラーを避けるために、このスクリプト処理では0フレームを除外しています。カレントフレームを判定し、0であれば`return`コマンドを呼び出して`process()`関数から抜け出します。

```
j = thisFrame;
if(thisFrame > pulserange)
{
  i = integer(thisFrame / pulserange);
  if(i != 0)
  {
    j = thisFrame - (i * pulserange);
  }
  k = pulserange / 2;
  if(j <= k)
  {
    // 明滅をオンにします
    sa.luminous = j / k;
    // 逆の値を生成します
    if (!keepdiffuse) sa.diffuse = 1.0 - (j / k);
  }
  else
  {
```

```
        // 明滅をオフにします
        // パーセンテージが正しくあるように折り曲げます
        j = j - k;
        sa.luminous = 1.0 - (j / k);
        if (!keepdiffuse) sa.diffuse = j / k;
    }
  }
```

上記コード部分は一見したところでは複雑に見えますが、全体を見て、それから細かく分けていきましょう。このコードでは現在のフレームとパルスの長さを読み込むことで、時間が経つにつれてサーフェイスの自己発光度と拡散レベルの値を変更していきます。では処理を二つの部分、パルス総時間の半分である明るい状態と、再び暗くなった状態とに分けて見ていきましょう。

拡散レベルの値はブール変数'keepdiffuse'によって保護されるかどうかに気をつけてください。これは自己発光度が増したときに、サーフェイスの拡散レベルを下げるのか、それともそのままの状態にしておくのかをコントロールするユーザー定義の値です。

コードセクションがどのように動作するのかを見るには、関係する変数全てによる表を構築し、時間の経過と共にそれらの変数がどのように変化していくのかを見てみるのが良いでしょう。下記の例においてサンプルフレームの表を作りました。この表ではパルスのレートを20フレームと仮定し（10フレームアップ、10フレームダウン）、自己発光度と同様にサーフェイスの拡散レベルも修正しましょう。

frame	*pulserange*	*j*	*k*	*sa.luminous*	*sa.diffuse*
5	20	j = this-Frame = 5	10	j / k = 0.5	1.0 - (j / k) = 0.5
7	20	j = this-Frame = 7	10	j / k = 0.7	1.0 - (j / k) = 0.3
16	20	j = (j-k) = 6	10	1.0 - (j / k) = 0.4	(j / k) = 0.6
20	20	j = (j-k) = 10	10	1.0 - (j / k) = 0.0	(j / k) = 1.0

このように2～3個のフレームで実行してみれば、最終的にどのようになるのかがわかるでしょう。実行する時には条件式を忘れずに全て処理するようにしてください。`if`文を誤ると、途中で値が変更してしまうからです。

```
save: what, io
{
  if (what == OBJECTMODE)
  {
    io.writeNumber(pulserange);
    io.writeNumber(keepdiffuse);
  }
}
```

`save()`関数を使用するとオブジェクトファイルの中にシェーダー情報を保存することが出来ます。今回の場合、変数'`pulserange`'と'`keepdiffuse`'の値を保存しましょう。

`save()`関数には二つの個別のモードがあります。`SCENEMODE`ではプラグイン情報をシーンファイルへと保存し、`OBJECTMODE`ではプラグイン情報をオブジェクト自身へと保存します。シェーダー情報は直接LightWaveのオブジェクトファイルへと保存されますので、このスクリプトでは`OBJECTMODE`を使用します。

'`io`' Agentを使用して、情報をオブジェクトファイルへと追加します。ファイル書き込みのコマンドに関してはリファレンスマニュアル第1章：共通コマンドの章を参照してください。

```
load: what,io
{
  if (what == OBJECTMODE)
  {
    pulserange = io.readNumber();
    keepdiffuse = io.readNumber();
  }
  setdesc(prog + " is Pulsing on "+string(pulserange) + "
Frame Intervals");
}
```

`load()`関数は`save()`関数とよく似ています。PULSE2.LSが適用されているオブジェクトが読み込まれると、`load()`関数が呼び出され、`save()`関数を使用して保存しておいた情報を取得します。今回の場合、この特定のサーフェイスに対し保存されていた`pulserange`と`keepdiffuse`の値を読み戻します。

また、ユーザーにフィードバックを与えるための処理も加えましょう。`create()`関数内で`setdesc()`が呼び出されるため、ユーザにはデフォルト値が表示されることになります。特定のサーフェイスに対する値がデフォルト値と異なる場合、読み込み時には表示されません。オブジェクトからサーフェイスの値を読み出した後で`setdesc()`を読み出せば、この問題を解決します。ユーザーがインターフェイスを開かなくても、シェーダープラグインリストの中に適切な値が表示されるようになります。

コードの最終部分は`options()`関数です。これはインターフェイス作成部分です。関数`options()`は、ユーザーがシェーダー用のオプションポップアップから選択したときに呼び出されます。

ここではインターフェイスの作成法については解説しませんが、2～3点指摘しておくことがあります。

```
options
{
  reqbegin(prog);
  reqsize(314,69);
```

`reqbegin()`コマンドはインターフェイス開始を示し、残りのコントロールに対するステージを設定します。`reqbegin()`に渡す値はウィンドウのタイトル領域に表示する名称です。これがスクリプト名称変数を作成した置いたもう一つの理由です。コードの一番上の箇所で変数の名称を変更することでウィンドウの名称も同時に変更することが可能なのです。コード処理という意味では特別面白い箇所ではありませんが、更新時間を縮める利点があります。

```
  c1 = ctlminislider("Pulse Range",pulserange,1,100);
  ctlposition(c1,13,12);
  c2 = ctlcheckbox("Keep Diffuse",keepdiffuse);
  ctlposition(c2,201,12);
  return if !reqpost();
  pulserange = getvalue(c1);
  keepdiffuse = getvalue(c2);
  reqend();
  setdesc(prog + " is Pulsing on "+string(pulserange) + "
Frame Intervals");
}
```

スクリプトのインターフェイス部分において指摘しておきたい最後の要素は終わり2行にあります。`reqpost()`関数はユーザーが"**OK**"ボタンまたは"**Cancel**"ボタンを押したときにブール値を返します。`reqend()`関数は実際にパネルを閉じるときに呼び出されます。

最後にもう一度`setdesc()`関数を呼び出して新しい値を表示設定しましょう。LightWaveのCD-ROMにはPulse2.lsスクリプトが入っています。ここまで解説を読みながら少しずつコードを入力してきた方のために、以下に完全なコードを提供します。間違いがないか、一通りチェックしてみてください。

PULSE2.LS

```
  // Pulse2
  // サーフェイスの自己発光度と拡散レベルを(交互に)反復することで
  // サーフェイスを"波動"のように見せます
  //
  // 更新日 04.21.98 Bob Hood
  // 更新日 07.20.01 Scott Wheeler
  @version 2.3
  thisFrame;
  pulserange = 2;
  prog = "Pulse v1.01";
  keepdiffuse = true;

  create
  {
    scene = Scene();
    pulserange = scene.fps;
    setdesc(prog + " is Pulsing on " + string(pulserange) + "
Frame Intervals");
  }

  newtime: frame, time
  {
    thisFrame = frame;
  }

  flags
  {
    return(LUMINOUS,DIFFUSE);
  }
```

```
process: sa
{
  if(thisFrame == 0)
  return;
  j = thisFrame;
  if(thisFrame > pulserange)
  {
    i = integer(thisFrame / pulserange);
    if(i != 0)
    j = thisFrame - (i * pulserange);
  }
  k = pulserange / 2;
  if(j <= k)
  {
    // 明滅オン
    sa.luminous = j / k;
    if (!keepdiffuse) sa.diffuse = 1.0 - (j / k);
  }
  else
  {
    // 明滅オフ
    j -= k;
    sa.luminous = 1.0 - (j / k);
    if (!keepdiffuse) sa.diffuse = j / k;
  }
}

save: what, io
{
  if (what == OBJECTMODE)
  {
    io.writeNumber(pulserange);
    io.writeNumber(keepdiffuse);
  }
}
```

```
  load: what, io
  {
    if (what == OBJECTMODE)
    {
      pulserange = io.readNumber();
      keepdiffuse = io.readNumber();
    }
    setdesc(prog + " is Pulsing on " + string(pulserange) + "
Frame Intervals");
  }

  options
  {
    reqbegin(prog);
      reqsize(314,69);
      c1 = ctlminislider("Pulse Range",pulserange,1,100);
      ctlposition(c1,13,12);
      c2 = ctlcheckbox("Keep Diffuse",keepdiffuse);
      ctlposition(c2,201,12);
      return if !reqpost();
      pulserange = getvalue(c1);
      keepdiffuse = getvalue(c2);
    reqend();
    setdesc(prog + " is Pulsing on " + string(pulserange) + "
Frame Intervals");
  }
```

第 15 章：イメージフィルタスクリプト

Image Filterスクリプトを使用すると、LightWaveのレンダリング出力に対し素晴らしい後処理コントロールが出来ます。画像を反転するだけの簡単なスクリプトから、望みどおりの効果を得るのに必要となる複雑なスクリプトまで作成可能です。

この章での最終目標は、Image Filterシステムの動作についての基本的な理解を深めることにあります。ですからImage Filterコントロールの簡単な側面をより重視します。さらに高度なスクリプトはLightWaveのCD-ROMやNewTekのWebサイトにおいてありますので、これらのソースを利用すると、Image Filterコントロールをより深く掘り下げていくのに役立つでしょう。

イメージバッファ

LightWaveからレンダリングされた画像には割り当てられた豊富なデータがあり、このデータはバッファという一連のスポットに保存されています。各バッファには画像内の個別のエレメントに関する情報が入っています。例えば、`LUMINOUS`バッファ内にあるデータを調べることで、ピクセルの自己発光度を見ることが出来るのです。

このデータ全てが読取専用であるわけではありません。2～3のバッファに対してはデータが読取専用とは限りません。後処理効果として振舞えるように特定のピクセルをコードで変更することが可能です。バッファされている情報を使用して後々合成ソフトなどで操作できるよう自己発光度のピクセルだけを個別の画像として保存することも可能なのです。

Image Filter Object Agentで利用可能なデータメンバとメソッドの完全なリストについては、リファレンスマニュアル第23章：Object Agent Referenceを参照してください。

Image Filter 関数

Image Filterスクリプトは、大半のLScriptのレイアウトスクリプトタイプと同じように設定します。開発者が機能追加用に使用する事前定義関数がいくつか含まれています。以下、これらの関数とImage Filterスクリプト内における役割を記していきます。

create() と destroy()

`create()`関数はスクリプトが最初に起動したときに呼び出されます。ユーザーが最初にスクリプトを起動したとき、またはシーンから読み込まれた時にも呼び出されます。通常、変数の初期化とコマンドの設定がここで行われます。

`destroy()`関数はユーザーによってアクティブな処理からスクリプトが除去されたとき、またスクリプトが割り当てられていたシーンがクリアされた時に呼び出されます。またこの関数の中で変数のクリーンアップを行う場合もあります。他の関数と同様、関数内に何のコードもなければ、スクリプトに含める必要はありません。

process()

`process()`関数はスクリプトの中では馬車馬のような関数です。LScriptは各フレームがレンダリングされた後に自動的にこの関数を呼び出します。LightWaveの内部イメージバッファ（上記参照）を使用することで、開発者はレンダリングされたフレームのイメージにアクセスし、ピクセルを修正するコードを実行し、イメージにもう一度値を戻したり、後で使用するために他へ保存することが容易に可能になります。これらのアクションは全てスクリプトの`process()`関数内で起こることです。

`process()`関数は、Image Filter Object Agent内に含まれているデータメンバやメソッドを通してフレームのバッファ情報にアクセスします。このObject AgentはLScriptから呼び出されるといつでも自動的に`process()`関数へと渡され、Image Filterスクリプトで必要とされる情報を全て保持しています。

```
process: ifo
{
}
```

引数`ifo`はImage Filter Object Agentです。このObject Agentを通して、利用可能なバッファの一つに情報を集めたり設定したり、またシーン特有のデータメンバを問い合わせたり、データを新規バッファへコピーペーストが出来たりします。

load() と save()

`load()`と`save()`関数を使うと、後々使用出来るようにシーンファイル内部に情報を保存しておくことが出来ます。オプションの状態、ユーザーカラーなど、後で読み込みなおしたい情報は何でも保存可能です。

例えば、レンダー時にスレートを作成するImage Filterを作成したとします。shotの名称、showの名称、takeなどを保存したくなるかもしれません。以下で情報を保存できます。

```
save: what, io
{
  io.writeln(shot);
  io.writeln(show);
  io.writeln(take);
  io.writeln(episode);
  io.writeln(date);
  io.writeln(slateframe);
}
```

`save()`関数はレイアウトから渡される二つの引数を受け取ります。最初の引数は、レイアウトがこの関数を現在呼び出している時点で二つの保存モードのうちのどちらかのモードを指定します。値が`SCENEMODE`であれば、ユーザーはシーンの保存を選択したことになります。しかし`OBJECTMODE`の場合には、スクリプトが適用されているオブジェクトを保存するように指示することになります。

どのモードで作業するのかを把握するのは重要なことです。`SCENEMODE`で保存する値は文字列（ASCII）形式、`OBJECTMODE`ではバイナリの値が要求されます。本来、引数は`SCENEMODE`の値を指定するのがほとんどです。`OBJECTMODE`でデータを保存するのは、シェーダースクリプトを書くときのみです。他のLScriptクラスではオブジェクト自身に割り当てられるタイプはありません。

`save()`関数へ渡される引数`io`はObject Agentです。これはシーンファイルに保存されるスクリプトのデータ行にアクセスするために使用されます。`writeln()`メソッドを使用して文字列データを保存するシーンファイルへと埋め込みます。いったん保存された情報から値を取り出すには、`load()`関数を使用してください。

```
load: what,io
{
  shot = io.read();
  show = io.read();
  take = io.read();
```

```
    episode = io.read();
    date = io.read();
    slateframe = integer(io.read());
  }
```

構造上、この関数は前述の`save()`関数と同じになります。`load()`は`save()`で使用されている引数と同じですが、情報を保存する`io` Object Agentを使用する代わりに、シーンファイルからデータを読むために`io`の`read()`メソッドを使用します。

注意
load()とsave()関数の双方において、データの割り当てや収集に使用される変数はグローバルスコープとなります。

options()

`options()`関数は、プラグイン用のインターフェイスを作成します。このインターフェイスはユーザーがプラグインのプロパティを選択した時に表示されるものです。プラグイン用のインターフェイス作成に関する詳細な情報は、このマニュアルの第24章：LSIDEを参照してください。

例：Black and White

作成する最初のスクリプトは本当に基本的なものです。画像からRGBの値を取得し平均化して黒白画像を作成するものです。

まずはLScript Editorを開きます。エディターを使えば、スクリプトを簡単に作成できる素晴らしいツールをいくつも使用できます。これらのツールの一つはツールメニューの下にあるテンプレート機能です。テンプレートを選択しImage Filterをハイライトにします。

ready-to-useな状態にあるテンプレートを読み込み、そのテンプレートを使用してImage Filterスクリプトを作成します。これでスクリプトの形式を覚えておく必要がなくなります。テンプレートが教えてくれるからです。しかし完成するまでにテンプレートからコードをいくつか変更したり削除する必要があります。それでも基礎は出来上がるのです。

```
// ---------------------------
// LScript Image Filter template
//
```

デフォルトのスクリプトにある最初の数行はスクリプトのコメントや名称を挿入するための領域です。`//`で始まる行はコメントです。この行の後にあるテキストは全てコマンドとして評価されることはありません。

以下の行を作成するスクリプトの名称で置き換えることにしましょう。

```
// ----------------------------
// Black and White
//
```

メインコメントの後に一連のコンパイラ指示子が続きます。この時点で私たちに関係する指示子は`@script`指示子のみです。このコマンドはスクリプトがLightWaveに追加された時にスクリプトの種類をLightWaveが分類できるようにします。

```
@version 2.5
@warnings
@script image
```

ご存知の通り、次のグループの最初の行はコメントです。このコメントではユーザー定義関数（UDF）外で定義されている変数は全てグローバルであり、スクリプト全体で利用可能であることを示しています。画像の幅と高さの値をここで定義しますので、後々`process()`コードにおいてこれらの変数や配列を定義しなくてはなりません。`process()`関数が呼び出される前に画像のサイズを割り出すことは出来ないからです。

```
// ここに大域変数
```

大域変数を初期化した後は、スクリプトが開始また終了した時点で何をするのかをLScriptに伝えるビルトイン関数を呼び出さなくてはなりません。

```
create
{
  // ここで一度初期化を行います
}
```

`create()`関数はスクリプトが最初に起動されたときに呼び出されます。ユーザーから最初に起動された時、またはシーンから読み込まれた時にも呼び出されます。ここでスクリプトの名称を設定するのがよいでしょう。

```
create
{
  setdesc( "Black and White" );
}
```

`setdesc()`コマンドはImage Filterプラグインリスト内のタイトル領域に配置するテキスト値を受け取ります。デフォルトのLScriptタイトルを独自のタイトルに置き換えることで、よりプロフェッショナルなスクリプトに見えます。この例ではスクリプトに"Black and White"という名前を付けましょう。どんな名前を付けても構いません。

理論上、`destroy()`は`create()`の後に来ます。スクリプトが処理を終えた後の大掃除をここで行います。今回のスクリプトではこの機能は必要としませんので、スクリプトから以下の行を削除することが出来ます。

```
destroy
{
  //  ここで気をつけて最終クリーンアップ処理を行います
}
```

前述の通り、`process()`関数は処理する画像に関する重要なデータをObject Agentの形式で渡します。まずはじめに見たいのはImage Filter Object Agentのデータメンバ`width`と`height`に表されている、レンダリングされた画像のサイズです。これらの値を使用してイメージ全体を処理を確実にするための変数のループ処理をコントロールします。

```
process: ifo
{
  //  画像の幅と高さの情報を取得します
  imgWidth = ifo.width;
  imgHeight = ifo.height;
}
```

集められたこのデータを使って、イメージ内の全ピクセルを通して処理出来ます。二つの`for`ループを作成し、処理を行います。一つは画像のX座標値を、もう一つはY座標値を反復処理します。まずはYから始めましょう。

```
imgHeight = ifo.height;
// y 座標値のループ処理
for(i = 1;i <= imgHeight; ++i)
{
}
...
```

次に X の反復処理です。

```
for(i = 1;i <= imgHeight; ++i)
{
  // x 座標値のループ処理
  for(j = 1; j <= imgWidth; ++j)
  {
  }
}
```

この二つのループを通して画像内のピクセル毎にデータを読み込んだり書き込んだりします。白黒イメージを作成するために使用する数学の方程式は、至って簡単なものです。各ピクセルに対しまずは赤、緑、青の三つの値を平均化し、ピクセルの合計値を変数 `avg` に代入します。

```
for(j = 1; j <= imgWidth; ++j)
{
  // 平均値を計算します
  avg = (ifo.red[j,i] + ifo.green[j,i] + ifo.blue[j,i]) /
3;
}
```

平均値が決定すれば、`avg` の値を各チャンネルの値へと置き換えます。

```
avg = (ifo.red[j,i] +
ifo.green[j,i] + ifo.blue[j,i]) / 3;
// 平均値をカラーチャンネルに代入します
ifo.red[j,i] = avg;
ifo.green[j,i] = avg;
ifo.blue[j,i] = avg;
```

赤、緑、青の配列位置を返し、グレースケール値を作成します。

では以下に現在のコードを全て書き出してみましょう。

```
// ----------------------
// Black and White
//
@version 2.5
@warnings
@script image
create
{
  setdesc( "Black and White" );
}
process: ifo
{
  // 画像の幅と高さの情報を取得します
  imgWidth = ifo.width;
  imgHeight = ifo.height;
  // y 座標値のループ処理
  for(i = 1;i <= imgHeight; ++i)
  {
    // x 座標値のループ処理
    for(j = 1; j <= imgWidth; ++j)
    {
      // 平均値を計算します
      avg = (ifo.red[j,i] +
      ifo.green[j,i] + ifo.blue[j,i]) / 3;
      // 平均値をカラーチャンネルに代入します
      ifo.red[j,i] = avg;
      ifo.green[j,i] = avg;
      ifo.blue[j,i] = avg;
    }
  }
}
```

進行モニターの追加

上記画像処理の例は極めて簡単な例ですが、新しい画像を計算するのはLScriptにとっては依然、時間を要するものです。処理コードがより複雑になればなる程、最終画像までレンダリングするのにさらに長い時間がかかるでしょう。ユーザーは処理がどの辺りまで来ているのか知りたいものです。レンダリング状況ウィンドウ上に完成までのパーセンテージを表示するための進行モニターを表示することで、この情報を提供することが出来ます。

Black and Whiteスクリプトでは画像サイズを取得するすぐ後に、このコードを配置します。

```
imgHeight = ifo.height;
// 進行モニターを設定します
if(runningUnder() != SCREAMERNET)
  moninit(imgHeight);
```

ここでやるべきことは二つあります。一つはこの画像がレンダーファーム内にレンダリングされているのかどうかをチェックすることです。レンダリングされていれば、進行モニターを使用する必要はありません。どのみち誰も見ることは出来ないのですから。レンダーファームにレンダリングされていない場合、`moninit()`関数へ画像の高さを渡します。

`moninit()`関数を呼び出すことにより、レンダー状況スクリーンへ進行状況を通知するスクリプトを設定します。変数`imgHeight`を`moninit()`へ渡してモニターの最大カウント値を決定します。こうしてレンダー状況ウィンドウは、処理がどれくらいの割合で完成しているのかを決定します。

例えば画像が640x480であれば、変数`imgHeight`の値は480となります。`imgHeight`を`moninit()`関数へ渡すことにより、モニターの最大カウント値は480となります。ですから480のうち12行目を通過すると、レンダー状況ウィンドウは正しい割合を表示することになります。

`monstep()`関数を呼び出すことでモニターの現在のカウント数を増やせます。画像の高さが100%となるように設定していますので、この関数を高さループの終端に挿入します。そうすれば高さループが1増えるごとにモニターの値も1ずつ増えることになります。

```
ifo.blue[j,i] = avg;
}
// 進行モニターの更新
if(runningUnder() != SCREAMERNET)
  if(monstep())
    return;
```

この関数は、モニターの現在のカウントを増やすだけでなくブール値も返します。値が`true`であれば"**Cancel**"ボタンまたは**ESC**キーが押されたことを示します。これによりユーザーは長時間の処理をキャンセルできるようになります。

注意
進行モニターは全てに対し適用可能なわけではなく、レイアウトプラグイン構造によっては異なる動作を行います。

第 16 章：アイテムアニメーション

LScript の **Item Animation（LS-IA）**は、アニメーション中にオブジェクトの位置や回転、スケール値を変更し、レイアウトで計算されていた元の値を上書きします。

process()

他の大半の LScript と同様、Item Animation スクリプトの主な動作箇所は `process()`関数です。LS-IA 用の `process()`関数は、引数を三つ受け取ります。

第 1 引数'`ma`'は MotionAccess と呼ばれる Object Agent です。これはスクリプト内でユーザー定義変数として表されます。スクリプト間の整合性を取るため、'`ma`'を使用することをお勧めします。この Object Agent ではスクリプトを適用するオブジェクトに割り当てられているメソッドやデータメンバ全てにアクセスが可能です。利用可能なメソッドの完全なリストについてはこのマニュアルのメソッドの章を参照してください。

`frame` にはアニメーションの現在のフレームが入っています。

`time` は各フレームに対し生成される浮動小数点数値のタイムインデックスが入っています。このタイムインデックス値はフレーム番号と違いシーンのフレーム / 秒設定に依存しています。

