

プログラミング言語 VCSSL

- リファレンス ガイド -



第 22 版

目次

第 1 部 基礎と文法	… 7
第 2 部 GUI	… 119
第 3 部 2DCG	… 148
第 4 部 3DCG	… 192
本文書内の商標について	… 331

はじめに

この文書は、プログラミング言語 VCSSL の概要、環境構築方法、基礎文法、標準ライブラリ諸機能など、これまで散在していたドキュメントを 1 冊にまとめ、一部を再編集したものです。この 1 冊で、VCSSL の文法や主要ライブラリの扱いなどの基本事項を、大まかにカバーできる内容を目指しています。

■ Web 版

この文書の大部分は、Web 版も用意されています。Web 版では、サンプルコードの関数コール部分などに、詳細仕様ページへのリンクが付いているなど、より便利な形になっています。ぜひご利用ください。

プログラミング言語 VCSSL 公式情報サイト: <https://www.vcssl.org/ja-jp/>

■ 本文書における VCSSL のバージョン

本文書の内容は、2019 年現在の最新版である VCSSL 3.4 を前提としています。

・旧バージョンとの互換性が問題となる処理

VCSSL は、旧バージョンに対して高い上位互換性を維持しています。つまり、旧来の VCSSL 仕様に準拠して開発されたプログラムは、新しい VCSSL 仕様に準拠した処理システム上で基本的に問題なく動作します。

しかしながら、仕様に問題が見つかったり、当初は予定されていなかった拡張が必要になったりなどが原因で、やむを得ず一部の仕様が変更され、新旧バージョン間で互換性が問題となってしまうような処理も、少数ですが存在します。具体的には、以下の通りです。

・剰余算の優先度(1.0 から 2.0 への間で仕様変更)

剰余演算子「%」は、VCSSL1.0 では乗算・除算よりも高い優先度となっていました。VCSSL2.x 系では、他の C 言語系の言語に合わせ、乗算・剰余算と同じ優先度となりました。

・ファイル入出力の write 関数と read 関数の改行(2.0 から 2.1 への間で仕様変更)

VCSSL2.1 から、write 関数でのファイル出力は改行が付かなくなり、代わりに改行付きの writeln 関数がサポートされました。read 関数も行単位読み込みではなくなり、代わりに行単位読み込みの readln 関数がサポートされました。

・非推奨となった機能

また、最新版でも問題なく動作するものの、仕様が混乱を招き、将来的に仕様変更の可能性があるなどの理由で、使用が推奨されなくなった機能もいくつか存在します。具体的には、以下の通りです。

(※なお、仕様変更の可能性がほぼ無い、単純に新しい機能で代替可能なために不要になった機能については、ここでは掲載していません。そのような機能は、互換性を保つため、特別な事情が無い限りサポートされます。)

・冪(べき)乗演算子の記号(2.0 から非推奨化)

VCSSL1.0で冪乗演算子に割り当てられていた「^」記号は、他のC言語系言語では排他的論理和演算子を意味するため、代わりに「**」記号の使用が推奨されるようになりました。

・配列の要素数を取得する length 関数(2.5 で引数追加、従来のものは非推奨化)

VCSSL1.0 から 2.4 まで、多次元配列の要素数を、配列にまとめて返す `length(array[])` 関数をサポートしてきました。しかしこの関数は、配列次元を右から 0,1,2,... と数えるという仕様が混乱を招く上に、結果を配列にまとめて返すために処理上のボトルネックになるという、複数の問題がありました。

この問題により、VCSSL2.5 から、次元指定のために 2 つめの引数を追加し、従来のものは非推奨となりました。新しい `length(array[], int dim)` 関数では、2 つめの引数 `dim` に要素数を知りたい次元を指定し、結果は非配列の `int` 値で返されます。そして次元は左から 0,1,2,... と数えます。

・8 進数リテラルの記法(3.4 で「0o」プレフィックスをサポート、従来のものは非推奨化)

VCSSL では、他の言語との互換性を考慮して、「0 で始まる整数リテラルは 8 進数として解釈する」という記法を採用してきました。しかしながら、0 で始まる整数は、一般には桁揃えされた 10 進数に見えるため、意図しない誤用を招きかねないと判断し、3.4 以降では非推奨となりました。それに代わって、新たに「0o」をプレフィックスとする 8 進数リテラル記法がサポートされました。従来の記法も引き続き使用できますが、実行時に警告メッセージが表示されます。

なお、この変更に伴い、整数値を 8 進数で表した文字列を返す `System.oct` 関数にも、「0」ではなく「0o」プレフィックスが付くようになりました。この変更は互換性が失われるため、`oct` 関数を使用しているコードでは、必要に応じて部分的な改修が必要となります。

第 1 部

基礎・文法



この章では、まずプログラミング言語 VCSSL の特徴や、開発環境の準備などについて解説します。続いて、あらゆる VCSSL プログラムの開発の基本となる、文法や各種規則など、いわゆる「プログラムの書き方」について解説します。

VCSSL の特徴 - VCSSL ってどんな言語?

はじめに、VCSSL の基本的な事項や、特徴などについて解説します。

■ VCSSL とは

VCSSL は、2011 年公開の、比較的新しい C 言語系のプログラミング言語です。名称は Visualization/Computation/Simulation Script Language (可視化・数値計算/シミュレーションのためのスクリプト言語) の略で、特殊な環境に依存せず、どこでも使用できる数値計算プログラムや科学技術系シミュレーションを、手軽に開発する事を主目的として生まれた言語です。

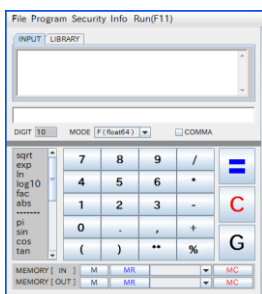
加えて現在では、より汎用的な一般用途においても、例えば簡易 GUI アプリや 3D 可視化ツールなど、比較的小規模なものを手軽に開発できる言語を目指しています (開発最初、VCSSL の「V」は Virtualized の略でしたが、後に上のようなコンセプトが加わったため、現在は Visualization の略となっています)。文法が C 系言語の中でも単純なため、プログラミング入門用にも適しており、C 系他言語へのステップアップも考えられます。

下記公式情報サイトでは、各種開発ガイドやサンプルコード、最新情報などを配信しています。

プログラミング言語 VCSSL 公式サイト: <https://www.vcssl.org/ja-jp/>

■ 開発経緯

・電卓ソフト用言語としての VCSSL 1.0



VCSSL の最初のバージョンである VCSSL 1.0 は、2011 年 1 月に電卓ソフト上で動作する言語として誕生しました。その当初の目的は、電卓ソフト上において、ユーザー定義関数や、ちょっとした短い数値計算プログラムなどを記述する事でした。そのため、数値計算分野でユーザーの多い C 言語系の文法を採用しつつ、電卓上で手軽に扱えるように極力シンプル化した言語となりました。

・GUI やグラフィックスをサポートし、単体のプログラミング言語として独立した VCSSL 2.0 以降

続く 2011 年 8 月公開の VCSSL 2.0 は、GUI や 2D/3D グラフィックスなどの諸機能を大幅に拡充し、数値計算だけでなく、ある程度汎用的な用途にも対応する事を目指したバージョンでした。それに伴い、電卓ソフト付属の言語という立場から、単体で使用する言語へと独立しました。

VCSSL の拡張が最も著しいペースで進んだのは 2.x 世代で、この時点で現在の VCSSL に繋がる基本的な部分はほぼ固まり、以降は現行版の 3.x 世代に至るまで細かな改良が続いています。

■ VCSSL の特徴

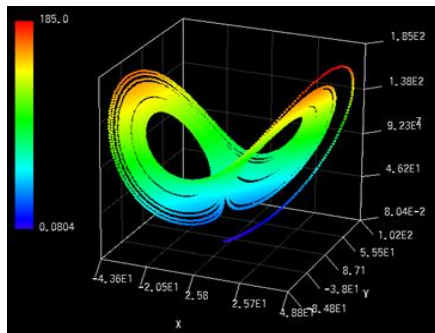
・C 言語(C 系言語)ユーザーが手軽に扱えるシンプルな文法 - プログラミング入門にも

```
int a = 1 ;  
int b = 2 ;  
int c = a + b ;  
print( c ) ;
```

VCSSL は電卓ソフト用の言語として誕生した経緯から、C 系のスタンダードな文法を採用しつつ簡略化し、手軽に扱えるシンプルな文法を目指しています。すでに C 系の言語を扱える方なら、VCSSL の文法はすぐに習得できるでしょう。

また、VCSSL はプログラミング入門にもおすすめです。文法がシンプルで入門しやすい事に加えて、他の C 系言語へのステップアップの足がかりにもなるからです。

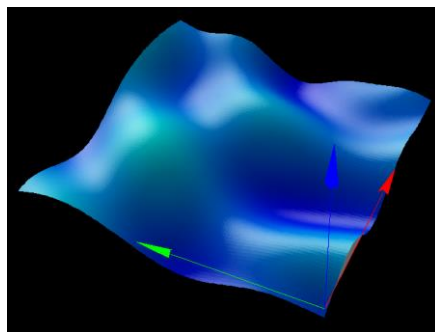
・ちょっとした数値計算やシミュレーションを記述するのに適した特性



電卓ソフト用言語であった事に由来して、VCSSL はちょっとした数値計算やシミュレーションに適した特性を持っています。ここでの「ちょっとした」とは計算負荷を指しており、具体的には C 言語で書いても 1 秒以内で終わる程度の計算をターゲットにしています。

具体的な特長としては、文法が静的型付けである事や、CSV/TSV 形式のファイルを手軽に扱えるファイル入出力機能の存在、2D/3D グラフをプロットするライブラリがサポートされている事などが挙げられます。また、計算速度も毎秒 1 億回 (100MFLOPS) を超え、それなりの水準に達しています。

・GUI や 2D/3D グラフィックス、サウンドなどの各種汎用機能もサポート - 簡易 GUI アプリ開発も



VCSSL は GUI や 2D/3D グラフィックスなどの汎用機能を標準ライブラリでサポートしているため、手軽な文法はそのまま、ある程度汎用的な用途もこなす事ができます。

といっても、VCSSL は比較的小規模なプログラムを想定した言語であるため、高機能なアプリケーションの開発などは得意ではありません。そういった用途では、最初から大規模開発を前提とした汎用言語にはかないません。

しかし、比較的単純な機能で、GUI も少しの入力項目が並ぶだけといった簡易アプリ的なものなら、VCSSL のシンプルさを活かして手軽に組むことができます。アニメーション対応の 2D/3D グラフィックスを手軽に扱えるのも長所です。サウンド再生や時間計測、テキスト処理なども扱えます。

・「 VCSSL エンジン 」 上で動作するスクリプト言語 – 書いてすぐに実行可能

VCSSL は、「VCSSL エンジン」という処理エンジン上で動作する、スクリプト言語の一種です。

(広義の)スクリプト言語とは、人間が記述したプログラム(ソースコード)を、実行前に機械語などに変換しなくても、その場でそのまま実行できる形式のプログラミング言語の事です。

そのため、ちょっとした計算や短い処理などを行いたい場合に便利です。例えば、メモ用のソフトなどでさっと数行書き、「.vcssl」の拡張子を付けて保存すれば、それだけで VCSSL プログラムの完成です。あとは、それをマウスでダブルクリックするだけで、自動的に実行できます。

・様々な環境で動作 - USB フラッシュメモリーで実行環境ごと持ち運びも

VCSSL の実行環境は、主要な各種デスクトップ向けオペレーティングシステムに対応しています。そのため、実行環境さえあれば、VCSSL で記述されたプログラムは、どこでも同様に動作します。スクリプト言語なので、環境に合わせて書き換えたり、変換したりする必要はありません。全く同一の



プログラムが動作します。これは、GUI や 2D/3D グラフィックスなどの機能も例外ではありません。

また、実行環境は USB フラッシュメモリーなどからも起動可能で、インストールも不要です。

従って、プログラムと実行環境を USB フラッシュメモリーに入れて持ち運び、外出先のコンピューターなどで使用する事もできます。

・実行環境は、自作プログラムに同梱して再配布可能

VCSSL の実行環境は、自作のプログラムに同梱して配布できます。この事は、VCSSL の実行環境に関するライセンス(使用許諾)でも、公式に認められています。

VCSSL の実行環境は、先に述べたように、各種デスクトップ向けオペレーティングシステムにおいて、インストール不要で動作します。つまり、自作のプログラムに実行環境を同梱して配布すると、どこの誰でも、そのプログラムを入手してすぐに使う事が可能です。

■ アプリケーション組み込み用サブセット(部分機能版)の「Vnano」も



現在、VCSSL の中心的な機能のみを抜き出したサブセット(部分機能版)として、ユーザーが手軽に自作のアプリケーション等に組み込み、スクリプト処理機能として使用できる事を目指した「Vnano (VCSSL nano の略)」が開発進行中です。Vnano のスクリプトエンジンはオープンソースで、下記 URL の公開リポジトリ上において開発 & 公開されています:

<https://github.com/RINEARN/vnano>

現時点ではまだ正式リリースには至っていませんが、既にライブラリとして呼び出し可能で、大枠では動作する段階になっています。興味がある方は、ぜひ気軽に試してみてください！

なお、将来的には、この Vnano のスクリプトエンジンを土台に、VCSSL のスクリプトエンジンを新実装に置き換えていく事で、VCSSL をオープンソースの言語にしていける計画もあります。

VCSSL の用途 – VCSSL では、こんなことができる

続いて、VCSSL で扱える様々な用途を紹介します。

■ 日常のちょっとした計算をすぐにできる

例えば、日常のあるとき、ちょっとした計算が必要になったとしましょう。電卓ソフトを起動して手動で計算してもいいですが、似たような計算を、値を変えて何度も行うような場合には面倒です。そんな時はメモ用のソフトなどを開き、計算内容を書いて拡張子「.vcssl」をつけて保存すれば、あとはそれをダブルクリックするだけで、すぐに計算を実行できます。

・例 - 素数判定を行うプログラム

<https://www.vcssl.org/ja-jp/code/archive/0001/0300-prime-test/>



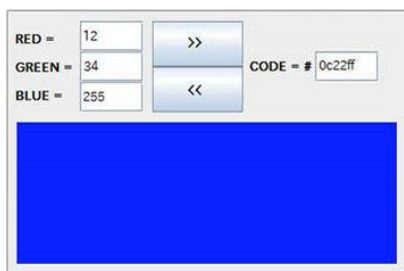
このプログラムは、ユーザーの入力値が素数かどうかを判定する計算プログラムです。50 行に満たない規模でありながら、起動後に入力フォームを表示して値を受け取り、対話的に素数判定を行います。

■ 簡易 GUI アプリを開発できる

高機能な汎用言語で GUI (画面) 部分を作成するには、高機能であるが故に、面倒な記述が必要となる場合があります。特に、中核の処理内容が単純である場合に、GUI 部分に多くの手間をかけるのは割に合わないという事も多いでしょう。VCSSL では、細かさや機能性では汎用言語に劣るものの、最短 3 行でボタンのあるウィンドウを作る事ができます。そのため、小規模な簡易 GUI アプリであれば手軽に開発する事ができます。

・例 - 16 進数カラーコードと RGB 値の相互変換・色表示アプリ

<https://www.vcssl.org/ja-jp/code/archive/0001/1000-color-display/>



このプログラムは、GUI 上で RGB 値とカラーコードを相互変換するプログラムです。実際の色を表示して確認する事もできます。全体の規模は空白・コメント込みで 450 行程度、GUI の制御に関する部分は 250 行程度です。

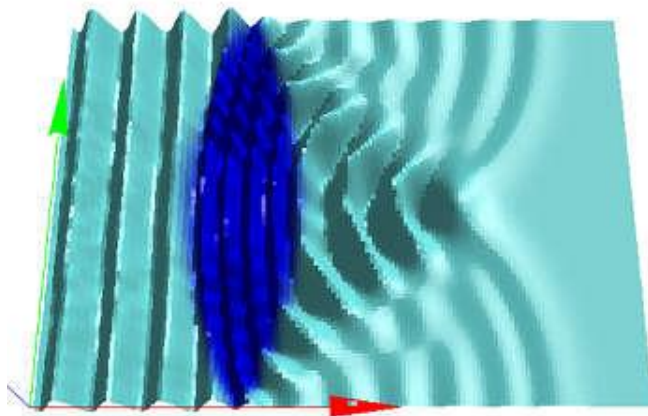
■ アニメーション対応の 2D/3D グラフィックスで、リアルタイムシミュレーションも

VCSSL は、2D/3D グラフィックスを標準ライブラリでサポートしています。共にリアルタイムでの処理に対応しており、アニメーション描画などを行う事が可能となっています。こうしたグラフィックス機能を活用すれば、他言語では複雑大規模になりがちなリアルタイム演算・アニメーション描画のシミュレーションなども、VCSSL のみで完結して開発する事ができます。

なお、VCSSL の 3D グラフィックス機能は、ソフトウェア実装のレンダラー（描画エンジン）で処理されるため、グラフィックスボードなどの特別なハードウェアの有無に関わらず、どのようなコンピューター上においても動作します。その分、ハードウェアを用いる場合と比較してパフォーマンスは制限されますが、それでも数十万ポリゴン/秒の性能水準を実現しています。

・例 - 凸レンズを通過する波のシミュレーション

<https://www.vcssl.org/ja-jp/code/archive/0001/1800-convex-lens/>

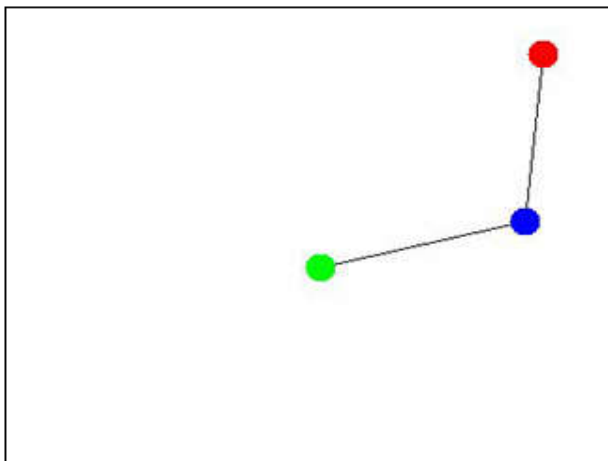


このプログラムは、弾性体の媒質中を伝わる力学波が、凸レンズ形状の密度分布で屈折し、焦点に集中する様子を再現したシミュレーションです。

物理演算はリアルタイムで行い、その結果を 3D アニメーションで表示しています。なお、密度分布は PNG 形式画像ファイルから読み込む設計になっています。

・例 - 二重振り子のシミュレーション

<https://www.vcssl.org/ja-jp/code/archive/0001/0200-duplex-pendulum/>



このプログラムは、二重振り子の運動を再現したシミュレーションです。こちらも物理演算はリアルタイムで行い、その結果を 2D アニメーションで表示しています。

リアルタイム物理演算では、ラグランジュ方程式の時間発展の計算を 1 フレームあたり約 5000 回処理し、振り子の運動を再現しています。

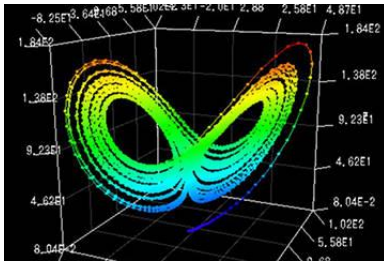
■ グラフプロット機能を活用して数値計算も

VCSSL は、電卓ソフト用の言語であった事の名残として、2D/3D のグラフプロット機能が充実しており、数値計算プログラムでは強力です。グラフ機能は標準ライブラリでこそありませんが、ほぼ全ての実行環境がサポートしており、手軽に使用できます。

ところで数値計算と言えば C 言語が主流です。もちろん処理速度では C 言語が圧倒的に有利で、C で数十秒～数分かかるといった場合、VCSSL は明らかに適していません。しかし短時間で終わるような軽い計算の場合なら、VCSSL の方がプログラム内からグラフにプロットでき、パラメータを変えてのアニメーションプロットも可能となるなど、効率的な解析を行えるため、用途によっては有利になり得ます。必要であれば GUI でパラメータ設定画面を作れるのも利点です。

・例 - ローレンツアトラクタ(ファイル出力版)

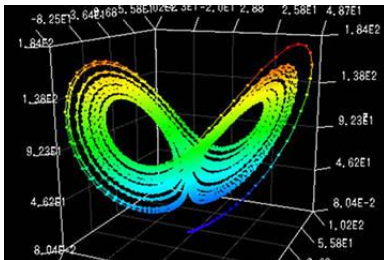
<https://www.vcssl.org/ja-jp/code/archive/0001/1600-lorenz-attractor-file/>



このプログラムは、ローレンツアトラクタを 4 次ルンゲ=クッタ法で計算し、結果をファイルに書き出して、それを 3D グラフにプロットさせる、いわゆる「典型的な数値計算」のプログラムです。このプログラムは C 言語でもほぼ同じ動作を実現できるため、VCSSL で記述するメリットはあまりありません。

・例 - ローレンツアトラクタ(GUI 版)

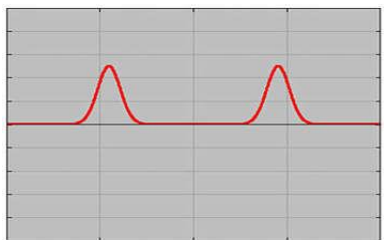
<https://www.vcssl.org/ja-jp/code/archive/0001/0100-lorenz-attractor/>



このプログラムは、上と同様ローレンツアトラクタを計算して 3D グラフにプロットするプログラムですが、ファイルを介さずに、座標値配列を直接グラフに転送しています。さらに GUI の設定画面を備えており、スライダーでパラメータを操作すると、自動で再計算されてグラフが更新されます。

・例 - 力学アルゴリズムによる波のシミュレーション(線上の波)

<https://www.vcssl.org/ja-jp/code/archive/0001/1100-string-wave/>



このプログラムは、弾性体の線上を伝わる波の様子を、2D グラフにアニメーションプロットするプログラムです。この計算は軽く、VCSSL でも一瞬で終わるため、毎秒数十回の頻度で再計算とプロットを繰り返し、アニメーションさせています。

実行環境の用意 – VCSSL をはじめよう

それではいよいよ、VCSSL の実行環境を用意し、VCSSL プログラムを開発・実行してみましょう。

■ 「 Hello, World ! 」と表示するプログラムを開発する

それでは、早速 VCSSL のプログラムを開発します。まず、メモ用のものでも何でもよいので、何かテキストを書くソフトを起動してください。そして、以下のように記述してみてください。

```
print ( "Hello, World !" );
```

記述が完了したら、「 **Test.vcssl** 」という名前を付けて適当な場所に保存してください。なお、ファイルの種類を選べる場合は、「 **すべてのファイル** 」などを選択する必要があります。

なお、保存したファイルの拡張子がちゃんと「 .vcssl 」になっているかご注意ください。もし拡張子が表示されない場合、オペレーティングシステムの設定で、拡張子を表示するようにしておくと便利です。方法は「 **拡張子 表示** 」などのキーワードで Web 検索してください。



以上で、最初のプログラムは完成です。このプログラムは、画面に「 **Hello, World !** 」とメッセージを表示する内容になっています。

つまりこのプログラムで実際に「 Hello, World ! 」と表示させる事ができれば、VCSSL プログラミングの環境が整った事になります。それでは、環境を用意しましょう。

■ 「 VCSSL ランタイム 」のダウンロードと展開

VCSSL の最も標準的な実行環境は、「 **VCSSL ランタイム** 」で、下記 URL から入手できます。

VCSSL ランタイム 入手先 URL : <https://www.vcssl.org/ja-jp/download/>

まずは上記ページにアクセスし、「 **いますぐダウンロード** 」のボタンを押してダウンロードします：



ダウンロードしたファイルは圧縮された ZIP 形式のファイルになっているため、**その ZIP ファイルを**右クリックし、メニューから「**すべて展開**」や「**ここに展開**」などを選択して展開してください。

※ 「問題を引き起こす可能性～」などのエラーが表示されて展開を完了できない場合、ZIP ファイルを右クリックして「プロパティ」を選択し、**プロパティの画面の下にあるセキュリティ項目の「許可する」にチェックを入れて「OK」**すると、以降は展開可能になります。このエラーは、インターネット等から入手した不明なプログラムを安易に実行しないための、OS のセキュリティ機能によるものです。

※ **Linux®等をご使用の場合で、展開結果のファイル名の日本語が文字化けしてしまう場合があります。**その場合、コマンドライン端末から以下のように unzip コマンドで展開してみてください：

```
cd ZIPファイルのある場所
unzip -O cp932 ZIPファイル名
```

効果が無かった場合は、他の展開ソフトを使用するか、展開後に convmv コマンドなどで文字化けの修復処理を試してみてください（日本語ファイル名は CP932 でエンコードされています）。