例：

シーン中で 24 フレーム / 秒に設定されている場合、フレームに対するタイムインデックスは以下のようになります。

```
フレームタイムインデックス（24 フレーム/秒）
1/24 もしくは .042
2/24 もしくは .083
3/24 もしくは .125
```

シーン中で 30 フレーム / 秒に設定されている場合、フレームに対するタイムインデックスは以下のようになります。

```
フレームタイムインデックス（30 フレーム/秒）
1/30 もしくは .033
2/30 もしくは .066
3/30 もしくは .1
```

LS-IA メソッド

MotionAccess Object Agentでは利用可能な一般的なメソッドに加え、Item Animation構造には特別に二つのビルトインメソッドが用意されています。この二つのメソッドでは指定した時刻におけるモーションの属性を読み込んだり（"get"）、カレント時刻において属性を書き込んだり（"set"）することが可能です。

get(attribute,time)

get()関数を使用すると、スクリプトが適用されているオブジェクトの位置データなどを取得することが出来ます。get()関数から取得できる情報の種類は、要求する'attribute'のタイプに依存します。属性のタイプとしてPOSITION, ROTATIONそれにSCALINGが指定可能です。

例：

オブジェクトから回転情報を取得したい場合には、以下のようにget()関数を呼び出します。

```
TheObjectRotation = get(ROTATION,time);
```

get()関数の呼び出しに対しベクトルタイプを返します。戻り値は選択された属性の値を表す三つの値<x,y,z>を一組としたものです。上記例では変数'TheObjectRotation'にはオブジェクトの回転に対する<h,p,b>なる三つの値を含むベクトルの値が返されます。

また個々の名称を使用することで、ベクトルの一つの値だけにアクセスすることも可能です。オブジェクトのヘディングの情報だけが欲しい場合には、以下のようにアクセスします。

```
ObjectHeading = TheObjectRotation.h;
```

get()関数自身にも同じテクニックを適用すれば、特定のチャンネル情報をより早くより読みやすい形で取得できます。

```
ObjectHeading = get(ROTATION,time).h;
```

注意
ROTATIONは度数で値を返します。POSITIONはメートル単位、SCALINGはパーセンテージとして1.0は100%を表す値を返します。

timeは'attribute'の情報を要求する時点におけるタイムインデックスです。上記例ではカレント時刻における値を全て取得しています。しかしタイムインデックスを増減させることで、現在時刻前後の値を取得することが可能です。以下のコードでは1秒先のヘディング情報を取得できます。

```
ObjectHeading = get(ROTATION,time+1.0).h
```

set(attribute,value)

`set()`関数では'`attribute`'（`POSITION`, `ROTATION`, `SCALING`）に対し指定したベクトル値'`value`'を設定することが出来ます。これは現在時刻におけるオブジェクトのプロパティに影響を及ぼします。

`objID (読取専用)`

このプラグインが割り当てられているオブジェクトのObject Agentのポインタです。

スクリプト作成例：

ROTATER.LS

Axis Rotation Control

以下の例では、モーションの特定の軸に沿って移動する距離に従い回転値を割り当てるスクリプトを調べていきます。このタイプのコントローラでは、ボルトの回転にナットを合わせるといった使用法が考えられます。ナットがボルトの長さにあわせ上下に動くにつれ、ネジに沿って回転します。

この例は非常に基本的なタイプのオブジェクトコントロールではありますが、LScriptのItem Animationでオブジェクトのモーションをいかにたやすく拘束できるかを示すには大変好都合です。

```
//----------------------------------------
// Axis Rotation Control
//
// Scott Wheeler
// 09/12/01
@version 2.3
@warnings
@script motion
```

全スクリプトと同様、上記部分は謝辞を述べたりコンパイラ指定用に予約されています。注釈の指示子の一つに`@name`があります。`@name`を使用すると、レイアウトで保持させるスクリプトの名称を割り当てられます。`@name`で指定しない場合には、スクリプトの保存名称が代用として使用されます。

この場合、`@name`指示子がありませんから、スクリプトは単にRotaterと呼ばれます。小さな点ではありますが、LightWaveプラグインをよりプロフェッショナルに見せてくれます。

```
rotspermeter = 1.0;
controlaxis = 1;
controlpos = 1;
```

スクリプトで使用する大域変数を宣言するには、コンパイラ指示子のすぐ後が良いでしょう。大域変数はスクリプト内のどの場所からも見える変数です。この変数と同系の制限バージョンとしてローカル変数があります。ローカル変数は、特定のUDF内部にあるローカルタスクを処理するよう定義されます。

注意
ローカル変数はループ処理におけるカウンター変数として、また一時的な記憶値として使用される場合が最も多いでしょう。

以下のアイテムは使用する大域変数です。

`rotspermeter`は1メートル動くごとの回転数を入れておく変数です、

`controlaxis`はオブジェクトが回転する軸、Heading, Pitchもしくは Bankです。`ctlchoice()`コマンドを通してスクリプトインターフェイス上からユーザーに軸を選択し確定される整数値です。`ctrlchoice()`関数はユーザーが選択した選択リストを表す整数値を返します。

注意
ctrlchoice()についての詳細は、このマニュアルのインターフェイスの章にあります。

`controlpos`は`controlaxis`と同様の変数です。回転の基となる位置移動軸を特定するXYZを表す整数値です。

変数は出来るだけ実行する処理と似せて名前を付けるようにして下さい。この機能はスクリプトに三つぐらいの変数しかなければそれほど重要ではありませんが、非常に多くの変数を持つスクリプトで作業する場合には大切なことです。後々スクリプトを更新する他の人にとってもスクリプトがより読みやすくなります。

```
create
{
  setdesc("Axis Rotation Control");
}
```

次のコード部分はスクリプトが作成時に実行されます。ですから関数名は'`create`'と名づけられているのです。

`setdesc()`コマンドはモーションオプションパネルでスクリプトの名称を表示することが出来ます。このステップは絶対必要ということはありませんが、スクリプトをより洗練されたものへとするための特別なアイテムのうちの一つです。

`setdesc()`を使用しない場合、モーションオプションパネルではスクリプト名称を表示し、LScriptとして識別します。

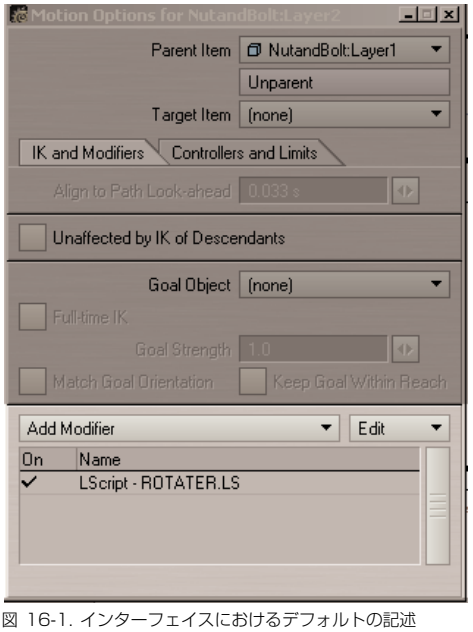

図 16-1. インターフェイスにおけるデフォルトの記述

`setdesc()`コマンドを使うと、デフォルトの代わりにウィンドウに読みやすい名称で置き換えることが出来ます。

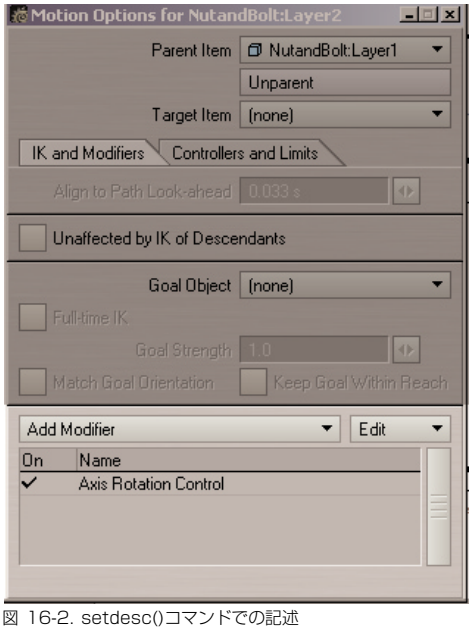

図 16-2. setdesc()コマンドでの記述

実をいうとスクリプト記述という大きな計画においてたいしたことではないのですが、こうすることでスクリプトには正当性とプロフェッショナリズム的な雰囲気が醸し出されるのです。

`create()`関数の中では変数へのデフォルト値の割り当てや、スクリプト追加時に一度だけ処理される一般的な処理も実行できます。

```
process: ma, frame, time
{
  rot = ma.get(ROTATION,time);
  roth = rot.h;
  rotp = rot.p;
  rotb = rot.b;
  pos = ma.get(POSITION,time);
```

大半のスクリプトと同様、`process()`関数では大量の処理が行われます。Item Animationの場合、`process()`関数はレイアウトで時刻が変化するたびに呼び出されます。レンダリング中であってもプレビューが再生中であっても同様です。

今回のスクリプトでは、まず現在時刻においてオブジェクトが何をしているのかを見つけ出しましょう。変数 rot はオブジェクトの回転情報が（ベクトルとして）代入されます。それからローカル変数 roth, rotp, rotb には、各々 rot ベクトルの一部が代入されます。変数 pos は Motion Access Object Agent からオブジェクトの位置情報を格納します。

お望みの情報を手に入れるため get()メソッドを使用します。回転の個別のチャンネルに対し影響を与えるようにしたいので、ROTATION を三つのコンポーネントヘディング（roth)、ピッチ（rotp)、バンク（rotb）へと割り振ります。

スクリプトの後半にありますが、ROTATION チャンネルを分けることで他の二つのチャンネルの値を保護することが出来ます。そうすれば、影響を受けない二つのチャンネル上に存在するキーフレーム付きモーションとスクリプトの拘束を受けるチャンネルとを合わせることが可能です。

```
if (controlpos == 1) rotamount = pos.x*(rotspermeter*360);
else
if (controlpos == 2) rotamount = pos.y*(rotspermeter*360);
else
rotamount = pos.z*(rotspermeter*360);
```

これで、受け取った POSITION 情報を使用してオブジェクトの回転を修正することが出来ます。しかし、まずは回転量をオブジェクトの位置に応じて割り当てることが必要です。以下の方程式を使用して値を割り出しましょう。

```
Rotational Amount =  Object Positional Axis
Location * (Rotations Per Meter User Input * 360)
```

移動軸として使用する軸をユーザーに選択させますので、その選択軸に応じた方程式を計算する用意をしておかなくてはなりません。上記の if()文ではＸに対しては1、Ｙに対しては2、Ｚに対しては3でコントロールします。

また変数 rotspermeter は360が掛け合わされている点に注意してください。これで"Rotation Per Meter"の入力値が度数へと変換されました。

```
if (controlaxis == 1) roth = rotamount;
else
if (controlaxis == 2) rotp = rotamount;
else
rotb = rotamount;
```

位置軸に対する if()文と同様、この if()文も割り当てられてた回転値を取得するのはどの回転軸なのかを確定します。

```
ma.set(ROTATION,<roth,rotp,rotb>);
```

全て計算し割り当て終えたら、変更点をオブジェクトのモーションに再び割り当てなくてはなりません。`set()`メソッドを使用して処理しましょう。オブジェクトの`ROTATION`のみに影響を与えたいので、`ROTATION`だけに`set()`を行いましょう。

`ROTATION`情報はベクトルとして書き直している点に注目してください。つまりヘディングとピッチ、バンクの値は一組となっているのです。そのために変数を`<`と`>`で囲っているのです。`<>`は三つの値を一つのベクトルとして結び付けます。ベクトルを作成する時にはその順番に注意してください。間違えてもエラーチェックは効きません。もし間違えて変数の位置を変更してしまうと、一つのチャンネル値が他のチャンネル値となってしまいます。例えば以下のように

```
ma.set(ROTATION,<rotp,roth,rotb>);
```

としてしまうと、ヘディングとピッチの情報が入れ替わってしまい、とんでもなく変な結果を引き起こしてしまいます。

```
options
{
  reqbegin("Axis Rotation Control");
    c1 = ctlchoice("Rotation
Axis",controlaxis,@"H","P","B"@);
    c3 = ctlchoice("Control Axis",controlpos,@"X","Y","Z"@);
    c2 = ctlnumber("Rotations Per Meter",rotspermeter);
    return if !reqpost();
    controlaxis = getvalue(c1);
    controlpos = getvalue(c3);
    rotspermeter = getvalue(c2);
  reqend();
}
```

`options()` UDFは、ユーザーがスクリプトのプロパティを尋ねる際のボタンやコントロールを全て設定する箇所です。

インターフェイスの設計はこのマニュアルの別の章でカバーしていますが、上記のコードは以下のようなインターフェイスを生成します。

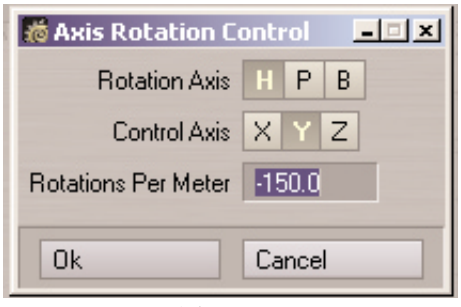

図 16-3. options()で生成されたインターフェイス

```
load: what,io
{
  if(what == SCENEMODE) // ASCII シーンファイルの処理
  {
    controlaxis = integer(io.read());
    controlpos = integer(io.read());
    rotspermeter = number(io.read());
  }
}
save: what,io
{
  if(what == SCENEMODE)
  {
    io.writeln(controlaxis);
    io.writeln(controlpos);
    io.writeln(rotspermeter);
  }
}
```

load()と save() UDF を使うと、シーン保存時に保持しておきたい変数を保存することが出来ます。これによりシーンが再び読み込まれたときに、保持しておきたい変数が自動的に復元されます。この機能がなければシーンを読み込むたびに値を設定しなおさなければならず、スクリプトの使い勝手がひどく制限されてしまいます。

以上が Item Animation です。上記の小さなチャンクでは全体として見通しづらい点もあるでしょうから、以下にスクリプト全部を記しておきます。

ROTATER.LS

```
///----------------------------------------
// Axis Rotation Control
//
// Scott Wheeler
// 09/12/01
@version 2.3
@warnings
@script motion
@name AxisRotationControl
rotspermeter = 1.0;
controlaxis = 1;
controlpos = 1;

create
{
  setdesc("Axis Rotation Control");
}

process: ma, frame, time
{
  roth = ma.get(ROTATION,time).h;
  rotp = ma.get(ROTATION,time).p;
  rotb = ma.get(ROTATION,time).b;
  pos = ma.get(POSITION,time);
  if (controlpos == 1) rotamount = pos.x*(rotspermeter*360);
  else
  if (controlpos == 2) rotamount = pos.y*(rotspermeter*360);
  else
    rotamount = pos.z*(rotspermeter*360);
  if (controlaxis == 1) roth = rotamount;
  else
  if (controlaxis == 2) rotp = rotamount;
  else
    rotb = rotamount;
  ma.set(ROTATION,<roth,rotp,rotb>);
}

options
```

```
  {
    reqbegin("Axis Rotation Control");
      c1 = ctlchoice("Rotation
Axis",controlaxis,@"H","P","B"@);
     c3 = ctlchoice("Control Axis",controlpos,@"X","Y","Z"@);
     c2 = ctlnumber("Rotations Per Meter",rotspermeter);
     return if !reqpost();
     controlaxis = getvalue(c1);
     controlpos = getvalue(c3);
     rotspermeter = getvalue(c2);
    reqend();
  }

  load: what,io
  {
    if(what == SCENEMODE) // ASCII シーンファイルの処理
    {
      controlaxis = integer(io.read());
      controlpos = integer(io.read());
      rotspermeter = number(io.read());
    }
  }

  save: what,io
  {
    if(what == SCENEMODE)
    {
      io.writeln(controlaxis);
      io.writeln(controlpos);
      io.writeln(rotspermeter);
    }
  }
```

第 17 章：チャンネルフィルタ

Channel Filterスクリプトでは、ユーザーがアイテムモーションの個別のチャンネルに対しアニメーションを修正出来るようになります。このタイプのスクリプトはチャンネルの値に対し読み書き両方においてアクセス可能です。チャンネル内で処理される調整を監視し作用します。

一般的なChannel Filterスクリプトは Channel Object Agentから値を読み取り、収集したデータに対しテストと計算を実行し、最終的には値を修正しチャンネルへと値を戻します。

LScriptに見られる大半のレイアウトスクリプト構造と同様、Channel Filterも、このスクリプトクラスで使用可能なオプションを利用するために事前定義された関数をいくつか使用しています。スクリプトを適切に動作させるためにこれら関数がいくつかは必要となりますが、他は単にオプションの機能を提供しているだけです。

create() と destroy()

`create()`と`destroy()`関数は全変数の初期化とスクリプトに対するクリーンアップ処理を扱います。Channel Filterスクリプトタイプ用として、`create()`関数はスクリプトが適用されているチャンネルから作成されたChannel Object Agentを受け取ります。

process()

`process()`関数は重要なチャンネル修正コードの大半が書かれている箇所です。この関数はレイアウトから呼び出された時に三つの変数`ca`, `frame`それに`time`を受け取ります。`process()`はレイアウトでフレームデータが修正された時もしくは評価されたときに呼び出されます。

Channel Access (`ca`) Object Agentには、チャンネルに保存される値を認識し編集するために使用可能なデータメンバ一つとメソッドが二つあります。他の構造特有のObject Agentと同様、Channel Accessは`process()`関数の外側では作成できません。

Channel Accessのデータメンバ`name`にはチャンネルの現在の名称が保存されています。例えば、アイテムのXチャンネルに対しChannel Filterスクリプトを適用している場合、`name`データメンバには文字列"`Position.x`"が入っています。

`get()`と`set()`メソッドで、Channel Access Objectでアイテムのチャンネルに保存されている数値を取得したり編集したり出来るようになります。`get()`関数は引数`time`を受け取り、`set()`では単にチャンネルの現在のフレーム上で動作します。

`process()`関数への二番目（`frame`）と三番目（`time`）のパラメータはチャンネルが評価される時刻を示しています。これらの値はユーザーインターフェイス内で表示されるシーンの現在の時刻に一致するかもしれないし、しないかもしれないという点を覚えておいて下さい。

load() と save()

`load()`と`save()`関数を使用すると、レイアウトはこのスクリプト操作に属する特定のデータを読み込んだり保存することが出来るようになります。スクリプトをどのように実行するのかを確定する選択オプションや設定は、シーンファイル内部に保存されなければなりません。シーンが保存されたり読み込まれる時に、このスクリプトが属するアイテムのチャンネル情報は破棄されます。

レイアウトがシーンを読み込み、スクリプト用のエントリが現れると、`load()`関数が呼び出されます。この関数内では、想定どおりのスクリプト処理を可能にする設定や値を読み込み、割り当てます。同様に`save()`関数はユーザーがシーンを保存するときに呼び出されます。この関数には、シーンファイルにスクリプトのデータを系統立て保存するときに必要なコードが全て含まれています。

options()

`options()`関数はプラグインリストでスクリプトのプロパティをダブルクリックしたり、編集を選択するたびに呼び出されます。そこにはスクリプトのインターフェイスを設定し実行するために必要なコードが全て含まれています。

Channel Object Agent

`create()`とイベントコールバックUDF関数の双方が、引数`channel`を受け取ります。このObject Agentにはキーフレーム情報の取得や設定、キーフレームの作成や削除、キーフレームのスプライン情報の検索などを行うデータメンバやメソッドがいくつか用意されています。

注意
このObject Agentはアイテムのfirst Channel()メソッドを呼び出すことでも作成可能です。

例 1: maxValue

まず手始めに、チャンネルの値を強制的に最大値へ設定するスクリプトを作成しましょう。スクリプトがアイテムチャンネルからオンに設定されると、チャンネルは事前に定義された値を越えることが出来なくなります。スクリプトのヘッダー情報から取り除いていきましょう。

```
@warnings
@version 2.5
@script channel
@name maxValue
```

ここでは新しく学ぶことは何もありません。プリプロセッサ指示子でLScriptに対しこのスクリプトで使用するオプションをいくつか設定しています。`@script`指示子の中での`channel`設定を使用して、このスクリプトがChannel Filterであると定義している点に注目してください。

レイアウトにおける大半のスクリプト構造と同様、Channel Filtersは利用可能なお馴染みの関数をいくつも持っています。まずは`create()`関数です。

```
create: channel
{
}
```

`create()`関数はスクリプトが最初に読み込まれたときに呼び出されます。ユーザーがチャンネルに対しプラグインリストからChannel Filterスクリプトを適用した時、またChannel Filterが適用されているシーンファイル内のオブジェクトが読み込まれた時にも、この関数が呼び出されます。通常、`create()`関数の中でスクリプトのキー変数をセットアップしたり、ユーザーエラーのチェック、スクリプト解説文の設定を行います。ではここでスクリプト解説文の設定を行いましょう。

```
create: channel
{
  // プラグインリストの解説文を設定します
  setdesc( "maxValue" );
}
```

見ておわかりのとおり、`create()`関数は呼び出されるたびに、引数`channel`を受け取ります。この引数はスクリプトが適用されるチャンネルから作成されたChannel Object Agentです。`create()`関数に対し簡単な`info()`関数を追加することで、テストしてみることが出来ます。

```
create: channel
{
  info( "We have attached this to the:  ",channel.name, "
channel." );
  // プラグインリストの解説文を設定します
  setdesc( "maxValue" );
}
```

ここではChannel Obejct Agentのデータメンバ`name`をテストしてみました。このスクリプトをアイテムのXチャンネルに適用している場合には、使用しているチャンネル名を告知する`info()`リクエスタが表示されるはずです。

注意
Channel Object Agentで利用可能なデータメンバとメソッドの完全なリストについては、リファレンスマニュアル第26章： Channel Object Agentsを参照してください。

さて次に、このタイプのスクリプトにおいて最も重要となる関数、`process()`関数へと移りましょう。この関数はフレームのモーションデータが変更された時に呼び出される関数なので、関数内にはフレーム特有のコードを全て書いておくようにして下さい。そうすればチャンネル情報が修正された時点で、スクリプトは新しい値へと更新されます。では何が利用出来るのかを見ていきましょう。

```
process: ca, frame, time
{
}
```

見てわかるとおり、`process()`関数には三つの引数が渡されます。最初の引数`ca`はチャンネルのChannel Access Agentです。これは単一のデータメンバ（`name`）と二つのメソッド（`set()`と`get()`）をエクスポートし、アイテムチャンネル内の値をやり取りすることが可能です。2番目と3番目の引数`frame`と`time`は単にユーザーが現在シーンにおいて編集している箇所を示しています。では見てみましょう。

```
process: ca, frame, time
{
  info(frame);
}
```

この小さなテストでは二つの作業を行います。まずはこれからLightWaveを無限ループに陥らせるので、実行する前に作業を保存するようにして下さい。2番目に、`process()`関数がどのくらいの頻度で呼び出されるのかを正確に示します。スクリプト実行時に、関数が各フレームに対し定期的に繰り返し呼び出されることがわかるはずです。OKボタンを押すことは出来ますが、他は何も出来ません。スクリプトを中断するにはレイアウトを強制終了するしか手はありません。どのように動くのかを見た後は、コードからこの行を忘れずに削除するようにして下さい。

`info()`関数へ送られるデータを修正するだけで、`process()`関数へと渡される他の引数を引き続きテストすることが可能です。渡された各引数によって異なる結果を表示しますが、`process()`関数が呼び出される頻度は変化しません。いつ、どのくらいの頻度で関数が呼び出されるのかを決定するデータは表示されません。それはChannel Filterスクリプト自身の設計だからです。

明らかに、`process()`関数内でデータを表示するために`info()`リクエスタを使用することは出来ません。スクリプトを変更する毎にレイアウトを終了させ再起動させるなんて絶対に嫌でしょう。このタイプのスクリプトを学ぶ間に、可視化したいデータの表示法を考えつかないといけませんね。常にチャンネルデータの値を取得するのではなく、処理するチャンネルに対し変更が加えられた場合にのみ、表示するようにすればよいのです。捜し求めているのはマスターのイベントハンドラーと同じです。幸運なことにchannel Obejct Agentはこのメソッドを持っています。

```
create: channel
{
  channel.event( "newEvent" );
  //  プラグインリストの解説文を設定します
  setdesc( "maxValue" );
}
```

使用する`event()`はパラメータとしてユーザー定義関数（UDF）の名称を受け取ります。この場合、スクリプトのチャンネルが修正された時点でスクリプト内部の`newEvent()`関数が呼び出されます。これをテストする前に、新しいUDFを定義しなくてはなりません。

```
newEvent: channel, event
{
  info(event);
}
```

これでチャンネルが修正されるたびに`newEvent()`関数が呼び出され、`info()`リクエスタは実際に何が起きているのかを表示するようになります。しばらくこの編集に時間をかけてみると、イベントがいつ生成されたのか、何のイベント生成されたのかが理解できるようになります。また`newEvent()`関数は渡されたイベントを取得するだけではなく、同様にChannel Object Agnetを受け取ることに気が付いたことでしょう。

この例で作成しているスクリプトはチャンネルの`event()`コールバック関数を使用しません。イベントパラメータを試し終えたら、`newEvent()`関数を完全に除去し、同様に`create()`関数内部における`event()`メソッドの呼び出しも除去してください。

最終的にはこのスクリプトに対するインターフェイスが必要となりますが、今はまず望みどおりに動作をスクリプトが実行出来るかについて心配しなくてはなりません。ただしこのスクリプトが正しく動作するためには、インターフェイスが提供するデータも必要となります。そこでインターフェイス作成に多大な時間をかけずに、大域変数を定義し初期値として与えることにしましょう。

```
@name maxValue
// Global variables:
maxValue = 1.0;

create: channel
{
```

スクリプトが正しく動作していることが評価され証明されたときには、インターフェイスを作成しましょう。次にチャンネルの値をモニターするために`process()`関数を設定していきます。

```
process: ca, frame, time
{
  // Channel Access Object Agentからの値を取得します
  currValue = ca.get(time);
}
```

`get()`メソッドにタイムインデックスを渡し、その時刻におけるチャンネルの値を取得します。`get()`メソッドは単一の浮動小数点数値を返します。チャンネルの値を問い合わせるだけで、キーフレーム全体の値をとるわけではないという点を覚えておいてください。上記例では返される値は指定した時刻におけるアイテムのX座標値です。

それでは値が取得できたところで、変数`maxValue`の値を比較してみましょう。

```
process: ca, frame, time
{
  // Channel Access Object Agentからの値を取得します
  currValue = ca.get(time);
  // 値を比較します
  if(currValue >= maxValue)
    ca.set(maxValue);
}
```

まずチャンネルの現在の値が変数 `maxValue` に格納されている値以上であるかを比較してみます。以上であればChannel Accessの `set()` メソッドを使用して、チャンネルの値を `maxValue` の値に設定します。このようにしてチャンネルの値を編集しますが、`maxValue` よりも小さな値であれば何も行いません。この設定は上限値設定を行っているのだと思って下さい。

この時点で実験してみましょう。

1 オブジェクトCowを読み込みます。

2 CowのXチャンネルにスクリプトを追加します。

3 `maxValue` の値を越える程度に、X軸に沿ってCowを動かしてみてください。

レイアウトでCowを動かすと、チャンネルの値が `maxValue` 以上の地点では、キーフレームを作成しても値を変更したりカレントフレームが更新されるとすぐに `maxValue` に固定されるのがわかるでしょう。

以下はこれまでのコードに `maxValue` 変数を設定するためのインターフェイスを追加したコードとなります。

```
@warnings
@script channel
@version 2.5
@name maxValue
// 大域変数
maxValue = 1.0;

create: channel
{
  // プラグインリストの解説文を設定します
  setdesc("maxValue");
}

process: ca, frame, time
{
  // Channel Access Object Agentからの値を取得します
  value = ca.get(time);
  // 値を比較します
  if(value >= maxValue)
    ca.set(maxValue);
}
```

例 2: dynLimiter

このスクリプトでは、先ほど作成した maxValue スクリプトの動的バージョンを作成していきます。上記と同じ例を使用してはじめましょう。しかし変数内にチャンネルの最大値を設定するのではなく、他のオブジェクトのチャンネルの値を最大値として使用します。こうすることで時間の経過とともに最大値をアニメーションさせることが出来ます。

先ほどの大域変数 maxValue を、プリファレンスとして使用するオブジェクトの名称を表す objName に置き換えましょう。ここでは文字列値"Null"を使用します。この値はインターフェイスからも取得可能です。

```
// 大域変数
objName = "Null";
create: channel
```

変数 objName には参照オブジェクトとして使用するオブジェクトの名称が入ります。まずは process()関数内においてこの変数から Mesh Object Agent を作成します。

```
process: ca, frame, time
{
  // Object Agent を作成します。
  obj = Mesh(objName);
  // Channel Access Object Agent から値を取得します
```

Object Agent を作成しましたので、可変チャンネルに保存されている値を取得することが出来ます。先を続ける前に、Object Agent が実際に作成されたのかどうかを確認します。確認したら、参照オブジェクト内に存在する最初のチャンネルから Channel Object を作成することが出来ます。

```
if(obj)
{
  // オブジェクトの最初のチャンネルを取得します
  chan = obj.firstChannel();
}
else
  error("No Mesh Object Agent created.");
```

firstChannel()メソッドは Mesh Object Agent 内にある最初のチャンネルを返すだけです。この例では一つのチャンネル、Position.X だけに関心があります。今回の例では非常に幸運でした。参照オブジェクト内に最初のチャンネルが Position.X である、引数 ca として process()関数に送信されるチャンネル名称と一致するからです。しかし、常にこういう事態になるとは仮定出来ません。

アイテムの全チャンネル名称を通して、`process()`関数へ渡されたチャンネル名称と一致するかどうかを確かめながら検索する必要があります。ですから二つの異なるオブジェクトの同じチャンネルを見ることになります。`while()`文を呼び出します。

```
// オブジェクトの最初のチャンネルを取得します
chan = obj.firstChannel();
// チャンネル名称 ca と chan を比較します
while(chan && chan.name != ca.name)
  chan = obj.nextChannel();
```

`while()`文を使うと、オブジェクト内の次のチャンネルから繰り返しChannel Object Agentを作成することが可能です、どのチャンネルが開始地点となるのかを把握するにはどうすればよいのでしょうか？最初のChannel Object Agentである`chan`を作成するのには、`firstChannel()`メソッドを使用しました。それ以降Mesh Object Agentの`nextChannel()`メソッドを呼び出すと、オブジェクト内の次のチャンネルが返されます。例えば最初のチャンネルの値がPosition.Xであれば、次のチャンネルから返される値はPosition.Yといった具合に続いていきます。

評価する双方のチャンネル名称が一致した場合にのみ、`while()`文はスクリプトの処理続行を許可します。二つのオブジェクト内部で同じチャンネルを比較するようにします。この場合においてChannel Obejct Agentである`chan`が作成されたかを確認してみましょう。

```
if(chan)
{
}
```

今や、このスクリプトが正しく動作するために必要となる情報の多くを保持しています。リファレンスオブジェクトの名称（`objName`）やスクリプトのチャンネルから作成されたObject Agentである`ca` Object Agentの値、リファレンスオブジェクトなどの情報です。あとはリファレンスオブジェクトのチャンネルから値を取得するだけです。

前述の例において、値を比較しチャンネルの値を調節するコードは既に書いています。今回は上記で開始した`if()`文内におけるコードを全て変更しましょう。

```
if(chan)
{
  // 参照オブジェクトの Channel Object Agent から値を取得します
  maxValue = chan.value(time);
  // Channel Access Object Agent(このチャンネルにおける)から値を取得します
  value = ca.get(time);
  // 値を比較します
  if(value >= maxValue)
    ca.set(maxValue);
}
```

処理を多数行いますが、どれもみな非常に簡単なものです。まず value()メソッドを使用して参照オブジェクトのチャンネル値を取得しました。次に ca Channel Access Object Agent 内に保存されていた値を取得しました。それから二つの値を比較し、ca の値が maxValue を超えていれば、set()メソッドを使用して ca に maxValue の値を代入するだけです。

このスクリプトをテストしてみましょう。

1. Cow オブジェクトを読み込んでいるか確認してください。
2. "Null"という名称のヌルオブジェクトを追加します。
3. このスクリプトをオブジェクト Cow へ適用します。
4. Null を X 軸に沿ってキーフレームをつけます。
5. 参照オブジェクト Null を越える箇所に Cow オブジェクトをアニメーションさせます。
6. キーフレームを作成し、カレントフレームを変更してみます。Cow オブジェクトが参照オブジェクト Null の X 座標位置に固定されているのがわかるますね。

最終スクリプト:

```
@warnings
@script channel
@version 2.5
@name dynLimiter
// 大域変数
ObjName = "Null";

create: channel
{
  // プラグインリストの解説文を設定します
  setdesc( "maxValue" );
}

process: ca, frame, time
{
  // Object Agent を作成します
  obj = Mesh(objName);
  // Channel Access Object Agent から値を取得します
  if(obj)
```

```
  {
    // オブジェクトの最初のチャンネルを取得します
    chan = obj.firstChannel();
    // caとchan Object Agentsのチャンネル名称を比較します
    while(chan && chan.name != ca.name)
    chan = obj.nextChannel();
    if(chan)
    {
      // 参照オブジェクトのChannel Object Agentから値を取得します
      maxValue = chan.value(time);
      // Channel Access(このチャンネル) Object Agentからの値を取得します
      value = ca.get(time);
      // 値を比較します
      if(value >= maxValue)
        ca.set(maxValue);
    }
  }
  else
    error( "No Mesh Object Agent created.);
}
```

第 18 章：ジェネリック（総括）スクリプト

全スクリプトタイプの中で最も制限のないスクリプトが、**LScript Generic（LSGN）**です。スクリプトタイプの大半がレンダー処理のある特定の側面に制限されていますが、Generic LScript はそれら全ての外側に存在しています。スクリプトはレイアウト内部の様々な側面の多くにアクセス出来るのです。

Generic スクリプトはモデラーの LScript のような動作を行います。これらのスクリプトは呼び出された時点で割り当てられる処理全体を完了する"**run-once**"操作です。LightWave 環境内部から実行するスタンドアローンプログラムのようなものですから、開発者は Generic LScript 内において、他のレイアウト LScript タイプでは利用できない機能を自由に実行することが可能です。

ジェネリックスクリプト構造

Generic スクリプトが"run-once（一回だけの実行）"イベントであるならば、プラグインの実行期間内においてある特定の時間に特定の関数を呼び出すことは出来ません。ですから`create()`や`destroy()`、`init()`関数は利用できません。Generic のスクリプトは任意のシーンやオブジェクトに制限されていないため、他のスクリプトタイプが使用可能な`save()`や`load()`関数もまた利用できないのです。

レイアウトの Generic スクリプトは、`generic()`関数という単一のエントリーポイントを持っています。スクリプトはこの関数の呼び出しから始まり、関数の処理が完了した段階で終了します。

さらに単純化するため、`options()`関数は Generic のスクリプトでは利用できません。これは、Generic ではインターフェイスが持てないと言っているのではありません。インターフェイスコントロールは全て`generic()`関数の中に配置されることになるのです。スクリプト処理がこのコードに到達した時点でリクエスタが表示されます。

LS-GN 関数

Genericのスクリプトには2～3の関数が用意されています。これらの関数を使用して、レイアウトのシーンを管理することが出来ます。

loadscene(filename[,title])

`loadscene()`関数では、指定されたファイル名称をレイアウトのシーンファイルとみなして読み込みます。オプション引数`title`はいったん読み込まれた後で内部的に保持されるシーンファイル名称として使用されます。このオプション名称がシーンの新規ファイル名称となります。このパラメータが省略されている場合には、引数`filename`が代用として使用されます。

savescene(filename)

`savescene()`関数は、レイアウトにあるカレントのシーンを、指定したファイル名称`filename`で保存します。

例： INCREMENTSCENE.LS

GenericのLScriptでは基本的にスクリプト関数は制限されていませんから、ここで解説していく例はどれも、その小規模な範囲以上のものをカバーすることは不可能です。それを頭に入れた上で、Genericのスクリプトタイプを使用してLightWaveに追加可能な機能の種類を表示する短いスクリプトを見ていきましょう。

以下のスクリプトはLightWaveに読み込まれたシーンを受け取り、スクリプトを実行するたびにバージョン番号を増やしながらシーンファイルを保存します。

```
//----------------------------------------
// Incremental Scene Saver v1.0
//
// Scott Wheeler
// 08-03-01
@version 2.3
@warnings
@script generic
@name LW_IncrementalSceneSaver
```

スクリプトの最初の部分は主としてコメント領域です。これについての詳細は、このマニュアルの第１４章：プロシージャルテクスチャを参照して下さい。

スクリプトラベルの後にはプラグマ指示子が来ます。このスクリプトを読みやすくするため、指示子は全てスクリプトの一番上に置いておきます。プラグマ指示子は行頭にある文字"@"によって認識され、その後に指示名称と指定可能な引数が続きます。

読み込まれたシーンの名称を変更する前に、現在のシーンの名称を把握しておく必要があります。シーン特有の情報にアクセスするためScene Object Agentを使用します。`Scene()`コンストラクタを呼び出し、現在のシーンからScene Objectを作成することが出来ます。

```
scene = Scene();
```

変数`scene`に入っているScene Object Agentには、データメンバがいくつもあります。データメンバの一つに`filename`があります。データメンバ`filename`にはカレントシーンの完全なパス付きファイル名称の情報が入っています。ディスクにシーンを保存するためには、この名称が必要となるのです。

例：

```
X:\Scenes\MyProjects\ThisScene.lws
```

特定のデータメンバを名称で要求することでScene Object Agentである'`scene`'の外へと情報を抜き出します。Object Agentコンテナ（この場合は`scene`）の終端にメンバ`name`を追加します。

例：

```
scene.filename  // Scene Object 'scene'のファイル名称を返します
scene.polycount // Scene Object 'scene'内にあるポリゴン総数を返します
```

変数'`scene`'にMesh Object Agentが含まれている場合には、データメンバ`polycount`にアクセスすると、オブジェクト内のポリゴン数を返します。

```
generic
{
  scene = Scene();
  if (scene.filename == "(unnamed)")
    SaveSceneAs();
  SaveSceneAs(setupscene(scene.filename));
```

`generic()`関数の開始行において、現在のLightWaveシーンに実際にその名称が割り当てられているのかをチェックします。`if()`文内においてScene Object Agentから直接情報にアクセスしています。こうするとコード内で必要とされる行数は圧縮出来ますが、スクリプトは理解しづらくなりますね。

では、`if()`文が実際にはどのような動作を行うのかを見ていきましょう。LightWaveは保存されていないシーンに対し"`(unnamed)`"という名称を自動的につけます。ですから名称に"`(unnamed)`"と設定されていないかをチェックし、それに応じて処理を行います。

シーンが保存されていない場合、Command Sequenceコマンドである`SaveSceneAs()`を呼び出します。Command Sequenceコマンドは直接LightWaveの内部関数とリンクしています。これらのコマンドにより、スクリプトはLightWaveのインターフェイスと内部関数双方をさらにコントロールできるようになります。ただしCommand Sequenceコマンドは`Generic`と`Master`クラススクリプト以外ではアクセス出来ません。

`SaveSceneAs()`は、保存するシーンのファイル名称を要求します。まだシーン名称を決めていないので、名称を空にしたままでコマンドを呼び出します。するとLightWaveはユーザーに対し"`Save As`"ファイルダイアログを開きます。その後は最初からシーンにファイル名称が付けられていたかのように処理されます。

今回`SaveSceneAs()`コマンドへと渡す情報は、シーン名称変更用に作成された関数の戻り値です。このユーザー定義関数（UDF）は`setupscene()`から呼び出され、`SaveSceneAs()`へ渡す変更後のシーン名称を返します。`SaveSceneAs()`内で`setupscene()`を呼び出し入れ子状態にすることで、ここでもコードの行数を削ります。

シーン名称を変更するための別個の関数を使用し、コードを一部切り離すことでより読みやすくすることも出来ます。UDFではある一つのセクション内に共有化する機能の領域を配置することが可能です。そうすればシーン名称を処理するコードは全て`setupscene()`関数内にありますので、コードの修正がはるかに容易になります。異なるセクションを探すためにメインコードを通して検索する必要がなくなるのです。

これほどまで極端にスクリプトを小さなサイズへとするのは賢いやり方だとは思えないかもしれませんが、あらゆる状況においてコードをより読みやすくするための方法を探し出していくにはとてもよい実習です。いずれは後々変更を加えないかもしれませんし、ある期間を経た後スクリプトへと戻っていくかもしれません。どちらにしろコードを読みやすくしておくほうが良いのです。

もう一つ、独自の関数を作成するとコードのローカライズを行うことで何度も使用出来るという利点があります。コード自体を反復させるのではなく、関数を一つ作成しスクリプト内部で何度も関数を呼び出すことが出来ます。

```
setupscene: sceneName
{
}
```

UDF は処理中に使用するパラメータ群を受け取ることが可能です。今回の場合では、`sceneName` というパラメータを一つ渡すことになります。`generic()`関数内で見たとおり、`setupscene()`へは`scene.Finename` のコンテンツを渡します。それからこの値はローカルで使用するために変数`sceneName`へと代入します。

```
cropsize = (sceneName.size()) - 4;
sceneName = strsub(sceneName,1,cropsize);
// .lws を切り取ります
```

`sceneName` には読み込むシーンのフルのファイル名称が入っているため、拡張子".lws"も含まれています。新しいバージョン番号をシーンファイル名称の終端につけたいのですが、拡張子".lws"の後ろにつけたくはありません。ですから処理を行う前にファイル名称から".lws"を切り取る必要があります。

まずは拡張子".lws"を取り除いたファイル名称の大きさを割り出します。LScript のビルトイン変数メソッドで、この処理が簡単になります。`size()`メソッドは全ての変数に対し利用可能であり、変数のサイズを返します。ファイル名称のような文字列の場合、名称内にある文字数を返します。全体の長さから 4 引くことで、拡張子".lws"を除いたファイル名称の長さを取得します。

ファイル名称の長さがわかったので、拡張子を取り除くことが出来ます。LScript では文字列処理とこれらの文字列の部分抽出を行うコマンド群を提供しています。これらコマンドの一つに`strsub()`コマンドがあります。`strsub()`では文字列を受け取り、引数として渡した切り取り用変数に基づいて切り取られた文字列のサブセットを返します。

例：

```
strsub()は以下のパラメータを受け取ります:
strsub(<sceneName>,<first character position>,<length>)
```

ファイル名称が以下の場合

```
sceneName = "X:/myscenes/scene.lws" .
```

ファイル名称の総文字列は 21 です。最後の 4 文字を除去したいので文字数は 21-4=17 になります。ですから最後 4 文字を取り除く`strsub()`への呼び出しは以下のようになります。

```
strsub(sceneName,1,17);
```

これは"文字列`sceneName`を受け取り文字列の最初の文字から 17 文字を抽出する"と解釈されます。名称全体は 21 文字の長さですから最後の 4 文字が返されないままとなります。

拡張子".lws"が取り除かれファイル名称にバージョン番号を付加できるようになりました。

```
if (strsub(sceneName,cropsize - 4,1) == "_")
```

スクリプトはファイル名称に"_vXXX"を追加します。このxxxの個所には番号が入ります（たとえばv001,v002）。4文字の固定した値ですので、ファイル名称内にバージョン番号が存在するかどうかが確認できます。ファイル名称終端からの4文字めはアンダースコア（_）となります。

これを頭に入れた上で、`if()`文は終端からの4文字が"_"であるかどうかをチェックします。

```
ver = integer(strsub(sceneName,cropsize - 2,3)) + 1;
```

終端からの4文字が"_"であれば、カレントシーンの終端にバージョン番号が存在しています。シーンを保存する前にバージョンの値がわかれば、バージョン番号を上げていくことが出来ます。

既に`strsub()`コマンドを使用していますので、文字列から一連の文字を抜き出す方法はわかっています。`integer()`コマンドを使用して組み合わせていきましょう。文字列値は`integer()`コマンドを通して値を渡すことにより整数へと変換が可能です。これでカレントシーンの整数バージョン番号は文字から整数値へと変換されます。その後バージョン番号に1を加算して、バージョン番号を上げていきます。

```
sceneName = strsub(sceneName,1,(sceneName.size()) - 5));
```

このコード行は既存の"_vXXX"文字列をファイル名称から除去し、新しいバージョン番号を適用させる準備を行います。

```
else
  ver = 1;
```

`if()`文の'`else`'部分は、終端からの4番目の文字が"_"でない場合、つまりバージョン名称がシーンに割り当てられていない場合に処理を行います。この場合にはシーンを保存する前にバージョン番号が'1'に設定されているかどうかを確認する必要があります。

```
pad = "_v" + ver.asStr(3,true);
```

可変メソッド`asStr()`を使用してデータを整数タイプから文字列タイプへと変換します。最初の引数(3)を渡すことで、文字列の長さを特定しています。こうすると文字列は常に文字数3となります。

```
return((sceneName + pad + ".lws"));
```

`setupscene()`関数を終了させるには、調整したシーンを保存可能にするため`return()`文が必要となります。

`return()`文の中で切り取られたシーン名称`scenename`は新規バージョン番号を受け取り、再び拡張子".lws"をつけます。

スクリプトを保存しレイアウトに読み込んだ後、スクリプトをキーストロークに割り当てておけば、キーを押すだけでバージョン付きシーンを保存することが出来ます。

INCREMENTSCENE.LS

```
//----------------------------------------
// Incremental Scene Saver v1.0
//
// Scott Wheeler
// 08-03-01
@version 2.3
@warnings
@script generic
@name LW_IncrementalSceneSaver

generic
{
  scene = Scene();
  if(scene.filename == "(unnamed)")
    SaveSceneAs();
  else
    SaveSceneAs(setupscene(Scene().filename));
}

setupscene : sceneName
{
  cropsize = (sceneName.size()) - 4;
  sceneName = strsub(sceneName,1,cropsize);
  //   .lws を切り取ります
  if (strsub(sceneName,cropsize - 4,1) == "_")
  {
    ver = integer(strsub(sceneName,cropsize - 2,3)) + 1;
    sceneName = strsub(sceneName,1,(size(sceneName) - 5));
  }
  else
    ver = 1;
  pad = "_v" + ver.asStr(3,true);
  return(sceneName + pad + ".lws");
}
```

第 19 章：マスタークラス

LScript の **Master Class（MC）**スクリプトは、開発者に対し全く異なるレベルの機能を提供します。これらのスクリプトではレイアウトの **Command Sequence（CS）**コマンドが実行でき（Generic スクリプトにおける処理と同様）、またユーザーによってアクションが更新できるという独自の能力があります。オブジェクト選択やフレーム変更、設定の調整といったアクションは、ユーザーがアクションを実行する際に CS コマンドへと全て変換されています。MC スクリプトはこれらコマンドがいつ発生しているのかを監視するだけで、特定の状況になった時点でアクションを更新するようコード化することが出来るのです。