■ VCSSL ランタイムを起動してプログラムを実行

展開できたら、実際に使用してみましょう。方法は複数ありますが、まずは最も簡単な方法です。

Microsoft® Windows® をご使用の場合は、ZIP ファイルを展開してできたフォルダ内にある「VCSSL_??.?.bat（バッチファイル、「?」の箇所はバージョン番号の数字）」をダブルクリックして実行してください。

Linux 等やその他の OS では、同フォルダ内までコマンドライン端末上で cd コマンドで移動し、「java -jar VCSSL.jar」とコマンド入力して「VCSSL.jar（JAR ファイル）」を実行してください（メモリ使用量を指定したい場合は、例えば 512MB なら「java -Xmx512m -jar VCSSL.jar」のように -Xmx オプションを追加してください）。

すると VCSSL ランタイムが起動し、プログラムを選択する画面が表示されます。そこで、先ほど開発した VCSSL プログラム「**Test.vcssl**」を選択してください。黒い画面が起動し、「Hello, World!」と表示されれば成功です:



※画像はイメージで、実際の見た目は環境により異なります。

この方法では、コンピューターに設定などを一切行わず、とにかく即席でプログラムを実行することができます。そのため、VCSSL ランタイムを USB メモリーに入れて持ち運ぶような場合には、大変便利です。**なお、Microsoft® Windows® をご使用の場合は、VCSSL プログラムを右クリックして「プログラムから開く」を選択し、そこで先ほどのバッチファイルを指定する事でも実行できます。**

※ java コマンドの使用でエラーになってしまう場合は…

Linux®等やその他の OS で、上記の手順に従って起動した場合、そもそも java コマンドが使用できない旨のエラーが出る場合があります。これはご使用の環境に Java®の実行環境 (JRE) が無いためで、使用するには導入が必要です。apt コマンドが使える場合は、コマンドライン端末上で:

```
apt search jre      (または apt の代わりに apt-cache)
```

と入力して入手可能なものの一覧を確認した上で:

```
sudo apt install default-jre      (または apt の代わりに apt-get)
または
sudo apt install openjdk-?-jre    (?の箇所にはバージョンの数字が入ります)
```

などとして簡単に導入できるかもしれません (環境によります)。

導入するのは他のものでも構いませんが、VCSSL ランタイムが動作しないものもあります (※末尾に **-headless** が付いているものでは動作しないので、付けないようご注意ください)。

■ 付属のインストーラでインストールして使用方法 (※ Windows をご使用の場合のみ)

Microsoft® Windows® をご使用の場合は、付属のインストーラ「**Install.exe**」を実行して、VCSSL をインストールして使用する事もできます。インストールすると、VCSSL プログラム(拡張子 .vcssl)が青色のアイコンで表示されて見分けやすくなり、また、VCSSL プログラムをダブルクリックすると自動で実行される状態になります。

※後から新しいバージョンをインストールする場合は、あらかじめ古いバージョンをアンインストールしてください。

なお、インストールした場合でも、コマンドプロンプトから使用するためのパス設定は、自動では行われません。通常のインストール場所(推奨)にインストールした場合、「C:¥VCSSL¥bin」などを環境変数 Path に設定する必要があります。方法については次の項目をご参照ください。

■ コマンドライン端末上から vcssl コマンドで起動し、プログラムを実行する方法

コマンドプロンプトやシェルなどのコマンドライン端末上で常用したい場合は、VCSSL ランタイムをダウンロード・展開したフォルダ内の「bin」というフォルダのパス(場所)を、環境変数 Path または PATH に設定します(後述)。すると vcssl コマンドが使用可能になり、

```
vcssl Test.vcssl
```

とコマンド入力するだけで、VCSSL ランタイムを起動してプログラム「Test.vcssl」を実行できるようになります。なお、main 関数を持つプログラムでは、末尾に引数を指定する事もできます：

```
vcssl Test.vcssl arg0 arg1 arg2
```

・Microsoft® Windows® をご使用の場合のパス設定

まずは、VCSSL ランタイムのフォルダをどこか適当な場所へ配置(ずっとそこへ置きます)してください。なお、インストールした場合は通常「C:¥VCSSL」に配置されています。

その後にパス設定を行いますが、バージョンによって画面の開き方や手順が異なるため、詳細は「Windows Path 設定」などで検索してみてください。Windows 10 をご使用の場合は：

スタートボタン > 歯車アイコン(設定) > 「環境変数を編集」と検索して移動 > 開かれた画面で「ユーザー環境変数」の一覧から「Path」(無ければ作成)を選び「編集」 > 「新規」を押し、VCSSL ランタイムの「bin」フォルダのパスを入力※
(※ Shift キーを押しながら bin フォルダを右クリックすると、メニューからパスのコピーを行えます)

で行えますが、誤って他の値をいじったり、削除してしまったりすると大変な事になるのでご注意ください(上の説明で、システム環境変数への登録もできますが、その点を考慮してユーザー環境変数に登録しています)。手慣れた方に依頼できる場合は、行ってもらう方が無難かもしれません。

・Linux® 等やその他の OS をご使用の場合のパス設定(および実行権限設定)

まずは、VCSSL ランタイムのディレクトリをどこか適当な場所へ配置(ずっとそこへ置きます)してください。ここでは例として以下の場所に配置したとします。

```
/usr/local/bin/vcssl/vcssl_?_?_?/ (??_?の箇所はバージョン番号です)
```

続いてcd コマンドでこの場所のさらにbin ディレクトリ内まで移動し、以下のコマンドを実行します:

```
chmod +x vcssl (起動用のシェルスクリプトに実行権限を付加しています)
```

最後に、ユーザーのホームディレクトリにある、「.bashrc(隠しファイル)」または「.bash_profile」もしくは「.profile」(どのファイルが有効かはオペレーティングシステムによって異なります)をテキストエディタで開き、最終行に下記の一行を追記してください。

```
export PATH=$PATH:/usr/local/bin/vcssl/vcssl_?_?_?/bin/  
(??_?の箇所はバージョン番号で置き換えて下さい)
```

\$PATH:以降の内容は、VCSSL ランタイムのディレクトリを配置した場所に合わせてください。

■ VCSSL の基本画面（VCSSL コンソール）

「 Test.vcssl 」を実行すると、画面に黒いウィンドウが表示され、白い文字で「 Hello, World ! 」と表示されます。



※画像は模式図です。実際の外観は、オペレーションシステムの種類などによって異なります。

この黒い画面は「 VCSSL コンソール 」というもので、VCSSL プログラムの実行中に出現し、ここにメッセージなどを表示する事ができます。VCSSL コンソールを閉じると、VCSSL プログラムの実行も終了します。なお、GUI を用いたプログラムでは、邪魔であれば非表示にする事もできます。

```
-----
エラー 1
[ 発生箇所 ] プログラム「 UNKNOWN_PROGRAM 」の 3 行目
[ 処理内容 ] r_x = 2.0
[ 詳細情報 ]
    宣言されていない変数「r_x」を呼び出しています。

-----
エラー 2
[ 発生箇所 ] プログラム「 UNKNOWN_PROGRAM 」の 11 行目
[ 処理内容 ] f = fun( x )
[ 詳細情報 ]
    関数「 fun 」
    を呼び出しましたが、引数の型/個数が適合しませんでした。
    以下に引数の候補を表示します：

    fun( int[] a, int[] b )
    fun( int a, int b )
```

VCSSL コンソールには、自分でメッセージを表示するだけでなく、システム側からメッセージが表示される事もあります。

それはプログラム中に記述ミスがあった場合や、実行中にエラーが発生した場合などで、その内容に関連したシステムメッセージが表示されます。

システムメッセージを表示している際には、**VCSSL コンソールの色が青色**になります。

■ その他の実行環境

VCSSL には、上で解説した VCSSL ランタイム以外にも、様々な実行環境があります。第 1 部の最後に、主要な実行環境をご紹介します。用途に応じて適したものを選択してください。

・VCSSL エディタ



VCSSL エディタ (VCSSL Editor) は、VCSSL ランタイムに同梱されている、プログラム記述用の簡易エディタです。

シンプルなテキストエディタとランタイムが一体化したソフトで、プログラムを書いてすぐに実行する事ができるため、短いプログラムを手軽に記述するのに適しています。

VCSSL ランタイムをインストールした場合は、「VCSSL Editor」のアイコンから起動できます。そうでない場合は、VCSSL ランタイムと同様、「VCSSL_Editor_~.bat (バッチファイル)」または「VCSSL_Editor.jar」を起動します。

・リニアプロセッサ (関数電卓ソフト)

<https://www.rinearn.com/ja-jp/processor/>



リニアプロセッサ (RINEARN Processor) は、VCSSL の開発の元となった関数電卓ソフトです。VCSSL のプログラムを開発・実行できるのはもちろん、VCSSL で記述した関数を、関数電卓の機能として使用する事ができます。さらに、開発したプログラムをメニューバーに登録しておき、いつでもすぐに使用する事ができます。

VCSSL で書いた計算コードなどを、日常で使いたい場合に便利です。なお、電卓向け機能だけでなく、常に最新版の VCSSL の機能がフルスペックで動作します。

プログラム

開発作業に入る前に、プログラミングの前提となる事をいくつか抑えておきましょう。

■ プログラム

・コンピューターとソフトウェア

誕生からわずか数十年、コンピューターはまるで夢物語のように急速な進化を続け、毎年のようにどんどん便利になり、今日ではまさに何でもできる魔法の機械となりました。その背景には、もちろん処理性能の向上といったハードウェア面での進化もありますが、ソフトウェア面での進化も決して見逃せません。

よく知られている通り、コンピューターは、ソフトウェアを入れる事で、次々と新しい機能を追加していけます。どんなに高性能なコンピューターも、生産されたばかりの時点では、何もできません。現在のコンピューターが、非常に便利で何でもできるのは、実に様々な種類のソフトウェアが普及しているおかげでもあるのです。

・プログラム

このように、コンピューター上で様々な機能を実現してくれるソフトウェアですが、そもそもソフトウェアとは、一体何なのでしょう。

実はソフトウェアの実体、つまり中身は、「プログラム」というものです。プログラムには、いわゆるマシン語から簡易スクリプトに至るまで、実に様々な種類のものが存在しますが、それら全てに共通しているのは、「コンピューターに対する命令が、規則正しく並んでいる」という事です。コンピューター（もしくはランタイムなどの実行環境）は、プログラム中の命令を規則正しく順々に実行していき、その結果としてソフトウェアの機能が実現されます。つまりプログラムとは、コンピューターに対する「処理の手順書」のようなものです。

私達がコンピューターでインターネットができるのも、ゲームができるのも、さらに言うなら画面に文字が表示されるのも、全てそういうプログラムがあるからで、誰かがそれを作ったからなのです。

■ プログラミング言語

プログラムについて、「コンピューターに対する命令が、規則正しく並んでいる」ものだと言いましたが、その「規則」となるのが、プログラミング言語です。プログラミング言語には、実に様々な種類のものが存在しますが、大別すると以下のように分類できます。

・マシン語、アセンブリ言語

最も直接的な形でコンピューターに命令し、全ての土台となっているのが、マシン語（機械語）です。マシン語は 0 と 1 の羅列で構成され、コンピューターが直接理解し、実行する事が可能です。というよりも、マシン語を読んでその通りに動作するように、コンピューターの中核部（CPU）は設計され、生産されているのです。

マシン語は 0 と 1 だけで構成されるため、人間が直接マシン語でプログラムを開発するのは困難です。そこで、各命令に対応する 0 と 1 の羅列に、英字の語句（ニーモニック）を割り当て、読みやすくしたものがアセンブリ言語です。実行する際には、ニーモニックを元のマシン語に置き換えます。この作業は人間の手でも可能ですが、一般には「アセンブラ」というソフトウェアが使用されます。

・コンパイラ型言語

マシン語やアセンブリ言語は、基本的に CPU に直接用意されている機能（基礎演算やメモリーアクセス、ジャンプなど）しか使用できないため、手作業でそれらを組み合わせて高機能なソフトウェアを開発するには、かなりの労力が必要となります。

そこで、もっと汎用的で便利な機能を言語としてサポートしておいて、それを用いて楽にプログラムを記述し、そのプログラムを、「コンパイラ」と呼ばれる特殊なソフトウェアでマシン語に変換して実行するのが、コンパイラ型言語です。有名な「C 言語」も、コンパイラ型言語です。

コンパイラ型言語を用いても最終的に仕上がるものはマシン語のプログラムですが、高機能な部分の実装はコンパイラが自動で行ってくれるので、マシン語やアセンブリ言語でゼロから開発するよりも、はるかに簡単に済みます。それでいて、マシン語として CPU 上で直接実行されるので、処理速度も非常に高速で、CPU の性能を最大限活用する事ができます。

・スクリプト言語

コンパイラ型言語で記述されたプログラムは高機能ですが、マシン語に変換しなければ実行できません。頻繁に書き換えるようなプログラムなど、いちいち変換するのが面倒な場合もあります。また、マシン語は CPU の世代や種類によって様々な種類が存在するため、マシン語のプログラムには機種依存性が生じます。

そこで、プログラムを事前にマシン語には変換せず、そのまま実行するのが、スクリプト言語です。もちろん、コンピューターはマシン語しか読めませんから、コンピューターがスクリプト言語のプログラムを直接実行する事はできません。そこで、スクリプト言語のプログラムは、「インタープリタ」という特殊なソフトウェアの上で実行されます。インタープリタは、プログラムを解釈し、実行できるように設計されたソフトウェアです。つまるところ、プログラムを動かすためのプログラムです。

・VCSSL はスクリプト言語

VCSSL は、最後に挙げたスクリプト言語に該当します。VCSSL で記述されたプログラムは、「VCSSL エンジン」というインタープリタ上で解釈され、実行されます。これにより、VCSSL ではプログラムを書いてそのまま実行でき、さらに様々なオペレーティングシステム上でそのまま動作し、機種依存性もありません。手軽に扱う事ができます。

反面、C 言語などと比較すると処理速度では劣ります。

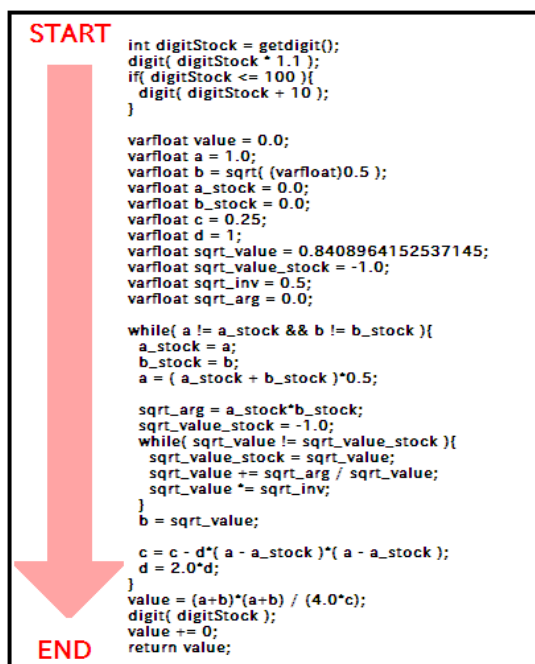
■ VCSSL プログラムの基本

それでは、いよいよ VCSSL でプログラムを開発してみましょう。まず、全ての基本となるルールをいくつか押さえておきます。

・プログラムは上から下へ実行される

VCSSLに限らず、多くのプログラミング言語では、プログラムの内容は基本的に上の行から下の行へ向かう流れで処理されていきます。1つの行(文)で1つの処理を意味します。これは人間が横書きの本を読む場合と同じです。

ただし、関数の呼び出しなどにより、処理が突然離れた行や、別のプログラムの内容に移る場合もあります。しかし、移った先では、また上から下へと処理されていきます。



・行(文)の終わりには「 ; 」(セミコロン)を付ける

VCSSLをはじめ、C 言語系の文法をもつプログラミング言語の多くは、行(文)の区切りを「 ; 」(セミコロン)で認識します。テキストエディタ上での改行の有無は処理に全く関係が無く、どんなに改行を含もうともセミコロンが打たれるまでが「一行(一つの文)」となります。これにより、長い式をテキストエディタの複数行に渡って記述する事ができます。しかしながら、改行してもセミコロンを打ち忘れるとエラーとなるため、慣れるまでは注意が必要です。

・プログラムは半角英数字で記述、ただし変数名や関数名には日本語も可

VCSSLのプログラムは、原則として半角英数字で記述しなければなりません。同じアルファベットでも、全角で記述すると処理できません。特に、全角スペースを誤って使用しないよう気を付けてください。ただし、変数名や関数名には、日本語を含む全角文字を使用する事も可能です。

・ダブルスラッシュ「//」よりも後は無視される(コメント)

プログラム中でダブルスラッシュ「//」を記述すると、その行の中で、そこよりも後の部分は無視されます。これを利用して、プログラム中にメモやコメントを記入する事が可能です。

・「/*」と「*/」で囲った箇所は無視される(コメント)

ダブルスラッシュ「//」によるコメントは、一行のみが無視されます。それに対し、「/*」と「*/」で囲った部分は、複数行にわたって無視されます。長いコメントを記述したい場合や、プログラム内の特定の処理を一時的に削除したい場合などに使用します。

・処理内容を画面に出力(表示)する

処理内容の出力は、プログラミングにおいて最も重要な操作です。ここでは、print 関数を用いて、VCSSL コンソールにメッセージを表示してみましょう。下記のように記述し、実行してみてください。

```
print ( "Hello, World !" );
```

このプログラムを実行すると、VCSSL コンソールに「 Hello, World ! 」と表示されます。この他

にも、数値や式の値などを表示する事も可能です。下記のように記述し、実行してみてください。

```
print ( 1 + 2 * 3 );
```

このプログラムを実行すると、VCSSL コンソールに「 7 」と表示されます。

なお、処理内容の出力には、print 関数の他にも、println 関数や pop 関数が存在します。println 関数は、print 関数に自動で改行を行う機能が付いた関数です。pop 関数は、VCSSL コンソールでは無く、独立したメッセージウィンドウにメッセージを表示する関数です。



なお、実際にどのような形でメッセージが表示されるかは、処理系やオペレーティングシステムに依存します。図のように独立したウィンドウでは無く、ソフトウェアの画面の一部にメッセージが表示されるような場合もあり得ます。

変数

■ 変数とは

プログラミングの世界では、基本的に、計算に用いる値は「変数」に入れて使います。変数は、値を格納しておく器のようなものです。つまり変数という器の中に値を入れておき、必要に応じて中身を読み取ったり、書き込んだりするわけです。

■ 変数の宣言

変数を使うには、まず宣言を行います。宣言は以下のような記述で行います。

```
値のデータ型 変数名 ;
```

データ型についてはすぐ後で解説しますが、とりあえず器の種類のようなものだと思ってください。変数名は変数に割り振る名前、自由に付ける事ができます。例えば：

```
string s ;
```

これで string 型(文字列を扱う型)の「s」という名前の変数が用意されます。このプログラムを実行しても、何も起こりません。ただ変数を用意しただけで、処理を行っていないからです。

それでは、変数 s に値を代入してみましょう。以下のようなプログラムを実行してみてください：

```
string s ;  
s = " Good Morning ! " ;  
print( s ) ;
```

上のプログラムを実行すると、VCSSL コンソールに「 Good Morning ! 」と表示されます。つまり print 関数で表示する内容を、変数「s」で渡したわけです。そして変数「s」を受け取った print 関

数は、その中に入っている値を読み込んで、VCSSL コンソールに表示したのです。

■ 変数の代入

さて、上の流れにおいて「 = (イコール) 」記号が登場しました。この記号は、算数的な意味でのイコールではなく、「変数の中に値を入れる」という意味を持つ記号です。もう少し詳しく述べると、「イコールの右側にある値を、イコールの左側の変数に代入する」という処理を行います。**どんな時でも必ず右から左へ代入**されます。

■ 初期化

宣言した変数に、初めて数値を代入する事を、一般に「変数を初期化する」と言います。初期化されていない変数は、正常に使用できない場合があります。変数を宣言した場合には、その変数を使用する前に必ず初期化するのを忘れないようにご注意ください。

■ 宣言と初期化を同時に行う

以下のように、変数の宣言と初期化を同時に行う事もできます。

```
string s = " Good morning. " ;
```

この記述方法には、行数を節約できるだけでなく、変数の初期化忘れを防ぐ効果があります。

■ 複数の変数宣言を一行で行う

変数型が同じ場合、以下のように、複数の変数の宣言を一行で行う事が可能です。

```
string a, b, c="Hello", d="World", e, f, g ;
```

データ型

■ データ型とは

プログラム内のデータには、変数に格納されているものや、式の中などに直接記述されている値も含めて、いくつかの種類があります。この種類の事を**データ型**と言います。VCSSL の変数では、格納する値のデータ型は宣言時に決める必要があり、その後は変えられません（いわゆる静的型付けです）。

データ型によって、それぞれ扱える種類の値と、扱えない種類の値があります。また、行える演算にも違いがあります。例えば整数を扱う `int` 型で文字列は扱えません。また、文字列を扱う `string` 型を用いて四則演算はできません。従って、変数を宣言する際は、扱う値の種類に適したデータ型を選んで宣言しておく必要があります。

VCSSL でサポートされているデータ型には、以下のようなものがあります。

型	格納できる値	容量	詳細
<code>string</code>	文字列	任意の長さ	文字列を格納するためのデータ型です。
<code>int</code> または <code>long</code>	整数	64bit / 18 桁程度	整数を格納するためのデータ型です。処理に 64bit の 2 進法が適用されるので、高速な計算が可能です。
<code>float</code> または <code>double</code>	小数	64bit / 15 桁程度	小数を格納するためのデータ型です。処理に 64bit の 2 進法が適用されるので、高速な計算が可能です。精度はあまり良くなく、計算する度に、末尾の数桁に計算誤差が生じます。
<code>complex</code>	複素数	<code>float</code> ×2 値	複素数を格納するためのデータ型です。 <code>float</code> 型の実数値と虚数値を持ちます。
<code>varint</code>	任意桁の整数	任意桁数	整数を格納するためのデータ型です。非常に高精度な計算方法が適用されるので、どこまでも長い桁数を扱う事ができます。反面、 <code>int</code> 型よりも処理速度は数十倍以上遅くなります。
<code>varfloat</code>	任意桁の小数	任意桁数	小数を格納するためのデータ型です。非常に高精度な計算方法が適用されるので、どこまでも長い桁数を扱う事ができます。反面、 <code>float</code> 型よりも処理速度は数十倍以上遅くなります。

varcomplex	任意桁の 複素数	varfloat×2 値	複素数を格納するためのデータ型です。 varfloat 型の実数値と虚数値を持ちます。
bool	真偽値 True/false	2 択	条件判定などの結果を格納するためのデータ 型です。値の true と false の意味は、そのまま YES と NO と解釈しても差し支えないでしょう。
struct	構造体		任意の型・個数の変数をまとめて扱う型です。

※他のプログラミング言語では、int 型と float 型の容量に比べて、long (または long int など) 型と double 型の容量が倍 (またはそれ以上) になっている場合がよくあります。VCSSL ではこれらは全て 64bit の精度を持つため、区別する必要はありません。long や double 型がサポートされているのは、あくまでも他の言語へのコードの移植性を考慮した仕様です。

■ 整数型を使用する

例として、整数を格納できる int 型の変数を使用してみましょう：

```
int i = 1 ;  
print ( i ) ;
```

このプログラムを実行すると、VCSSL コンソールに「 1 」と表示されます。

ところで、int 型に整数以外のものを代入すると、どうなるのでしょうか。以下のようなプログラムを実行してみてください：

```
int i = 2.883 ;  
print ( i ) ;
```

このプログラムを実行すると、VCSSL コンソールに「 2 」が表示されます。つまり、代入された「2.883」の小数点以下を切り捨て、無理矢理に整数に変換して代入された事がわかります。このように、**変数は、自分の型では扱えない値が代入されると、自分が扱える値に変換してから受け取ります。**このようなルールを「**暗黙の型変換**」と呼びます。なお、変換不能な値が代入された場合は、エラーが出力され、プログラムが強制終了されます。

さて、小数(実数、浮動小数点数)を扱うには、本来なら double 型または float 型を使用すべきです。それには以下のように記述にします：

```
float f = 2.883 ;  
print ( f ) ;
```

このプログラムを実行すると、VCSSL コンソールに「 2.883 」と正しく表示されます。

■ キャスト演算(明示的な型変換)

上の例において、整数型に小数値を代入すると、無理矢理に整数へと変換される事を述べまし

た。このような変換は、以下のように、任意の場所で明示的に行う事もできます：

```
print ( (int)2.883 );
```

このプログラムを実行すると、VCSSL コンソールに「 2 」と表示されます。このプログラムの「 (int) 」の部分に着目してください。このようにデータ型をカッコで囲んだ記述の事をキャスト演算子と呼びます。キャスト演算子は、すぐ右側の値を、任意の型へ強制的に変換する演算子です。上の例では、「2.883」を強制的に整数へ変換し、「2」にしたのです。

■ 複素数型を使用する

続いて、複素数を格納できる complex 型を使用してみましょう：

```
complex c = 1.0 + 2.0 * I; // 1 + 2 i  
print( c );
```