ユーザーがアニメーションや設定編集、シーンのレンダリングを行っている一方で、バックグラウンドではアクティブな Master スクリプトはアクションの記録と更新を連続的に実行しています。MC スクリプトをこんなにも強力なツールにしているのはこの機能なのです。レイアウトでアニメーションさせている時にアニメーターを手助けするツールが書ければ、レイアウトのユーザーインターフェイスを拡張することが出来ます。ツールの選択からマクロの記録まで、MC スクリプトの能力にはほとんど際限がありません。

追加されているこの機能は信じられないほど強力な機能ではありますが、スクリプトは常に実行可能な状態にしておかなくてはなりません。優れた Master スクリプトは、アニメーション中にユーザーが挿入する多くの状況を扱えるようにしてあります。より大切なことは、これらのスクリプトは常時実行しているわけですから、コードを速く効率的に実行しなくてはならず、レイアウトのユーザーインターフェイスのインタラクティブ性が損なわれないよう注意しておかなくてはならないということです。コード作成におけるこの側面を見落としてしまうと、便利な MC スクリプトを作るという目標が大きく覆されてしまいます。

サポートされている関数

MC スクリプトではレイアウトスクリプト内にある一般的な関数、create()や destroy()、save()、load()それに options()に加え、flags()関数が使用できます。

flags()

flags()関数はレイアウトから使用され、MC スクリプト実行時のオプションを設定します。スクリプトが初期化された度に LScript から自動的に呼び出されます。一度でも flags()関数が呼び出されればスクリプトの設定を LScript に再通知する必要はないので、もう一度スクリプト内部で呼び出されることもありません。

SCENE はレイアウトに対し、このスクリプトをシーンと共にクリアするよう指示します。flags()関数が定義されていない場合はデフォルト設定となります。

LAYOUT を指定すると、シーンがクリアされようと新規シーンが読み込まれようと、MC スクリプトは読み込まれ起動状態のままでいることが可能です。

例：

```
flags
{
  return(SCENE);
}
```

process()

Master Class スクリプトにおける process()関数は、レイアウトが新規コマンドを発行するたびに自動的に呼び出されます。この関数が呼び出されると、シーン内でユーザーが実行した処理をスクリプトが正確に確定出来るよう、レイアウトから二つの引数を受け取ります。

引数 event は process()関数へ送られるイベントの種類を返します。現在この引数には NOTHING, COMMAND, TIME, SELECT もしくは RENDER_DONE という定数値が返されます。

文字列 command には、レイアウトが一番最近発行したコマンドが引数付きで入っています。返される文字列には個々のデータを区別するためスペースが使用されています。

例：

```
process: event, command
{
  info(command);
}
```

アイテムが選択された場合、`process()`関数の`info()`リクエスタは以下のように状況を表示します。

```
"SelectItem 10000000"
```

例 １ ： masterTest.ls

Master Classのスクリプトは今まで解説してきたスクリプトタイプの機能とは異なっていますので、最初のスクリプトでは新しい機能についてのみを解説していくようにしましょう。最初のスクリプトは単純なものですが、二つの効果をもたらしてくれます。一つはMasterスクリプトの構造が他のスクリプトタイプとどのように違うのかを把握しやすくしてくれること、もう一つは完成版スクリプトを作業ツールとして実行できるようになることです。このツールを二番目の例として使用し、コマンドをより理解できるように解説していきます。

masterTest.lsスクリプトは、レイアウトのコマンド履歴ウィンドウと似たような動作を行います。発行されるレイアウトコマンドの動作はスクリプトで捕らえられ、`info()`リクエスタに表示されます。どのアクションも表示されると言いましたが、各々のアクションという意味です。ですからスクリプト実行が決断されると、行った特定の動作は全て`info()`リクエスタを表示します。言うまでもないことですが、これは実用的なスクリプトではないものの、目的は達成できます。

ではMaster Classスクリプトにヘッダーを追加していきましょう

```
//
// masterTest.ls: レイアウトからMaster Classスクリプトへと
// 送信されたコマンドを紹介するスクリプト
//
@version 2.3
@warnings
@script master
@name masterTest
```

何も新しいことは行っていません。短い解説文、バージョン番号、警告レベル、作成するスクリプトの種類、スクリプトの名称などを指定しています。一般的なスクリプトの開始部分と同じです。

前述の通り、Master Classスクリプトはレイアウト内でユーザーがコマンドとして実行する動作を監視することが出来ます。Genericスクリプトではなく Master Classスクリプトを使用する際の主な利点の一つが、この機能にあります。しかしユーザーからの情報を取得する前に、まずスクリプトとレイアウト間の関係を定義しておく必要があります。`flags()`という関数の中でその処理を行います。

`Flags()`関数は唯一、LScriptから内部的に使用される関数です。コードの任意の場所から手動でこの関数を呼び出すことはありません。この関数は、スクリプトが適用されているシーンがクリアされたときに、レイアウトがスクリプトをどのように扱うかを設定するための関数です。

Master Classスクリプト起動時には、LScriptはスクリプトのコードを通して自動的にこの関数を検索し、定義されている`flag()`関数を呼び出し、レイアウトとスクリプトの関係を確定します。この関数には`return()`文が含まれており、レイアウトのオプションを設定するために使用されます。前述通り、この関数には二つのオプション、`SCENE`と`LAYOUT`が利用出来ます。

以下の例では、このスクリプト用の`flags()`関数がどのように見えるのかを示しています。

```
flags
{
  // レイアウトのイベント全てを監視するためのフラグを設定します
  return(SCENE);
}
```

ここまで見てきたレイアウトスクリプトの大半と同様、スクリプトの動作のほとんどが`process()`関数内で処理されます。Master Classスクリプトの一部として、`process()`関数のジョブはユーザーからコマンドが発行される度にレイアウトから呼び出されます。スクリプトの`process()`関数ではレイアウトから自動的に渡される二つの引数`event`と`command`を受け取ります。以下の例で関数を解説していきます。

```
process: event, command
{
  // 二つの渡された引数の値を表示します
}
```

目標を達成するには、`info()`関数へと値を入れるだけです。`process()`関数がレイアウトから呼び出されるたびに二つの値を表示します。

```
...
// 二つの渡された引数の値を表示します
info(event, " ", command);
...
```

このスクリプトを実行してみると、何かをクリックしたり、値を修正したり、フレームを変更する度に`info()`リクエスタが現れ、`event`の数値や使用されているコマンド、コマンドで使用されているパラメータ（もしあれば）が表示されます。引数`event`から返される整数値は定数`NOTHING`, `COMMAND`, `TIME`, `SELECT`または`RENDER_DONE`を表しています。この表示データによって何がレイアウトコマンドなのか、そうでないのかをよく表してくれます。さらに重要なのは、`process()`関数がアニメーション時にどのくらいの頻度で呼び出されるのかがわかることでしょう。アニメーション中にこのスクリプトを使用してみると、大量の`info()`リクエスタが表示されてしまい、物凄くいらいらしてしまいますね。

このため`process()`コードで莫大な処理時間を浪費したくはないと思うでしょう。現在ユーザーが処理を行う度に`info()`リクエスタを手動で閉じなければならないため、スクリプトの生産性が著しく落ちているような状態です。しかし`info()`文ではなく、プロセッサ集中型コマンドが多くある場合にも、レイアウトとの対話は遅くなってしまいます。これを実演するため、`info()`コマンドを以下のコードで置き換えてみましょう。

```
...
for(i = 0; i < 100000; i++)
{
}
...
```

このコードではLScriptに時間ループ処理を挿入しています。勿論、`for`ループ文にはコマンドや関数は何も割り当てられていません。何の処理も行いませんが、時間だけはかかります。時間がかかることで、ループもまたレイアウトが何かの処理を行うのを妨害してしまいます。レイアウトが`process()`関数を呼び出す度に瞬間、感知できる程度の一時停止を引き起こします。

注意
一時停止を生成する時間の長さは、実際にはコンピューターの処理速度とforループの実行時間に依存しています。

スクリプトを実行し、画面上の周りをヌルが移動するシーンを作ってみてください。ループの処理がレイアウトのインタラクティブ性をどれほど遅らせているのかが体感できるでしょう。レイアウトが`process()`関数を呼び出すたびにタイムループ処理を実行しているからです。では以前のように`for`ループを`info()`文へと置き換えます。

ここに最終的なコードがあります。

```
//
// masterTest.ls: レイアウトからMaster Classスクリプトへと
// 送信されたコマンドを紹介するスクリプト
//
```

```
@version 2.3
@warnings
@script master
@name masterTest

flags
{
  // レイアウトのイベント全てを監視するためフラグを設定します
  return(SCENE);
}

process: event, command
{
  // 二つの渡された引数の値を表示します
  info(event, " ", command);
}
```

このスクリプトを保存し取って置いてください。後ほど例 2 で使用します。

例 2 ： selected.ls

次に取り組む例は、インターフェイス内で選択されているアイテムを表示するだけのものです。このスクリプトでは、レイアウトからのコマンドを監視し選択状態を更新することで、Master Class スクリプトについて既に学んだことを広げていきます。具体的には、選択アイテムを扱うコマンドを探していきます。

この例では、`process()`関数でスクリプト処理の大半を実行します。しかしこのスクリプトの最終版ではユーザーインターフェイスが必要となります。このマニュアルの章全体で LScript におけるインターフェイス作成の議論にささげてきました。ですから何が起きているのかを詳細に解説するよりも何をしているのかだけをなぞるだけにします。インターフェイスについては、この後の"インターフェイス"の章、および"LSIDE"の章でカバーします。

スクリプトの設定

まずはヘッダー情報から始めましょう。

```
//
// selected.ls - 現在選択されているアイテムを
// 列挙する Master Class のスクリプト
//
```

```
@version 2.2
@warnings
@script master
@name selected
```

ではこの関数で使用する関数を挿入しましょう。最終的には、これらの関数にはさらにコードが増えることになりますが、現段階ではスクリプトの設定だけです。

```
create

{
  // ここに初期化作業が必要となります
}

flags
{
  // この関数ではシーンがクリアされた場合に
  // レイアウトにスクリプトをクリアする必要があるかどうかを指示します
  return(SCENE);
}

process: event, command
{
  // ここではレイアウトが送信するコマンドを調べます
}
```

前例で`info()`リクエスタを除去して記述したスクリプトによく似ていますね。何度スクリプトを実行してみても何も起こりませんが、この時点では何もない状態で正しいのです。

次にユーザーが現在何を選択しているのかを調べる方法を設定します。アイテムの選択などに関しては、Scene Object Agentを作成すればシーン設定を取得することが可能です。これは一度だけ、しかもスクリプト内のかなり速い段階で処理しておきたいので、`create()`関数の中に追加することにしましょう。この関数はスクリプトが呼び出されたときに自動的に実行されます。

```
create
{
  // Scene Object を作成します
  sceneObj = Scene();
}
```

`sceneObj` Object Agent を `create()` 関数内部においてローカルに宣言していますので、スクリプトにおいて他の関数内ではそのデータにアクセスすることは出来ません。ローカルではなくグローバルで変数を宣言しなくてはなりません。そのために `@name` プラグマと `create()` 関数の間に変数の宣言文を挿入します。

```
...
@name selected
// ここで大域変数の宣言
sceneObj;
create
...
```

この変数はグローバルとして宣言されましたので、スクリプト内部にある関数は全て Object Agent のメソッドとデータメンバを共有できます。

注意
Scene Object Agent で利用可能なデータメンバとメソッドの完全なリストについては、リファレンスマニュアルの 第 11 章： Scene Object Agent を参照してください。

作成された `sceneObj` Object Agent を使えば、`getselected()` メソッドを使用してユーザーが現在選択している全アイテムを表す Object Agent（`Mesh`、`Light`、`Camera`）の配列を取得することが出来ます。`process()` 関数内に以下のコードを追加します。

```
...
process: event, command
{
  // 現在選択されているオブジェクトを取得します
  If(event == COMMAND)
  {
    selItems = sceneObj.getSelect();
    ...
```

まずは引数 `event` が定数 `COMMAND` を返しているかどうかを確かめましょう。こうすれば監視する必要のない他の種類のイベント全てを除くようになります。作成されたこの配列で様々な Object Agent を反復することが出来、`for` ループを使用してメソッドやデータメンバにアクセスが可能になります。

```
  ...
  {
    selItems = sceneObj.getSelect();
    for(i = 1; i <= selItems.size(); i++)
```

```
        info(selItems[i].name);
    }
}
```

このforループ文は、1番目のインデックス(1)からselItems[]配列のサイズをカウントしています。配列のサイズは格納されているアイテム数により確定されます。選択アイテムが三つある場合には、三つのアイテムがselItems[]配列に保存されています。ですから配列のサイズは3となります。この文を設定すれば、forループ文を使用して配列内の各インデックスを通じて、メソッドやデータメンバにアクセスできるようになります。

現時点では簡単にするため、forループ文にはinfo()関数だけをつけています。このコード行はレイアウトから取得するデータの種類が何であるかを見るためだけの一時的なものです。これで正しい軌道上にいるのかどうかを確定することが出来ます。スクリプトを組み始めたばかりの頃は、これらの状況判断を設定するのが非常に重要なことなので、スクリプトが想定したとおりに動いているかどうかもわからないまま様々な処理は出来ません。数行もしくはスクリプト全体をデバッグするよりも、バックアップを取り誤っている1行だけを修正する方がはるかに簡単なのです。

ここにコードを紹介します。

```
//
// selected.ls - 現在選択されているアイテムを
// 列挙するMaster Classのスクリプト
//
@version 2.2
@warnings
@script master
@name selected
// ここで大域変数の宣言
sceneObj;

create
{
  // Scene Objectを作成します
  sceneObj = Scene();
}

flags
{
  // この関数ではシーンがクリアされた場合に
```

```
  // レイアウトにスクリプトをクリアする必要があるかどうかを指示します
  return(SCENE);
}

process: event, command
{
  // 現在選択されているオブジェクトを取得します
  selItems = sceneObj.getSelect();
  if(event == COMMAND)
  {
    for(i = 1; i <= selItems.size(); i++)
      info(selItems[i].name);
  }
}
```

スクリプトのテスト

現在のシーンをクリアし、Masterプラグインパネル上のプラグインリストへMaster Classスクリプト"selected.ls"を追加して下さい。

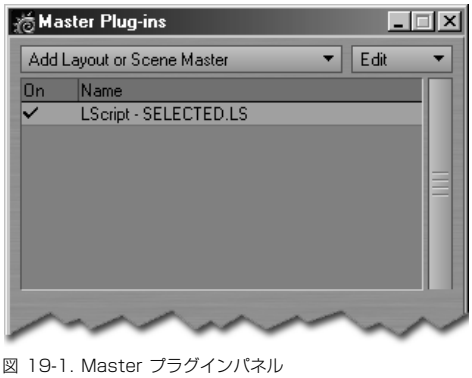

図 19-1. Master プラグインパネル

1. Cameraアイテムを選択します。info()リクエスタが現れ、カメラの名称を表示します。
2. OKボタンを押しパネルを閉じます。
3. Lightアイテムを選択します。同様のリクエスタが出てきますが、今度はライトの名称が表示されます。

これらのステップでは、単一のアイテムが選択されているときにスクリプトが動作するかどうかをテストしています。スクリプトは予測どおりに動いているように見えますが、これで終わりではありません。複数のアイテムが選択されている状態でスクリプトがうまく動くのかどうかをテストしてみましょう。カレントシーンに2～3個オブジェクトを読み込んでみるのが一番簡単な方法でしょう。

1 最初のオブジェクトを選択します。選択しているオブジェクトの名称が表示されているリクエスタ一つ現れます。

2 OKボタンを押してパネルを閉じます。

3 SHIFT選択で二番目のオブジェクトを選択します。もう一度選択されているオブジェクトを表示するリクエスタが現れます。

4 OKボタンを押すとパネルが閉じます。しかし二番目のリクエスタが現れ選択していた最初のオブジェクトが表示されているリクエスタが現れます。

選択しているオブジェクトの数だけこの作業が繰り返され、前回選択したオブジェクト表示されているリクエスタが連続して開くのがわかりますね。最終的にリクエスタが表示される個数は選択したアイテムの個数に依存します。見てわかるとおりデータ表示にこのメソッドを使用することで、スクリプトはもしかすると多量のリクエスタを生成することになるかもしれません。これは効果的とはとても言えませんが、このテストはスクリプトの`for`ループ選択が適切に動作していることを証明しています。まずい設計ではありますが動作はしています。

このスクリプトでは一つ大きな問題があります。問題を解説していくためにカレントシーンの0フレームを1フレームに変更してみてください。リクエスタ群が再び現れてきましたね。これは前述の通り、`process()`関数をトリガーしている全てのイベントに対して発生するからです。しばらくの間、複数の`info()`リクエスタが出てきますが、この問題を我慢すると想定するなんてユーザーには耐えがたいことです。

CS コマンドの決定

選択しているアイテムを表示するコードを、いつ実行するのかを決定しなければいけません。実演して見せたとおり、process()関数が呼び出されるたびにコードを実行させたくはありません。情報がユーザーにとって価値を持つ場合にだけ、表示用コードを実行させたいですね。今回の場合、コードはオブジェクトが選択されたときに実行されるようにしましょう。それではこのコード実行の時期を選択させるにはどのように取り組めばよいのでしょうか？

前にも解説しましたが、process()関数に渡される引数の一つには、変数command内に記憶されている値が入っています。例"masterTest.ls"ではprocess()関数が呼び出されるタイミング、引数commandには文字列が含まれていることなどを学びました。この文字列はレイアウトから発行されたコマンドとその引数が含まれています。この文字列を解析することにより、どのコマンドが発行されたのかを把握することが出来、またコードがそのコマンドに対して注意を払うべきかどうかも確定できます。

前述の例masterTest.lsを使用すれば何百という使用可能なコマンドの中から、一体どのコマンドがコードで必要となるのかを確定することが出来ます。masterTest.lsスクリプトを実行し、Cameraアイテムを選択します。以下のような文字が描画されているinfo()リクエスタが表示されましたね。

```
"SelectItem 300000000"
```

ここで使用しているコマンドはSelectItemで、その引数は300000000です。現時点で引数はそれほど重要ではありませんが、この引数の値はLightWaveが内部的にアイテムを識別するために使用している番号です。今回のスクリプトではこの引数を使う必要はありません。SelectITemというコマンドだけが必要なのです。

探しているコマンドが見つかりましたので、このコマンドに遭遇した場合にのみforループコードの処理を行うようにしましょう。forループの真中に行を追加します（括弧を忘れないようにして下さい）。

```
...
if(event == COMMAND)
{
  if(command == "SelectItem")
  {
    // items[]配列のコンテンツを表示します
    for(i = 1; i <= selItems.size(); i++)
      info(selItems[i].name);
  }
}
```

スクリプトを実行し今までどおりテストして、どのように処理されるかを確認してみてください。奇妙なことに、このコードを実行してみても何も起こらないことに気がついたでしょう。アイテムをいくつ選択してもスクリプトはinfo()リクエスタが表示されなくなりました。どうしてでしょう？どこが悪いのでしょうか？

問題は、`if-then`文が二つの文字列、変数`command`内の文字列と"`SelectItem`"が一致するかどうかを比較しているところにあります。演繹法を使用すると、`for`ループと`info()`文が実行されない場合、この二つの文字列値が等しくないのです。そのためにコードは`if-then`文に割り当てられている文を見ることがないのです。`if-then`文で何かまずいことが起こっているのです。

`if-then`文の両側は決して等しくはなりません。なぜなら変数`command`の実際の値は"`SelectItem`"ではなく"`SelectItem 300000000`"だからです。このため文字列が等しくなることはなく、`for`ループ文が実行されることもありえません。これを証明するため、`if-then`文の前にもう一つ`info()`コマンドを挿入して、変数`command`の値の現在値を表示するようにしましょう。コードを実行すると、`for`ループ文が実行される機会が来ないことがわかるでしょう。

文字列`command`全体から、引数部分を無視し実際のコマンド部分だけを取り出す必要があります。`parse()`関数を使用すれば取り出すことが出来ます。`parse`という言葉は、あるもの（言葉など）を構成している部品へと分解していくことを意味しています。これがこの関数の動作となります。区切りの文字と分解される文字列を指定すると、`parse()`関数は初期文字列を構成していた全ての部分文字列を含む配列を返します。区切り文字列は単に`parse()`関数が指定した文字列を分解するためだけに検索する文字列です。例えば

```
word = parse( " ", "This is a test." );
```

区切り文字列はスペース（" "）で、分解用文字列は " This is a test. " です。結果は以下のように区切られた文字列が返されます。

```
word[1] = "This"
word[2] = "is"
word[3] = "a"
word[4] = "test."
```

`process()`関数内の`if-then`文の前に以下の行を挿入してください。

```
...
selItems = sceneObj.getSelect();
// 変数 command を解析します
command = parse( " ", command);
if(event == COMMAND)
{
```

```
  if(command == "SelectItem")
  ...
```

これらの行ではレイアウトから送られてきた変数 command を受け取り、parse()関数へと送り出しています。parse()関数から返される配列には、変数 command が分解され保存されています。スペース（" "）文字を使用して変数 command を分解しているため、最初のインデックスはコマンドを、2番目のインデックスは引数を表す配列が生成されるようになります。前述の例では文字列 command は以下のとおりです。

```
"SelectItem 300000000"
```

変数 command は以下のように分解されます。

```
command[1] = "SelectItem"
command[2] = "30000000"
```

command[1]に保存されている値と文字列"SelectItem"を比較することにより、info()リクエスタを多数生成することになります。では、if-then 文を配列 command[]の1番目のインデックスと比較するように修正しなくてはいけませんね。新しいコードは以下のようになります。

```
// 変数 command を解析します
command = parse( " ", command);
if(event == COMMAND)
{
  if(command[1] == "SelectItem")
  ...
```

もう一度スクリプトを実行しアイテムを選択します。スクリプトは選択処理の場合にのみ更新されることに気がついたでしょう。フレームの変更やウィンドウを開くといった処理に対しては、for ループ文に割り当てられている文が実行されなくなりました。表示用のコードを実行するアクションを単に限定することで、このスクリプトの有用性を証明しました。

不幸なことに、この小さな修正には欠点があります。info()リクエスタはオブジェクトを選択したときには表示されますが、SHIFT クリックメソッドを使用してオブジェクトの選択を追加してみてください。何も起こらないでしょう。さらにテストを重ね、最初のアイテムが選択された時だけ望みどおりの結果となり、複数のオブジェクトが選択されたときには表示されないことがわかるますね。問題はコード中のエラーではなく、基本検索にエラーがあります。

アイテムが選択されたときにレイアウトが発行するコマンドは"SelectItem"コマンドであると最初に発見した時に、すぐにそれを実装してコマンドの実例を見てみました。もう少し時間をかけてレイアウトにおける選択システムがどのように動作しているのかを理解するべきでした。コードを修正する羽目になる前に、必要となる追加情報へと導いてくれるはずだったのです。レイアウトがアイテム選択を扱うためのコマンドは一つだけではなく、実際には三つのコマンドを使用しているということを理解していませんでした。

もう一度MasterTest.lsスクリプトを実行してテストを繰り返してみると、アイテムを選択に追加したり除去したりするのにレイアウトは三つのコマンドを使用しているのがわかりますね。これらのコマンドが重要となるのです。

```
"SelectItem"
"AddToSelection"
"RemoveFromSelection"
```

コードが適切に動作するためには一つだけではなく、三つのコマンド全てをテストする必要があるのです。幸運なことに、新しい要求を満たすためにスクリプトを修正するのはかなり簡単です。単に`if-then`文を一つではなく三つのコマンドと比較するように修正するだけでよいのです。

```
if( command[1] == "SelectItem" or
command[1] == "AddToSelection" or
command[1] == "RemoveFromSelection")
```