このプログラムを実行すると、VCSSL コンソールに「 (1.0,2.0) 」と表示されます。ここで「 I 」は、虚数単位の値をもつ変数(定数)で、システムによって自動的に定義されています。小数に I を乗算すると虚数になります。上の例での値は数学的に $1 + 2i$ に相当します。

なお、複素数型の実部や虚部を取得するには、re 関数と im 関数を使用します：

```
complex c = 1.0 + 2.0 * I; // 1 + 2 i  
float r = re( c ); // c の実部を取得  
print( r );
```

このプログラムを実行すると、VCSSL コンソールに「 1.0 」と表示されます。これはcの実部です。re 関数の代わりにim 関数を使用すると、コンソールにはcの虚部である「 2.0 」が表示されます。

■ 可変精度変数を使用する

小数には、有限精度の double 型 または float 型の他にも、任意の精度を持つ varfloat 型も用意されています。これは、使用する桁数を無制限に指定でき、なおかつ実行中の桁数変更も可能な変数です。桁数は、プログラム中で digit 関数を呼び出します：

```
digit( 100 ); // 桁数を 100 桁に設定
varfloat v1 = 10.0vf ;
varfloat v2 = 3.0vf ;
varfloat v3 = v1 / v2 ; // 100 桁で除算を行う
print( v3 );
```

これを実行すると、100 桁まで 3.33333333… が出力されます。

なお、varfloat 型の桁数を制御する digit 関数は、設定した桁数を値として返します。この性質を利用し、digit 関数を空で呼び出すことにより、現在の演算桁数を取得できます：

```
int d = digit( ) ; // 現在の設定桁数を d に取得
```

■ 可変精度型の数値

2 つ上のプログラムでは、「 10.0vf 」のように、数値の後に「 vf 」が付加されていました。これは、その数値が varfloat 型である事を明示するためのものです。

通常は、式の中に直接記述した数値にも、適した変数型が自動で割り当てられます。しかし、varfloat 型と varint 型を使用する際には注意が必要です。式の中に少数や整数が存在する場合、それが float 型になるのか varfloat 型になるのか、または int 型になるのか varint 型になるのかは、言語仕様として定まってはいません。

従って、確実に varfloat 型の数値を割り当てたい場合には末尾に「 vf 」を、また確実に varint 型を割り当てたい場合は末尾に「 vi 」を付加しておく事が推奨されます。

例として、以下の 2 つのプログラムを比較してみましょう。

```
digit( 100 );
print( 1.0 / 3.0 );
```

```
digit( 100 );  
print( 1.0vf / 3.0vf );
```

上のプログラムでは、VCSSL コンソールに 16 桁程度「 0.33333... 」と出力されます。つまり式の中の数値が double 型 または float 型として扱われた事を意味します。

下のプログラムでは、今度は 100 桁程度「 0.33333... 」と出力されます。数値の末尾に「 vf 」を付けたため、数値が varfloat 型として扱われた事を意味します。

■ 可変精度型の複素数

可変精度の複素数 varcomplex 型は、以下のように使用します。

```
digit( 30 );  
  
// 1/3 + (1/7) i  
varcomplex c = 1.0vf/3.0vf + 1.0vf/7.0vf * VCI ;  
  
varfloat re = re( c ); // c の実部を取得  
varfloat im = im( c ); // c の虚部を取得  
println( re );  
println( im );
```

上のプログラムでは、VCSSL コンソールに 30 桁「0.33333...」、次に行に「0.14285...」と出力されます。上でVCIは可変精度の虚数単位で、complex型の場合のIに相当するものです。re 関数 / im 関数を varcomplex 型に対して使用した場合、実部 / 虚部を varfloat 型で取得できます。

算術演算

■ 算術演算子

プログラムは、コンピューターに様々な計算処理を行わせるためのものです。変数に対して計算処理を行うためには、算術演算子を記述します。VCSSL でサポートしている算術演算子には、以下の種類があります。

記号	意味	扱える型	詳細
+	加算 (足し算)	int, float, complex, varint, varfloat, varcomplex, string	数値を足す演算です。string 型に対しては例外で、文字列を接続する演算となります。
-	減算 (引き算)	int, float, complex, varint, varfloat, varcomplex	数値を引く演算です。
*	乗算 (掛け算)	- と同様	数値を掛ける演算です。
/	除算 (割り算)	- と同様	数値を割る演算です。
%	剰余算 (余り)	- と同様	数値を割った余りを求める演算です。
**	冪乗算 (指数)	float, varfloat	指数演算 (Math.pow 関数の糖衣構文) です。 ※ 使用するには Math ライブラリをインポートする必要があります。 ※ 複素数には使用できません。

※ 冪乗算は、VCSSL2.1 以前では「^」記号が割り当てられていましたが、この記号は他の C 言語系言語では排他的論理和 (XOR) を意味する事から、VCSSL2.1 以降では代わりに「**」記号の使用が推奨されるようになりました。

また現時点では、整数同士の指数演算は、少数型に変換されてから実行されるため、結果が小数になります。この事は混同を招くため、VCSSL3.0 以降で整数同士の指数演算は非推奨となり、小数型に明示的にキャストしてから演算する事が推奨されるようになりました。整数同士の指数演算は、当面の間は互換性が維持されますが、将来的には整数の結果を返すように仕様が修正される可能性があります。

■ 複合演算子と特殊な演算子

演算と同時に代入する(複合代入演算子)など、使用頻度の多い演算に対しては、以下のように特別な演算子が用意されています。この演算子は多くの C 言語系の言語においても採用されているものです。

略記号	意味	使用例
+=	左の値に、右の値を足して、それを左の値に代入する	a += b ; (a には a+b の結果が代入される)
-=	左の値に、右の値を引いて、それを左の値に代入する	a -= b ; (a には a-b の結果が代入される)
*=	左の値に、右の値を掛けて、それを左の値に代入する	a *= b ; (a には a*b の結果が代入される)
/=	左の値に、右の値を割って、それを左の値に代入する	a /= b ; (a には a/b の結果が代入される)
++	左か右の値に 1 を足して、それを左の値に代入する	a ++ ; または ++ a ; (a には a+1 の結果が代入される)
--	左か右の値に 1 を引いて、それを左の値に代入する	a -- ; または -- a ; (a には a-1 の結果が代入される)

最後の 2 つの演算子は、ループの制御で用いるため、使用頻度が非常に高くなります。このため、一般に「++」演算にはインクリメント、「--」演算にはデクリメントという呼称が付いています。

インクリメント(とデクリメント)は、変数の前にも後にも記述できます。前者を前置インクリメント、後者を後置インクリメントと呼びます。インクリメント結果を同じ式中で別の変数に代入するような場合、前置インクリメントでは加算後の値が、後置インクリメントでは加算前の値が代入されます。

■ 加算を行う

例として、加算を行ってみましょう。以下のように記述してみてください：

```
print ( 1 + 1 );
```

このプログラムを実行すると、VCSSL コンソールに「 2 」が表示されます。これを以下のようにする
とどうなるでしょう：

```
print ( 1.0 + 1.0 );
```

今度は、メッセージウィンドウに 2.0 が表示されます。

上の例では 2 と整数だったのが、下の例では 2.0 と小数になりました。これは、最初の演算の結果は整数値となり、後の演算の結果は小数値となった事を意味しています。このように、算術演算結果の型は、演算対象の値の型によって異なります。

具体的には、**整数同士の演算結果は整数になります。そして、小数同士の場合と、整数と小数の混合演算の場合は、小数型の値になります。**これを表にすると以下ようになります：

A と B の演算結果	A = 整数	A = 小数
B = 整数	A と B の演算結果 = 整数	A と B の演算結果 = 小数
B = 小数	A と B の演算結果 = 小数	A と B の演算結果 = 小数

さらに同じようにして、**複素数を含む混合演算は複素数になります。**つまり上の表に複素数を加えると、以下ようになります。

A と B の演算結果	A = 整数または小数	A = 複素数
B = 整数または小数	A と B の演算結果 = 上の表	A と B の演算結果 = 複素数
B = 複素数	A と B の演算結果 = 複素数	A と B の演算結果 = 複素数

■ (重要) 整数の除算には注意が必要

上の規則は、除算、つまり割り算の場合に、特に注意が必要です。

一般に整数同士の加減算と乗算の結果は、数学的にも整数なので、何も問題はありません。しかし整数同士の除算はそうではありません。にもかかわらず、整数同士の乗算結果は必ず整数として返されてしまうわけです。このために、慣れるまで厄介な落とし穴が生じます。

例えば、以下のようにしてみてください：

```
print ( 1 / 2 );
```

これを実行すると、0 が表示されます。数学的には 0.5 となるべきなのですが、これが整数に変換されて 0 となってしまったのです。このような振る舞いは VCSSL に限らず、様々な言語においても見られるものです。

正しい(小数の)答えを得るには、以下のように小数へキャストしてから演算してください。

```
print ( (float)1 / (float)2 );
```

これを実行すると、正しい答えである 0.5 が表示されます。変数ではなく値を直接演算する場合は「.0」を付けて小数にしても良いでしょう：

```
print ( 1.0 / 2.0 );
```

これも正しく 0.5 を表示します。しかし、変数に「.0」を付けてもキャストはされないので注意してください。

■ 演算順序の指定

複数の演算を行う場合、演算の順序が問題となります。例えば、以下のような場合を考えてみましょう：

```
int i = 1 + 2 * 3;  
print (i);
```

こういった場合、加算・減算よりも、乗算・除算のほうが先に計算されます。つまり、 $2*3=6$ の値が、1 に足されて、結果は 7 が出力されます。こういった場合、 $1+2$ を先に計算させたいなら、次のように () で囲みます：

```
int i = ( 1 + 2 ) * 3;  
print (i);
```

こうすれば、 $1+2=3$ が先に計算され、それに 3 が掛けられて、結果は 9 が出力されます。

■ 文字列の加算

string 型の場合は、他と仕様が大きく異なります。まず、string 型には加算以外の演算が行えません。また、string 型と他の型との加算は、演算対象をすべて string 型に変換した上で行われます。例えば、以下の例を実行してみてください。

```
print ( "1" + 2 );
```

これを実行すると、12 が表示されます。これは「 "1" という文字列と "2" という文字列を接続せよ 」という処理となったためです。

■ 算術演算子の優先度

異なる演算子が混在する式の場合、優先度の高い演算子から先に処理されます。優先度は、以下のように設定されています。

冪乗算 > 剰余算 = 除算 = 乗算 > 減算 = 加算

つまり乗算や除算は、加算や減算よりも優先度が高く、先に処理されます。なお、乗算と除算のように、同じ優先度の演算子が並んでいる場合は、左から順に処理されます。

比較演算

■ 比較演算子

プログラムでは、値の大小を比較して、その結果によって異なる処理に分岐させたい場合などが頻繁に生じます。このような処理を行うのが、比較演算子です。VCSSL には、以下のような比較演算子が用意されています。

記号	意味	扱える型	詳細
<	左より右が大きいか？	int, float, varint, varfloat,	演算記号の左右にある、2 つの値の大小関係を比較します。その結果は bool 型となります。
>	左より右が小さいか？	< と同様	< と同様
<=	左より右が大きいか、もしくは等しいか？	< と同様	< と同様
>=	左より右が小さいか、もしくは等しいか？	< と同様	< と同様
==	左と右は等しいか？	int, float, complex, varint, varfloat, varcomplex, string	演算記号の左右にある、2 つの値が一致するかを確かめます。その結果は bool 型となります。
!=	左と右は異なるか？	== と同様	演算記号の左右にある、2 つの値が異なるかを確かめます。その結果は bool 型となります。

■ bool 型変数

上の表にも述べられている通り、比較演算の結果は bool 型となります。bool 型は、「true (真)」と「false (偽)」の 2 つの値をとる変数型で、一般に真偽型などとも呼ばれます。真や偽というのは一般にあまりピンとこない言葉ですが、つまるところ真は「YES」で、偽は「NO」の事だと思っても差し支えないでしょう。

■ 比較を行う

それでは、実際に比較演算を行ってみましょう。ここでは、「左より右が大きいか？」を調べる記号「<」を使ってみましょう：

```
bool b ;  
b = 1 < 2 ;  
print ( b ) ;
```

これを実行すると、true が表示されます。true はつまり YES のことなので、「左より右が大きいか？」の答えとして YES が得られた事になります。これに対して、以下の例ではどうなるでしょう：

```
bool b ;  
b = 2 < 1 ;  
print ( b ) ;
```

これを実行すると、false が表示されます。false はつまり NO のことなので、「左より右が大きいか？」の答えとして NO が得られた事になります。

■ 一致を確かめる

続いて、値の一致を確かめてみましょう。ここでは、「左と右は等しいか？」を調べる記号「==」を使ってみましょう：

```
bool b ;  
b = ( 1 == 1 ) ;  
print( b ) ;
```

これを実行すると、true が表示されます。これに対して、以下の例ではどうなるでしょう：

```
bool b ;  
b = ( 1 == 2 ) ;  
print( b ) ;
```

この場合は false が表示されます。また、以下のようにして不一致を確認する事もできます：

```
bool b ;  
b = ( 1 != 2 ) ;  
print( b ) ;
```

この場合は true が表示されます。つまり「左と右は異なるか？」の結果として YES が得られたわけです。

ところで、こういった条件判定は、後に扱う否定演算子「！」を使用して記述する事もできます。否定演算子は、その後ろが真ならば偽を、偽ならば真を返します。例えば、「左と右は異なるか？」は「左と右は等しいか？」の逆なので、上のプログラムと同一の結果を返すものとして、以下のように記述する事も可能です。

```
bool b ;  
b = !( 1 == 2 ) ;  
print( b ) ;
```

この場合も同じ意味の条件となるので、true が表示されます。

論理演算

■ 論理演算子

複数の比較演算結果などの真偽値（論理値）が、同時に成立しているかなどを調べるためには、論理演算子を使用します。VCSSL では、以下のような論理演算子をサポートしています。

記号	意味	扱える型	詳細
&&	論理積 / AND (かつ)	bool	右の値と左の値が、ともに true である場合のみ、結果が true となります。どちらか一方が false なら、結果は false となります。
	論理和 / OR (または)	bool	右の値と左の値のうち、どちらか一方が true なら、結果は true となります。右の値と左の値が、ともに false である場合のみ、結果が false となります。
!	否定 / NOT (でない)	bool	右側の値が true なら、結果は false となります。また、右側の値が false なら、結果は true となります。

■ 2 つの比較結果を調べる

それでは、実際に 2 つの比較結果を調べてみましょう：

```
bool b1 ;  
b1 = 1 < 2 ;  
  
bool b2 ;  
b2 = 2 < 3 ;  
  
bool b3 ;  
b3 = b1 && b2 ;  
  
print ( b3 ) ;
```

これを実行すると true が表示されます。b1 と b2 はともに true なので、b1 && b2 も true となったのです。それでは、以下のようにするとどうでしょうか：

```
bool b1 ;  
b1 = 1 < 2 ;  
  
bool b2 ;  
b2 = 2 > 3 ;  
  
bool b3 ;  
b3 = b1 && b2 ;  
  
print ( b3 ) ;
```

今度は false が表示されます。b1 は true でしたが、b2 は false だったので、b1 && b2 は false となったのです。このように「&&」演算を用いると、複数の比較結果が、共に成立している場合のみ、true を得る事ができます。

■ 比較と論理演算の混合

複数の比較結果を、論理演算で判断するような処理は、プログラミングにおいて頻繁に登場します。そのたびに bool 型変数をいくつも用意するのは大変です。そこで、以下のように混合記述する事も可能です：

```
bool b ;  
b = ( 1 < 2 ) && ( 2 < 3 ) ;  
print ( b ) ;
```

これを実行すると true が表示されます。

■ 複数の論理演算の混合

複数の論理演算を組み合わせて処理するには、以下のように()を用いて区切る必要があります：

```
bool b ;  
b = ( ( 1 < 2 ) && ( 2 < 3 ) ) || ( ( 1 > 2 ) && ( 2 > 3 ) ) || ( 1 > 2 ) ;  
print ( b ) ;
```

これは (true && true) || (false && false) || false 、つまり true || false || false の演算となるので、実行すると true が表示されます。

■ 否定

比較結果の間逆を求めたい場合があるかもしれません。そのような場合には、否定の記号「!」を使用します：

```
bool b1 = true ;  
bool b2 = !( b1 ) ;  
print ( b2 ) ;
```

これを実行すると false が表示されます。

スコープとブロック

■ プログラムの長さと変数名

長いプログラムを書くと、変数を数百個も使用する事になります。そういった場合、変数名をいちいち数百個も用意するのは非常に面倒です。重要な変数にはきちんとした名前を付ける事が推奨されますが、一時的な雑用変数には、1 文字からせいぜい数文字程度と同じ変数名を使いまわす事も多いでしょう。しかしながら、同じ名前の変数がいくつも存在する場合、呼び出しに何のルールも無い状態では、いったいどの変数を呼び出しているのか不明瞭になります。VCSSL をはじめ一般的なプログラミング言語では、これを「スコープ」という概念で判断します。

■ 変数を呼び出せるスコープと、その範囲や優先度を区切るブロック

例えばあなたの家族が、家の中であなたの名前を呼ぶと、あなたは「自分が呼ばれた」とすぐに判断できます。しかし、日本中であなたと同じ名前の人はたくさんいます。それにもかかわらず、あなたが「自分が呼ばれた」と判断できたのは、そこが家の中だったからに違いありません。

この「家」と全く同様に、プログラム内において変数呼び出しの可能な範囲や優先順位を司る、「スコープとブロック」というものが存在します。

例えば、次のようなプログラムを実行してみてください：

```
int i = 2 ;
{
    int i = 3;
    print(i);
}
```

これを実行すると 3 が表示されます。上のプログラムにおいて、記号「 { 」と記号「 } 」で区切られた部分をブロックと呼びます。上では print 関数で変数 i を呼び出していますが、ブロックの内側にある変数 i の値が呼び出された事がわかります。これは、家の中で名前を呼ぶと、家の中の人が答えてくれるのと同じです。例えるならば「ブロックでそこに家を建てる」わけです。

このように、ブロックは変数を呼び出す優先度を上げる効果があります。ブロックが何重にも階層的になっている場合でも同様です。その場合、適当な位置で変数を呼び出すと、まず呼び出し地点に最も近いブロック内から探されます。ブロック内に呼び出したい名前の変数が存在しない場合は、

次にブロックの外側が探されます。

続いて、スコープについてです。スコープとは簡単に言うと、変数を呼び出し可能な範囲の事です。上の例のように、ブロックの中で宣言された変数のスコープは、そのブロックの内側に限定されます。「家の外から家の中にいる人を呼んでも、気づいてもらえない」というわけです。実際に試してみましょう。次のようなプログラムを実行してみてください：

```
{  
    int i = 3 ;  
}  
print( i );
```

この場合、エラーが表示され、プログラムが強制終了されます。変数のスコープの外側から変数を呼び出す事は決してできません。しかしながら、ブロックは、内側にある変数のスコープが「外に漏れないように」区切って制限しますが、外側にある変数のスコープについては何も制限しません。つまり、ブロックの内側から、外側にある変数を呼び出す事は可能です。実際に試してみましょう：

```
int i = 2 ;  
{  
    print( i );  
}
```

この場合、確かにブロックの外側の `i` が呼ばれ、`2` が表示されます。

■ グローバル変数とローカル変数

何らかのブロック内で宣言された変数の事を、ローカル変数と呼びます。これに対し、すべてのブロックの外側で宣言されている変数は、グローバル変数と呼びます。ローカル変数は自分の属するブロック内からしか呼び出せず、ブロック外では存在しないのと同じです。いわば、その場だけの使い捨ての変数です。これに対してグローバル変数はどこからでも呼び出せるので非常に便利ですが、使いすぎてグローバル変数の量が多くなりすぎると、プログラムが読み辛くなってしまうという欠点もあります。

制御構文

■ 制御構文（制御文）

プログラム中の特定の部分を、一定の回数だけ繰り返し処理したり、ある条件が成立した場合のみ処理したりする場合には、制御構文を使用します。VCSSL でサポートしている制御構文は、必ずブロックとセットになっており（一行でも省略不可）、以下のような文法を持ちます：

```
制御構文型( 制御式 ){  
    処理内容 ;  
}
```

VCSSL でサポートしている制御構文には、以下のものがあります：

制御構文型	制御式の内容	詳細
if	条件式を記述する	条件式の結果が true の場合のみ、ブロックの中身を実行します。
if - else	条件式を記述する	複数の条件式を使用し、成立した条件式に対応するブロックの中身を実行します。
while	Ifと同じ	条件式の結果が true の間、ブロックの中身を繰り返し実行し続けます。
for	セミコロン「;」記号で挟んで、初期化式・条件式・反復式を記述する	while にカウンターが付いたものです。最初に初期化が呼ばれ、その後は条件式の結果が true の間、ブロックの中身が繰り返し実行されます。その際、繰り返し式が毎回呼ばれます。

■ if 制御構文 (if 文)

if 制御構文を使ってみましょう。以下のように記述して、実行してみてください：

```
if( 1 < 2 ){  
    print( " Hello " );  
}
```

これを実行すると、「 Hello 」と表示されます。つまり if 制御構文のブロック内にある、print 関数が実行された事が分かります。これを以下のように書き換えるとどうなるでしょうか：

```
if( 1 > 2 ){  
    print( " Hello " );  
}
```

今度は、何も表示されずに終了します。つまり if 制御構文は、条件式 (ここでは $1 < 2$) の結果が true の場合のみ、ブロックの中身が実行されるのです。なお、条件式の部分に bool 型変数を記述する事も可能です。その場合、bool 型変数の値が true の場合のみ、ブロックの中身が実行されます。

■ if - else 制御構文 (if - else 文)

if - else 制御構文は、if 制御構文に複数の条件式を持たせたもので、複雑な分岐を行う場合に使用します。「else」というのは「それ以外の、その他の」といった意味です。この制御構文は、ある条件式が不成立であった場合に、別の条件式による判定を何個も続けて行います。

実際に if - else 制御構文を使ってみましょう。以下のように記述して、実行してみてください：

```
float f = 2.5;  
  
if( f < 1 ){  
    print( " f < 1 " );  
}
```

```
}else if( f < 2 ){
    print( " f < 2 " );
}else if( f < 3 ){
    print( " f < 3 " );
}else{
    print( " 3 <= f " ); // ここはどの条件にも一致しなかった場合に実行される
}
```

これを実行すると、「 f < 3 」と表示されます。このように、if 制御構文の下に else 制御構文を連ねて記述すると、上から下へと順に条件式が判定されていきます。そして、条件式が成立した所で、そのブロック内の処理が実行されます。それ以降に続く条件式は判定されず、実行もされません。

■ while 制御構文 (while 文)

while 制御構文を使ってみましょう。以下のように記述して、実行してみてください：

```
int i = 0;
while( i <= 10 ){
    println( i );
    i = i+1 ;
}
```

これを実行すると、入出力端末に 0 から 10 までの数値が出力されます。

つまり while 制御構文は、条件式 (ここでは $i \leq 10$) の結果が true の間、ブロックの中身が繰り返し実行されるのです。

■ for 制御構文 (for 文)

上の while 制御構文の説明で用いた、一定の回数だけ繰り返す処理は、プログラミングにおいて非常に頻繁に使用されます。しかし、上で用いたような記述は少し面倒です。これをもっと短く記述するために、for 制御構文が用意されています。例として、while 制御構文の説明と全く同じ処理を、for 制御構文で記述してみましょう：

```
for( int i=0; i<=10; i=i+1 ){  
    println( i );  
}
```

このように、非常に簡潔に繰り返し処理を記述する事ができます。for 制御構文の制御式には、セミコロン記号「 ; 」で区切って、3 つの式を記述します。これをそれぞれ初期化式、条件式、反復式と呼びます。つまり上の例では：

初期化式： int i=0

条件式： i<=10

反復式： i=i+1

でした。for 制御構文では、最初に初期化式が実行され、その後は条件式の結果が true の間、スコープの中身が繰り返し実行されます。その際、ブロックの中身が実行し終わる度に、反復式が毎回実行されます。

初期化式には、繰り返しをカウントする変数を宣言するのが一般的です。このような変数は一般にループカウンタと呼ばれます。初期化式で宣言された変数は、スコープがそのブロック内に限定されたローカル変数となります。

■ 制御構文の組み合わせ

制御構文は、いくつも組み合わせて使用する事も多々あります。特に、if 制御構文は繰り返しの中で使用される事が多いでしょう。例として、偶数を出力するプログラムを書いてみましょう：

```
int max = input( "偶数を探す上限値を入力してください。" );  
for( int i=0; i<=max; i=i+1 ){  
    if( (i%2)==0 ){  
        println( i );  
    }  
}
```

上の例では、2 で割った余りが 0 の整数、つまり偶数だけが出力されます。

配列

■ 配列

複数の値を一つの変数名で表現したい場合には、配列を使用します。配列を宣言するには、以下のように記述します：

```
変数型 配列名[ 要素数 ] ;
```

ここで要素数は、配列に格納できる値の個数を指定します。なお、以下のように変数型の部分に要素数を付ける記述も可能です。

```
変数型[ 要素数 ] 配列名 ;
```