注意
コードを読みやすくするため、通常１行で足りるコード行を三つに分けました。どちらでも動作しますので、実装は開発者に委ねられます。

速度と効率性を保つためには、よく見てみると`process()`関数が呼び出される度に配列`selItems[]`を作成する必要がないことがわかるでしょう。選択が変更された時にのみ、この配列をもう一度作成する必要が出てくるのです。ですから`selItems[]`配列を作成するコード2行を`if-then`ブロックの中へと移します。

これが作業コードとなります。

```
//
// selected.ls - 現在選択されているアイテムを
// 列挙するMaster Classのスクリプト
//
@version 2.2
@warnings
@script master
@name selected
```

```
// ここで大域変数を宣言
sceneObj;

create
{
  // Scene Object を作成します
  sceneObj = Scene();
}

flags
{
  // この関数ではシーンがクリアされた場合に
  // レイアウトにスクリプトをクリアする必要があるかどうかを指示します
  return(SCENE);
}

process: event, command
{
  // 変数 command を解析します
  command = parse(" ", command);
  if(event == COMMAND)
  {
    if( command[1] == "SelectItem" or
    command[1] == "AddToSelection" or
    command[1] == "RemoveFromSelection")
    {
      // 現在選択されているオブジェクトを取得します
      selItems = sceneObj.getSelect();
      // 配列 items[]のコンテンツを表示します
      for(i = 1; i <= selItems.size(); i++)
        info(selItems[i].name);
    }
  }
}
```

シンプルに

スクリプトは正常に動くようになりました。ここまでが第1段階です。今度はコードをじっくりと眺めてみて、より小さく、より速く、より効率的に実行出来ないかを調べてみます。スクリプトは小さなものですが、使用するメモリ量を減らせる方法が一つありましたね。ある特定の環境下では実行速度が速くすることが出来るかもしれません。

まずは配列`selItems[]`をどのように扱っているのかを調べてみましょう。Scene Object Agentの`getselected()`メソッド実行時に、前述の`selItems[]`配列に保存されているObject Agentsの配列を生成します。これらObject Agentはデータメンバとメソッドの形式でそれぞれにかなりな量のデータを保持しており、それぞれ個々のアイテムに対する様々なプロパティを記述しています。

さらによく見てみると、それぞれObject Agentのアイテム名称だけが必要であるとわかりますね。配列から不必要なデータを破棄すれば、保存されているデータは動的に小さくなり、メモリ量も小さくなります。Object Agentからデータを削除することは不可能ですが、アイテムの名称だけをもつ新しい配列を作成することは出来ます。こうすれば望みどおりの結果も得られますし、後でスクリプトに追加する機能も設定可能です。

新規配列を作成するために、`selItems[]`配列からアイテムの各名称だけをコピーし、`items[]`という新しい配列に保存しましょう。`selItems[]`配列構築の後で、もう一つ別の`for`ループ文を配置すれば作成できます。

```
...
selItems = sceneObj.getSelect();
for(i = 1; i <= selItems.size(); i++)
  items += selItems[i].name;
  // items[]配列の中身を表示します
  ...
```

この場合、(`+=`)命令子で配列を作成すると一つの重要な欠陥が生じてしまいます。関数が呼び出されるたびに同じ配列を使用しています。配列の値は各呼び出しの間で保存されており、配列`items[]`は重複した名称で急激に崩れていきます。この問題を実証するため、現在の状態でスクリプトを実行してみましょう。

この問題を修正するには、使用する間に変数へ`nil`の値を代入して配列の値を壊すだけでよいのです。そうすれば`for`ループが変数内へアイテム名称を保存するたびに、常に最初のインデックスから始まることになります。

```
...
selItems = sceneObj.getSelect();
// 変数 items をクリアします
items = nil;
// items[]配列をより小さく、より効果的に作成します
...
```

問題が解決されると同時にスクリプトもより効率的になりました。selItems[]配列で使用されているメモリを開放するため、変数の破棄が必要になります。items[]配列を破棄するのと同じ方法で行います。selItems[]変数にも、値 nil を単に割り当てるだけです。

```
...
for(i = 1; i <= selItems.size(); i++)
  items += selItems[i].name;
// selItems 配列で使用されているメモリを開放します
selItems = nil;
// 配列 items[]の中身を表示します
...
```

修正箇所はもう一つだけあります。二番目の for ループにおいて使用される配列の名称です。selItems[]配列の代わりに、items[]配列を反復処理するように設定しなおします。

```
...
// 配列 items[]の中身を表示します
for(i = 1; i <= items.size(); i++)
  info(items[i]);
```

process()関数の中身のみを修正した最新版です。

```
process: event, command
{
  // 変数 command を解析します
  command = parse(" ", command);
  if(event == COMMAND)
  {
    if( command[1] == "SelectItem" or command[1] ==
"AddToSelection" or command[1] == "RemoveFromSelection")
    {
      // 現在選択されているオブジェクトを取得します
```

```
      selItems = sceneObj.getSelect();
      // 変数 items をクリアします
      items = nil;
      // より小さくより効果的な items[]配列を作成します
      for(i = 1; i <= selItems.size(); i++)
        items += selItems[i].name;
      // selItems 配列で使用されているメモリを開放します
      selItems = nil;
      // 配列 items[]の中身を表示します
      for(i = 1; i <= items.size(); i++)
        info(items[i]);
    }
  }
}
```

よりきれいに

解決し残している最も明らかな問題点は、items[]配列の中身をより効率よく表示するにはどうすればよいのかということです。現在使用している info()リクエスタでも動作するにはしますがあまりにも扱いにくいですし、最終スクリプトで使用するには邪魔です。アイテム名称の全リストを一度で表示したいと思います。残念なことに、これを叶えるためにはインターフェイスを作成しなければなりません。

注意
インターフェイス作成にはただひたすら努力が必要となるでしょう。入出力の設計やユーザーインターフェイスの実装にこのマニュアルの１章丸ごと費やしました。ですからこの議題に関しては理論や解説をここではあまり深くは行いません。

基本的には、リクエスタを作成し items[]配列の中身を表示するためのリストコントロールを使用します。このコントロールは複数のアイテムをスクロール可能な一つのリストで一度に表示させることができます。でもこれで問題が解決すると同時に、リストボックスコントロールは設定するインターフェイスコンポーネントのうち、かなり複雑なコントロールの一つでもあります。適切に動作するためにはサポート関数をいくつも記述しなくてはなりません。

まずは item[]配列をグローバルにしておく必要があります。こうすれば全ての関数、特にインターフェイスコードを扱う options()関数から配列のコンテンツにアクセス出来るようになります。items[]配列をグローバルにするため、スクリプト冒頭において変数 sceneObj の前に変数を追加します。

```
...
@name selected
items, sceneObj;
create
...
```

次に`options()`関数を設定しましょう。この関数は、ユーザーがMasterプラグインパネル上のプラグインのインスタンスをダブルクリックした時、もしくはプラグインの編集ドロップリストからプロパティオプションが選択されたときに、自動的に呼び出されます。ここではリクエスタを適切に描画し更新するため必要となるインターフェイスコードが全て含まれています。

```
options
{
  // リクエスタ既に開かれていないかどうかをチェックします
  if(reqisopen())
    // 開かれている場合には閉じます
    reqend();
  else
  {
    // "Selected:"という名称のリクエスタを作成します
    reqbegin("Selected:");
    // リクエスタのサイズは130x300とします
    reqsize(130, 300);
    // リストボックスコントロールを作成します
    c0 = ctllistbox("Selected Items:", 100, 290, "c0_count",
"c0_name");
    // リクエスタをノンモーダルで開きます
    reqopen();
  }
```

その後リストボックスコントロールが適切に動作するために必要な関数を追加します。

```
c0_count
{
  // この関数は変数のサイズを返します
  return(items.size());
}
```

それに

```
c0_name: index
{
  // この関数は index で指定された配列の値を返します
  return(items[index]);
}
```

最終コード

これで全てです。スクリプトが完成しました。以下はインターフェイスコードや機能をいくつか追加し、ほんの少し構造を修正した最終スクリプトです。

```
//
// selected.ls - 現在選択されているアイテムを
// 列挙する Master Class のスクリプト
//
@version 2.2
@warnings
@script master
@name selected
items, sceneObj;

create
{
  // Scene Object Agent を作成します
  sceneObj = Scene();
  // getSelected() UDF を呼び出します
  getSelected();
  // プラグインリストに表示される解説文を設定します
  setdesc("Selected Items: ", items.size(), "item(s)");
}

flags
{
  // この関数ではシーンがクリアされた場合に
  // レイアウトにスクリプトをクリアする必要があるかどうかを指示します
  return(SCENE);
}
```

```
  process: event, command
  {
    // 文字列 command を解析します
    command = parse(" ", command);
    // どのコマンドが更新をトリガーするのか確定します
    if( command[1] == "SelectItem" or command[1] ==
"AddToSelection" or command[1] == "RemoveFromSelection")
    {
      // getSelected UDF を呼び出します
      getSelected();
      if(reqisopen())
        requpdate();
    }
  }

  options
  {
    // リクエスタが既に開かれているかどうかをチェックします
    if(reqisopen())
      // 開かれている場合にはリクエスタを閉じます
      reqend();
    else
    {
      // "Selected:"という名称のリクエスタを作成します
      reqbegin("Selected:");
      // リクエスタのサイズは 130x300 とします
      reqsize(130, 300);
      // リストボックスコントロールを作成します
      c0 = ctllistbox("Selected Items:", 100, 290, "c0_count",
"c0_name");
      // リクエスタをノンモーダルで開きます
      reqopen();
    }
  }
```

```
c0_count
{
  // この関数はインターフェイスのリストボックスコントロール用の
  // 関数であり items[]配列のサイズを返します
  return(items.size());
}

c0_name: index
{
  // この関数はインターフェイスのリストボックスコントロール用の
  // 関数であり インデックスで指定した配列の値を返します
  return(items[index]);
}

getSelected
{
  // この UDF はこのスクリプトにおいて二つの機能を提供するために追加されました
  // これは単にグローバル変数である item[]配列を構成するためのものです
  // items[]配列を破棄します
  items = nil;
  // 現在選択されているオブジェクトを取得します
  selItems = sceneObj.getSelect();
  // items[]配列をアイテム名称で構築します
  for(i = 1; i <= selItems.size(); i++)
    items += selItems[i].name;
}
```

第 20 章：インターフェイス概論

このマニュアルを通し、LScriptがLightWaveアーティストへ信じられないほどの力を提供出来るということを実証してきました。ここまでで、ポイントやポリゴンの修正、サーフェイスの調節、シェーダーの作成、アニメーション作成のスクリプトなど様々なスクリプトが書けるようになりました。LightWaveコミュニティに提供する用意が万端整ったと思うかもしれませんが、スクリプト能力は使いやすさという新たな局面に突入しようとしています。

このセクションでは、インターフェイスコード記述の入出力についてカバーしていきます。前章までと同様、簡単な例から始めインターフェイスコード独自の新規コマンドや関数などを解説していきます。インターフェイスコマンド全リストについての解説は行いませんが、コマンドの少数のサブセットを学ぶことで、残りのコマンドについてもどのように動作するかといった一般的な概念を取得できるようになります。インターフェイスのセクション全体を読み終える頃には、かなり高度なユーザーインターフェイス（UI）に取り組んだり、少なくともそれがどういう動作をしているのか、理解できるようにはなっているでしょう。

インターフェイスとは？

インターフェイスとはどういうものかというのは皆さんお分かりですね。コンピュータをつけたり、LightWaveなどのアプリケーションを実行するたびにインターフェイスを使用しています。インターフェイスとはユーザーがボタンを押したりスライダを動かしたり文字を入力することが出来る領域を指します。この発明によりコンピュータ上の生活はより容易になるのです。不運にもインターフェイス無しのソフトウェアを使用しなければならない人に尋ねて御覧なさい。大抵、大量にタイプを打たねばならず、何時間も黒画面を見入ることになります。ちっとも面白くありませんね。

スクリプトにインターフェイスを追加すれば、スクリプトに対する親切さが増しますが、新しいレベルの複雑さをもコードに追加することになるでしょう。インターフェイスのコードは冗長で、一度は美しくすっきりとしていたスクリプトが、いとも簡単にぐちゃぐちゃになってしまうほどの反復性コードとなる場合がほとんどです。コードが騒然となってしまうことで、正しく動作させるために終にはスクリプトのデバッグまでもを必要とするエラーを生成してしまいかねません。では何故インターフェイスを使用したがるのでしょう？もっと大切なことは、スクリプトに対しインターフェイスは何を行うというのでしょう？

インターフェイスは単にコードへの窓または入口でしかありません。この入口でユーザーは変数に保存されている特定の値を見たり変更することが可能になるのです。そう、コード内の変数を変えるだけで以上の処理は可能ではありますが、インターフェイスはスクリプト実行中にこの処理を可能にしてしまうのです。以下の状況を想像してみてください。

実際に素晴らしいポリゴンリダクションスクリプトを作成したとします。モデラーに存在するツールの大半と同様、処理するリダクションのレベル（1, 2, 3もしくは4）の設定を直接コードに設定出来ます。処理回数の値は `for` ループ文をコントロールする変数によって制御されています。では4回の反復処理を望むユーザーと同様、1回だけの反復処理を望むユーザーを満足させるためのスクリプトを作成するにはどうしたらよいのでしょうか？

ループ変数にそれぞれの値を持たせたスクリプトを4つ書くことも出来るでしょう。これでそれぞれぞれの問題は解決されますが、コードの効率的な使用は難しくなりますね。ユーザーが5回の反復処理を望む場合にはどうなるのでしょう？答えはなんでしょうか？

答えはループをコントロールする変数にインターフェイスをリンクすることなのです。インターフェイスはユーザーからの処理する分割用数値を要求します。インターフェイスからの変数はコード内で変数へとリンクされているため（実際には同じ変数となりますが）、画面上の値を変更すれば、コード内の値も変更されます。この機能はスクリプトに信じられないほど多大な量の柔軟性を追加してくれます。基本的に同じ処理を行う101個の異なるスクリプトを作成するのではなく、柔軟性を持つスクリプトを一つ持つことになります。これが静的なスクリプトと動的なスクリプトを作成する違いとなります。

静的なスクリプト

前章で作成された例は静的なスクリプトです。毎回スクリプトが実行される度にスクリプトの変数に保存されているのは同じ値のままです。当然のこととして、スクリプトは同じ処理を行うことになります。シーンが読み込まれようとサーフェイスが作成されようとも、スクリプトは同じ方式で何度も何度も処理します。スクリプトの変数が静的変数に固定されて組み込まれているからです。以下のコードの例で静的変数を解説します。

```
main
{
  objFile = "c:/newtek/objects/animals/cow.lwo" ;
  load(objFile);
}
```

何度このスクリプトを実行しようとも、常に同じ処理を実行します。静的変数である`objFile`に保存されている値は変更できませんので、変わることは決してありません。しかし、もう一つのオブジェクトを指すパス情報を修正したとしたら、スクリプトの関数は変更され、異なるオブジェクトを読み込むことになります。

```
main
{
  objFile = "c:/newtek/objects/animals/spider.lwo" ;
  load(objFile);
}
```

spiderオブジェクトを読み込むとても便利なスクリプトを作っても、異なるオブジェクトを読み込ませる度にコードを修正したり異なるスクリプトを作成しなければならないという、相変わらず静的なスクリプトのままです。スクリプト実行中にこの変数を修正する方法が欲しいのです。変数に保存されている値の中身によって異なる動作を実行するスクリプトであって欲しいのです。これを行うのが動的なスクリプトです。

動的スクリプト

スクリプト内部においてデータを固定してしまうのではなく、変数に対してインターフェイスを作成することにより、変数の値を動的に作成することが可能です。スクリプト実行中に変数の値を変更出来るようにすれば、様々な機能やオプションを提供することが出来ます。以下の例では一箇所だけ修正を加えた前述の例をもう一度見てみましょう。

```
main
{
  // インターフェイスコード
  // ここでユーザーからオブジェクトのパス情報を取得し
  // 変数 objFile に保存します
  load(objFile);
}
```

注意
コードを読みやすくするために、実際のインターフェイスコードはまだ書いていません。コードはこれから作成していきますのでご心配なく。

変数 `objFile` に入る値を特定の値に決める代わりに、インターフェイスを作成してユーザーからの情報を要求することが出来ます。インターフェイスを実行した後、パス情報が変数 `objFile` に保存されます。どのような手段でデータを取得するかに関係なく、`objFile` が有効な LightWave のオブジェクトファイルである限り、モデラーはパスに保存されているオブジェクトを読み込みます。

スクリプトは動的であるとみなされます。スクリプトの関数を実行する度に変更しなくても、結果的にはそうなります。開発者ではなくユーザーがどのオブジェクトを読み込むのかを確定出来るため、より使いやすいスクリプトになっています。これがインターフェイスを記述する最も大きな利点の一つです。

リクエスタ

インターフェイスは全てリクエスタから始めなければなりません。リクエスタとはインターフェイスが最初に作成される時点で表示されるパネルです。リクエスタには、スクリプトからデータを要求したり表示したりするためのコントロールと呼ばれるエレメントを配置することが出来ます。各リクエスタには何十、時には何百ものコントロールがあります。画面上に必要となるコントロール全てを表示する領域だけが、コントロール数の制限となります。

注意
何百ものエレメントを持つ、大きなインターフェイスを作成するのは賢いやり方ではありません。このようなスタイルは、ユーザーを圧倒し困惑させるだけです。

リクエスタウィンドウを画面上で動かしてみると、リクエスタに付けられているコントロールも全てパネルと一緒に移動しますね。本来、コントロールはリクエスタにリンクされているのです。LScriptは追加したコントロールを編集中のカレントリクエスタに対し自動的に割り当てます。ですから、インターフェイスに割り当てずにコントロールを作成することは不可能です。

ユーザーとの対話のレベルは、リクエスタのインターフェイスを使用してスクリプトが何を必要としているのかに依存します。例えば、頻繁に使われる`info()`関数は、リクエスタを作成し文字列を表示するだけです。ウィンドウを閉じるために"OK"ボタンを押す以外、インタラクティブにやり取りすることは何もありません。ですが、他のスクリプトではユーザーがデータを入力したり値を見るなどの様々なコントロールを持っている場合があります。

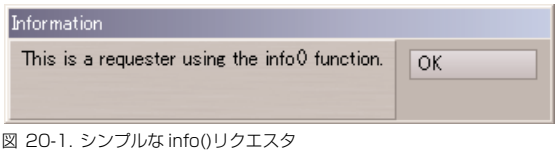

図 20-1. シンプルなinfo()リクエスタ

各リクエスタには、パネルのタイトルバー表示部分に名称を割り当てることが出来ます。パネルのサイズとリクエスタ領域は画像の座標値と同様、二次元ピクセル値（XとY）で測定されます。パネルに対しサイズを指定することも出来ますし、LScriptが適切なサイズを自動で確定することも出来ます。LScriptはどのコントロールが使用されるのか、各コントロールが正しく表示されるにはどのくらいの領域が必要となるのかなどを考慮に入れます。時間が絶対必要となる場合、短時間で終了するインターフェイスへとたどり着きます。

図 20-2. 空のリクエスタ

LScriptは画面上でのコントロールの配置、各パネルやコントロールの色、リクエスタを開いたり閉じたり、最小化、最大化を行うために必要とされる全てのボタンなど、ありとあらゆるパネル属性を自動的に扱います。ですからリクエスタは全て適切に動作するように作成され、他のスクリプトやLightWave内にあるプラグインと同じような外観となるのです。これらのデフォルトがなければインターフェイス作成処理にはさらに時間がかかり、不測のエラーを引き起こしがちになるでしょう。

このパネルは単独で何か重要な処理を行うわけではありませんが、この章が終了する時点ではユーザーが対話を必要とするコントロールが全て含まれることになるでしょう。上記リクエスタはインターフェイスが適切に動作するため必要最小限にしてあります。

コントロール

コントロールとは、ユーザーが対話を行うインターフェイス内のエレメントを指します。コントロールそれぞれは、スクリプトからのデータを収集したり、表示するために設計されています。LScriptはスクリプトで使える多くの利用可能なコントロールを用意しています。ボタンやデータを通してスクロールできるスライダ、文字入力可能なフィールド、それにカスタムメードの画像コントロールなどです。これらコントロールはスクリプトリクエスタ上に配置されます。

図 20-3. リクエスタとサンプルのコントロール

上記リクエスタでは、選択可能な様々なタイプのLScriptのコントロールを図解しています。これらのコントロールには直接フィールドにデータを入力できるものもあれば、オプションリストからの選択が可能になるものもあります。スクリプトを実行することで、これらの値を全て取得することが出来、この値はオプションやユーザーが選択した値を確定することもあります。

注意
インターフェイスのコントロールや関数の完全なリストについては、リファレンスマニュアルの第２章を参照してください。

インターフェイスの種類

LightWaveには二種類のLScriptがあります。ツールやユーティリティのように特定のアイテムデータのフレームから独立した機能を持つスクリプトと、Item Animationスクリプトのように、各フレームに対しアイテムデータを評価するスクリプトです。どちらのインターフェイスの種類も同じような方法でコード化され、同じインターフェイスのコマンドや関数を使用することになります。しかし、この二つの共通点も実際のスクリプト構造を比較する段階までです。

前章では、モデラーやGenericクラスでスクリプトのコマンドを含む単一関数、`main()`や`generic()`の使用法について解説してきました。この単一の関数にはスクリプトを開き処理し、閉じるまでに必要となるコード全てが含まれています。理論的にはこの関数内にスクリプトのインターフェイスコードも全て配置されることになります。

注意
もちろん、モデラーまたはGenericクラスのスクリプトは実行中、他にいくつものユーザー定義関数（UDF）を呼び出すことが可能です。

このタイプのスクリプトを実行すると、コード化されているアクションを実行するのみで、コントロールは即座にレイアウトまたはモデラーへと返されます。コードが実行終了したら、スクリプトも終了となるのです。スクリプトはユーザーが再び起動したときにのみ再実行されます。

もう一方で、レイアウトのスクリプトは複数の関数を使用し、追加したい機能全てをサポートします。レイアウトスクリプトでは、処理の大半を行うコードは`process()`関数内にあります。重要なコードの大半が`process()`関数にあっても、他の関数も変数を設定したり、読み込みや書き込みを扱ったり、スクリプト終了後のクリーンアップ処理が行えます。ひとえに何を処理したいのかによるのです。

注意
以下の章を通し、レイアウトクラスのスクリプトを参照していきます。このスクリプトクラスはレイアウトにあるクラスであり、複数の特定化された関数を使用してスクリプトのアクションを実行しています。しかしGenericスクリプトの構造はレイアウトスクリプトよりもモデラーのスクリプトと似通っています。ですからレイアウトスクリプトを参照するというのは、Genericクラスを除いた全レイアウトのスクリプトクラスを参照することになります。

レイアウトスクリプト実行時には、アニメーションの各フレームにおいてコードとシーンデータを計算します。これはシーンのレンダリング時と同様、ユーザーがアニメーションをつけたりプレビューさせたりしている時点でも発生します。これらのスクリプトはフレームが変更される度に呼び出されますから、アニメーション作成の経過途中において何百、何千回も呼び出されることになります。インターフェイスコードの構造の議論において、これは大きな問題を提起します。

例えば、Item Animationスクリプトでインターフェイスを作成するのに、何も考えずにアプローチしてみましょう。重要なコードは`process()`関数内にあるとわかっていますから、インターフェイスを作成するコードを`process()`関数に配置してみようかと思うかもしれませんね。こうすればインターフェイスは確実に呼び出されますし、ユーザーから変数の設定が確実になるでしょう。スクリプトは万事うまくいきそうです。