これら 2 通りの宣言方法は、どちらも全く同じ処理となります。

例として、int 型の値を格納する配列を用意してみましょう：

```
int a[5] ;
```

または、

```
int[5] a ;
```

とします。これで 5 つの整数値を格納できる配列が用意できました。

配列に値を代入したり、呼び出したりするには、変数名の右に [インデックス番号] をつけて、普通の変数と同じように行う事が可能です：

```
int a[5];  
a[3] = 1;  
print ( a[3] );
```

これを実行すると 1 が表示されます。つまり a の中の、3 番目の要素に値を格納したわけです。

インデックス番号は、配列の要素に割り振られる番号で、0 番から(要素数-1)番までが用意されています。要素数番まででは無い事に注意してください。例えば、上の例では 0,1,2,3,4 番までが用意されており、5 番は存在しません。

配列はその用途から、繰り返しの制御構文と併用される事が多いでしょう。例として、0 から 10 までの整数の 2 乗を、配列に格納してみましょう：

```
int a[11];  
for( int i=0; i<=10; i=i+1 ){  
    a[i] = i * i;  
}  
print ( a[8] );
```

これを実行すると 64 が表示されます。

■ 配列同士の代入

配列のインデックス番号を指定せずに、配列名のみを用いて代入を行った場合は、配列の全要素が代入されます。

```
int a[10];  
int b[10];  
a = 1 ; // a の全要素に 1 を代入  
b = 2 ; // b の全要素に 2 を代入  
a = b ; // a の全要素に b の全要素を代入  
print( a[ 5 ] );
```

このプログラムを実行すると 2 が表示されます。

要素数の異なる配列を代入する場合には、少し注意が必要です。そういった場合、代入先の配列の要素数が変更され、代入したものと同じになります。

```
int a[10];
int b[20];
a = 1; // a の全要素に 1 を代入
b = 2; // b の全要素に 2 を代入
a = b; // a の全要素に b の全要素を代入
print( a[ 15 ] );
```

このプログラムを実行すると、結果は「 2 」が表示されます。そして、a の要素数は 20 になります。

ところで、多くの C 言語系の他言語では、配列は参照型であり、代入先と代入元の変数は、同一の変数のように振舞うようになります(同じデータ領域を参照するようになる)。この場合、代入後に行った、代入先の配列に対する変更が、代入元の配列にも反映されます。

一方、VCSSL では配列も値型であり、代入は単純に、全要素値のコピーとなります。同じデータ領域を参照するようにはなりません。従って、代入後に、代入先の配列に対して変更を加えても、それは代入元の配列に反映されません。

VCSSL は基本的に、C 系の他言語となるべく同じ要領で扱えるようになっていますが、この違いについては注意が必要です。

■ 多次元配列

配列は、インデックスを複数付ける事もできます。これを多次元配列と呼びます。多次元配列の宣言・使用法は 1 次元の場合と基本的に同じで、[] を複数付けるだけです：

```
int a[ 11 ][ 11 ];
for( int i=0; i<=10; i=i+1 ){
    for( int j=0; j<=10; j=j+1 ){
        a[ i ][ j ] = i * j;
    }
}
```



```
}  
print ( a[ 2 ][ 8 ] );
```

これを実行すると 16 が表示されます。上の例の a は 2 次元の配列ですが、[] の数を増やすだけで何次元でも利用可能です。

■ 配列の要素数を取得する

プログラムの実行中に、配列の要素数を調べたい場合はよくあります。そういう場合には、length 関数を使用します。length 関数は、配列の指定された次元の要素数を調べて、結果を int 型で返します。最初の引数に配列変数を、続く引数に次元を指定します。次元の指定は、配列の左から 0 次元目, 1 次元目, 2 次元目, ...と数えます。

```
int a[ 1 ][ 2 ][ 3 ];  
int i = length( a, 2 );  
print ( i );
```

これを実行すると 3 が表示されます。これは a の左端から数えた 2 番目(0, 1, 2 と数える)、つまり右端の次元の要素数です。同様に length(a, 0)は 1 を、length(a, 1)は 2 を返します

なお、array 関数で 2 つめの引数を省略すると、各次元の要素数をまとめた配列が返されます。上の例では、length(a) とすると、要素数 3 の int 配列が返されます。しかし、この機能は古い互換性維持のためのもので、配列の次元を右から 0, 1, 2 と数える仕様となっているため、混同を招くという問題があります。

従って、length 関数で 2 つめの引数を省略するのは、現在では非推奨となっています。

■ 配列の要素数を変更する

配列の要素数を変更したい場合は、alloc 演算子を使用します。この演算子は使用方法が特殊で、以下のように使用します：

```
alloc[要素数] 配列変数 ; // 1 次元の場合  
alloc[要素数][要素数]...[要素数] 配列変数 // 多次元の場合
```

このように記述すると、配列変数の要素数が、alloc の後の[] 内に記述した要素数に変わります。具体的に使用してみましょう。以下のように記述し、実行してみてください：

```
int a[ 10 ] ;  
alloc[ 20 ] a ; // a の要素数を [ 20 ] に変更  
int i = length( a, 0 ) ;  
print ( i ) ;
```

これを実行すると 20 が表示されます。このように、alloc 演算子によって、配列変数 a の要素数を、10 から 20 に変更できました。

なお、alloc の前後では、配列の各要素の値は維持されます。つまり、もし上で a[1] に最初 255 が入っていたとすると、alloc 後の a[1] も 255 のままです。多次元の場合でも同様に、同じインデックスに対応する要素は、同じ値に保たれます。

■ 要素数 1 の配列を、配列でない変数に代入する

配列を、配列でない普通の変数に代入する事は、一般にはできません。しかし例外があります。それは配列の要素数が 1 の場合です。要素数が 1 の配列は、データの的には普通の変数と変わらないので、普通の変数へ代入する事ができます：

```
int i[1] ;  
i[0] = 1 ;  
int j = i ;  
print ( j ) ;
```

上の例を実行すると 1 が表示されます。

■ 配列の全要素に対する代入と演算

配列の全要素に同じ値を代入したい場合、

```
配列 = 値 ;
```

といった処理が可能です。例えば、

```
int i[10] ;  
i = 1 ;  
print( i[5] );
```

これで i の全要素に 1 が代入されます。

また、配列の全要素に、同じ値で同じ演算を行いたい場合は、以下のような略記も可能です：

```
int i[10] ;  
i = 1 ;  
i = i + 2 ;  
print( i[5] );
```

上の処理では、 i の全要素に 2 が加算され、3 となります。

ベクトル演算

■ ベクトル演算

ベクトル演算とは、大量の値に対して、同じ処理規則を適用するような演算のことです。これに対し、1 つずつの値に対する演算をスカラ演算と呼びます。VCSSL では、配列変数同士の演算において、全要素に同じ演算を行うベクトル演算をサポートしています。

ベクトル演算を行うには、原則として、式の中にある全配列変数の次元と要素数が等しい必要があります。ただし代入(=)では要素数が異なっても構いません。また、式には配列でない変数を含ませる事も可能で、その場合、配列でない変数は、要素が全て等しい配列変数と同様に扱われます。

■ ベクトル演算子

VCSSL では、以下のようなベクトル演算子がサポートされています：

記号	意味	扱える配列型	詳細
=	ベクトル代入算	全ての型	右の配列の中身を、左の配列に代入します。 代入後の左辺の要素数は、右辺と同じになります。
+	ベクトル加算	int, float, complex, varint, varfloat, varcomplex, string	右の配列の中身と、左の配列の中身を、同じインデックスのもの同士で全て加算します。 配列の次元と要素数が異なる場合には使用できません。
-	ベクトル減算	int, float, complex, varint, varfloat, varcomplex	ベクトル加算が減算になったものです。
*	ベクトル乗算	- と同様	ベクトル加算が乗算になったものです。
/	ベクトル除算	- と同様	ベクトル加算が除算になったものです。
%	ベクトル剰余算	- と同様	ベクトル加算が剰余算になったものです。
**	ベクトル冪乗算	- と同様	ベクトル加算が冪乗算になったものです。

■ ベクトル加算を行う

実際に、ベクトル加算を行ってみましょう：

```
int a[11];
int b[11];

for( int i=0; i<10; i=i+1 ){

    a[i] = i * i ;
    b[i] = i ;

}

int c[11];
c = a + b ;

print ( c[5] );
```

これを実行すると 30 が表示されます。少し長いプログラムなので、処理内容を追ってみましょう。

まず、要素数が 11 個の配列 a,b が用意され、そして a の中には 1 から 10 までの整数の 2 乗が代入され、b の中には 1 から 10 までの整数がそのまま代入されます。続いて、要素数が 11 個の配列 c が用意されます。

続いて「 c = a + b 」というベクトル加算が実行され、これにより、c の中身には、

```
c[0] = a[0] + b[0] ;
c[1] = a[1] + b[1] ;
...
c[10] = a[10] + b[10] ;
```

という内容が代入されます。そして c[5]を出力する関数が呼ばれ、プログラムが終了します。c[5]には 5*5+5 が入っているはずですから、確かに 30 という値は正しいものです。つまり、正常にベクトル演算が実行できた事がわかります。

関数

■ 関数

例えば、ある適当な数 a, b に対して、「 $a*a+b*b$ 」といった値をプログラム中のいくつかの箇所で行いたい場合があったとします。このような場合、必要な箇所すべてに「 $a*a+b*b$ 」を書いていてもいいのですが、それは少し面倒です。特に、あとでやはり「 $a*a+b*b-2$ 」にしたい、等といった事が生じると、いくつかの箇所を書き直す事になってしまいます。

このような場合は、あらかじめ関数を定義しておくと便利です。関数は、変数の組を受け取って、それに対する処理を行い、結果を出力するための仕組みです。関数は、以下のような文法で定義されています：

```
戻り値の変数型 関数名( 引数 1, 引数 2, ... ){
    処理内容 ;
    return 戻り値 ;
}
```

ここで引数とは、関数が処理に使用するために受け取る変数のことです。恐らく「引き受ける数」などから省略された用語なので、「ひきすう」と発音します。

また、戻り値とは、関数が処理結果として出力する変数の事です。恐らく「関数から戻ってくる値」などから省略された用語です。

■ $a*a+b*b$ を処理する関数の例

例として、ある適当な整数 a, b に対して、「 $a*a+b*b$ 」といった値を返す関数を定義してみましょう：

```
int fun( int a, int b ){
    int value = a * a + b * b ;
    return value ;
}
```

この関数は、int 型の a,b を受け取り、内部で value という値に $a*a+b*b$ を計算して代入し、それを戻り値として返します。

それでは、実際にこの関数を、プログラム中から呼び出してみましょう。プログラム中から関数を呼び出すには、そこに「関数名の後にカッコを付けたもの」を記述します。そしてそのカッコの中に、引数をカンマ記号「 , 」で区切って指定します。

以下のように記述し、実行してみてください：

```
int fun( int a, int b ){  
    int value = a * a + b * b ;  
    return value ;  
}  
  
int i;  
i = fun( 1, 2 );  
print( i );
```

これを実行すると 5 が表示されます。関数 fun の中で正しく $1*1+2*2$ が実行され、戻り値として返ってきた事が分かります。

■ 何も返さない関数

関数は、必ずしも戻り値を返す必要はありません。戻り値を返さない場合は、「 void 型 」で関数を宣言します。void とは英語で空 (から) の意味で、void 型はその名の通り、戻り値が空である事を意味する、特別な型です。

```
void fun( int a ){  
    ...  
}
```

■ 引数を持たない関数

引数についても、何も持たない事が可能です。その場合は、引数記述部を空白にします。

```
int fun(){  
    ...  
}
```

■ 配列を引数とする関数

関数の引数には、配列を指定する事もできます。それには以下のように記述します：

```
int fun( int a[ ], int b[ ] ){  
    ...  
}
```

または、変数型の部分にブラケットを付けた、以下のような記述も可能です。

```
int fun( int[ ] a, int[ ] b ){  
    ...  
}
```

これで、int 型の配列 a,b を引数に受け取れます。配列の要素数は、プログラム中から呼び出し時に入力された引数のものとなります。必要であれば、length(array[], int dim) 関数で要素数を確認する事もできます。

ところで、上のように配列を引数に渡した場合、多くの C 言語系の他言語では配列が参照型であるため、参照渡し(後述)のような挙動(参照の値渡し)となりますが、VCSSL では単純な値渡しとなります。つまり、配列の中身が全てコピーして引数に渡され、関数内で引数の値を変更しても、それが呼び出し元には反映されません。この点については、一般的な他言語と異なるので注意が必要です。

■ 配列を戻り値とする関数

VCSSL の関数は、配列を戻り値に返す事もできます。それには以下のように、関数の型宣言の部分に [] を付けて宣言します。

```
int[ ] fun( int a, int b ){  
    ...  
}
```

これで、戻り値に配列を返す事ができます。プログラム中からこの関数を呼び出し、戻り値を配列に代入すると、その配列の要素数は戻り値と同じものに変更されます。

■ 引数の値渡しと参照渡し

関数の引数は、これまでの例のように普通に定義すると、呼び出し時に値がコピー（代入）される「値渡し」となります。つまり、関数内で引数の値を変更しても、それが呼び出し元には反映されません。それに対して、変数名の前にアンパサイド記号「 & 」 を付けて定義すると、呼び出し元にも変更が反映される「参照渡し」となります：

```
// 引数を「値渡し」で受け取る関数  
int callByValue( int a ) {  
    a = 1 ;  
}  
  
// 引数を「参照渡し」で受け取る関数  
void callByRef( int &a ) {  
    a = 1 ;  
}  
  
int i = 0 ;  
int j = 0 ;  
callByValue( i ) ; // i を値渡しして関数を呼び出す(関数内での変更は反映されない)  
callByRef( j ) ;   // j を参照渡しして関数を呼び出す(関数内での変更は反映される)  
print( "i=" + i + " j=" + j ) ;
```

実行するとコンソールに「 i=0 j=1 」と表示されます。参照渡しで j の変更が反映されました。

■ 配列の参照渡し

配列をまるごと参照渡しする事もできます。その場合は、以下のように記述します。

```
int fun( int (&a)[ ] ){  
    ...  
}
```

このように、引数名とアンパサイド記号「 & 」 をまとめて () でくくります。ただし参照渡しには、後で述べるように、いくつかの注意点や制限、デメリットなどがあります。それについては配列の場合でも同様です。

■ 参照渡しの注意点

VCSSL における参照渡しの機能はまだ新しく、処理系の実装に依存している不完全な挙動があります。この点に関しては、使用の際に注意が必要です。

・配列要素の参照渡しには特に注意が必要

まず注意が必要なのは、配列の要素を参照渡しする場合です。例えば、参照渡しで引数を受け取る関数 fun があつたとして、fun(a[i]) ように配列の要素を参照渡しした場合、関数の処理が終わるまで、関数外においてインデックス i の値を変更してはいけません。つまり、fun(a[i]) をコールした時点で i が 2 であつたものを、処理途中に外部から i を 3 に切り替えるような事はしてはいけません。実際にこのような処理を行った場合、fun 内部での(参照渡しで受け取った)引数が、ずっと a[2] を参照し続けるのか、もしくは i の変更後から a[3] を参照するよう切り替わるのかは、現時点で処理系依存です。現在の VCSSL エンジンの実装では後者の挙動が中心です。しかし他言語では前者の挙動が一般的であるため、VCSSL エンジンでも今後は前者の挙動に切り替わっていく可能性があります。

上述のような、配列要素の参照渡しについては、構造体やそのメンバに関しても同様に注意が必要です。例えば参照渡しで引数を受け取る関数 fun に、fun(a[i].b[j].c[k]) のように引数を渡した場合、その関数の処理が終わるまで、i も j も k も、値を変更してはいけません。

最後に、配列要素を参照渡しする場合、その関数の処理が終わるまでは、alloc など配列を再

確保してはいけません。

以上をまとめると、参照渡しについては、渡す際に含まれる変数を（それが配列のインデックスや、渡すメンバ変数を保持する構造体変数であっても）、関数の処理が終わるまで、外部から一切変更しないよう、注意する必要があります。

・参照渡しができない場合

参照渡しは、常に行えるわけではなく、できない場合もあります。例えば別の関数の戻り値を、参照渡しで直接受け取る事はできません。そういった場合は、関数の戻り値を一度別の変数に格納してから、その変数を参照渡しする必要があります。

・参照渡しのデメリット

参照渡しで受け取った引数に対する演算は、処理効率が悪く、一般に処理速度が低下します。引数に渡す変数のメモリーサイズが大きい場合は、値渡しよりも参照渡しのほうが、データのコピーが不要になる分、軽くなる場合がありますが、そうでない場合、VCSSL における参照渡しはパフォーマンス面で不利になります。

また、普通に値渡し引数を受け取って、結果を戻り値で返せば済むような場面などで、あまり頻繁に参照渡しを用いると、プログラム全体の可読性を低下させる要因になり得ます。

main 関数

■ main 関数とは

通常の関数は、プログラムの中で明示的に呼び出さない限り、実行される事はありません。しかし、システム側から特定のタイミングで自動的に呼び出され、実行される特別な関数がいくつか存在します。その一つが、main 関数です。main 関数は、実用的なプログラムを開発する際において重要な役割を担います。例えば、コマンド入出力端末から VCSSL プログラムを実行する際、コマンドのパラメータを、main 関数の引数として受け取る事ができます。

例えば、任意のファイルを読み込んで処理するようなプログラムを、GUIではなくコマンドライン用を開発する場合、開くファイルパスはコマンドライン引数として受け取るようにするのが一般的でしょう。このようなコマンドライン引数は、VCSSL プログラムをコマンドラインから(vcssl コマンドで)起動する際、main 関数の引数に渡されます。つまりこのような場合、main 関数は必須となります。

また別の例として、GUI やグラフィックスを用いた実用的なプログラムの開発においても、メインループ(画面更新や、ユーザーからの操作に対応するための無限ループ)をグローバル領域の中には直接書かず、なるべく main 関数の中か、そこから呼ばれる関数の中に記述する事が推奨されます。なぜなら、メインループがグローバル領域に存在すると、そこより下はいつまでたっても処理されないため、それ以降で宣言されたグローバル変数が初期化されないという不具合が生じるからです。

■ main 関数の呼び出されるタイミング

main 関数は、関数外領域の処理が全て完了した後に、システムから自動的に呼び出されます。従って、関数外領域で変数の宣言や初期化などを記述した場合、main 関数が呼ばれた時点では、それらは全て初期化が完了した状態となっています。

■ main 関数の文法

main 関数は、以下のような文法で定義されています：

```
void main ( string args[ ] ) {  
    処理内容 ;  
}
```

この形の関数を記述しておく、システム側から自動で呼び出されます。その際、引数 `args` にはシステム側からの情報が渡されます。この情報は一般の場合、何も入っていません。この情報は一般の場合、何も入っていません。しかし、例えばプログラムをコマンド入出力端末から起動した場合などは、起動時のパラメータなどが格納されています。

■ main 関数の使用

実際に、`main` 関数を使用してみましょう。以下のように記述し、実行してみてください。

```
int FACTRIAL_ARGS = 5;

void main ( string args[ ] ) {
    int value = 1;
    for( int i=FACTRIAL_ARGS; 0<i; i-- ){
        value *= i;
    }
    println( value );
}
```

このプログラムは、変数 `FACTRIAL_ARGS` にセットした値の階乗を求めます。`main` 関数をプログラム中のどこからも呼び出していないにもかかわらず、自動で実行されます。

■ main 関数と移植性・可読性

スクリプト言語の `VCSSL` では、`main` 関数を使用しなくても、関数外領域で特に制限なく処理を記述する事が可能です。本文書のサンプルコードでもそうしてきました。しかし他の言語、特に `C` 系のコンパイラ型言語では、関数外領域で宣言以外の処理を行う事を禁止しているものも多くあります。従って `VCSSL` でも `main` 関数を使用すれば、そのような言語への移植性が向上します。

なお、関数外領域で宣言以外の処理も記述できるのは、比較的短いプログラムでは手軽で便利です。しかし、ある程度長くて関数も多いプログラムでは、関数外領域に宣言以外の処理が混在していると、処理の流れが読みづらくなります。そのような場合、関数外領域で行うのは宣言のみに限定しておいて、それ以外の(手続き的な)処理を `main` 関数にまとめると読みやすくなります。

システム関数と定数

■ システム関数

システム関数は、ユーザーが定義する関数とは異なり、VCSSL を処理するシステムによって、標準でサポートされている関数です。

これまで扱ってきた内容でも、システム関数を何度も使っていました。というのも、結果の出力に使用していた `print` 関数なども、全てシステム関数です。つまり、VCSSL を処理するシステムと通信し、そのシステムの機能により画面出力を行っていたのです。

■ 代表的なシステム関数

よく使用する代表的なシステム関数は以下の通りです：

システム関数	引数	詳細
<code>print</code>	任意型、任意個数	引数の内容を VCSSL コンソールに表示します。
<code>println</code>	任意型、任意個数	引数の内容を VCSSL コンソールに表示し、改行します。
<code>output</code>	string 型のメッセージ	引数の内容を、結果出力用の端末に表示します。端末が存在しない場合、 <code>println</code> 関数と同様の動作をします。
<code>pop</code>	任意型、任意個数	引数の内容をポップアップウィンドウに表示します。
<code>input</code>	string 型のメッセージ	入力ウィンドウを表示し、ユーザーに数値の入力を要求し、結果を string 型変数で返します。
<code>confirm</code>	string 型のメッセージ	ユーザーに YES/NO の選択を促します。結果を bool 型で返します。
<code>select</code>	string 型の選択肢 1, string 型の選択肢 2, … (任意個数)	引数の中から 1 個を選択するようユーザーに要求します。結果を string 型で返します。
<code>beep</code>	なし、または int 型の回数、 または int 型の回数と間隔	アラーム音を鳴らします。引数が 1 個の場合は回数、2 個の場合は回数と間隔(ミリ秒)とみなされます。
<code>choose</code>	string 型のメッセージ	ファイルまたはフォルダの選択をユーザーに要求し、ファイルの絶対パスを string 型で返します。
<code>sleep</code>	int 型の停止時間	プログラムを一定の時間だけ停止します。引数には、停止時間をミリ秒単位で指定してください。

exit	なし	プログラムの実行を終了します。
reset	string 型のプログラム名, string 配列型の main 引数	現在実行中のプログラムを終了し、別のプログラムを実行します。一つめの引数には、別のプログラムの名称または相対パスを、二つめの引数にはメイン関数コールの引数を指定してください。
eval	string 型の式内容	string 型変数の中身を式と見なし、処理した結果を string 型配列で返します。
evaltr	string 型の式内容	eval 関数の処理過程を、string 型変数で返します。 ※主にデバッグ用であり、内容は処理システムに依存します。
override	string 型の関数名, string 型配列の引数型名, string 型の関数内容	関数の処理内容を、プログラム実行中に変更します。 2 つめの引数には、"int"や"float"など、引数の型を保持する string 型配列を指定してください。 3 つめの引数には、新しい関数の処理内容を、VCSSL プログラムの文法で記述した string 変数を指定してください。
hide ※	なし	VCSSL コンソールを隠します(不可視化)。
show	なし	VCSSL コンソールを出現させます(可視化)。
clear	なし	コンソールの内容を全てクリアします。
save	string 型のファイルパス, string 型の内容(省略可)	2 つめの引数の内容を、ファイルに出力します。内容の指定を省略した場合、コンソールの内容をファイルに出力します。
load	なし、または string 型のファイルパス	指定されたファイルの内容を読み込み、string 型で返します。ファイルパスの指定を省略した場合、コンソールの内容を返します。
re	vomplex 型または varcomplex 型の、 実部を知りたい変数	複素数の実部を返します。
im	vomplex 型または varcomplex 型の、 虚部を知りたい変数	複素数の虚部を返します。

※ hide 関数を使用すると、VCSSL コンソールを閉じる事ができなくなり、そのままではユーザーが手動でプログラムを終了させる手段が無くなるため、設計上の注意が必要です。GUI を用いたプログラム等で hide 関数を使用する場合は、メインウィンドウを閉じた際に exit 関数をコールさせるなどして(onWindowClose イベントハンドラを用います)、プログラム側から自身を終了させるように設計してください。

■ システム定数

システム関数と同様、システムによってあらかじめ定義されている定数値をシステム定数と呼びます。システム定数には、機種依存の改行コードや、整数・小数の最大最小値など、不可欠な値が定義されています。

■ 代表的なシステム定数

よく使用する代表的なシステム定数は以下の通りです：

システム定数	型	詳細
NULL	特別な型	無効な値です。文字列や構造体などは無効値を保持する事があり、その場合に NULL と比較すると true が得られます。代入する事も可能です。 なお、int 型との比較・代入に関しては、特例的に -1 と等価な振る舞いをします。
EOL	string 型	環境で使用されている標準的な改行コードです。
LF	string 型	環境依存の改行コード LF (0x0A) です。
CR	string 型	環境依存の改行コード CR (0x0D) です。
I	complex 型	complex 型の虚数単位(0.0,1.0)です。
VCI	varcomplex 型	varcomplex 型の虚数単位(0.0,1.0)です。
INT_MAX	int 型	int 型の最大値です。
INT_MIN	int 型	int 型の最小値です。
FLOAT_MAX	float 型	float 型の最大値です。
FLOAT_MIN	float 型	float 型の最小値です。

標準ファイル入出力

■ 標準ファイル出力

ここでは、単純なファイル入出力を扱うためのシステム関数を扱います。例として、「world.txt」というファイルに文字列を書き込んでみましょう：

```
int file = open( "world.txt", "w" );    // ファイルをテキスト書き込みモードで開く
write( file, " Hello world !" );    // ファイルに書き込む
close( file );                        // ファイルを閉じる
```

このプログラムを実行すると「world.txt」というファイルが生成され、中に「 Hello world ! 」と記述されているはずです。

上の open 関数では、最初の引数にファイル名 "world.txt" を、次の引数にテキスト書き込みモード "w" を指定しています。モードには、他にも様々なものがあります。

open 関数は、開いたファイルに固有の識別番号「ファイル番号」を割り当てて返します。上ではそれを int 型変数 file で受け取っています。

ファイルへの書き込み/読み込み処理や、ファイルを閉じる場合には、上の例のように、対象ファイルのファイル番号を指定します。このように、ファイル番号で対象を識別する仕組みにより、複数のファイルを同時に操作する事ができます。

■ 標準ファイル入力

今度は、先のファイルを読み込んでみましょう：

```
int file = open( "world.txt", "r" );    // ファイルをテキスト読み込みモードで開く
string text = read( file );    // ファイルから読み込む
close( file );                // ファイルを閉じる
print( text ); // 読み込み内容を表示
```

上の open 関数の引数 "r" はテキスト読み込みモードを意味します。read 関数は、読み込んだファ

イルの内容を返します。read 関数の戻り値は、一般には string 型配列ですが、テキストモード ("r") モードでは要素数 1 の配列を返すので、非配列の変数 text で受け取れます。

ただしバイナリモード ("rb") ではバイト区切り、TSV モード ("rtsv") では空白・タブ区切り、CSV モード ("rcsv") ではカンマ区切りの配列で返されるため、配列変数で受け取る必要があります。

■ 1 行ごとの読み込み

ファイルを 1 行ごとに読み込むには、以下のようにします：

```
int file = open( "world.txt", "r" ); // ファイルをテキスト読み込みモードで開く
int n = countln( file ); // ファイル行数を取得
string line ;
for( int i=0; i<n; i++ ){
    line = readln( file ); // ファイルから一行読み込む
    println( i + "行目=" + line ); // 読み込み内容を表示
}
close( file ); // ファイルを閉じる
```

このように、まず countln 関数でファイル行数をカウントし、そして readln 関数で 1 行ずつ読み込みます。先の read 関数はファイル全体を一度に読み込むためのものでしたが、readln 関数は 1 行単位で読み込むためのもので、それ以外は同じです。

■ 標準ファイル入出力関数の種類

システム関数に用意されている標準ファイル入出力関数には、以下のものが存在します：

関数	引数	詳細
open	string ファイル名, string モード	<p>指定されたファイル名のファイルを、指定されたモードで開き、ファイル番号を割り振って返します。</p> <p>モードには、以下のようなものがあります。"w" のように文字列リテラルで指定する他に、WRITE のようにシステム定数を用いて指定する事もできます。</p> <p>- 書き込みモード -</p> <p>"w" テキスト書き込みモード (定数 WRITE)</p> <p>"a" 追記テキスト書き込みモード (定数 APPEND)</p> <p>"wtsv" TSV 書き込みモード (定数 WRITE_TSV)</p> <p>"wcsv" CSV 書き込みモード (定数 WRITE_CSV)</p> <p>"wb" バイナリ書き込みモード (定数 WRITE_BINARY)</p> <p>- 読み込みモード -</p> <p>"r" テキスト読み込みモード (定数 READ)</p> <p>"rtsv" TSV 読み込みモード (定数 READ_TSV)</p> <p>"rcsv" CSV 読み込みモード (定数 READ_CSV)</p> <p>"rb" バイナリ読み込みモード (定数 READ_BINARY)</p> <p>テキストモードは、普通のテキストファイルを扱うためのモードです。バイナリモードは、バイト値ごとにダイレクトに入出力を行うためのモードです。</p> <p>TSV モードは、タブ・空白区切りの数値データファイルを扱うためのモードで、CSV モードはカンマ区切り数値データファイルのモードです。基本的に数値データを扱う事を想定したもので、単純な挙動を行います。複雑な処理を要するテキスト CSV/TSV ファイルを扱う機能は、代わりに open.file.TextFile ライブラリが提供します。</p>

write	int ファイル番号, string 内容 1, string 内容 2, ...	書き込み対象ファイルに内容を書き込みます。引数を複数指定した場合、テキストモードなら連続で、TSVモードならタブ区切りで、CSVモードならカンマ区切りで、バイナリモードならバイトごとに書き込まれます。
writeln	write 関数と同じ	書き込み対象ファイルに内容を書き込み、改行します。引数を複数指定した場合の振る舞いは、上の write 関数と同様です。
read	int ファイル番号	読み込み対象ファイルの中身を 全行 読み込み、string 型配列として返します。 テキストモードの場合は、要素数 1 の配列が返され、0 番要素にファイル内容が格納されています。 その他モードの場合、TSV モードでは空白・タブ区切り、CSV モードではカンマ区切り、バイナリモードではバイト区切りで、配列にまとめて返されます。 なお、TSV モードでは空白とタブを区別せず、かつ連続した空白・タブは 1 つと同様に扱われます。これは数値データ TSV ファイルを前提とした挙動です。より厳格なテキスト TSV ファイルを扱う機能は、open.file.TextFile ライブラリが提供します。
readln	read 関数と同じ	読み込み対象ファイルの中身を 一行だけ 読み込み、string 型配列として返します。 配列の内容は、上の read 関数と同様です。
countln	int ファイル番号	ファイル内容の行数をカウントし、int 型で返します。
close	int ファイル番号	ファイルを閉じます(この時点で書き込み完了)。

※CSV ファイルの場合はカンマ記号(,)/改行区切りで処理されます

■ $y=x^2$ の値を CSV 数値データファイルに書き出す

例として、CSV 数値データファイルを扱います。まずは書き込みです。

例として、 $y=x^2$ の計算結果を、ファイルに出力してみましょう：

```
int fileID = open( "x2.csv", "wcsv" ); // ファイルを CSV 書き込みモードで開く
for( int i=0; i<=10; i++){
    writeln( fileID, i, i*i ); // ファイルに一行書き込み、改行
}
close( fileID );
```

これを実行して「x2.txt」を開くと、以下のような内容となっています：

```
0,0
1,1
...
10,100
```

このような形式のファイルは、一般的なグラフソフトを用いてグラフ化する事ができます。

■ $y=x^2$ の CSV 数値データファイルを読み込む

上で作った $y=x^2$ の数値データファイルを、今度は読み込んでみましょう：

```
int fileID = open( "x2.csv", "rcsv" ); // ファイルを CSV 読み込みモードで開く
int n = countln( fileID ); // ファイル行数を取得

int line[2]; // 各行の値(カンマ区切り)を受け取る変数
int x[n];
int y[n];
```

```

for( int i=0; i<n; i++){
    line = readln( fileID ); // 次の行を読み込み
    x[i] = line[0]; // 行の左から 1 個目の値を x に
    y[i] = line[1]; // 行の左から 2 個目の値を y に
}

close( fileID );

for( int i=0; i<n; i++){
    println( x[i], y[i] );
}

```

このプログラムを実行すると、先ほど作った $y=x^2$ のデータファイルを読み込んで配列に格納し、その内容をコマンドライン端末などに表示します。

上の `readln` 関数が最初に呼ばれると、ファイルの先頭行をカンマ記号「 , 」で区切って、`string` 配列に格納して返ってきます。それを `int` 型配列の `line` で受け取っています（この際、暗黙の型変換により、文字列が整数へキャストされます）。行の 1 列目の `line[0]` は `x` の値なので、`int` 型配列 `x` に値を格納し、同様に `line[1]` を `y` に格納しています。これで 1 行の読み込みが終わります。

`readln` 関数は `for` 関数の中にあるので、10 回繰り返して呼ばれます。2 回目に呼ばれた際はファイルの 2 行目が呼ばれます。このように、10 行目まで読み込み、プログラムが終了されます。

■ 文字コードの指定

異なるオペレーションシステム上で作成したファイルをやり取りする場合などでは、文字コードの違いによって、いわゆる文字化けが問題となる場合があります。このような場合のために、`open` 関数には、引数の最後に `string` 型の引数を追加して文字コードを指定する事ができます。

```

int file = open( "s_jis.txt", "w", "Shift_JIS" );
write( file, "こんにちは !" );
close( file );

```

使用できる文字コードには以下のものがあります：

文字コード	引数への記述	詳細
シフト JIS	Shift_JIS	シフト JIS でファイルにアクセスします。Shift と JIS の区切りはアンダーバー「_」である事にご注意ください。
EUC-JP	EUC-JP	EUC-JP でファイルにアクセスします。EUC と JP の区切りはハイフン「-」である事にご注意ください。
Unicode	UTF-8 UTF-16 UTF-32	Unicode でファイルにアクセスします。引数へは「Unicode」ではなく、「UTF-8」など、エンコーディングの実装を指定します。

上の文字コードは、正式な表記で記述しなければ無効となります。例えば Shift_JIS を Shift-JIS と書くと向こうなのでご注意下さい。同様に UTF-8 を UTF_8 と書いても無効となります。大文字と小文字も区別されるため、完全に上記の通りに記述する必要があります。

汎用ファイル入出力

■ より汎用的な入出力（テキスト CSV/TSV ファイル）

ここまで扱ってきた標準入出力関数は、高速ですが、非常に単純な挙動のものです。例えば CSV/TSV ファイルは数値データを想定したもので、単純に数値をカンマなどで区切って書き込み、同様に区切って読み込むだけです。しかし、数値に限らずより汎用的な内容、つまりテキストを CSV/TSV で書き込むには、複雑な処理が必要です。まずは、基本的な項目について説明します。

・デリミタ

CSV ならカンマ記号、TSV ならタブ記号などのように、値を区切る文字の事を**デリミタ**と呼びます。デリミタには、他にも半角スペースやセミコロンなどがよく使われます。

・エンクロージャ

例えば CSV 形式で、書き込む値がカンマ記号を含む場合、そのまま書き込んだのでは、それが「区切り文字(デリミタ)」としてのカンマ記号なのか、それとも「値の一部」としてのカンマ記号なのか、区別のつかない曖昧なファイルになってしまいます：

```
あいう,えお,かきくけこ,さしすせそ  
ABC,DE,FGHIJ,KL,MNO
```

上の例では、「あいう,えお」という値を書き込んだとしても、それは区切りのカンマと区別できません。そのため、テキストを扱う CSV 形式では、そのような場合に値をダブルクォーテーションなどで区切ります。このように囲む記号を**エンクロージャ**と呼びます。

```
"あいう,えお",かきくけこ,さしすせそ  
ABCDE,FGHIJ,"KL,MNO"
```

こうすると、最初の値は「あいう,えお」という意味なのだと区別できます。改行を含む場合も同様に：

```
"あいう  
えお",かきくけこ,さしすせそ  
ABCDE,FGHIJ,"KL  
MNO"
```


こうすると最初の値は「 あいう(改行)えお 」という意味であると区別できます。なお、値がエンクロージャと同じ文字を含む場合、エンクロージャを 2 つ並べるなどして区別します。

・コメント行

コメント行など、特定の文字で始まる行は無視して読み込ませたい場合もあります。

```
"あいう,えお",かきくけこ,さしすせそ  
#この行はコメントなので無視する  
ABCDE,FGHIJ,"KL,MNO"
```

コメント行の行頭文字は、そのファイルを扱うソフトによって多種多様です。

■ 汎用入出力ライブラリ

上で述べたような項目は、テキスト主体の汎用的な入出力を行うためには避けられません。しかし、前章で述べたような標準入出力関数は、数値ファイルを想定した単純なものであるため、これらの機能がありません。そのため VCSSL 3.1 以前では、load 関数などでファイル内容をまるごと読み込み、文字列解析を行って、こうした機能の処理を自分で記述する必要がありました。

しかし VCSSL3.2 からは、こうした機能を提供する汎用入出力機能が標準サポートで用意されたため、標準入出力関数とほぼ同様の簡単な操作で、こういった処理を自動で行えるようになりました。

・CSV 形式テキストファイルの書き込み

例として、CSV 形式でテキストファイルを書き込みます。

```
import file.TextFile; // 汎用入出力を行うために必要  
  
TextFile file = openTextFile( "text.csv", "wcsv" ); // ファイルを CSV 書き込みモードで開く  
setCommentLineCode( file, "#" ); // コメント行の行頭文字を # に設定  
  
writeln( file, "あい,うえお", "かきくけこ", "さしすせそ" ); // ファイルに内容を書き込んで改行  
commentln( file, "この行はコメントなので無視する" ); // コメント行を書き込んで改行  
writeln( file, "ABCDE", "FGHIJ", "KL,MNO" ); // ファイルに内容を書き込んで改行  
  
close( file ); // ファイルを閉じる
```

実行すると、以下の内容のファイルが書き出されます。

```
"あいう,えお",かきくけこ,さしすせそ  
#この行はコメントなので無視する  
ABCDE,FGHIJ,"KL,MNO"
```

カンマ記号を含む値が、正しくエンクロージャで囲まれています。また、コメント行は設定した「#」を付けて書き出されています。

プログラムの最初の行「`import file.TextFile;`」は、汎用入出力を行うために必要です。具体的には、汎用入出力機能を提供するプログラム(ライブラリ)を読み込んでいますが、これについては後の「モジュールとライブラリ」の章で説明します。

内容については、全体的に標準入出力とかなり似ているので、特に違和感無く使用できると思いますが、以下の1行は形が異なります。

```
TextFile file = openTextFile( "text.csv", "wcsv" ); // ファイルを CSV 書き込みモードで開く
```

この部分は、標準入出力の場合なら以下になるでしょう:

```
int file = open( "text.csv", "wcsv" ); // ファイルを CSV 書き込みモードで開く
```

このように、標準入出力(下)ならばファイル番号を `int` 型の変数 `file` で受け取るわけですが、汎用入出力(上)では `TextFile` という見たことの無い型の変数で受け取っています。これはすぐ後の章で説明しますが、構造体というもので、いわば新しい型です。ここではとりあえず `int` 型の代わりに `TextFile` 型を用いるだけなので、あまり深く考える必要はありません。

なお、標準入出力と同様、モードは以下のように定数で指定する事もできます。定数名も標準入出力と全く同じです。タイプ数は増えますが、タイプミスが構文エラーで分かるので防げます。

```
TextFile file = openTextFile( "text.csv", WRITE_CSV ); // モードを定数で指定
```

・CSV 形式テキストファイルの読み込み

続いて、先ほど作成した CSV 形式でテキストファイルを読み込みます。

```
import file.TextFile; // 汎用入出力を行うために必要

TextFile file = openTextFile( "text.csv", "rcsv" ); // ファイルを CSV 読み込みモードで開く
setCommentLineCode( file, "#" ); // コメント行の行頭文字を # に設定

// ファイル行数をカウント(コメント行は読み飛ばされる)
int n = countln( file );

string line[3]; // 各行の値を格納する配列
for(int i=0; i<n; i++){
    line = readln( file ); // ファイルから一行を読み込む
    println( i + "行目: " + line[0] + "/" + line[1] + "/" + line[2] ); // 読み込んだ内容を入力
}

close( file ); // ファイルを閉じる
```

実行すると、コンソールに以下の内容が出力されます。

```
0 行目: あいう,えお/かきくけこ/さしすせそ
1 行目: ABCDE/FGHIJ/KL,MNO
```

このように、エンクロージャを認識した上で、「 あい,うえお 」を 1 つの値として読み込みました。また、コメント行は無視されています。

ファイル行数をカウントする countln 関数も、コメント行は完全に無視して数えるてくれるめ、ファイル行末を超過して読み込んでしまう事ありません。なお、エンクロージャで囲まれた値が改行コードを含む場合についても、countln 関数はそれを 1 行にはカウントせず、あくまで値に含まれた改行コードであると識別してくれます。そのため、標準入出力と同様の感覚で、まず countln 関数で行数を取得し、その回数だけ readln 関数で読み進めるだけで、簡単にテキスト CSV/TSV ファイルの入出力を行う事ができます。

・入出力時の挙動に関する細かな設定

汎用入出力では、細かな設定が可能です。詳細は下記 URL の公式仕様をご参照ください。

<https://www.vcssl.org/ja-jp/lib/file/TextFile>

実行時評価

■ 文字列をプログラムとして実行する

string 型変数が保持する文字列を、プログラムとして実行したい場合があるかもしれません。例えば電卓ソフトウェアを製作する場合、ユーザーによって入力された数式(文字列)を処理し、値を算出する必要がありますが、そういった処理を行うエンジンを自力で書き下すのは面倒です。

そのような場合、ここで述べる実行時評価の機能を利用し、文字列をそのまま VCSSL プログラムの一部として実行すると便利です。

■ eval 関数

一行の式を計算して値を求めたい時は、eval 関数を使用します：

```
int i = eval( "1+2" );  
print( i );
```

上のプログラムを実行すると、VCSSL コンソールに「 3 」と表示されます。

eval 関数は、結果を string 型の配列で返します。上の例では、要素数 1 の配列で、0 番要素が "3" の string 配列が返され、それが暗黙の型変換によって int 型非配列の i に代入されたわけです。なお、式が構文的に正しくなかった場合は無効値(NULL との比較が真)が返されます。

また、以下のように、代入を行う事もできます。

```
int i = 0;  
eval( "i = 1+2" );  
print( i );
```

この例でも VCSSL コンソールに「 3 」と表示されます。結果が構造体になる場合などは、string 型配列では返せないため、このように eval 内で代入する方法を取ります。

なお、eval 関数は、複数行(文)に渡る内容を実行する事はできません。

■ ランタイムオーバーライド(実行時オーバーライド)

複数行に渡る内容を実行するには、ランタイムオーバーライド(実行時オーバーライド)を使用します。

ランタイムオーバーライドは、関数の処理内容を、プログラム実行中に上書きする機能です。この機能を利用する事で、string 型変数が保持する文字列を、関数の内容として処理する事が可能となります。引数も取得でき、戻り値も返す事ができます。さらに、**ローカルを宣言する事も可能**です。

ランタイムオーバーライドを行うには、override 関数を呼び出し、その一つ目の引数に関数名、二つ目の引数に(オーバーライドする関数の)引数情報、三つ目の引数にプログラム記述内容を指定します。

```
override( string name, string[ ] args, string program ) ;
```

最初の引数 name には関数名を、最後の引数 program には新しいプログラム記述内容を指定します。

問題は中央の引数 args ですが、これにはランタイムオーバーライド対象となる関数の引数情報を指定します。args の要素数は引数の個数に一致し、各要素には引数の変数型を記述します。

例えば、以下のような関数を想定するとします。

```
int fun( int a, float b, float c[ ] )
```

このような関数をランタイムオーバーライドするには、args には以下のような string 配列を指定します。

```
string args[ 3 ] ;  
args[ 0 ] = "int" ;  
args[ 1 ] = "float" ;  
args[ 2 ] = "float[ ]" ;
```

なお、引数が void 型の関数をオーバーライドするには、args に要素数 0 の配列を指定します。

実際にランタイムオーバーライドを行って見ましょう。例として以下のように記述し、実行してみてください。

```
println( " オーバーライド前 : " + fun( 8 ) );

// オーバーライドする関数名
string name = "fun" ;

// オーバーライド後のプログラム記述内容
string program = " int x = i * i ; return x ; " ;

// 引数の型/個数指定
string args[1] ;
args[ 0 ] = "int" ;

// 関数 fun の処理内容をランタイムオーバーライド
override( name, args, program ) ;

println( " オーバーライド後 : " + fun( 8 ) );

void fun ( int i ) {
    return i ;
}
```

このプログラムを実行すると、「 オーバーライド前 : 8 」と「 オーバーライド後 : 64 」という結果が 2 行に渡って出力されます。このプログラムでは、関数 fun は受け取った引数をそのまま返すよう記述されています。しかしそれをランタイムオーバーライドにより、引数の 2 乗を返すように変更したわけです。

■グローバル変数へのアクセスや関数呼び出しも可能

ランタイムオーバーライドで上書きする処理内容の中で、グローバル変数(すべてのスコープの外側で宣言され、プログラム中のどこからでもアクセスできる変数)を使用する事も可能です。また、別の関数を呼び出す事も可能です。

■ ランタイムオーバーライドで宣言されるのはローカル変数

例に挙げたプログラムでは、ランタイムオーバーライドで上書きする新しい処理内容の中で、int 型の変数 `x` を宣言しています。このように、ランタイムオーバーライドする処理内容の中で、新しい変数を宣言して使用する事も可能です。

ただし、ランタイムオーバーライドで宣言した変数はローカル変数となるため、ランタイムオーバーライドの対象となった関数の中でしか使用できません。

■ 新たな関数の宣言はできない

ランタイムオーバーライドは既存の関数を上書きするものであるため、別の新しい関数を宣言する事はできません。

構造体

■ 構造体

関連する意味を持った変数を、まとめて一つにして扱いたい場合はよくあります。これを実現するのが構造体です。

例えば Box 形状 (四角形) の幅と高さは、別々に宣言するよりも、2 つをまとめて「 Box 型 」などと扱えれば便利です。構造体では、このような Box 型を作る事ができます。

VCSSL における構造体は、以下のような文法で定義します：

```
struct 構造体の名前 {  
    メンバ変数 a の宣言 ;  
    メンバ変数 b の宣言 ;  
    ...  
}
```

メンバ変数とは、構造体にまとめる (構成要素となる) 変数の事です。

■ 幅と高さを持つ構造体の定義例

実際に、上でも述べた、幅 (width) と高さ (height) をメンバ変数に持つ構造体「 Box 型 」は、以下のように定義します：

```
struct Box {  
    int width ; // 幅  
    int height ; // 高さ  
}
```

これで、幅を表すメンバ変数「 width 」と、高さを表すメンバ変数「 height 」をまとめた構造体「 Box 」が定義できました。

■ 構造体の使用方法

構造体を使うには、普通の変数と同様に、変数名を付けて宣言を行います。変数型の部分が構造体の名前(上の例では Box)になるだけです。

そして、構造体のメンバ変数を使うには、構造体の変数名の後にドット記号「.」を付け、その後にメンバ変数の名前を記述します。つまり「構造体変数名.メンバ変数名」で1つの変数のように扱います。

具体的に、先の Box 型を使ってみましょう。以下のように記述し、実行してみてください：

```
struct Box {  
    int width ; // 幅  
    int height ; // 高さ  
}  
  
Box b ; // Box 型変数 b を宣言  
b.width = 100 ; // b のメンバ変数 width に 100 を代入  
b.height = 200 ; // b のメンバ変数 height に 200 を代入  
  
print( "幅=" + b.width + " 高さ=" + b.height ) ;
```

上のプログラムを実行すると、VCSSL コンソールに「 幅=100 高さ=200 」と表示されます。このように、構造体を用いて、幅と高さという2つの値を、1つの変数にまとめて扱う事ができました。

■ 構造体の代入演算は、すべてのメンバ変数のコピー

構造体変数の代入は、同じ構造体(型)のもの同士でのみ可能です。その際、代入演算の内容は、全てのメンバ変数値を代入するのと同じになります。つまりは中身のコピーに相当します。

■ 配列や関数でも使用可能

構造体の配列を作る事も可能です。また、関数の引数や戻り値に構造体を使用する事もできます。

ジェネリクス

■ 変数型そのものを引数にする

関数や構造体の用途によっては、引数やメンバ変数の型が異なるだけで、それ以外は全く同じ関数や構造体が必要になる場合があります。このような場合、必要な型すべてに対して、同じような関数や構造体を用意するのは面倒です。また、未知の型の構造体を、広く一般的に扱いたい場合もあります。こういった場合に役立つのが、ジェネリクスです。ジェネリクスは、関数の引数と同じような感じで、型そのものを引数に指定する機能です。

■ ジェネリクスを用いた関数

ジェネリクスを用いた関数は、以下のような仕様で定義します。

```
戻り値の変数型 関数名<型引数 1, 型引数 2, … >( 引数 1, 引数 2, … ){
    処理内容 ;
    return 戻り値 ;
}
```

上で通常関数と異なるのは、<型引数 1, 型引数 2, … > の部分です。ここに型の名前を格納する引数を宣言します。名前は実在の型名ではなく、自由に付けます。いわば仮の型です。そこに、呼び出し時に実際の型が入力されます。