しかし、この方法ではフレームが変化したという情報を受け取るたびに、`process()`関数が呼び出されてしまうという大きな問題が生じます。ですからItem Animationスクリプトが割り当てられているオブジェクトを編集すると、インターフェイスが現れます。これはすぐに古くなってしまうでしょう。ここで登場するのが`options()`関数です。

インターフェイスコードを`process()`関数から切り離して、`options()`と呼ばれる関数内に配置することで、パネルはユーザーが要求した時にのみ呼び出されるようになります。ユーザーはプラグインのリストウィンドウにあるスクリプトのインスタンスをダブルクリックするか、プラグインのコマンドリストから**Edit（編集）> Properties（プロパティ）**を選択することによって、インターフェイスを呼び出すことができます。

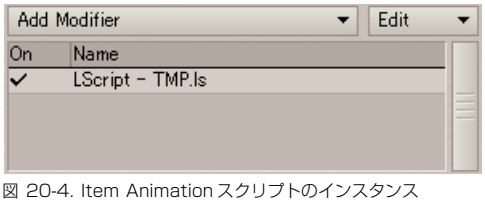

図 20-4. Item Animationスクリプトのインスタンス

この二つのアクションどちらかを実行することで、LScriptはスクリプトコード全体を通して`options()`関数を検索します。関数がスクリプト内部に存在すればそのコードを実行し、その間はスクリプトを停止させます。しかし、関数がスクリプト内に存在しなかった場合、開発者はこのスクリプトに対しインターフェイス描画を望んでいなかったのだと想定し、ユーザーに対しオプションは利用できないとの旨を通知します。

LSIDE ： Interface Designer（インターフェイスデザイナー）

LScript Integrated Developement Environment（LSIDE）には非常に強力なインターフェイス設計アプリケーションが含まれています。"見たままのものをお手元に"（WYSIWYG）というコンセプトのコーディング環境を使用すれば、インターフェイスを手早く簡単にレイアウトし作成することが出来ます。**Interface Designer**では描画したりドラッグしたりリクエスタ上のコントロールを整列させることで、簡単なまたは複雑なインターフェイスを作成することが出来ます。処理が完了すると、`Interface Designer`はリクエスタをLScriptコードとして、スクリプトへコピーアンドペーストしてエクスポートします。

図 20-5. LScript Interface Designer(インターフェイスデザイナー)

この処理は恐ろしいほど素早く効率的ですが、その中で一体何が起きているのかを知っておくべきであると考えます。ある状況下では、Designerからエクスポートされるコードの目的や機能などについて把握し、理解することが必要となるでしょう。また理解することにより、Designerを再起動し古いコードの上に新しいコードをカットアンドペーストするなどの厄介な作業を行うことなく、コードを追加したり修正したり出来るようになります。

インターフェイスがどのように動作するのか、またどのようなコードで作成されるのかしっかりと理解出来るようになる頃には、Interface Designerは間違いなく強力なツールとなり、あなたはDesignerを使用して必要なインターフェイスコードを全て用意できるようになるでしょう。ですから今はDesignerを使うという欲求と戦い、まずはより困難な道を選ぶことにします。初めての大きなプロジェクトを書こうとする時には、次の章で蓄えていく知識が非常に貴重なものとなるはずです。

注意
Interface Designer のより詳細な情報については、このマニュアルの最終章 "LSIDE" を参照してください。

第21章：モデラークラスと Generic（総括）クラスのインターフェイス

Generic（総括）クラス、およびモデラーのスクリプトクラスは、根本的に全く異なる関数やコマンドを使用しているものの、どちらのスクリプトに対するインターフェイスも同じ様式でコード化されます。どちらもユーティリティとして機能するため、インターフェイスはユーザーから必要となる情報を取得すると実際に機能を実行するコードへと進むような形に設計されています。この簡易性のおかげで、インターフェイスのコーディングはかなり易しく、構造や仕掛けをそれほど必要とはしません。このタイプのインターフェイスが他のスクリプトタイプのインターフェイスに比べ強力でないとは言っていません。ただ、より簡易化されているのです。

注意
クラスのスクリプト処理に対する制限と同じ制限が、インターフェイスコードにもつけられているということが重要です。例えばモデラーのインターフェイスは、レイアウトインターフェイスで見られるカレントのフレーム情報についてはアクセスすることが出来ません。

モデラークラスのインターフェイス

モデラースクリプトは、ジオメトリのモデリングをしやすくするために設計されたツールです。このタイプのスクリプトではシーンフレームやライト、カメラの設定などにアクセスする必要がないですし、シーンのフレーム情報が変化するたびに処理する必要もなく、インターフェイスコードの学習を始めるには非常に良い出発点となるでしょう。スクリプトの構造、特にインターフェイスコードを限りなく簡易化しています。

注意
再教育講習が必要であれば、リファレンスマニュアルのモデラースクリプトの章を参照してください。

モデラースクリプト用にインターフェイスを作成するのは非常に簡単です。いったんヘッダーや変数などの通常のスクリプト形式が処理されれば、インターフェイスコードが始まります。インターフェイスデータがユーザーから収集され適切な値が設定されたら、スクリプトのメイン処理コードが始まります。他のパネルを開かない限りは、スクリプトにおけるインターフェイス部分はこれで終了です。

始めてみよう！

この例ではインターフェイスコードの導入に焦点をあてていますので、驚くほど簡単なスクリプト用のモデラーインターフェイスを作成していきましょう。このスクリプトは二つの数値を足して、その結果を`info()`パネルに出力するというものです。それではモデラークラスのスクリプト"ModInterfaceTest"を設定していきましょう。

```
@script modeler
@name ModInterfaceTest
@version 2.3
main()
{
}
```

より簡単にするため、大まかな輪郭を挙げていきます。

1. インターフェイスを描画します
2. インターフェイスから値を取得します
3. 二つの値を加算します
4. 結果を表示します

これをスクリプトに挿入してみます

```
...
main()
{
  // インターフェイスを描画します
  // インターフェイスから値を取得します
  // 二つの値を加算します
  // 結果を表示します
}
```

リストの1番目の事項はインターフェイスの描画となっていますが、まずはコードがうまく動くかを確認します。テストの値と変数をいくつか作成する所から始めましょう。スクリプトの主要部分が動作していれば、一時的な値の割り当てを削除してインターフェイスコードを書きます。こうすることで、確実に動作が保証されているコード上でインターフェイスが記述することになります。後でも解説しますが、長い目で見ればこの方法を取ることで時間が節約できます。ここでは重要なことは何も行いませんが、同じページ上に全て存在しているか確認するために、機能コードは以下のようになります。

```
...
main()
{
  // インターフェイスを描画します
  // インターフェイスから値を取得します
  val1 = 10;
  val2 = 5;
  // 二つの値を加算します
  val3 = val1 + val2;
  // 結果を表示します
  info( "sum =  ", val3);
}
```

注意
ここには一時変数 val1 と val2 があります。この二つの変数はインターフェイスが挿入される時点で除去されます。

このコードを実行すると、info()パネルには15と表示されているでしょう。10+5の合計値を表示するスクリプトが必要だったのですが、これで完了です！スクリプトは単純ですが、ユーザーに二つの値を要求し、合計値を表示するインターフェイスを作成する場合には、さらに便利にすることが出来ます。

LScriptインターフェイスにおいて、最も重要でなおかつ根本となるものがリクエスタです。パネルが一つもなければ、コントロールを挿入する場所がありません。リクエスタを作成するため、インターフェイスコードは`reqbegin()`関数を使用してLScriptからリクエスタモードへと切り替える必要があります。この関数は二つの処理を行います。一つは指定されたタイトルでリクエスタを作成すること、もう一つは、LScriptに対しどのパネルが有効になっているのかを伝えることです。この呼び出しの後で追加されたコントロールは、アクティブなパネルに割り当てられることになります。

インターフェイスパネルの上部に表示されるタイトルは、`reqbegin()`に渡される文字列により設定されます。ご想像どおり、`reqbegin()`関数は`reqend()`という関数と一組になっています。この関数はリクエスタモードから抜け出てインターフェイスを閉じます。カレントのスクリプトでこの二つのコマンドを動かしてみましょう。

```
...
// インターフェイスを描画します
reqbegin( "Sum of two numbers:" );
// インターフェイスから値を取得します
reqend();
  val1 = 10;
  val2 = 5;
  ...
```

ファイルを保存し、このコードをモデラーでテストしてみます。インターフェイスが現れなくても心配しないで下さい。処理中における大変重要なステップを残したままなのです。LScriptに伝えることはリクエスタを作成し、それを閉じるということだけでした。LScriptにインターフェイス作成の作業が終了したことを通知し、インターフェイスを表示させる必要があります。これには`reqpost()`関数を使用します。

```
...
reqbegin( "Sum of two numbers:" );
  reqpost();
  // インターフェイスから値を取得します
reqend();
...
```

これでリクエスタが表示されるようになり、LScriptは二つのボタンのどちらかが押されるまで、どんなコードも実行しません。スクリプトを実行すると、以下のようなインターフェイスが表示されていることでしょう。

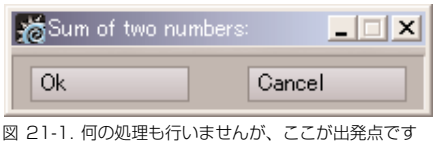

図 21-1. 何の処理も行いませんが、ここが出発点です

注意
"OK"と"Cancel"ボタンは自動的に作成されます。

ではこのリクエスタにいくつかコントロールを挿入してみましょう。ユーザーから数値をいくつか取得したいので、`ctlinteger()`コントロールを使用します。この関数は呼び出されると二つの値を受け取ります。一つはコントロールのラベルを表す文字列、そしてもう一つはデフォルト値を表す整数値です。新規2行を既存のスクリプトへと挿入します。

```
...
reqbegin( "Sum of two numbers:" );
  c0 = ctlinteger( "Value2:  ", 5);
  c1 = ctlinteger( "Value1:  ", 10);
  reqpost();
  // インターフェイスから値を取得します
  ...
```

有効で実用的なデフォルト値でコントロールを作成することが大切です。この値は、ユーザーに対しリクエスタ内に入力されるべき値の手がかりを示すものです。ユーザーがツールの適切な使用法がわからない場合、"OK"ボタンを押してみてデフォルト値ではどのような動作になるのかを見るかもしれません。デフォルト値が無効であったり間違った選択をしている場合、スクリプトの良い試運転とはいきません。エラーさえ引き起こすことになりかねません。

上記で追加した二行では、リクエスタ上のコントロールを配置する方法を紹介しています。`ctlinteger()`関数は二つの引数値と共に呼び出されます。コントロール作成に成功すれば、関数は`Handle`という値を返します。この`Handle`は基本的にLScriptが各コントロールを一意に認識するために使用する内部的な値です。この`Handle`は変数`c0`に保存されます。

スクリプトを実行してみると、二つのコントロールが画面上に正確に配置されているのがわかりますね。

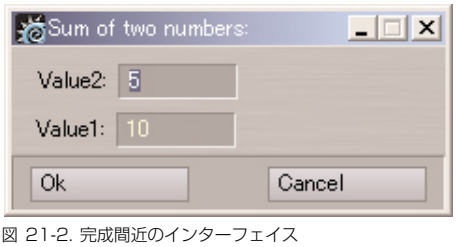

図 21-2. 完成間近のインターフェイス

`Value1`の上に、`Value2`とラベルがつけられたコントロールが挿入されているのがわかりますね。これにより二つのことが実証されました。一つは、最初に作成されたコントロールがデフォルトとしてリクエスタ上に（上から下に向かって）配置されるという点です。二つ目はリクエスタ内部におけるコントロールの位置やサイズです。コントロールの移動やサイズ変更の方法を紹介するために、ここで`ctlposition()`関数を使用してみます。

```
...
c0 = ctlinteger( "Value2:  ", 5);
c1 = ctlinteger( "Value1:  ", 10);
ctlposition(c0, 35, 30);
ctlposition(c1, 35, 5);
reqpost();
// インターフェイスから値を取得します
...
```

`ctlposition()`関数は三つの引数（三つのオプション引数）を要求します。1番目は影響を及ぼしたいコントロールの`Handle`です。この場合、変数`c0`、`c1`に保存されている`Handle`を使用します。2番目と3番目の引数は列数と行数です。画面上のコントロールを修正しようと思うと、修正結果を確認するためにスクリプトを実行させなければならず、大変な時間の浪費となる場合もあります。このためにLSIDEインターフェイスエディターを使用すれば、時間を著しく節約できます。

スクリプトを実行し結果を見てみましょう。

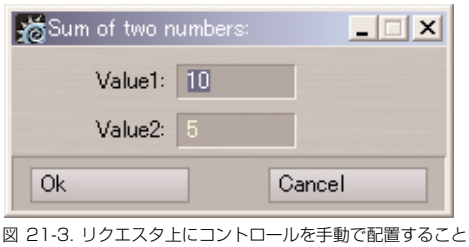

図 21-3. リクエスタ上にコントロールを手動で配置することにより、全く違う外観を作成できます

コントロールの位置が正確に修正されているのがわかりますね。`ctlposition()`関数を使えば、スクリプト作成時においてインターフェイスの外観達成に必要となる柔軟性が与えられます。しかし多くの簡単なスクリプトでは、良い計画と適切なコードを持ちさえすれば、リクエスタ上にコントロールを自動的に配置させ、このステップを回避することが可能なのです。

あとはコントロールから値を取得するだけです。`getvalue()`関数から返される値を、宣言した変数へと保存し、値を取得します。この関数は値を取り出したいコントロールの`Handle`を要求します。今回の場合は

```
...
  // インターフェイスから値を取得します
  val1 = getvalue(c0);
  val2 = getvalue(c1);
reqend();
// 結果を表示します
...
```

注意
以前使用していた一時変数 val1 と val2 を忘れずに削除してください。

これでコントロールの値は変数`val1`と`val2`に保存されました。これらの宣言部分が終了すれば、スクリプトは実行時に想定どおりの処理を行うようになります。テストしてみてください。

うまく動いているように見えますが、もう一度テストしてみてください。今度は二つの値を入力した後、"**Cancel**"ボタンを押してみてください。依然として結果が表示されていますね。ユーザーインターフェイスの慣例に反しています。"**Cancel**"ボタンが押されたときには、アクションは無視されるべきなのです。この問題を修正するため、スクリプトを変更しましょう。

この問題が引き起こしている間違いの一つは、現在`reqpost()`関数の戻り値を無視していることにあります。この値は、ユーザーがリクエスタパネルを閉じるのに"OK"ボタンを押したのか、"Cancel"ボタンを押したのかを示す値です。現在の`reqpost()`関数を以下のコードで置き換えることにより、この値を取得することが可能です。

```
...
return if !reqpost();
...
```

このコードでは`reqpost()`関数がブール値'`false`'（つまり"**Cancel**"ボタンが押下）の場合には、スクリプトが呼び出された時点へと関数を返すという処理を行います。この場合、モデラーへと返されます。基本的にはこれでスクリプトは終了です。ではスクリプトを動かしてみましょう。

最終コード：

```
@script modeler
@name ModInterfaceTest
@version 2.3

main
{
  // インターフェイスを描画します
  reqbegin("Sum of two numbers:");
    c0 = ctlinteger("Value2: ",5);
    c1 = ctlinteger("Value1: ",10);
    ctlposition(c0, 35, 30);
    ctlposition(c1, 35, 5);
    return if !reqpost();
    // インターフェイスから値を取得します
    val1 = getvalue(c0);
    val2 = getvalue(c1);
  reqend();
  // 二つの値を加算します
  val3 = val1 + val2;
  // 結果を表示します
  info("sum = ", val3);
}
```

Generic（総括）クラスのインターフェイス

Generic（総括）クラスとモデラークラスのスクリプトにおけるインターフェイス作成法には、何ら違いはありません。構造的に大変似通っているため、インターフェイスのコーディングは事実上、同一のコードとなります。これを実証するため、前述の例をモデラークラスのスクリプトからGenericクラスのスクリプトへと変更し、レイアウトから実行してみましょう。モデラースクリプトとすべて同じように動作し、表示されるのがわかりますね。

インターフェイスのコードの箇所でのみ比較できます。本来、Genericクラスのスクリプトで利用可能な関数やコマンドは、レイアウトのみに限定されているからです。

第 22 章：レイアウトインターフェイス

インターフェイスには明確な機能が二つあります。ひとつはユーザーが作成したり修正を加えたい設定を取得すること、もうひとつは保存されている値をユーザーに表示し、現在値や設定を通知することです。優れたインターフェイスというのはこの双方を実現しています。その点ではレイアウトスクリプトもモデラースクリプトも同類です。

しかし前述した通り、モデラーとレイアウトのスクリプトの間にある大きな違いは、その構造にあります。モデラースクリプトには事前定義された一つの関数が含まれますが、レイアウトのスクリプトが適切に動くためには、複数の関数が必要となります。これはレイアウトスクリプトに対するインターフェイスのコーディングの仕方に直接影響を及ぼします。Item Animation スクリプトのテンプレートを見てください。この例では何も処理していませんが、スクリプトのスケルトンコードを設定しています。

```
@version 2.2
@warnings
@script motion

create: obj
{
}

destroy
{
}

process: ma, frame, time
{
}

load: what,io
{
```

```
  if(what == SCENEMODE) // ASCII シーンファイルを処理
  {
  }
}

save: what,io
{
  if(what == SCENEMODE)
  {
  }
}
```

注意
関数内で何の処理も行わない場合、関数を宣言する必要はありません。しかし今回はこれらの関数がどのようにサポートされるのかを見るために、あえて関数を宣言するようにします。関数宣言の方法を理解したら、コード部分を含んでいる関数だけを使用するように、コードを切り取ることが出来ます。

見てわかるとおり、関数`create()`、`destroy()`、`process()`、`load()`、それに`save()`は全てスクリプトの実行時に特化された機能があります。重要度に違いはあるものの、全ての関数をあわせてスクリプトの機能を完全なものにしています。LScriptはこれらの関数をそれぞれいつ実行するのかを把握し、スクリプト実行箇所で呼び出します。ではどこにインターフェイスのコードを挿入すればよいのでしょうか？

インターフェイス概論の章で解説した通り、`process()`関数にインターフェイスコードを追加するだけでは出来ません。これでは絶えずリクエスタが現れてしまい、ユーザーはすぐにいらいらしてしまうでしょう。しかし`options()`という関数を宣言することで、LScriptに対しインターフェイスコード内で探しに行く場所を正確に指示することが出来ます。前述の例の終端に、次のコードを単に追加してください。

```
options
{
  reqbegin("An Animation Test");
    return if !reqpost();
  reqend();
}
```

これらのコマンドに馴染みはありますか？解説したように、モデラーとレイアウトインターフェイスで使用されるコマンドと関数は同じものです。上記で使用されるコマンドは、最初に動作するインターフェイス用に最低限必要なものです。事実、この特定の状況においては`reqend()`関数が存在する必要はありません。LScriptは`options()`関数を通り過ぎる時に自動的にインターフェイスを終了させるからです。しかしパネルを追加する場合、またはインターフェイスコードを後で追加する場合に備えて、手動でインターフェイスを閉じるよう習慣付けておく方が良いでしょう。

このItem Animationスクリプトを実行してみてプラグインリスト内にあるこのインスタンスをダブルクリックしてみましょう。以下の画像のようなパネルが現れましたね。

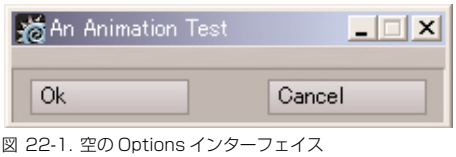

図 22-1. 空の Options インターフェイス

`options()`関数内部において、前章で行ったのと同じ方法でコントロールを追加し、望みどおりに機能するインターフェイスを作成することが出来ます。モデラーインターフェイスで使用したときと全て同じような外観と機能を持ちます。しかしレイアウトインターフェイスがモデラーのインターフェイスと全く同じような動作し始める前に、取り組むべき箇所がもう一つあるのです。

変数スコープ

関数内に変数を宣言した時に、その変数に対するデータは同じ関数内部のコードでのみ利用可能です。例えばAとBという二つの関数がある場合、関数Aの仲で変数TEMPを宣言すると、関数BではTEMPの存在を知ることなく、カレント値を正しく与えようと気にかけることは決してないでしょう。これがレイアウトインターフェイスで現在直面している問題なのです。

インターフェイスからデータを集めるコードは`options()`関数内に置かれます。ところが、実際のスクリプトのタスクを実行するコードは`process()`関数内に置かれます。そのため、インターフェイスから集められた値は、スクリプトの処理コードからアクセスできません。変数は`options()`関数でローカルに宣言されており、`create()`や`process()`関数ではその値を読み取れないのです。つまり可変関数の間で共有される変数は、グローバル変数として宣言されなければならないのです。

変数名称を指示子セクションの後、コードに記されている最初の関数の前に宣言するだけでよいのです。例として以前のコードを使用してみましょう。

```
@version 2.2
@warnings
@script motion
// ここに大域変数を挿入
option1 = "Interface Test.";
option2 = .01;
option3 = 1;

create: obj
{
}
```

全ての変数を大域変数として宣言したくはありませんが、変数`option1`、`option2`、`option3`は、`options()`関数だけではなく全関数内からでも利用可能になりました。もっと大切なことは、`process()`関数内のコードはインターフェイスから集められたデータを使用することが出来るようになったということです。

解説文

最後にインターフェイスで選択した重要なオプションをまとめて、スクリプトの解説リスト上に表示し、ユーザーに通知しなければなりません。`setdesc()`コマンドを使用して処理を行います。表示したい情報は、文字列引数として保存されている`info()`関数と同様の働きをします。

注意
この関数はモデラースクリプトでは利用できません。

例えば、`ptions()`関数の終端にこの行を配置してみます。

```
  ...
  reqend();
  setdesc( "Script: "  + option1);
}
...
```

リクエスタを閉じると、この関数は`option1`に何を選択したのかをユーザーに対し正確に通知します。お気づきのとおり、選択された値全てを表示することは出来ません。何を選択したかが問題となるような、重要な値だけを表示するようにします。

例：

アイテムのキーフレームの値から厳密に最大座標値を設定する、簡単な拘束アニメーションスクリプトを作成してみましょう。これはItem Animationスクリプトで、アイテムが移動するに従って全ての動きを封じるようにします。いつもどおりヘッダー設定から始めましょう。

```
@version 2.3
@warnings
@script motion
```

最大X座標値を保存するための大域変数を宣言します。

```
maxXPos = 1.0;
```

まずは`process()`関数から始めましょう。最初に処理コードがすべて正常に動くようにします。それからインターフェイスを開発し、ユーザー入力として使用する値を全て扱うインターフェイスを開発します。Item Animationスクリプトの内部作成を終わらせるよりも、インターフェイスコードの中へ直接飛びましょう。しかしスクリプトは同様の動作を実行しますので、以下の処理機能を用意します。

```
process: ma, frame, time
{
  // ma Motion Object Agent を使用して現在位置を取得します
  currPosition = ma.get(POSITION,time);
  // アイテムの位置が maxXPos より大きければ
  // maxXPos に変更します
  if(currPosition.x > maxXPos)
    currPosition.x = maxXPos;
  // 位置情報を設定します
  ma.set(POSITION, currPosition);
}
```

注意
この関数内で使用している馴染みの薄いコードについては、コメントで解説します。コメントがない場合は、リファレンスマニュアル 25 章の Motion Object Agent を参照してください。リファレンスマニュアルのこのセクションでは、この関数内で使用されているコマンド全てが解説されています。