実際に、加算を行う関数を、ジェネリクスを用いて記述してみましょう：

```
Type add<Type>( Type a, Type b ){
    Type value = a + b ;
    return value;
}

int x = add<int>( 1, 2 ); // add 関数の型引数「 Type 」を int 型として呼び出す
println( x );
```

このプログラムを実行すると、VCSSL コンソールに「 3 」と表示されます。

上の例で、add 関数で用いている「 Type 」型というのは、VCSSL に標準では存在しない型です。しかし関数内で、Type value = a + b ; のように、Type 型の変数を宣言しています。さらに、関数の引数も戻り値も Type 型となっています。これでエラーにならないのは不思議に思えるかもしれません。これが成立する理由は、Type が型引数であって、仮の型だからです。

この関数を呼び出す時に、add<int>(1, 2) としています。この<int>という所で、型引数 Type に int を指定しています。これにより、add 関数の中で、Type 型が int 型と見なされて実行されます。

つまり、上の add 関数が実際に実行される際には、以下の関数と等価なものとなっています：

```
int add ( int a, int b ){  
    int value = a + b ;  
    return value;  
}
```

今度は float 型として呼び出してみましょう：

```
Type add<Type>( Type a, Type b ){  
    Type value = a + b ;  
    return value;  
}  
  
float x = add<float>( 1.1, 2.2 ); // add 関数の型引数「 Type 」を float 型として呼び出す  
println( x );
```

この例では VCSSL コンソールに「 3.3 」と表示されます。型引数に float を指定したため、Type 型が float 型と見なされて実行されたためです。

このようにして、ジェネリクスを用いると、同じ関数を、複数の型に対して使用する事ができるようになります。

■ ジェネリクスを用いた構造体

ジェネリクスは、構造体でも使用する事ができます。それは以下のような仕様で定義します。

```
struct 構造体の名前<型引数 1, 型引数 2, ... > {  
    メンバ変数 a の宣言 ;  
    メンバ変数 b の宣言 ;  
    ...  
}
```

このように、構造体名の後ろに型引数を付けます。例えば、型が未定のメンバ変数を持つ構造体は、以下のように定義して使用します：

```
struct Box<Type> {  
    Type width ;  
    Type height ;  
}  
  
Box<float> b ;  
  
b.width = 1.2 ;  
b.height = 2.4 ;  
  
println( "幅=" + b.width + " 高さ=" + b.height ) ;
```

上のプログラムを実行すると、VCSSL コンソールに「 幅=1.2 高さ=2.4 」と表示されます。

上の例の構造体「 Box 」は、型が未知のメンバ変数「 width 」と「 height 」を持ちます。それを変数宣言時に Box<float> b ; というように、型パラメータに float 型を指定した段階で、Type 型が float 型と見なされて用意されたわけです。

上の例で、b の宣言を Box<int> b ; に変更して実行すると、メンバ変数が int 型となるため、結果は VCSSL コンソールに「 幅=1 高さ=2 」と表示されます。

モジュールとライブラリ

■ ライブラリとは

ライブラリとは、よく使う関数や定数などをまとめて記述した、部品的なプログラムの事です。

■ モジュール

これまでの内容では、プログラムは全て 1 枚のファイルに記述し、処理内容もその中で完結していました。しかし、ライブラリを使用したプログラムの場合は、いくつものファイルに記述された処理が、組み合わせあって動作する事になります。

この時、各ファイルの記述内容を、1 枚のファイルに結合して実行する事もできます。しかし、それでは同じ変数名や関数名が競合していた場合に不都合が生じます。そのため、通常は、ファイルごとに独立した、「モジュール」という形で読み込まれて実行されます。このように書くと抽象的で分かり辛いですが、VCSSL では基本的に 1 枚のファイルが 1 つのモジュールに自動対応する仕組みになっているため、モジュールとはそれぞれのファイル内容の事だと思っても問題はありません。A という名前のファイルに書かれたプログラムは「A モジュール」という具合です。

■ 実行モジュール と ライブラリモジュール

実行モジュールとは、これまでのように（別のモジュールから使用されるのでは無く）、普通に直接実行されるモジュールの事です。それに対してライブラリモジュールとは、そのものを直接実行されるのでは無く、別のモジュールから読み込まれて使用される、部品的なモジュールの事です。

■ ライブラリのインポート

ライブラリモジュールに用意された関数を、別のモジュール（別ファイルに記述されたプログラム）から使用するには、そのモジュールの先頭領域で `import` 宣言を行います：

```
import ライブラリパス ;
```

ライブラリパスは、**実行モジュールからの相対パス**を、ドット記号「`.`」区切りで指定します。また、拡張子「`.vcssl`」は**不要**です。ライブラリモジュールのファイルが実行モジュールと同じ場所（フォルダ

内)に存在する時は、ライブラリモジュールのファイル名を記述するだけで使用できます。

例えば実行モジュールと同じフォルダ内にある「lib.vcssl」を使用するなら：

```
import lib ;
```

これで、lib.vcssl に記述されている全ての関数と定数が利用可能になります。

また、実行モジュールから見て、「aaa」フォルダ内の、さらに「bbb」フォルダ内にある「lib.vcssl」を使用するなら：

```
import aaa.bbb.lib ;
```

とします。

■ 別モジュールの関数や変数を使用する

別モジュールの関数や変数を使用するには、**同じ関数名や変数名が競合していない場合**、何も特別な事をする必要はありません。これまでと変わらず、普通に変数や関数を呼ぶ事ができます。

例えば、実行モジュール A から、モジュール B の関数を呼ぶ場合は、以下のようにします：

- 実行モジュール A (A.vcssl) -

```
import B ; // ライブラリモジュール B を読み込む

fun() ;
```

- ライブラリモジュール B (B.vcssl) -

```
void fun() {
    print( "call B.fun" ) ;
}
```

上の例で、**実行モジュール A**（ **A.vcssl** ）を実行すると、VCSSL コンソールに「 call B.fun 」と表示されます。

モジュール A には fun 関数は宣言されていませんが、fun()と呼び出しています。そこでモジュール B に宣言されている fun 関数が呼ばれたわけです。

■ モジュールの優先度

上に示した例では、関数や変数の名前（関数の場合は引数仕様も）が競合している場合、問題になる事があります。例えば、以下のように記述し、実行してみてください。

- 実行モジュール A （ A.vcssl ） -

```
import B ; // ライブラリモジュール B を読み込む

void fun() {
    print( "call A.fun" );
}

fun(); // ※ fun 関数の名前はモジュール A と B で競合している
```

- ライブラリモジュール B （ B.vcssl ） -

```
void fun() {
    print( "call B.fun" );
}
```

上の例で、**実行モジュール A**（ **A.vcssl** ）を実行すると、VCSSL コンソールに「 call A.fun 」と表示されます。つまり、モジュール B の fun 関数ではなく、モジュール A の fun 関数が呼ばれたわけです。

このように、関数を呼び出す場合、**自身のモジュール内に宣言された関数が最優先**されます。そして、**自身のモジュール内に見つからない場合、import しているモジュール内が探**されます。複数存在した場合は、後に import されたものが優先されます。

それでも見つからないという場合は、別モジュール間の `import` などによって、処理系に読み込まれた全てのモジュールから探されます。優先度は、後に読み込まれたものが高くなります。以下に例を挙げます：

- 実行モジュール A (`A.vcssl`) -

```
import B ; // ライブラリモジュール B を読み込む

fun();
```

- ライブラリモジュール B (`B.vcssl`) -

```
import C ; // ライブラリモジュール C を読み込む
```

- ライブラリモジュール C (`C.vcssl`) -

```
void fun() {
    print( "call C.fun" );
}
```

上の例で、**実行モジュール A** (`A.vcssl`) を実行すると、VCSSL コンソールに「 `"call C.fun"` 」と表示されます。

このように、モジュール A からはモジュール B しか `import` していないにも関わらず、モジュール C の `fun` 関数を呼ぶ事ができています。これは、モジュール B がさらにモジュール C を `import` していて、処理系がモジュール C の存在を知っているためです。

ただしこの機能は、モジュール数が増加すると優先度の把握が難しくなり、可読性も低下します。そのため、**原則として、使われる側のモジュールは、使う側のモジュールで明示的に `import` しておく事が推奨されます。**

つまり上の例では、モジュール A でも `import C ;` しておく事が推奨されます。

■ 所属モジュールの明示

これまでに示した、優先度による自動的なモジュール判定は、小規模なプログラムの場合には便

利です。しかしモジュールの数が多くなってきたり、第三者が開発したライブラリモジュールを組み合わせて使用するような場合、どのモジュールにこういった名前・引数仕様の関数があるのかを、常に完璧に把握しておくのは困難です。その場合、関数名の競合が発生しているの見逃してしまうと、意図していたのとは異なるモジュールの関数をコールしてしまうミス招きかねません。

このような場合、関数名の前にドット記号「.」区切りでモジュール名を明示する事ができます：

- 実行モジュール A (A.vcssl) -

```
import B ; // ライブラリモジュール B を読み込む
import C ; // ライブラリモジュール C を読み込む

B.fun() ;
C.fun() ;
```

- ライブラリモジュール B (B.vcssl) -

```
void fun() {
    println( "call B.fun" ) ;
}
```

- ライブラリモジュール C (C.vcssl) -

```
void fun() {
    println( "call C.fun" ) ;
}
```

上の例で、**実行モジュール A (A.vcssl)** を実行すると、VCSSL コンソールに、改行を挟んで「 call B.fun 」 「 call C.fun 」と表示されます。

このようにモジュールを明示する事で、競合を見落として意図しないモジュールの関数と呼んでしまうのを防げる上に、可読性も向上します。

■ アクセス修飾子

上で述べた所属モジュールの明示は、意図しないモジュールの関数・変数を呼んでしまう(アクセスする)事を防ぐ手段ですが、いわば「呼ぶ側」の手段です。これに対して「呼ばれる側」でアクセスを制御する手段も存在します。それがアクセス修飾子です。VCSSL のアクセス修飾子には、「private」と「public」の2種類が存在します。

例えば、モジュール外からは呼んでほしくない関数・変数や、全く呼ぶ必要が無い関数・変数があったとします。こうした場合、関数・変数宣言の先頭に「private」と記述する事で、モジュール外からは呼べなくなります(存在しないのと同じに扱われます)。

逆に、関数・変数宣言の先頭に「public」と記述すれば、モジュール外から自由に呼ぶ事ができます。ただし、VCSSL ではこれまでのように普通に関数や変数を宣言すると、標準で public になります。具体的な例を見てみましょう:

```
// この変数はモジュール外からアクセスできない
private int a ;

// この変数はモジュール外から自由にアクセス可能
public int b ;

// この関数はモジュール外から呼べない
private void funA(){
    println("call funA") ;
}

// この関数はモジュール外から自由に呼べる
public void funB(){
    println("call funB") ;
}
```

上の例では、変数 a 及び関数 funA はアクセス修飾子「private」が付いているので、モジュール外から呼ぶ事はできません。逆に、変数 b 及び関数 funB はアクセス修飾子「public」が付いているので、モジュール外から自由に呼ぶ事ができます。

ところで、「public」をわざわざ付けても、普通にアクセス修飾子を省略して宣言した時の挙動と同じなので、機能的にはあまり意味はありません。しかしながら明示的に付けておく事で、「private」を付け忘れたのでは無く、安心して外部から呼んでも大丈夫な変数・関数である事を表明する事ができ、可読性に貢献します。

■ モジュールの唯一性

モジュールには唯一性が保たれます。具体的には、複数回 import されたモジュールでも、その実体は 1 つしか無いという事が保証されます。例えば以下のように記述し、実行してみてください：

- 実行モジュール A (A.vcssl) -

```
import B ; // ライブラリモジュール B を読み込む
import C ; // ライブラリモジュール C を読み込む

C.x = 2 ;

B.fun() ;
```

- ライブラリモジュール B (B.vcssl) -

```
import C ;

void fun() {
    println( C.x ) ;
}
```

- ライブラリモジュール C (C.vcssl) -

```
int x = 0 ;
```

このプログラムを実行すると、VCSSL コンソールに「 2 」と表示されます。これはつまり、モジュール A から import したモジュール C と、モジュール B から import したモジュール C は、**同一のも**のであるということです。

■ ライブラリモジュール内での相対パスの扱い

ライブラリモジュールの中から、何らかのファイルへのアクセスを行う場合は、ファイルパスの指定に注意が必要です。というのも、ライブラリモジュールからファイルへアクセスする際のファイルパスには、**そのライブラリから見た相対パスではなく、実行モジュールから見た相対パスを指定しなければならない**からです。

例えば、ライブラリの中でさらに別のライブラリをインポートする場合、ライブラリパスには、実行モジュールから見た相対パスを指定する必要があります。さらに、ライブラリの中でファイル入出力関数を使用する場合や、画像のロード等を行う場合にも、実行モジュールから見た相対パスを指定する必要があります。

■ 各モジュールのグローバル領域実行順序

本文書の前半でもずっと行ってきた通り、VCSSL では、モジュールのグローバル領域に通常の処理を記述する事ができます。この時に問題になるのが、各モジュールのグローバル領域が、どのような順序で実行されるかです。VCSSL では、import の階層をツリー状に表した際、深い階層から順に実行されていきます（実行モジュールが最も浅い階層となります）。同じ深さの階層では、先に import されたものから順に実行されます。

■ グローバル定数の初期化順序

const キーワードを付けて宣言された変数は、初期化後に値を変更できない定数となります：

```
const int CONSTANT_VALUE = 255 ;
```

グローバル領域に宣言された定数、いわゆるグローバル定数は、グローバル領域の実行よりも早いタイミングで初期化されます。そのため、グローバル領域の実行が始まった時点では、全てのモジュールのグローバル定数は初期化されています。

■ 複数ファイルを 1 つのモジュールにまとめて読み込む

import に似ている機能に、include というものがあります。使い方は、import の代わりにそのまま include と記述するだけです。include は、別ファイルを独立したモジュールとして読み込むのではなく、ファイルの記述内容を、その場所にそのまま埋め込む機能です。つまり、include したファイルと、include されたファイルは、内容が結合され、同じ 1 つのモジュールとして読み込まれます。

この機能は、似ているけれども微妙に異なるモジュールが複数必要な場合などに、共通部分を

別ファイルに記述して使用するためにサポートされています。VCSSL では、同一モジュール内で、同名・同引数仕様の関数を複数宣言した場合、後に（ファイル下方で）宣言したもので上書きされます。これを利用して、include したファイルの関数を上書きし、微妙に異なるモジュールを作ることができます。

標準ライブラリ

■ 標準ライブラリとは

標準ライブラリとは、あらかじめ用意されていて、標準で使えるライブラリの事です。

通常、ライブラリを使用するためには、ユーザーがライブラリを入手し、起動モジュールと同じフォルダ(ディレクトリ)内や、処理系指定のフォルダ内に配置するなどの準備を行う必要があります。しかし標準ライブラリでは、このような準備は不要で、いつでもすぐに使用できます。

■ 標準ライブラリの役割

標準ライブラリの役割は、現実的なプログラムを開発する上で、よく必要とされるような、基本的な機能を提供する事です。そのような機能を、全てのユーザーが一から自力で作成しなければいけないのは不便ですし、他人の書いたプログラムを読む際にも、何から何まで人によって全く異なるというのは読み辛くなってしまいます。従って、そういった機能は標準ライブラリとしてあらかじめ用意されているわけです。

また、グラフィックスや GUI(グラフィカルユーザーインターフェイス)、オペレーティングシステムや処理系側で処理する必要があるような機能も、標準ライブラリとして提供されます。

■ print などのシステム関数も、「 System 」標準ライブラリで提供されている

実はこれまでのプログラムでも、標準ライブラリの機能を使用していました。それは、画面にメッセージを表示する print 関数などのシステム関数です。システム関数は、「 System 」という標準ライブラリによって提供されています。試しに以下のように記述し、実行してみてください：

```
import System ;  
  
System.print( "Hello, World !" );
```

これを実行すると、VCSSL コンソールに「 Hello, World ! 」と表示されます。この通り、print 関数が System ライブラリによって提供されている事が分かります。VCSSLでは、全てのモジュールを読み込む前に、System ライブラリが自動で読み込まれる決まりになっています。そのため、System ライブラリをインポートしなくても、print 関数などを呼び出す事ができたわけです。

■ System 以外の標準ライブラリは、明示的にインポートして使用

インポートすらも不要で利用できるのは、標準ライブラリの中でも System だけです。

System ライブラリは、あまりに基本的な機能を提供し、ほとんどの全ての場面においてインポートする必要が生じるため、特例的にインポートしなくても使用できるようになっているのです。

System 以外の標準ライブラリは、普通のライブラリと同様、インポートしてから使用する必要があります。これは、使用するライブラリが多いほど、解釈すべき内容が増え、起動に時間がかかってしまうためです。また、関数名の競合などの面倒を避けるためでもあります。ライブラリは基本的に、そのモジュールで使用するものだけをインポートするのが、実行時でも開発時でも有利です。

■ 標準ライブラリの一覧と詳細情報

VCSSL の各標準ライブラリに関する詳細情報は、下記 URL にて参照できます：

<https://www.vcssl.org/ja-jp/lib/>

標準ライブラリの一覧は以下の通りです。

- ・System (<https://www.vcssl.org/ja-jp/lib/System>) - 根幹機能を提供
- ・Math (<https://www.vcssl.org/ja-jp/lib/Math>) - 数学関数を提供
- ・Graphics (<https://www.vcssl.org/ja-jp/lib/Graphics>) - 描画リソース関連機能を提供
- ・Graphics2D (<https://www.vcssl.org/ja-jp/lib/Graphics2D>) - 2 次元描画機能を提供
- ・Graphics3D (<https://www.vcssl.org/ja-jp/lib/Graphics3D>) - 3 次元描画機能を提供
- ・GUI (<https://www.vcssl.org/ja-jp/lib/GUI>) - ウィンドウやボタンなど、GUI 機能を提供
- ・Color (<https://www.vcssl.org/ja-jp/lib/Color>) - 色の制御を提供
- ・Sound (<https://www.vcssl.org/ja-jp/lib/Sound>) - サウンドの制御を提供
- ・Text (<https://www.vcssl.org/ja-jp/lib/Text>) - 文字列の基本的な処理機能を提供
- ・File (<https://www.vcssl.org/ja-jp/lib/File>) - ファイル一覧やパス処理等の機能を提供
- ・Time - (<https://www.vcssl.org/ja-jp/lib/Time>) - 時間や日時に関する機能を提供
- ・Process (<https://www.vcssl.org/ja-jp/lib/Process>) - プロセス実行機能を提供
- ・Thread (<https://www.vcssl.org/ja-jp/lib/Thread>) - マルチスレッド処理機能を提供

これらの中で、(System 以外で)恐らく最も頻繁に使用するのは Math, Text, File, Time ライ

ブラリでしょう。以下では、実際にそれらを扱ってみます。

■ Math ライブラリ

Math は、sin や cos など、基本的な数学関数・定数を提供するライブラリです。

<https://www.vcssl.org/ja-jp/lib/Math>

実際に使用してみましょう。以下のように記述し、実行してみてください：

```
import Math ; // 標準ライブラリ「 Math 」をインポート

float value = sin( PI/2.0 ) ; // sin(  $\pi/2$  ) つまり sin(90°) = 1
print( value ) ;
```

上のプログラムを実行すると、sin($\pi/2$)の値である「 1.0 」が表示されます。必要に応じて、値を数桁以上の精度で求める varfloat 型の関数も使用できます。

■ Text ライブラリ

Text は、文字列の検索や置き換えなど、基本的なテキスト操作機能を提供するライブラリです。

<https://www.vcssl.org/ja-jp/lib/Text>

実際に使用してみましょう。以下のように記述し、実行してみてください：

```
import Text ; // 標準ライブラリ「 Text 」をインポート

// 文字数をカウント
int n = countText( "AAA/BBB/CCC" ) ;
println( n ) ; // 11 と表示される
```



```

// 2 番目から(5-1)番目までで文字列を切り取る
string crop = cropText( "AAA/BBB/CCC", 2, 5 );
println( crop ); // 「A/B」と表示される

// 文字列「a/a」が初めて出現する位置を取得
int first = findText( "aaa/aaa/aaa", "a/a", FIRST );
println( first ); // 2 と表示される(先頭から 0,1,2…とカウント)

// 文字列「a/a」が最後に表示する位置を取得
int last = findText( "aaa/aaa/aaa", "a/a", LAST );
println( last ); // 6 と表示される(先頭から 0,1,2…とカウント)

// 文字列「a/a」が出現する位置を全て取得
int all[ ] = findText( "aaa/aaa/aaa", "a/a", ALL );
println( all ); // 「 2      6 」と表示される

// 正規表現「./」に一致する位置を全て取得
int patternAll[ ] = findText( "AAA/BBB/CCC", "./", ALL_PATTERN );
println( patternAll ); // 「 2      6 」と表示される

// 正規表現「./」に一致する部分文字列を全て取得
string ext[ ] = extractText( "AAA/BBB/CCC", "./", ALL_PATTERN );
println( ext ); // 「 A/B      B/C 」と表示される

//「/」を全て「@」に置き換え
string repl = replaceText( "aaa/aaa/aaa", "a/a", "@", ALL );
println( repl ); // aa@a@aa と表示される

```

このように、文字列に対して基本的な操作を行う事ができます。

■ File ライブラリ

File は、ファイルパスを取得したり、ファイル一覧を取得したりするなど、ファイルに関する情報を提供するライブラリです。

<https://www.vcssl.org/ja-jp/lib/File>

実際に使用してみましょう。以下のように記述し、実行してみてください：

```
import File ; // 標準ライブラリ「 File 」をインポート

string base = getFilePath( "." ); // 相対パスの基準(実行モジュールの場所)を取得
println( base ); // 実行モジュールのあるフォルダ(ディレクトリ)の絶対パスが表示される

string parent = getFilePath( ".." ); // 親フォルダを取得
println( parent ); // 実行モジュールフォルダのさらに親フォルダの絶対パスが表示される

string file = getFilePath( "test.txt" ); // ファイルパスを取得
println( file ); // 実行モジュールと同じフォルダにある test.txt の絶対パスが表示される

string rel = getFilePath( "./bbb/test.txt", "../aaa" ); // 相対パスを絶対パスに変換

// 「 実行モジュールフォルダの親フォルダの下にある aaa フォルダ 」から見て、
// 「 bbb フォルダの中にある test.text 」の絶対パスが表示される
println( rel );
```

上の通り、ファイルの場所を取得するなど、ファイルの基本的な情報を扱う事ができます。なお、ファイルの指定では、「 / 」をフォルダ(ディレクトリ)の区切りとして使用できます。また、「 . 」を相対パス基準フォルダ、「 .. 」を親フォルダとして使用できます。

■ Time ライブラリ

Time は、時間や日付などに関する機能を提供するライブラリです。

<https://www.vcssl.org/ja-jp/lib/Time>

以下は時間を計測する例です：

```

import Time ; // 標準ライブラリ「 Time 」をインポート

int start = time( ) ; // この時点の時間を取得

alert( "時間を計っています..." ) ; // メッセージを表示している間は実行が止まる

int end = time( ) ; // この時点の時間を取得

int sec = second( end - start ) ; // 所要時間 (end-start) を秒単位に変換

print( sec ) ; // 所要時間を表示

```

上のプログラムを実行すると、「 時間を計っています... 」というメッセージが表示されます。メッセージを閉じた時点で、閉じるまでに要した時間が、VCSSL コンソールに秒単位で表示されます。

上の例で使用した time 関数は、プログラムの実行が開始されてからの経過時間を返す関数です。time 関数が返す時間の単位は、通常はミリ秒 (1/1000 秒) ですが、厳密には処理系に依存します。

そのため、得られた結果を任意の単位に変換するための関数も提供されており、上で用いた second 関数もその一つです。単位の変換には、millisecond (ミリ秒単位)、second (秒単位)、minute (分単位)、hour (時間単位) などがあります。

なお、second 関数や minute 関数、hour 関数などは、引数を省略すると、時計における時刻を返す機能もあります。これを用いて、何時何分何秒かを取得する事ができます。さらに、day で日付を、month で月を、year で年を取得する事ができます：

```

import Time ; // 標準ライブラリ「 Time 」をインポート

println( "日付=" + year( ) + "/" + month( ) + "/" + day( ) ) ; // 年/月/日を表示
println( "時刻=" + hour( ) + ":" + minute( ) + ":" + second( ) ) ; // 時間:分:秒を表示

```

上のプログラムを実行すると、その時点の日付と時刻が表示されます。なお、month 関数は、0～11 ではなく、1～12 までを返すと決まっています。つまり 1 月であれば 0 ではなく 1 を返します。Day

関数も同様で、(月の始めの)1 日であれば 0 ではなく 1 を返します。他言語では 0 から数えるものも存在するため、混同しないように注意が必要です。

その他のライブラリ

■ より幅広い機能を提供する標準ライブラリ

標準ライブラリは、前回扱ったものが全部ではありません。前回扱ったライブラリは、VCSSL 2.0 の頃から存在し、言語として最低限必要になる基板機能を提供しているものです。そして VCSSL 3.0 以降では、より幅広い機能を提供するライブラリも順次増えていっています。それらは、最初は標準ライブラリとは別扱いの位置づけでしたが、VCSSL 3.4 の時点で多くが標準ライブラリに統合されました。