任意のアイテムモーションプラグインリストでこのスクリプトを実行させ、アイテムを動かしてみてください。アイテムが maxXPos の値以上になったら動かなくなるのがわかりますね。今回の場合、最大X位置（maxXPos）は大域変数として値1.0で宣言されています。変数 maxXPos の値を手動で変更することで、最大X位置を変更します。

テストを十分に行い、コードの動作を確認したら、インターフェイスを追加していきましょう。まずは options()関数と初期リクエスタを作成するところから始めます。

```
options
{
  // リクエスタを開きます
  reqbegin("Clamp");
    return if !reqpost();
  // インターフェイスを閉じます
  reqend();
}
```

三次元空間内においてX座標値を表現したいので、浮動小数点数を使用します。数値コントロールを追加します。

```
...
reqbegin("Clamp");
  // 数値コントロールを作成します
  c0 = ctlnumber("Max. X Position",xPos);
...
```

数値コントロールのデフォルト値が変数 xPos の値に設定されているのがわかりますね。xPos は値1.0を持つように宣言されていますので、コントロールのデフォルト値は1.0となります。

パネルを閉じた後にコントロールの値を収集し、グローバル変数 xPos に値を保存しましょう。

```
...
return if !reqpost();
// インターフェイスからの値を取得し大域変数 xPos に保存します
xPos = getvalue(c0);
// インターフェイスを閉じます
...
```

スクリプトを実行すると、下のパネルのようなインターフェイスが現れます。

図 22-2. Clamp インターフェイス

プラグインリストに解説文を追加しましょう。

```
  ...
  reqend();
  // プラグインの解説文に xPos の値を送信します
  setdesc("xPos:" + xPos);
}
```

これでインターフェイスは完成です。

機能追加

この単純なインターフェイスは適切に動作するようになりましたので、他の二つの座標値もカバーするよう簡単に拡張できます。コピーアンドペーストし、変数の名称を変更し、大域変数をいくつか追加するだけで、下記に示すようなインターフェイスを持つスクリプトが、いとも簡単に出来あがります。

図 22-3. 拡張された Clamp インターフェイス

他に二つの数値フィールドとそれをサポートするコードを追加しましたので、完全な座標システムをサポートすることになり、スクリプトはより使いやすくなりました。最大値と最小値を拘束するスクリプトを作成すればさらに拡張することが出来ます。インターフェイスは以下のようになります。

図 22-4. さらに拡張されたインターフェイス

このスクリプトとインターフェイスを作成するコードは、先ほど動かしたサンプルのスクリプトを派生させただけに過ぎません。概念も関数もコマンドも、新しいものは何も導入されていません。

このように、インターフェイス作成法は段階を踏んで組み立てていくことでスクリプトの構築法を強化してくれます。最初に実用的なコードを、それからインターフェイスを、その後でもう一歩先に進めたコードを書くことにより、非常に柔軟性があってしっかりしたコードを作成しました。この手法に従えば、より大きなスクリプトやインターフェイスを作成する場合においても、さほど頭を悩ませることはないはずです。

第 23 章：インターフェイス設計における規則

初期のスクリプトでは、概念と設計という面で非常に単純なインターフェイスを作るべきです。インターフェイスで複雑な動作を実装させようとする前に、まずコントロールの基本とコントロールがどのように機能するのかを学び理解して下さい。インターフェイスのコーディングは信じられないほど難しいというものではありません。スクリプトにおける他の全ての側面と同様、全てがうまく動かせるようになるには、時間とそれに適用するための知識が必要となります。インターフェイスを書き続け、サンプルを研究すればするほど、独自のインターフェイスがよりよい外観を持ち、動作を行うようになるでしょう。

インターフェイスのコードを学習する一方で、従わなければならない規則がいくつかあります。

規則 #1: まずは機能性

インターフェイスはユーザーが直接見て対話するものですから、設計するのはとても楽しい作業となります。しかし開発の技術はインターフェイスデザインだけで判断されるなどという馬鹿げた考えは信じないで下さい。初心者にはスクリプトの実際のコードを書くよりもインターフェイスのプログラムにより時間をかけるという過ちを犯す人もいます。インターフェイスは素晴らしいものですし、つまらないインターフェイスよりかはプロフェッショナル的に見せてくれはしますが、コードを便利にするため以外の何者でもないのです。

開発を始めたばかりですから、スクリプトは最初に想定していたようには動作しないかもしれないと言っても差し支えないでしょう。過ちは経験不足によるものですから安心してください。過ちを犯すことで時間を浪費してしまうでしょうが、それも全て学習過程の一部なのです。

間違いは避けられないものなのですから、インターフェイスを追加する前にまずコードが適切に機能しているのかを確かめて下さい。不必要なインターフェイスコードを書く時間の無駄は省きたいでしょう？インターフェイス作成に時間をかける必要はないと言っているわけではありません。しかしスクリプトは設計どおりに動作しないのですから、全ての強力なインターフェイスの一部もしくは全部を処分するより負担が重くなることなどないのです！

可能であれば、インターフェイスを書き始める前に、ダミーオブジェクトと固定値を持つ変数で動作させるようなスクリプトを書いてみてください。コントロールの代わりにこれらの値を使用することによって、主なタスク、スクリプトの記述とテストなどに焦点を当てることが出来るようになります。この作業が終了すれば、実際の処理を全て行う根本となるコードが固定されることになり、コードは想定どおり機能することが保証されます。スクリプトの機能を担うコードを書き終えバグからも開放された時点で、インターフェイスの作成を始めてください。この時点で、機能を全て備え、全てが実装された後でもそれほど変更を加えないテストコードを基に、インターフェイスコードを書きます。このように作業することで、より効率的に時間を使用できます。

規則 #2: シンプルさを保つ

インターフェイスは視覚的な手段ですから、大なり小なり視覚的なものです。スクリプトやプラグインにはどれも素晴らしいデザインを持つインターフェイスを割り当てることが出来ます。インターフェイスは画面上にあまりの多くのボタンがあっても、もしくは配置がばらばらで混乱させるようなデザインであっても、目障りなものになってしまいます。実際、デザインが悪いとツールは難しくなり、場合によっては使用するのが苦痛でさえあります。

スクリーン上の情報量を最小化することで、インターフェイスは使いやすくなるのです。仕切り線を追加することでコントロールを系統立てて配置したり、タブを追加することでリクエスタのサイズ全体を小さくすることが出来ます。このようなトリックは必要であればあるほど複雑になってきます。

インターフェイスコードを書き始める前に、ユーザーから取得しなければならない値のリストを作成しておくのが良いでしょう。このリストから設定や変数を固定しておいて、コードの機能をテストするようにしてください（規則 #1 参照）。その情報からどのコントロールが必要となるのかを確定します。その後でリクエスタ上でどう配置するのか、タブを使ってコントロールをグループ化するのかなどを確定していきます。インターフェイスの輪郭はこのように考えていって下さい。

規則 #3: 美しさよりも速度

"**Less is More**"（過ぎたるは及ばざるが如し/出来るだけシンプルに）という黄金律を極限まで極めます。スクリプトにインターフェイスが必要でなければ、作る必要はありません！出来る限り作らないで下さい。スクリプトをより動的に使いやすくするためにインターフェイスを作成するのです。やるべきことは、速くそして効率の良いツールを作ることなのです。もう一つ、にわか仕立ての処理段階を追加することほど、ユーザーを悩ませるものはありません。常識を用いてスマートに設計するようにして下さい。一日に一回、もしくは一月に一回走らせるかどうかぐらいの"一度限り"のスクリプトであれば、一日に何百回も使用するスクリプトのように効率良くする必要はないのです。

規則 #4: 一度に一つのアイテム

パネルのサイズであろうとコントロールの配置であろうと、一度に一つのアイテムだけのコードを書いてください。そうすれば頻繁に動作をテスト出来るようになり、間違いも早く見つけることが出来ます。インターフェイスのデバッグに費やす時間を大幅に短縮出来、より簡単になります。一度に多くのものを実装しようとすると、簡単な間違いでバグ探しに莫大な時間を費やしてしまいかねません。

規則 #5: 業界標準

どのコントロールを使用するのか、画面上でそのコントロールをどう整えていくのかを決めるのは芸術形式です。大半のソフト会社には、ソフトウェアインターフェイスの設計と実装を担当する専門家がいるものです。彼らはクリエイティブな処理を手助けするために何年もの間研究を重ね、顧客からのフィードバックを元にインターフェイスを設計しています。こういった人々から多くのことが学べるはずです。事実、もう学んでいるはずです。

コンピュータ上で数え切れない時間を過ごすうちに、ソフトウェアがどのように動作するのか想定できるようになります。WindowsであれMacであれ、一つのアプリケーションから別のアプリケーションを通して、ある特定の関数が同じ動作を行っています。これはあなた自身の創作物にも有効であるはずです。LightWaveのプラグインやその他のスクリプトが、インターフェイスをどのように設定しているのかに注意を払うのはさらに重要です。ユーザーはそれらコンポーネントの対話に慣れており、想定どおりのインターフェイスであれば、スクリプトをより受け入れやすいからです。

規則 #6: 研究例

お好みのスクリプトのインターフェイスを見つけたり、どのように動作しているのか不思議に思ったら、コードを調べてみてください。理解したい領域、自身のスクリプトへと組み込みたいために研究が必要となる部分のコードを解析してみてください。これはインターフェイスに限らずコーディング一般に関して言えることです。時間をかけて他の素晴らしいスクリプト作品を学んで下さい。

注意

謝辞を授与すべき箇所には忘れずに記述するようにして下さい。あなた独自のものでないアイデアやコードの一部を使用している場合には、コメントの形でオリジナルの著者の謝辞を与えるようにして下さい（インターフェイス上ではなくコード内部で述べること）。

規則 #7: ユーザーの批判に耳を傾ける

アニメーターは、開発者が決して想定していなかったような方法でコードやインターフェイスを使おうとします。コードの目的が理解できないと考えるのではなく、その正反対に捉えるようにして下さい。アニメーター達はスクリプトの限界を押し上げようとしているのです。対象とする聴衆と対話を始めるのと同時に、彼らのコメントを、次期バージョンのスクリプトのための新鮮なヒントとして利用してください。これらがスクリプトのアイデアやベータテストコードとして跳ね返り、これらの人々は大変貴重なリソースとなるのです。

規則 #8: あなた自身が一番の批評家

自分のスクリプトを使用し、実際にテストしていってください。自身のコードのデバッグに役立つと共に、自らスクリプトを実行することで即時にフィードバックが提供されます。インターフェイスの動作方式が自分で気に入らないようであれば、LightWave コミュニティでもその動作方式が気に入られない可能性が高いでしょう。

注意
この意見の逆が必ずしも真であるとは限りません。インターフェイスの動作方式が自分では気に入っていても、コミュニティも気に入るとは保証出来ないのです。

第 24 章：LSIDE

LScript Integrated Development Environment（LSIDE）は、LScriptプログラマがスクリプトを作成するためのツール群の一つです。

LightWaveのツールキットとLScript内部で開発された技術に対し、LSIDEは完全にプラットフォーム独立です。つまり、LSIDE内にあるツールは全てLightWaveをサポートしている全てのプラットフォームで上で同じ動作を行うということです。

このセクションで見ていく通り、LSIDEのツールもまた完全に統合されています。これらはワークフロー向上のため、スクリプトの開発処理におけるキーとなる時間に相互に通信します。LSIDEにおける統合されたツールを使用するにつれ、スクリプト作成はより速く簡単になります。

LScript Editor（エディター）

まず最初に、LSIDEの基幹をなす**LScript Editor（エディター）**を見ていきましょう。どのテキストエディタでもスクリプトを作成することは出来ますが、LScript Editorの威力を目の前にしたら、このエディタを使用して全スクリプトを作りたくなることでしょう。以下のリストでは素晴らしい機能を紹介しています。

・色構文ハイライト化
・マウスサポート
・LScript テンプレート
・複数ドキュメント
・LScript構文チェック
・検索置換
・ブロックによるテキスト移動
・アンドゥの複数レベル
・ドキュメントをまたがっての検索置換

LScript Editorの主となるエリアは5つあります。メニュー、テキスト、コマンド、メッセージそれに状態エリアです。それでは、それぞれのエリアがどういう動作をするのか順番に見ていきましょう。

メニューエリア

想像どおり、メニューエリアを構成しているのは主に4つのポップアップです。

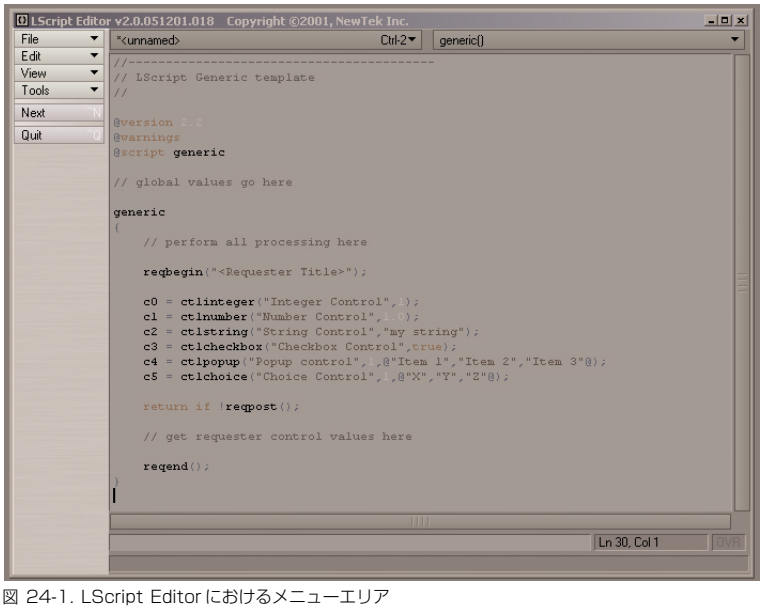

図 24-1. LScript Editor におけるメニューエリア

File メニューではテキストエディタからファイルを扱えます。新規ファイルの作成やファイルの保存、別名で保存、ファイルを閉じるなどの動作です。

このメニューにある唯一新規タイプのコマンドは **Toggle Write** コマンドです。このコマンドでは不注意により上書きしないよう、スクリプトの書き込みを許可するトグルを使用できます。いくつものスクリプトを開いている時に、これは非常に便利な機能となります。不測のエラーを作る機会を無くします。

Edit メニューには **Undo** や **Delete**、**Cut**、**Copy**、**Paste** それに **Search and Replace** があります。**Cut**、**Copy** それに **Delete** の機能は、使用する前にハイライト状態になっているテキストが必要です。処理を実行したいアイテムを選択し、**Edit** メニューからオプションを選択します。例えばコードのある一行を削除したい場合には行を選択してから **Edit > Delete** を選択します。これらのコマンドはWindowsのクリップボードとともに動作します。

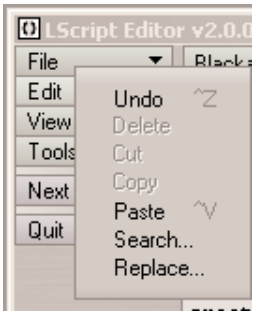

図 24-2. Edit メニュー

View メニューでは、エディタ内部のビューオプションを変更することが出来ます。まず最初のオプションは **Font** で表示エリアに対するフォントのサイズを変更することが出来ます。サイズを増減させます。

図 24-3. Font は文字サイズを変更します

Highlight セクションではLScirpt特有の構文を判読しやすいように色付けするかどうかを設定できます。**syntax** をハイライトにするよう選択すると、様々なタイプのコードがそれぞれ目立つように異なる色で表示されます。ハイライトの種類はコードを通して探しやすくし、リファレンスとしてカラーを使用することが出来ます。

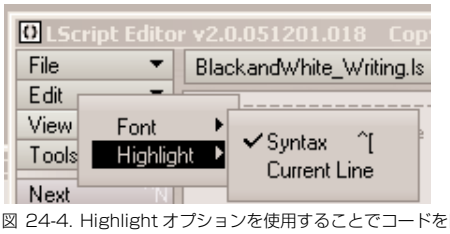

図 24-4. Highlight オプションを使用することでコードを目立たせます

Tool メニューには、エディターにおいて最も重要なツールの一つである **Templates** オプションがあります。

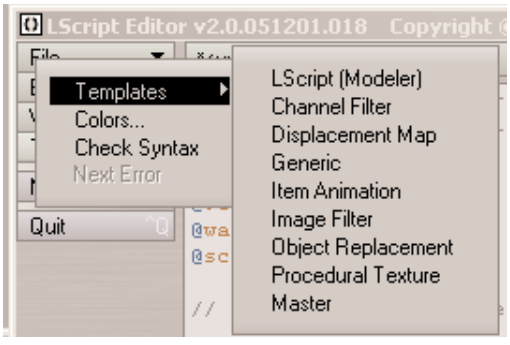

図 24-5. スクリプト開始時にはテンプレートを選択

お望みのタイプのスクリプトに対し事前に作成されたテンプレートを取得することが出来ます。スクリプト構造に関する心配をする必要がないわけですから、テンプレートを使うのはスクリプト処理をあっという間に始められる素晴らしい方法です。

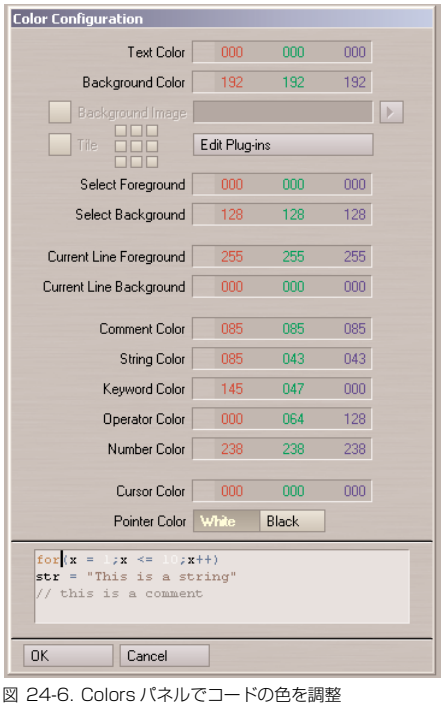

図 24-6. Colors パネルでコードの色を調整

Colorsパネル上では、テキストの色調整を行うためインターフェイス上で見えている通りにカラーを修正することが可能です。参照するカラーでコメントが表示されるように、また命令子をさらに目立たせるためにインターフェイスを修正することが出来ます。またエディターに対して**Background Image**を追加することも出来、背景内部の方向を設定することも可能です。

でも待って下さい、これで全てじゃないんです。**Tools**メニューからスクリプトの構文チェックも行えます。スクリプトの構文チェックを行えば、スクリプトの処理時間を向上できます。LightWaveにおけるスクリプト実行の必要がなくなり、20もの行の終端にセミコロンを付け忘れたのではないかと探すこともなくなるので、生産性が向上します。

LScript Editorはメモリ内に一度に複数のスクリプトを置いておけるため、**Next**ボタンを使用して読み込まれているスクリプト全ての間を切り替えることが可能です。

図 24-7. Next ボタンで読み込まれているスクリプトを移動

もちろん、アプリケーションを終了するときには**Quit**を使用します。

図 24-8. Quit ボタンで LScript Editor を終了

テキストエリア

LScript Editor のメインウィンドウはテキストエリアです。このウィンドウ内部で全スクリプトを編集します。どのワードプロセッサで編集されたテキストを持ってきても、問題なく内容を表示することが出来ます。実際、LScript エディタはどのテキストファイルでも読み込んで編集することが可能です。

図 24-9. スクリプト編集用のテキストエリア

編集エリアにおける独自の機能は、メインとなるテキストエリアの上にある二つのドロップメニューにあります。

左側のドロップメニューはスクリプトドロップメニューです。メニューエリアにある **Next** ボタン同様、このコントロールを使用してメモリに読み込まれているスクリプト間を切り替えることが出来ます。**Next** ボタンとは異なり、順に切り替えるのではなくリストから自由に任意のスクリプトを選択することが出来ます。

またスクリプトドロップメニューはスクリプトの状態を知らせるためにスクリプト名称の頭に文字を配置します。

* : **アスタリスク**はスクリプトが修正されまだ保存されていないことを表しています。

^ : **キャラット**はスクリプトが読取専用であることを示しています。

図 24-10. 上: スクリプトドロップメニュー 下: 関数リスト

右側のドロップメニューは関数リストです。スクリプトドロップメニューと同様、リスト内の関数名称にジャンプすることが出来ます。ここでの違いは、アクティブスクリプト内部において定義されている関数があればその関数へとジャンプすることができることです。Generic スクリプトのテンプレートが LSED に読み込まれている場合、ユーザー定義関数はありませんから、リストには `generic()` 関数のみが表示されます。

コマンドエリア

テキストエリアの下にあるフィールドがコマンドエリアで、特定のコマンドを入力する場所です。

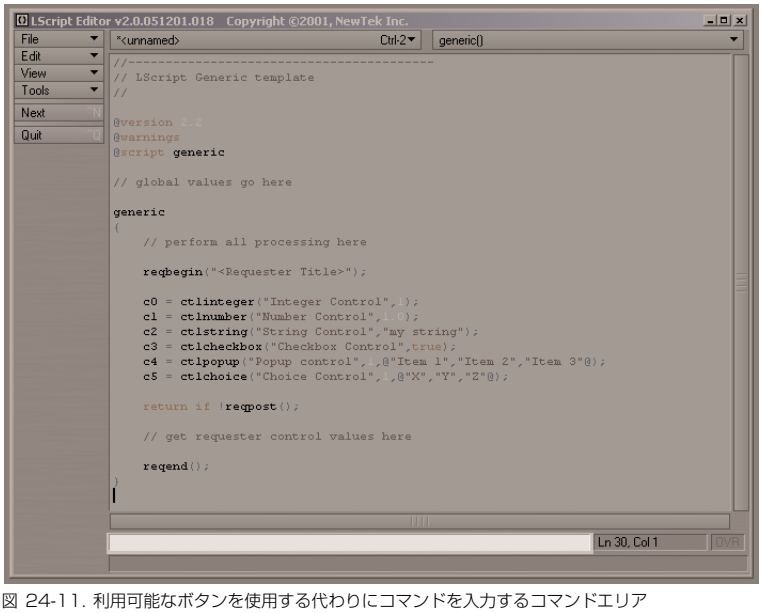

図 24-11. 利用可能なボタンを使用する代わりにコマンドを入力するコマンドエリア

コマンドエリアでは、LSED に対し何を処理させるのかを正確に指定することが出来ます。このエリアは、厄介な小さなボタンを使用するよりも直接コマンドを打ち込みたい人のためのエリアです。

以下がCommand入力ウィンドウで認識されるコマンドとなります。

```
goto <行番号>
open
s~<検索文字列>~
s~<検索文字列>~<置換文字列>~
home
end
match (brace, bracket or parentheses)
filename
new
close
```

インターフェイス上のボタンでも上記コマンドと同じ動作が出来ますので上記コマンドはは使用しないかもしれませんが、ある一つのコマンドはとても役に立つでしょう。

```
help
```

helpコマンドを入力すると以下のLSED用のキーボードショートカットが画面に溢れ出します。

CTRL+\	カレントの編集ドキュメントとコマンドウィンドウを切り替えます
CTRL+]	一致する括弧文字にジャンプします
CTRL+[	カレントスクリプトにおいてハイライトになっている構文をトグルします
SHIFT+TAB	選択されているブロックを4文字分左に移動します
TAB	選択されているブロックを4文字分右に移動します
CTRL+F7	選択されているブロックを1文字分左に移動します
CTRL+F8	選択されているブロックを1文字分右に移動します
CTRL+X	選択されているブロックをクリップボードへ切り取ります
CTRL+C	選択されているブロックをクリップボードへコピーします
CTRL+V	クリップボード内のコンテンツをカレントのドキュメントへ貼り付けます
CTRL+D	選択されているブロックを削除します
CTRL+Q	終了します
CTRL+O	新規ドキュメントを開きます
CTRL+S	カレントのドキュメントに修正が加えられていれば保存します
CTRL+R	バッファとそれに割り当てられているファイルの書き込み状態をトグルします
CTRL+N	次のドキュメントへ切り替えます
CTRL+G	カレントドキュメント内にある指定した行番号にジャンプします
CTRL+1-9	最初の9個のドキュメントの一つを選択します
CTRL+F9	フォントサイズを大きくします
CTRL+F10	フォントサイズを小さくします