以下では、実際にそらの内のいくつかを使用してみます。

■ 基本的なデータ構造

まずは、「スタック」や「キュー」などのデータ構造を扱うライブラリです。スタックとキューは、値を出し入れして、溜めておくための仕組みです。両者で異なるのはデータの取り出し順序で、**キューは入れた順に、スタックは逆順に取り出します**。つまりスタックでは、最後に入れたデータが最初に取り出されます。

データ構造を扱うための機能は、一般にジェネリクスを用いた構造体と関数によって提供されます。そのため、任意型の値（自作の構造体を含む）を扱う事ができます。

以下に使用例を掲載します：

```
import data.Stack ; // スタックの機能を提供するライブラリ
import data.Queue ; // キューの機能を提供するライブラリ

Stack<int> s ; // 整数値のスタックを扱う構造体
push<int>( s, 1 ) ; // スタック s に値「1」を入れる
push<int>( s, 2 ) ; // スタック s に「2」を入れる

println( "スタックから値を取り出します:" ) ;
println( pop<int>( s ) ) ; //スタック s から値を 1 つ取り出す
println( pop<int>( s ) ) ; //スタック s から値を 1 つ取り出す
```

```
Queue<int> q; //整数値のキューを扱う構造体
enqueue<int>(q, 1); // キュー q に値「1」を入れる
enqueue<int>(q, 2); // キュー q に「2」を入れる

println("キューから値を取り出します:");
println(dequeue<int>(q)); //キューq から値を 1 つ取り出す
println(dequeue<int>(q)); //キューq から値を 1 つ取り出す
```

上のプログラムを実行すると、VCSSL コンソールに以下のように表示されます:

```
スタックから値を取り出します:
2
1
キューから値を取り出します:
1
2
```

上の通り、スタックでは入れたのと逆順に、キューでは入れた順に、値が取り出された事が分かります。data.Stack/Queueはスタックとキューを扱うライブラリで、詳細情報は以下のURLで参照できます。

<https://www.vcssl.org/ja-jp/lib/data/Stack>
<https://www.vcssl.org/ja-jp/lib/data/Queue>

■ 軽量フレームワーク

描画やアニメーションなど、特別な処理を記述する必要があるものの、頻繁に用いられるような用途に関しては、毎回同じような内容を記述しなくても済むように、軽量フレームワークが用意されています。用途に応じたフレームワークを用いる事で、ゼロからプログラムを開発しなくても、最低限必要な部分だけを記述すれば済むようになります。

実際に 3D 描画を扱うためのフレームワークを使用してみましょう:

```

import Math;
import Color;
import Graphics3D;
import graphics3d.Graphics3DFramework; // 3D 描画用フレームワーク

// モデルの ID 格納変数や時刻変数など
int axis, box, sphere, cone;
float time=0.0;

/*
 * 初期化を行う関数(プログラムの最初に一度だけ呼び出される)
 * ここには、3D モデルの生成や配置など、
 * プログラムの最初に行いたい準備処理を記述する
 */
void onStart( int rendererID ){

    // フレームワークのパラメータ設定
    setBackgroundColor( Color.BLACK ); //背景色の設定
    setWindowSize( 800, 600 ); //ウィンドウサイズ(幅,高さ)の設定

    // 座標軸モデルを配置
    axis = newAxisModel( 3.0, 3.0, 3.0 ); //生成
    mountModel( axis, rendererID ); //配置

    // 箱モデルを配置
    box = newBoxModel( 1.0, 1.0, 1.0 ); //生成
    setModelColor( box, 255, 0, 0, 255 ); //色設定(赤,緑,青, $\alpha$ )
    mountModel( box, rendererID ); //配置

    // 球モデルを配置
    sphere = newSphereModel( 0.5, 0.5, 0.5, 38, 24 ); //生成
    moveModel( sphere, 1.5, 1.5, 0.0 ); //移動(dX,dY,dZ)
    setModelColor( sphere, 0, 0, 255, 255 ); //色設定(赤,緑,青, $\alpha$ )
    mountModel( sphere, rendererID ); //配置

```

```

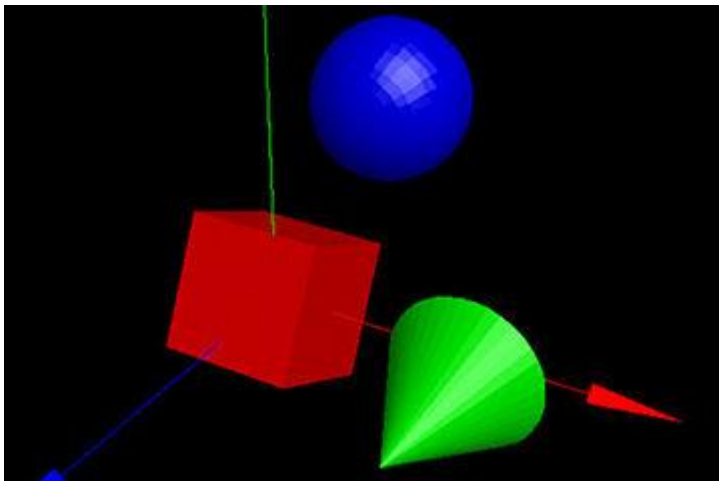
// 円錐モデルを配置
cone = newConeModel( 0.5, 0.5, 1.0, 38, 2 ); //生成
moveModel( cone, 0.0, 0.0, 1.0 ); //移動(dX,dY,dZ)
setModelColor( cone, 0, 255, 0, 255 ); //色設置(赤,緑,青, $\alpha$ )
mountModel( cone, rendererID ); //配置
}

/*
 * 更新処理を行う関数(繰り返し呼び出される)
 * ここには、3D モデルの移動など、
 * 毎ループ行いたい任意の処理を記述する
 */
void onUpdate( int rendererID ){
    rotModelX( box, 0.08 ); //箱モデルを X 軸まわり回転
    rotModelZ( sphere, 0.01 ); //球モデルを Z 軸まわり回転

    //円錐モデルを適当に移動(dX,dY,dZ)
    moveModel( cone, 0.1*sin(time), 0.1*sin(3.0*time), 0.0 );
    time += 0.1; // 時刻を少し進める
}

```

上のプログラムを実行すると、下図のような 3D アニメーション映像が表示されます。



上で用いた `graphics3d.Graphics3DFramework` は、3D 描画用途をサポートする、アニメーション対応のフレームワークです。詳細情報は以下の URL で参照できます。

<https://www.vcssl.org/ja-jp/lib/graphics3d/Graphics3DFramework>

VCSSL には、3D 描画を扱うための標準ライブラリ「 Graphics3D 」が存在します。しかし、それだけで現実的な 3D プログラムを開発するには、表示画面や描画エンジンの生成や、アニメーションループの制御、ライトの設定など、面倒な処理を自分で記述する必要があり、簡易用途ではやや不便です（逆に、細かい部分まで自分で調整したい場合には便利です）。

そこで graphics3d.Graphics3DFramework を使用すると、そういった面倒な部分はフレームワークが自動で行ってくれます。そして、準備が完了したタイミングで onStart 関数、アニメーション中の画面更新タイミングで onUpdate 関数を呼んでくれます。なので、これらの関数内に目的の処理を記述するだけで、手軽に 3D アニメーションプログラムを開発できます。

なお、上で宣言した onStart 関数や onUpdate 関数は、一度名称変更された、比較的新しい関数名であり、以前は initialize や update という関数名で宣言する必要がありました。この旧称の initialize や update も、互換目的のために引き続き使用できます。

ところで、このフレームワークでは、あくまで面倒な部分を自動化してくれるだけなので、独自に記述する部分に関しては標準ライブラリ「 Graphics3D 」の関数を使用する必要があります。上のプログラムでも、モデルの生成や配置、移動などに Graphics3D ライブラリの関数を使用しています。

Graphics3D ライブラリに関する詳細情報は以下の URL で参照できます。

<https://www.vcssl.org/ja-jp/lib/Graphics3D>

2D 描画用にも、3D 同様のフレームワークが用意されています。使用例を以下に掲載します：

```
Import Graphics2D;
import graphics2d.Graphics2DFramework; // 2D 描画用フレームワーク

/*
 * 初期化処理を行う関数（プログラムの最初に一度だけ呼び出される）
 * ここには、プログラムの最初に行いたい準備処理を記述する
 */
void onStart( int rendererID ){
    // フレームワークのパラメータ設定
    setAnimationState( false ); //アニメーションはしないので無効化
    setWindowSize( 300, 300 ); //ウィンドウサイズの設定
}
```

```

/*
 * 描画処理を行う関数(繰り返し呼び出される)
 * ここには、2DCG の描画処理などを記述する
 */
void onPaint( int rendererID ){
    string imagePath = "fighter.png"; //画像ファイル

    // 青色で直線を描画
    setDrawColor( rendererID, 0, 0, 255, 255 ); //赤,緑,青,α
    drawLine( rendererID, 0, 0, 100, 100 ); //x,y,w,h

    // 緑色で長方形を描画
    setDrawColor( rendererID, 0, 255, 0, 255 ); //赤,緑,青,α
    drawRect( rendererID, 0, 100, 100, 100, true ); //x,y,幅,高さ,塗りつぶし

    // 赤色で楕円を描画
    setDrawColor( rendererID, 255, 0, 0, 255 ); //赤,緑,青,α
    drawOval( rendererID, 50, 0, 150, 100, false ); //x,y,幅,高さ,塗りつぶし
}

```

上のプログラムを実行すると、ウィンドウに線、長方形、楕円が表示されます。

上で用いた `graphics2d.Graphics2DFramework` は、2D 描画用途をサポートする、アニメーション対応のフレームワークです。詳細情報は以下の URL で参照できます。

<https://www.vcssl.org/ja-jp/lib/graphics2d/Graphics2DFramework>

描画関数などに関しては、標準ライブラリ「`Graphics2D`」を使用する必要があります。詳細情報は以下の URL で参照できます。

<https://www.vcssl.org/ja-jp/lib/Graphics2D>

■ グラフ描画など、処理系依存のライブラリ

独立したソフトウェアのようなツールとやり取りする API ライブラリも存在します。例えば、グラフソフトと通信し、グラフをプロットする機能を提供するライブラリなどが挙げられます。

ただし、そうした機能は実装規模が比較的大きいため、サポートされるかどうかは、処理系開発元の判断によります。グラフソフト関連の機能などは、数値計算分野に向けた処理系ではサポートされるかもしれませんが、全く必要の無い用途に向けてはサポートされないかもしれません。

以上のような事を前提とした上で、数値計算用の 2D/3D グラフソフトを制御するためのライブラリ仕様が、以下の URL の通りに用意されています。

<https://www.vcssl.org/ja-jp/lib/tool/Graph2D>

<https://www.vcssl.org/ja-jp/lib/tool/Graph3D>

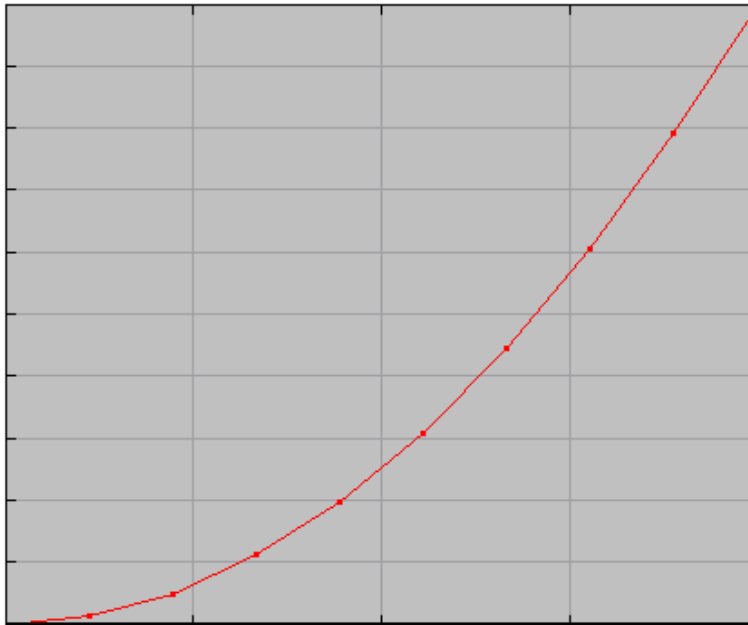
上の「 tool.Graph2D/Graph3D 」ライブラリがサポートされている処理系においては、上記仕様の通りに、2D/3D グラフをプロットする事ができます。

実際の使用例は、細かい部分は処理系依存ですが、大まかに以下ようになります。まずは、2D グラフの場合です：

```
import tool.Graph2D ; // 2D グラフソフト制御用ライブラリ

// ファイル「data2d.tsv」にデータを書き出す
int fileID = open( "data2d.tsv", "wtsv" ); // wtsv は TSV 形式の書き込みモード
float dx = 0.1 ;
float x ;
float y ;
for( int i=0; i<10; i++){
    x = i * dx ;
    y = x * x ;
    writeln( fileID, x, y ) ; // ファイル書き込み
}
close( fileID ); // ファイルを閉じる(忘れると書き込みが完了しないので注意)
int graph2D = newGraph2D( ) ; // 2次元グラフソフトを起動
setGraph2DFile( graph2D, "data2d.tsv" ) ; // ファイル「data2d.tsv」をグラフにプロット
```

実行すると、以下のようなグラフが表示されます：



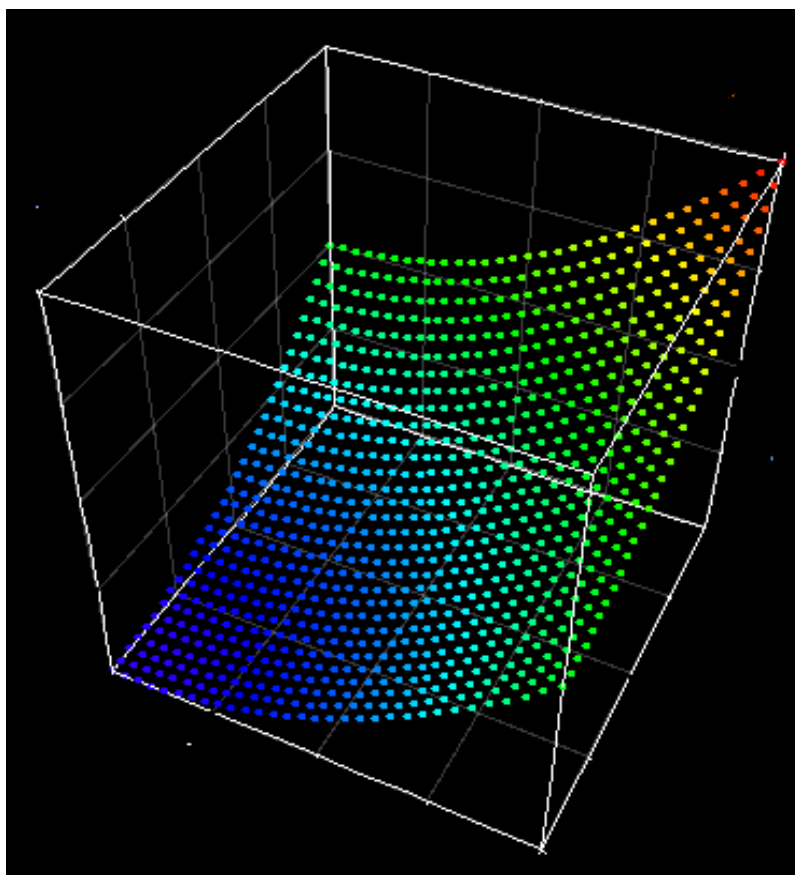
座標軸の目盛りやラベルがどのように表示されるかは処理系依存です。

続いて、3D グラフの場合です：

```
import tool.Graph3D ; // 3D グラフソフト制御用ライブラリ

// ファイル「data3d.tsv」にデータを書き出す
int fileID = open( "data3d.tsv", "wtsv" ); // wtsv は TSV 形式の書き込みモード
float dx = 0.1 ;
float dy = 0.1 ;
float x, y, z ;
for( int i=0; i<=30; i++ ){
    for( int j=0; j<=30; j++ ){
        x = i * dx ;
        y = j * dy ;
        z = x * x + y * y ;
        writeln( fileID, x, y, z ) ; // ファイル書き込み
    }
    writeln( fileID, "" ) ;
}
close( fileID ) ; // ファイルを閉じる (忘れると書き込みが完了しないので注意)
int graph3D = newGraph3D() ; // 3 次元グラフソフトウェアを起動
setGraph3DFile ( graph3D, "data3d.tsv" ) ; // ファイル「data3d.tsv」をグラフにプロット
```

実行すると、以下のようなグラフが表示されます：



座標軸の目盛りやラベル、色などがどのように表示されるかは処理系依存です。

プラグイン

■ プラグインとは

プラグインとは、VCSSL だけでは実現できないような、独自の機能を提供するための部品的なソフトウェアです。また、VCSSL から、他の言語で記述された処理を呼ぶためにも使用します。例を挙げるなら、本書で扱ったファイル入出力関数なども、VCSSL の処理系に標準で接続されたファイル入出力プラグインが処理しています（それが標準ライブラリ「System」によりラッピングされています）。プラグインが独自にサポートする関数の事を、そのプラグインのプラグイン関数と呼びます。

一般に、プラグインは VCSSL 以外の言語で開発されたプログラムです。プラグインがどのような言語で開発され、また VCSSL 処理系との間でどのような通信形態を採用しているかは、処理系によって異なります。従って、プラグインを接続するには、使用している処理系に対応したプラグインを用意する必要があります。

■ プラグインを接続するには

プラグインをシステムに接続するには、プログラムの先頭領域で connect 宣言を行います：

```
connect プラグインパス ;
```

プラグインパスは、ライブラリのインポートと同様に、実行モジュールからの相対パスを、ドット記号「.」区切りで指定します。プラグインのファイルが実行モジュールと同じ場所（フォルダ内）に存在する時は、プラグインのファイル名を記述するだけで使用できます。

例えば実行モジュールと同じフォルダ内にある「conn」プラグインを使用するなら：

```
connect conn ;
```

これで、conn がサポートするプラグイン関数を用いて、conn が提供する機能を使用する事ができるようになります。また、実行モジュールから見て、「aaa」フォルダ内の、さらに「bbb」フォルダ内にある「conn」を使用するなら：

```
connect aaa.bbb.conn ;
```

とします。なお、プラグインは、実行モジュールと同じ場所に配置するのではなく、処理系によって指定された場所に置く場合もあります。

■ プラグイン関数のラッピング

プラグインの機能を利用するプログラム内において、プラグイン関数を直接的にコールする事は推奨されません。なぜなら、プラグインはオペレーティングシステムなどの環境への依存性が強く、さらにプラグイン関数は個々のプラグイン独自の仕様を持つため、結果としてプラグイン関数を直接コールしているようなプログラムは、特定の動作環境に強く依存するものになってしまうからです。

例えば、接続しているプラグインを、同じような機能を提供する別のものに切り替えようと思った場合にはどうなるでしょう。この場合、プログラム内のプラグイン関数コール部分をすべて書き換えなくてはなくなってしまいます。そして、プラグインは特定のオペレーティングシステムなどの環境に依存し易い存在であるため、環境ごとにプラグインを切り替えなければいけない可能性は大いに考えられます。

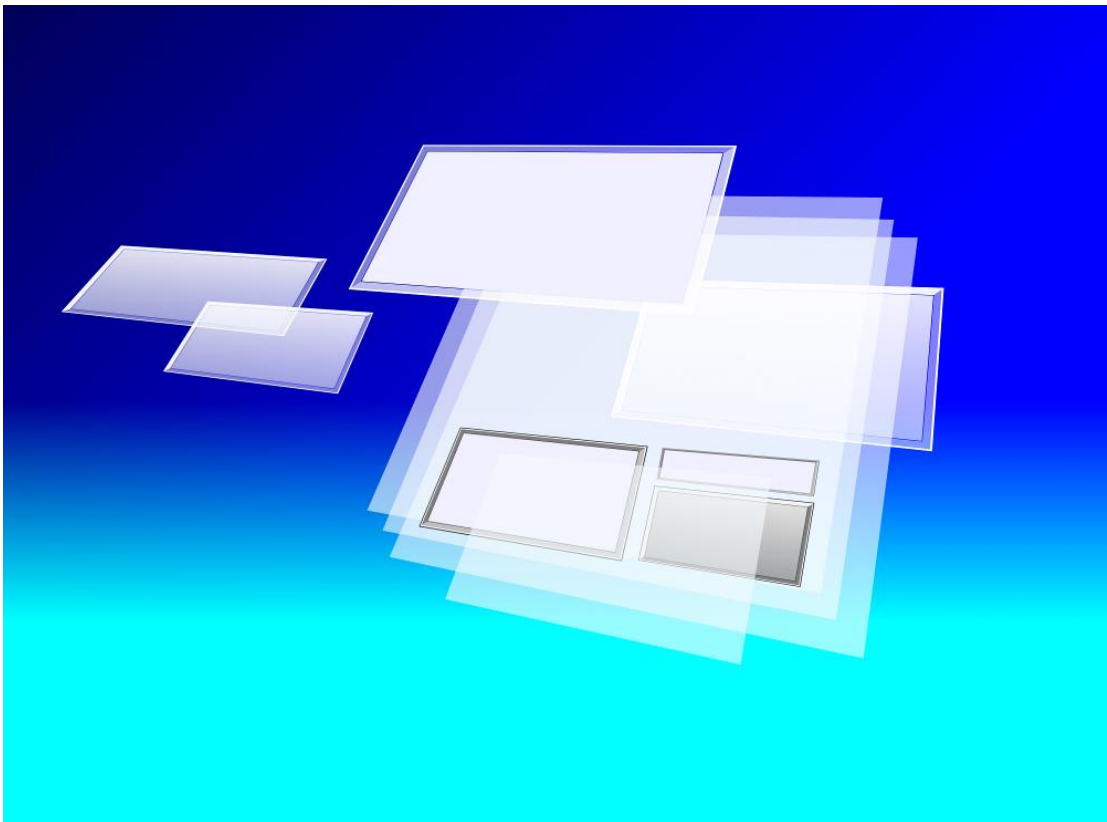
このような特定のプラグインへの依存を回避するための方法が、プラグイン関数のラッピングです。プラグイン関数のラッピングとは、プラグイン関数をコールするだけの関数を用意するという手法です。この関数をラッパー関数と呼びます。そしてプログラム内では、プラグイン関数を直接コールするのではなく、代わりにラッパー関数の方をコールします。つまり、プラグイン関数を VCSSL の関数でまると包んでしまうわけです。

このようにしておけば、プラグインを別のものに交換する際、ラッパー関数の中身を書き換えるだけで済みます。ラッパー関数をライブラリにまとめておけば、そのライブラリ 1 枚を書き換えるだけで済み、従ってプラグインの機能を利用している大量のプログラムを書き直す必要は無くなります。このようにして、特定のプラグインへの依存を回避する事が可能になります。

第 2 部

GUI

- グラフィカル ユーザー インターフェイス -

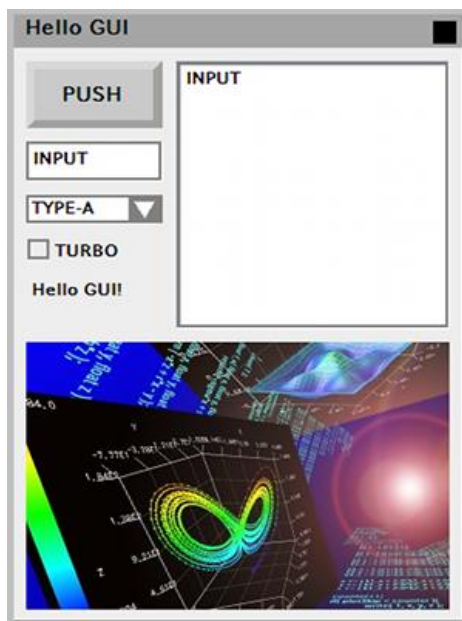


この部では、ウィンドウやボタンなどの GUI (グラフィカル・ユーザー・インターフェイス) を備えた VCSSL プログラムを開発する方法を解説します。この文書の内容を扱うには、VCSSL の基本的な文法をすでに習得している必要があります。

GUI

■ GUI とは

ユーザーがプログラムを操作するための仕組みを、一般にユーザーインターフェイスと呼びます。ユーザーインターフェイスにはいくつかの種類がありますが、現在の主流となっているものの一つに GUI (グラフィカルユーザーインターフェイス) が挙げられます。GUI では、ウィンドウやボタンなどの視覚的な表現によって、ユーザーがプログラムを操作します。GUI は、一般的な用途のアプリケーション・ソフトウェアなどで広く用いられています。



■ CUI から GUI へ

GUI 以外で有名なインターフェイスとして、CUI (キャラクターユーザーインターフェイス) というものもあります。

CUI では、プログラムは文字列の表示端末を通してユーザーとやり取りします。例えば VCSSL プログラムでも、GUI を使用しない場合は、VCSSL コンソールを用いた文字列表示が主体となります。そういった「黒い画面に白い文字が流れていく」といったものが CUI です。