メッセージエリア

LSIDEエディタの一番下の部分がメッセージエリアです。

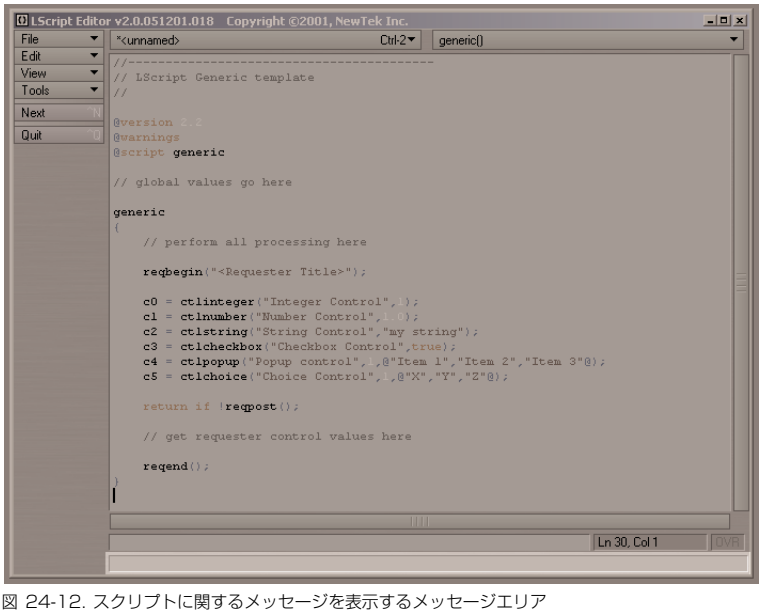

図 24-12. スクリプトに関するメッセージを表示するメッセージエリア

LSEDがドキュメントに関して何かを伝える必要がある場合には、このメッセージエリアに表示されます。

ポジションエリアとモードエリア

ウィンドウ右下はポジションエリアとモードエリアです。ポジションエリアにはスクリプト内における現在のカーソル位置が表示されています。位置は行番号と列番号で表されています。

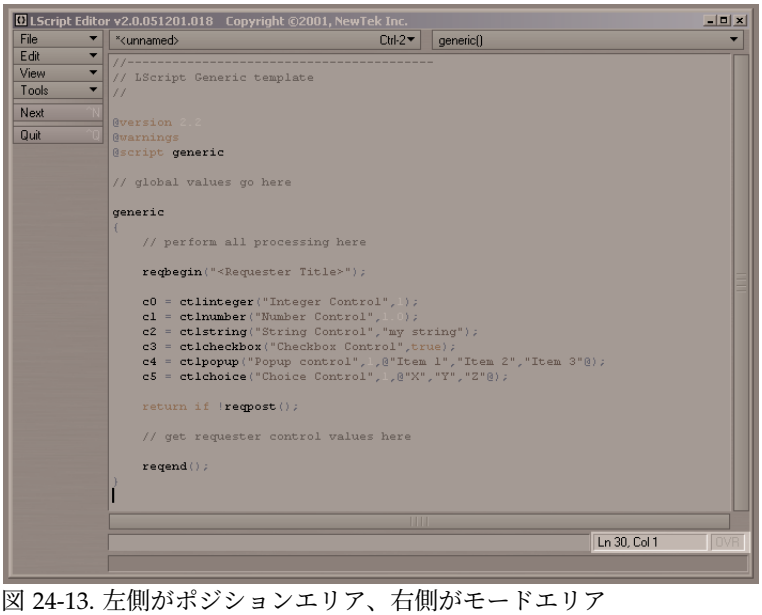

図 24-13. 左側がポジションエリア、右側がモードエリア

モードエリアでは **Insert** モードかまたは **Overwrite** モードかを表示します。**Insert** モードでは編集時においてカーソルの位置に文字を挿入します。**Overwrite** モードでは編集時に文字を上書きします。

LScript Interface Designer（インターフェイスデザイナー）

LScript Interface Designer（LSID）は"見たままのものをお手元に"（WYSIWYG）というコンセプトの基に作られた、スクリプト用のインターフェイスをデザインできるツールです。LSID を使用すると、リクエスタコントロールを編集することが出来ます。インタラクティブに好きな場所へコントロールを配置し、複雑なインターフェイスを作成する時間を劇的に減らすことが可能です。

図 24-22. LScript Interface Designer(インターフェイスデザイナー)

他の LSIDE ツールと同様、**LScript Interface Designer**は幾つかのエリアに分けられます。メニューエリア、コンポーネントツリーエリア、それにシステムメッセージエリアです。それぞれのエリアを見ていきましょう。

メニューエリア

Interface Designerにあるメニューはインターフェイスを処理する関数に対応しています。まず始めにインターフェイス用の空白テンプレートを作成するか、以前に保存してあるインターフェイスを読み込まなくてはなりません。

図 24-23. LScript Interface Designerのメニューエリア

テンプレートを読み込んだり保存するためのコマンドは**File**メニューの下にあり、自己説明形式のメニューとなっています。ここでの一つ大きな新機能として、サブメニュー**Export**があります。**Save**や**Save As**メニューオプションがInterface Designerだけが読み込み可能なインターフェイスファイルを保存するのに対し、**Export**は実際のスクリプトコードを保存します。

図 24-24. 左: Fileメニューにおけるオプション. 右: Exportメニューと利用可能なオプション

Interface DesignerとLScript Editorが双方とも開かれている場合、**Export**関数を使用してインターフェイスをエディターへと直接エクスポートすることが可能です。**Export**関数に書き込むコードタイプ（**モデラーLScript**、**レイアウトLScript**または**Panel**）を指定します。**Panel**を選択した場合、LSIDはCのコードをエクスポートします。

コンポーネントメニューでは様々なインターフェイスコンポーネントを選択できます。インターフェイス上に追加したいコンポーネントを選択するだけで、テンプレート上に作成されます。

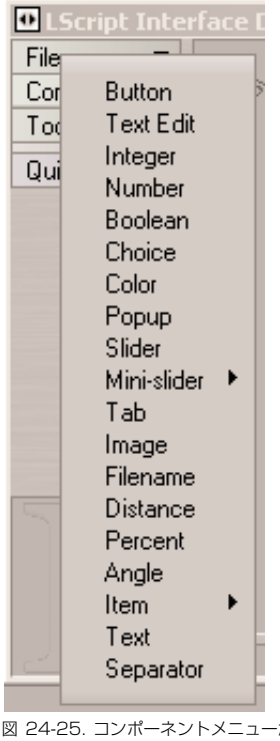

図 24-25. コンポーネントメニューから追加出来る使用可能なコンポーネント

ツールメニューではインターフェイスに対し新規レイヤーを追加できます。スクリプトの見え方には影響を及ぼしませんが、個別のレイヤーに似たようなボタンを挿入することが出来るため、複数のボタンを持つインターフェイスをより簡単に編集できるようになります。これはモデラーにおけるレイヤーと同じようなものです。各レイヤーにはオブジェクトの個別の部品がありますが、全て同一オブジェクトの一部とみなされます。

図 24-26. ツールメニューの New Layer オプション

インターフェイス上のコンポーネント群を選択して整列させることが出来ます。**Align** のサブメニューではどのように整列させるのかを選択します。

Align（整列）オプション

垂直方向にコントロールを積み上げたい場合には、**Left** または **Right** で整列させます。

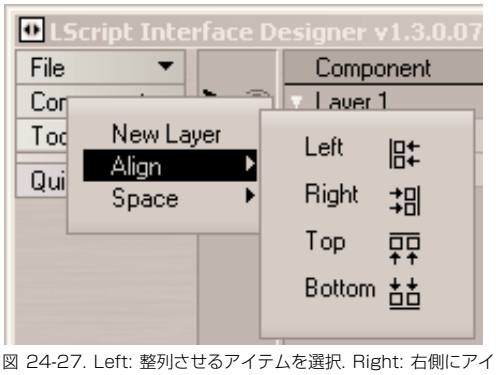

図 24-27. Left: 整列させるアイテムを選択. Right: 右側にアイテムが整列

コントロールを水平方向に修正したい場合には、**Top** または **Bottom** で整列可能です。

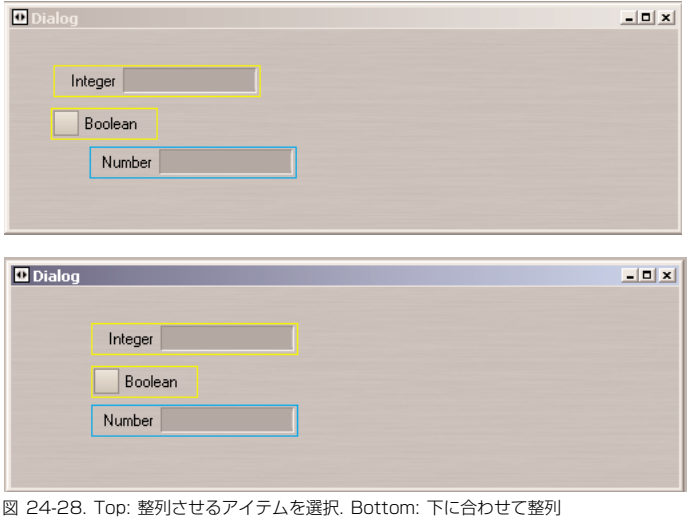

図 24-28. Top: 整列させるアイテムを選択. Bottom: 下に合わせて整列

SHIFT を押しながらクリックするとコントロールを複数選択できます。**Space** メニューオプションは、水平方向または垂直方向に複数のコントロールを均等に配置します。

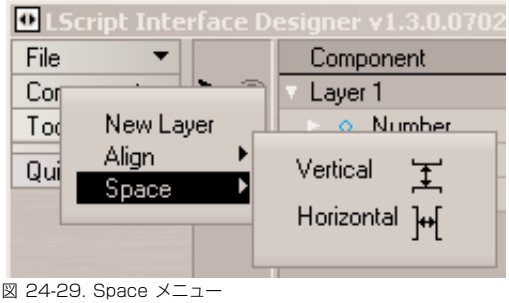

図 24-29. Space メニュー

注意
整列メニューツールと一緒に矢印キーを使用すれば、一度に位置ピクセルずつ矢印の方向へとコントロールをずらすことが出来ます。

コンポーネントツリー

コンポーネントツリーではインターフェイスの外観を組織的に見ることが出来ます。左側の列には二つのトグルがあります。その一つである **Quill** は現在どのレイヤーを修正しているのかを示し、もう一つのトグル **Eyeball** は、各レイヤーを可視状態にします。レイヤーの可視状態は **Interface Designer** においてのみ影響を受けます。レイヤーを見えない状態のままでエクスポートしても、コントロールはインターフェイスの一部としてエクスポートされます。ですから必要のないコントロールはエクスポートする前に削除しておいてください。

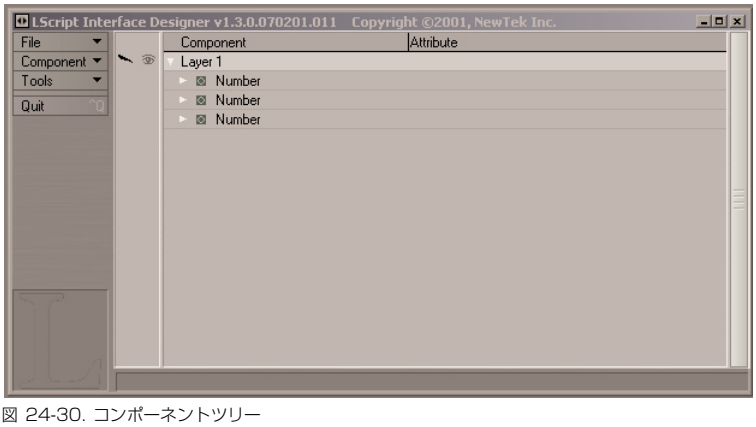

図 24-30. コンポーネントツリー

コントロールを拡張することにより、任意のコントロールの編集可能な値にすべてアクセスできます。矢印を下向きにクリックすることでコントロールを拡張します。値を編集するには修正を加えたい値をダブルクリックするだけでダイアログボックスが表示されます。

図 24-31. 矢印が下向きになるようにクリックするとコントロールが拡張され値が見えます

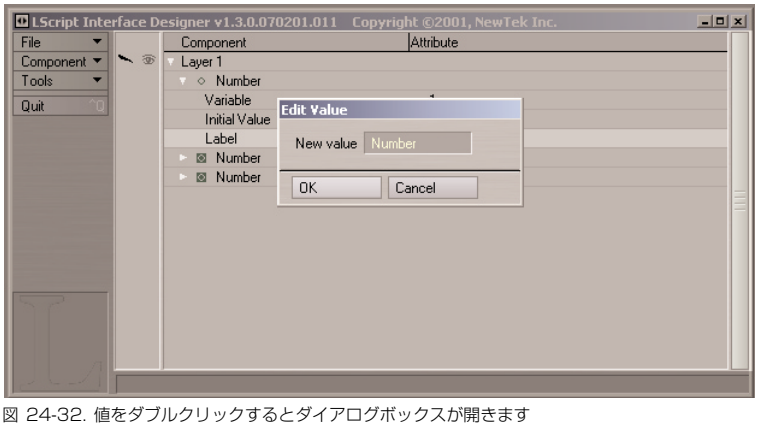

図 24-32. 値をダブルクリックするとダイアログボックスが開きます

コンポーネントツリーのもう一つのすばらしい機能として、コントロールを他のコントロールに対し親子関係を設定することが出来ます。この機能はLightWaveの**Scene Editor（シーン編集）**でアイテムに親子関係を設定するのと同じ方法で動作します。コンポーネントツリーウィンドウでは子コンポーネントを親コンポーネントの下にドラッグします。すると親アイテムの下に黄色い線が表示されます。 選択したアイテムに親子関係がつけられたことを示すため、コンポーネントは少しずれて表示されます。

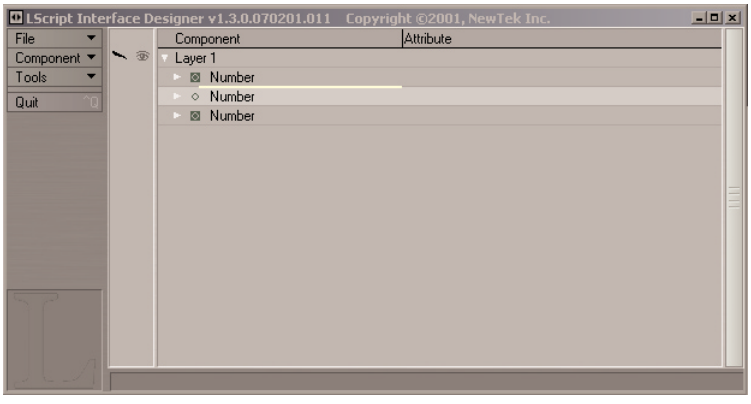

図 24-33. 一番目のコンポーネント Number の下にある短い線は子アイテムを示しています

親アイテムを選択したとき、インターフェース上では子アイテムにそれぞれ直線が引かれます。

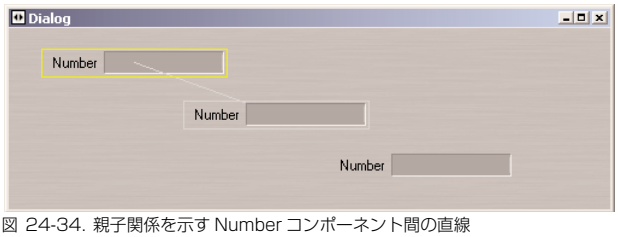

図 24-34. 親子関係を示す Number コンポーネント間の直線

今は親アイテムが移動しても子アイテムはその場に張り付いたままになっています。これは`ctlgroup()`コマンドのインタラクティブ版です。LSID においてインターフェイスを作成するときに親子付けがすべて発生するために、`ctlgroup()`関数はエクスポートされるコードに含まれません。

タブつきのインターフェイスを作成する場合には、親子関係もまた重要な要素となります。親子関係を使用して各タブにコントロールを配置することが出来るのです。

タブと親子付けを見てみましょう。

1 まずはTabコントロールを追加します。

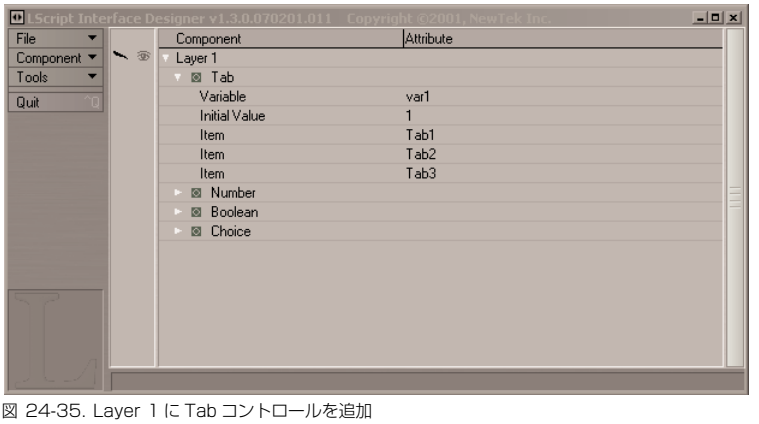

図 24-35. Layer 1 に Tab コントロールを追加

2 初期状態でタブが3つありますから、それぞれのタブに異なるコントロール、**Boolean**に**Number**それに**Choice**を追加しましょう。

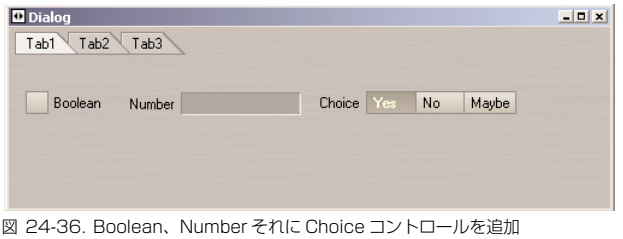

図 24-36. Boolean、Number それに Choice コントロールを追加

3 何もコントロールが割り当てられていませんので、インターフェイス上のすべてが可視状態になっています。コントロールをタブへと親子付けするために、コントロールをクリックし**Item**の行までドラッグします。たとえば、**Boolean**をTab1に、**Number**をTab2に、**Choice**をTab3に挿入してみると、インターフェイスとコンポーネントツリーは以下の画像のようになっているはずです。

図 24-37. コントロールをそれぞれ Item 行の下へドラッグ

4 各タブの下にあるコントロールを見るため、タブをクリックしてみてください。ただしツリービューにおいて個々のタブを選択しなければなりません。作業ダイアログにおいて個別のタブを選択することは出来ないのです。

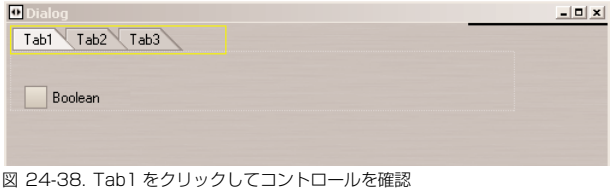

図 24-38. Tab1 をクリックしてコントロールを確認

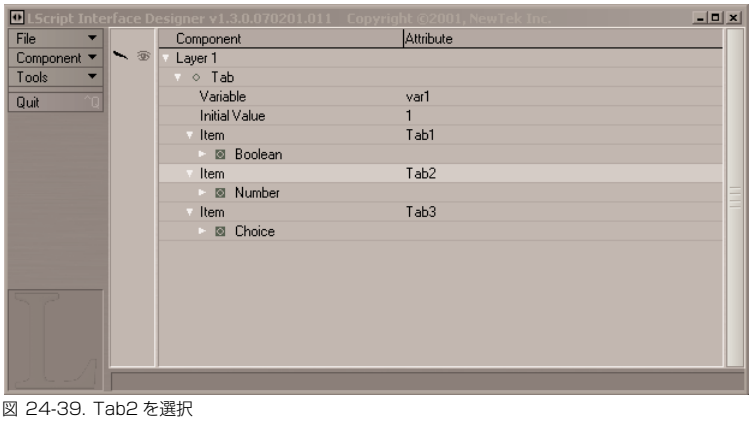

図 24-39. Tab2 を選択

図 24-40. Tab2 をクリックしてコントロールを確認

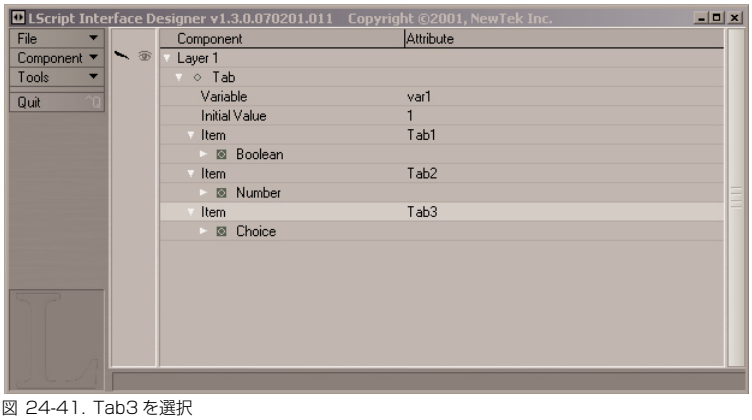

図 24-41. Tab3 を選択

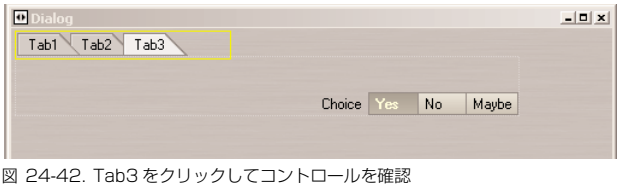

図 24-42. Tab3 をクリックしてコントロールを確認

5 **Export > Layout LScript** を選択すると、**LScript Editor** へとインターフェイスを送信することが出来ます。もしくは Editor が現在起動していない場合には、LSID は生成されたコードをディスクファイルへと保存するために保存名称の入力を促します。

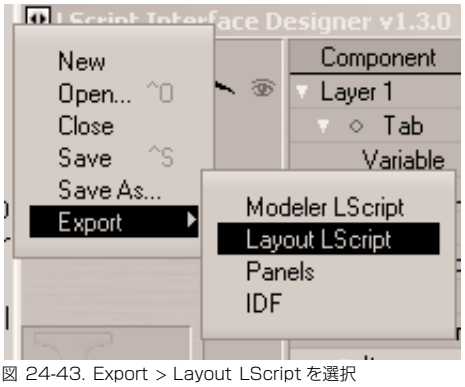

図 24-43. Export > Layout LScript を選択

LSIDは以下の画像にあるようなコードを作成しますので、コードを切り取り、スクリプト内に貼り付けることが出来ます。

図 24-44. エクスポート時に作成されるコード

この時点で少し時間を取り、Interface Designer（.id）ファイルとしてダイアログを保存するのがよいでしょう。そうすれば後日、Designerでプロジェクトを再編集することが可能になるからです。先ほど編集・エクスポートして生成されたコードは、スクリプト内部においてのみ使用可能になっています。これは今まで作業してきたプロジェクトを保存するということではありません。

メッセージエリア

ウィンドウ下にあるメッセージエリアはLSIDEのシステムがユーザーに対し関連する任意の情報を提供するエリアとなります。