CUI は、GUI が登場する以前の時代に、主流として普及していました。その後、GUI の登場によって、一気に主流の座を奪われました。これにはいくつかの理由がありますが、最大の理由は GUI が初心者ユーザーにやさしかった事でしょう。つまり VCSSL でも CUI から GUI 主体の設計へ移行する事で、より初心者ユーザーにやさしいプログラムを開発する事が可能になります。

なお、一般には目にする機会の少なくなった CUI ですが、熟練者にとっては高効率なため、プログラミングや科学技術計算など、比較的専門的な分野においては現在も広く用いられています。

■ VCSSL プログラムで GUI を扱うには

VCSSL プログラムで GUI を扱うには、VCSSL 標準ライブラリの一つである、VCSSL GUI ライブラリをインポートする必要があります。これには、プログラムの先頭行に以下のように記述します。

```
import GUI ;
```

これで、プログラム中から GUI を扱うための関数が利用可能になります。

GUI コンポーネント

■ GUI コンポーネント

ウィンドウやボタンなど、GUI において画面を構成する部品のことを、一般に GUI コンポーネントと呼びます。



■ GUI コンポーネントの生成

・GUI コンポーネントの生成

VCSSL プログラム中で任意の GUI コンポーネントを生成するには、new～といった形の関数を呼び出します。～の部分には、newWindow 関数や newButton 関数など、作成する GUI コンポーネントの種類に固有の名称が入ります。

```
int new～ ( ～ )
```

こういった関数をコールすると、GUI コンポーネントが生成され、それに割り当てられた GUI コンポーネント ID が返されます。

・GUI コンポーネント ID

GUI コンポーネント ID とは、全ての GUI コンポーネントに割り振られる識別番号です。つまり VCSSL 処理システムにとっての、GUI コンポーネントの名前のようなものです。

GUI コンポーネント ID は、そのままでは数字で分かり辛いので、通常は int 型の変数に入れて使います。

■ GUI コンポーネントの配置

GUI コンポーネントは、いくつも組み合わせて使用します。その土台となるのがウィンドウです。その上にボタンやラベル、入力項目などを作成して配置し、一般的な GUI ソフトウェアの形となります。

このように、GUI コンポーネントの上に、別の GUI コンポーネントを配置するには、mountComponent 関数を使用します。

```
void mountComponent ( int childID, int parentID )
```

この関数をコールすると、GUI コンポーネント上に別の GUI コンポーネントが配置されます。引数 childID には配置する GUI コンポーネントの ID を、続く引数 parentID には配置先 GUI コンポーネントの ID を指定します。つまり、parentID の上に childID を配置します。

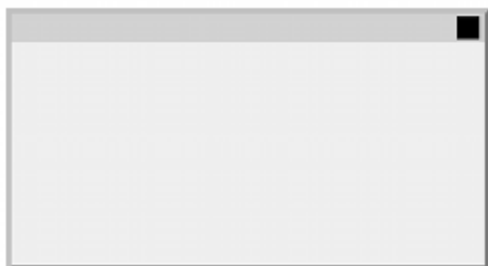
なお、VCSSL3.0 以前の世代では、配置には addComponent 関数を使用していました。しかし、add~という関数名としては引数の順序が混乱を招くという理由により、VCSSL3.1 以降では、関数名を上記の mountComponent に変えたものが追加されました。つまり addComponent 関数と mountComponent 関数は、名称が異なるだけで全く同一のものです。

各種 GUI コンポーネント

GUI コンポーネントには、様々な種類のものが存在します。ここでは、各種 GUI コンポーネントの作成について扱います。なお、ここで扱うのは代表的なものであり、実際にはより多くの GUI コンポーネントが利用できます。

※この章の画像に掲載されている GUI コンポーネントのデザインは模式図です。実際のデザインは、オペレーティングシステムの種類や、VCSSL 処理システムのバージョン、その他環境によって異なります。

■ ウィンドウの作成



GUI 開発において最初に作成するのが、ウィンドウです。ウィンドウを作成するには、`newWindow` 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newWindow( int x, int y, int width, int height, string title )
```

引数の `x` と `y` でウィンドウ左上頂点の座標を指定し、`width` と `height` でウィンドウの幅と高さを指定します。また、引数 `title` でウィンドウのタイトルバー表示文字列を指定します。

■ ボタンの作成



続いて扱うボタンは、頻繁に使用される代表的な GUI コンポーネントです。ボタンを作成するには、`newButton` 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newButton(int x, int y, int width, int height, string text )
```

引数には、x と y でボタン左上頂点の座標を指定し、width と height でボタンの幅と高さを指定します。最後の引数 text には、ボタンに表示する文字列を指定します。

■ テキストフィールドの作成

A rectangular text field with a light gray background and a thin black border. It contains the text "ABCDE" in a bold, black, sans-serif font.

次に、文字列を入力する一行の入力項目、テキストフィールドの作成です。こちらもボタンと並ぶ、代表的な GUI コンポーネントの一つです。テキストフィールドを作成するには、newTextField 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newTextField ( int x, int y, int width, int height, string text )
```

引数には、x と y でテキストフィールド左上頂点の座標を指定し、width と height でテキストフィールドの幅と高さを指定します。最後の引数 text には、項目に初期状態で表示する文字列を指定します。

■ テキストエリアの作成

A rectangular text area with a light gray background and a thin black border. It contains five lines of text in a bold, black, sans-serif font: "ABCDEFGH", "IJKLMNOP", "OPQRSTU", "VWXYZ", and "1234567890".

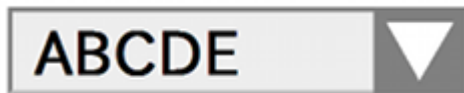
続いて、文字列を入力する広い領域、テキストエリアの作成です。テキストエリアはテキストフィールドと違い、複数行の文章を表示/入力する事が可能です。テキストエリアを作成するには、

newTextArea 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newTextArea ( int x, int y, int width, int height, string text )
```

引数には、x と y でテキストエリア左上頂点の座標を指定し、width と height でテキストエリアの幅と高さを指定します。最後の引数 text には、項目に初期状態で表示する文字列を指定します。

■ セレクトフィールドの作成

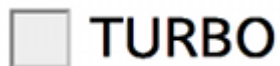


ユーザーに自由に文字列を入力してもらうのではなく、いくつかの候補の中から選択してほしい場合には、セレクトフィールドを使用します。セレクトフィールドを作成するには、newSelectField 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newSelectField ( int x, int y, int width, int height, string[ ] text )
```

引数には、x と y でセレクトフィールド左上頂点の座標を指定し、width と height でセレクトフィールドの幅と高さを指定します。最後の引数 text には、項目に初期状態で表示する選択項目を string 型の配列で指定します。

■ チェックボックスの作成



ユーザーに ON/OFF の 2 択を提示するには、チェックボックスを使用します。チェックボックスを作成するには、newCheckBox 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newCheckBox ( int x, int y, int width, int height, string text, bool b )
```

引数には、xとyでチェックボックス左上頂点の座標を指定し、widthとheightでチェックボックスの幅と高さを指定します。続く引数 text には、チェックボックスに表示するテキストを指定します。最後の b には、初期状態での ON/OFF 状態を指定します。

■ テキストラベルの作成

画面上に文字列を埋め込んで表示するには、テキストラベルを使用します。テキストラベルの作成には newTextLabel 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newTextLabel ( int x, int y, int width, int height, string text )
```

引数には、xとyでテキストラベル左上頂点の座標を指定し、widthとheightでテキストラベルの幅と高さを指定します。続く引数 text には、テキストラベルに表示するテキストを指定します。

■ 画像ラベルの作成

続いて、画面上に画像（プログラム内で描画したものを含む）を表示する方法を扱います。

それにはまず、VCSSL プログラム中で「グラフィックスデータ」を作成する必要があります。これは、画像ファイルの内容や、描画エンジンの描画結果を格納するデータで、いわば画用紙のようなものです。全くの白紙から作成する事も可能ですが、ここでは画像ファイルから読み込みましょう。

・Graphics ライブラリのインポート

画像ファイルを読み込んでグラフィックスデータを作成するには、まず、VCSSL 標準ライブラリの一つである Graphics ライブラリをインポートする必要があります：

```
import Graphics ;
```

Graphics ライブラリは、VCSSL でグラフィックスデータを扱うための基盤となるライブラリです。別のライブラリと連携してグラフィックスデータを動的に生成したり、画像ファイルから読み込んだり、

逆にグラフィックスデータを画像ファイルに出力するなどの機能が用意されています。

・画像ファイルの読み込み

画像ファイルからグラフィックスデータを生成するには、newGraphics 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newGraphics ( string filePath )
```

この関数は、引数に指定された場所から画像ファイルを読み込んでグラフィックスデータを生成し、グラフィックスデータ ID を戻り値として返します。グラフィックスデータ ID とは、グラフィックスデータに割り振られる識別番号です。

この newGraphics 関数の引数 filePath には、読み込む画像ファイルの場所を、実行モジュールから見た相対パスで指定します。画像ファイルが実行モジュールと同じ場所にある場合には、画像ファイル名だけで読み込み可能です。読み込みが可能な画像形式は、環境やバージョンによって異なります。PNG 形式の画像ファイルは、大抵の環境で読み込む事が可能なので、機種依存性を回避したい場合には、PNG 形式画像の使用が推奨されます。

・画像ラベルの作成

ここまでで準備は終了です。読み込んだグラフィックスデータを画面上に表示するには、画像ラベルという GUI コンポーネントを使用します。画像ラベルの作成には newImageLabel 関数を呼び出します。この関数は以下の仕様を持っています。

```
int newImageLabel ( int x, int y, int width, int height, int graphicsID )
```

引数には、x と y で画像ラベル左上頂点の座標を指定し、width と height で画像ラベルの幅と高さを指定します。最後の graphicsID には、表示対象のグラフィックスデータ ID を指定します。

■ プログラム例

これまでに扱った各種 GUI コンポーネントを、実際に使用してみましょう。下記のように記述し、実行してみてください。

```

import GUI ;
import Graphics ;

// ( 0, 0 ) の位置に 800×600 のウィンドウを生成
int windowID = newWindow( 0, 0, 360, 480, "Hello GUI" ) ;

// ( 10, 10 ) の位置に 100×50 のボタンを配置
int buttonID = newButton ( 10, 10, 100, 50, "PUSH" ) ;
mountComponent( buttonID, windowID ) ;

// ( 10, 70 ) の位置に 100×50 サイズのテキストフィールドを配置
int textFieldD = newTextField( 10, 70, 100, 30, "INPUT" ) ;
mountComponent( textFieldD, windowID ) ;

// ( 120, 10 ) の位置に 200×200 サイズのテキストエリアを配置
int textAreaID = newTextArea( 120, 10, 200, 200, "INPUT" ) ;
mountComponent( textAreaID, windowID ) ;

// ( 10, 110 ) の位置に 100×20 サイズのセレクトフィールドを配置
string text[ 3 ] ;
text[ 0 ] = "TYPE-A";
text[ 1 ] = "TYPE-B";
text[ 2 ] = "TYPE-C";
int selectFieldID = newSelectField( 10, 110, 100, 20, text ) ;
mountComponent( selectFieldID, windowID ) ;

// ( 10, 140 ) の位置に 100×20 サイズのチェックボックスを作成
int checkBoxID = newCheckBox( 10, 140, 100, 20, "TURBO", true ) ;
mountComponent( checkBoxID, windowID ) ;

// ( 10, 170 ) の位置に 100×20 サイズのテキストラベルを配置
int textLabelID = newTextLabel( 10, 170, 100, 20, "Hello GUI !" ) ;
mountComponent( textLabelID, windowID ) ;

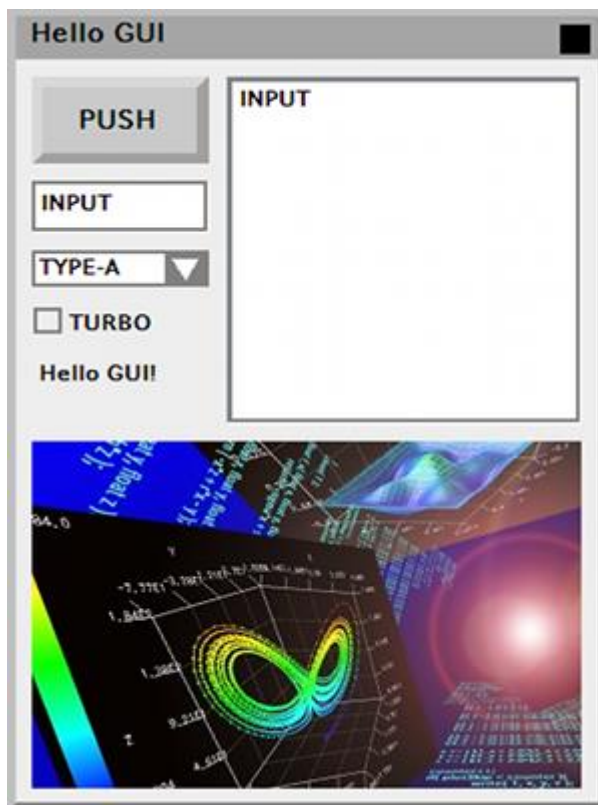
```

```
// 「Test.png」という名前の PNG 形式画像ファイルを読み込み
int graphicsID = newGraphics( "Test.png" );

// ( 10, 220 )の位置に 310×200 サイズの画像スラベルを配置
int imageLabelID = newImageLabel( 10, 220, 310, 200, graphicsID );
mountComponent( imageLabelID, windowID );
```

このプログラムを実行すると、ウィンドウが起動し、その上に様々な GUI コンポーネントが表示されます。

▼実行結果



※この画像は模式図です。実際の GUI デザインは、オペレーティングシステムの種類や、VCSSL 処理システムのバージョン、その他環境によって異なります。

GUI コンポーネントの設定

GUI コンポーネントは、ただ作成して配置しただけではあまり意味がありません。ここではテキストフィールドから文字を取得したり、グラフィックスを変更したりするなど、配置後の GUI コンポーネント制御に関して解説します。

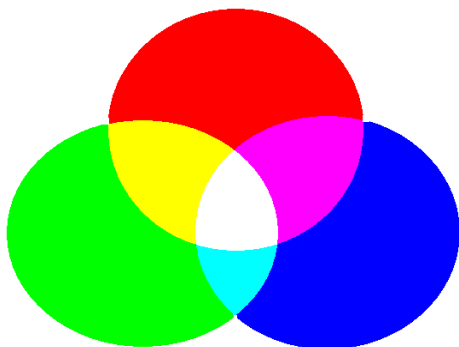
■ 色の変更/取得

GUI コンポーネントの色を変更するには `setComponentText` 関数を呼び出します。この関数は以下の仕様を持っています。

```
void setComponentColor (  
    int id,  
    int fgRed, int fgGreen, int fgBlue, int fgAlpha,  
    int bgRed, int bgGreen, int bgBlue, int bgAlpha  
)
```

引数 `id` には、色を変更する対象の GUI コンポーネント ID を指定します。続く引数 `fgRed`～`fgAlpha` には、前景色の色成分をそれぞれ 0～255 の範囲で指定します。さらに引数 `bgRed`～`bgAlpha` には、背景色の色成分をそれぞれ 0～255 の範囲で指定します。

ここで色成分は、赤、緑、青成分で表現する「RGB カラー」形式に、不透明度 `Alpha` を加えた「RGBA カラー」形式で指定します。`Alpha` は 0 で完全な透明、255 で不透明になります。特別な意図が無い場合、通常は常に 255 を指定します。なお、赤、緑、青成分の合成は、下図のように加法混色で処理されますのでご注意ください。



なお、GUI コンポーネントの色を取得するには `getComponentColor` 関数を呼び出します。この関数は以下の仕様を持っています。

```
int[ ][ ] getComponentColor ( int id )
```

引数 `id` には、色を取得する対象の GUI コンポーネント ID を指定します。この関数は要素数が `[2][4]` の `int` 型配列を戻り値に返します。戻り値の配列には、要素`[0][0]～[0][3]`にそれぞれ前景色の RGBA 値が、要素`[1][0]～[1][3]`にそれぞれ背景色の RGBA 値が格納されています。

`setComponentColor` および `getComponentColor` 関数は、以下の GUI コンポーネントなどに対して使用できます。

ウィンドウ / ボタン / テキストフィールド / テキストエリア / チェックボックス /
セレクトフィールド / テキストラベル / 画像ラベル

■ 位置の変更/取得

GUI コンポーネントの位置を変更するには `setComponentLocation` 関数を呼び出します。この関数は以下の仕様を持っています。

```
void setComponentLocation ( int id, int x, int y )
```

引数 `id` には、サイズを変更する対象の GUI コンポーネント ID を指定します。続く引数 `x, y` には、それぞれ変更後の左上頂点の `x` 座標、`y` 座標を指定します。

また、GUI コンポーネントのサイズを取得するには `getComponentLocation` 関数を呼び出します。この関数は以下の仕様を持っています。

```
int[ ] getComponentLocation ( int id )
```

この関数は要素数が 2 の int 型配列を戻り値に返します。戻り値の配列には、要素[0]に x 値、要素 [1]に y 値が格納されています。

setComponentLocation および getComponentLocation 関数は、以下の GUI コンポーネントなどに対して使用できます。

ウィンドウ / ボタン / テキストフィールド / テキストエリア / チェックボックス /
セレクトフィールド / テキストラベル / 画像ラベル

■ サイズの変更/取得

GUI コンポーネントのサイズを変更するには setComponentSize 関数を呼び出します。この関数は以下の仕様を持っています。

```
void setComponentSize ( int id, int width, int height )
```

引数 id には、サイズを変更する対象の GUI コンポーネント ID を指定します。続く引数 width, height には、それぞれ変更後の幅、高さを指定します。

また、GUI コンポーネントのサイズを取得するには getComponentSize 関数を呼び出します。この関数は以下の仕様を持っています。

```
int[ ] getComponentSize ( int id )
```

この関数は要素数が 2 の int 型配列を戻り値に返します。戻り値の配列には、要素[0]に幅、要素 [1]に高さの値が格納されています。

setComponentSize および getComponentSize 関数は、以下の GUI コンポーネントなどに対して使用できます。

ウィンドウ / ボタン / テキストフィールド / テキストエリア / チェックボックス /
セレクトフィールド / テキストラベル / 画像ラベル

■ テキストの取得/変更

GUI コンポーネントのテキストを変更するには `setComponentText` 関数を呼び出します。この関数は以下の仕様を持っています。

```
void setComponentText( int id, string text )
```

引数 `id` には、テキストを変更する対象の GUI コンポーネント ID を指定します。また、引数 `text` には変更後に表示するテキストを指定します。

また、GUI コンポーネントからテキストを取得するには、`getComponentText` 関数を呼び出します。この関数は以下の仕様を持っています。

```
string getComponentText( int id )
```

引数 `id` には、テキストを取得する対象の GUI コンポーネント ID を指定します。この関数は、取得したテキストを `string` 型の戻り値として返します。

`getComponentText` および `setComponentText` 関数は、以下の GUI コンポーネントなどに対して使用できます。

ボタン / テキストフィールド / テキストエリア / チェックボックス /
セレクトフィールド / テキストラベル

■ グラフィックスデータの変更 / 取得

画像ラベルなど、グラフィックスデータを扱う GUI コンポーネントにおいて、表示するグラフィックスデータを変更するには `setComponentGraphics` 関数を呼び出します。この関数は以下の仕様を持っています。

```
void setComponentGraphics( int componentID, int graphicsID )
```

引数 `componentID` には、グラフィックスデータを変更する対象の GUI コンポーネント ID を指定します。続く引数 `graphicsID` には、変更後に表示するグラフィックスデータ ID を指定します。

また、GUI コンポーネントの表示するグラフィックリソース ID を取得するには、`getComponentGraphics` 関数を呼び出します。この関数は以下の仕様を持っています。

```
int getComponentGraphics( int componentID )
```

引数 `componentID` には、グラフィックスデータを取得する対象の GUI コンポーネント ID を指定します。この関数は、表示しているグラフィックスのリソース ID を返します。

`setComponentGraphics` および `getComponentGraphics` 関数は、以下の GUI コンポーネントに対して使用できます。

画像ラベル

■ 論理値など、その他のパラメータの変更 / 取得

チェックボックスのように、ON/OFF の論理値（二択値）を持つ GUI コンポーネントの値は、`setComponentBool` 関数で変更できます。この関数は以下の仕様を持っています。

```
void setComponentBool ( int id, bool state )
```


引数 id には、論理値を変更する対象の GUI コンポーネント ID を指定します。続く引数 state には、変更後の論理値を指定します。

また、GUI コンポーネントの論理値を取得するには、getComponentBool 関数を呼び出します。この関数は以下の仕様を持っています。

```
bool GetComponentBool ( int id )
```

引数 id には、論理値を取得する対象の GUI コンポーネント ID を指定します。この関数は、論理値を bool 型変数で返します。例えばチェックボックスなら、選択されていると true を、されていないければ false を返します。

setComponentBool および GetComponentBool 関数は、以下の GUI コンポーネントに対して使用できます。

チェックボックス

なお、このガイドでは扱いませんが、論理値(bool)以外にも int 値や float 値のパラメータを操作するような GUI コンポーネント(スライダーなど)も存在します。そういった GUI コンポーネントについては、ここで扱った setComponentBool / GetComponentBool と同様に、setComponentInt / GetComponentInt や、setComponentFloat / GetComponentFloat 関数などでパラメータの設定・取得を行えます。詳細は GUI ライブラリの仕様をご参照ください。

イベント処理

ここでは、ユーザーが GUI コンポーネントを操作した際に処理を行う、「イベント処理」について解説します。

■ イベントとは

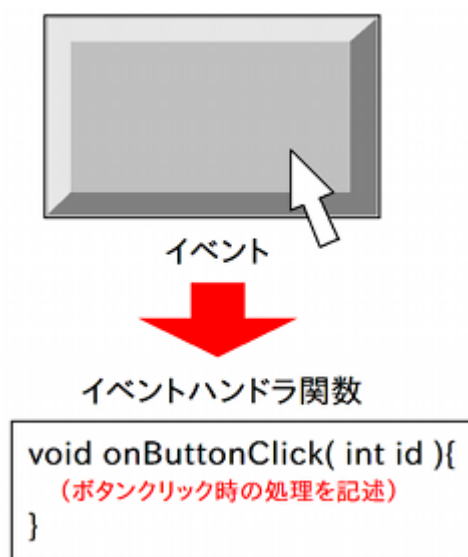
イベントとは、プログラムの外側から、プログラムに対して行われる操作の事です。例えば、ユーザーがボタンをクリックする事や、マウスを動かす事、ウィンドウを閉じる事などが挙げられます。

GUI を備えたプログラムが、アプリケーション・ソフトウェアとして意味を持つためには、イベントが発生した際に何らかの処理を実行しなければなりません。さもなければ、GUI コンポーネントはただの飾りとなってしまいます。

■ イベントハンドラ関数

VCSSL を含む一般のプログラミング言語では、イベントが発生すると、それに対応する関数がシステム側から呼ばれます。このような関数の事を一般に「イベントハンドラ関数」と呼びます。イベントハンドラ関数の内容はあらかじめ用意されているのではなく、プログラマが独自に記述します。

VCSSL では、処理したいイベントの種類に応じて特定の関数名と引数を持たせると、それが自動的にイベントハンドラ関数と見なされ、イベント発生時に呼び出されます。呼び出し時の引数には、イベントに関する情報が送られてきます。イベントハンドラ関数の中身には、イベントが発生した際に行わせたい処理を記述します。なお、VCSSL のイベントハンドラ関数は基本的に void 型であり、戻り値を返しません。



■ ウィンドウイベント処理

ウィンドウに関するイベントを処理するには、以下のイベントハンドラ関数を使用します。

イベントハンドラ関数	引数	イベント
onWindowOpen	int id	ウィンドウが起動した際に呼び出されます。
onWindowClose	int id	ウィンドウが閉じた際に呼び出されます。
onWindowMove	int id, int x, int y	ウィンドウが動いた際に呼び出されます。
onWindowResize	int id, int width, int height	ウィンドウサイズが変わった際に呼び出されます。
onWindowShow	int id	ウィンドウが表示された際に呼び出されます。
onWindowHide	int id	ウィンドウが非表示になった際に呼び出されます。

引数の id には、イベントが発生したウィンドウの GUI コンポーネント ID が渡されます。また、引数の width, height には、それぞれウィンドウの幅、高さが渡されます。

ここで特に重要なのが onWindowClose 関数です。このイベントハンドラ関数内で exit 命令をコールするようにすると、一般的なソフトウェアのような「ウィンドウを閉じると処理を終了する」という機能を実現できます。

これまで、VCSSL プログラムを終了させるには、VCSSL コンソールを手動で閉じるしかありませんでした。しかし、ウィンドウを閉じればプログラムが自動終了するようにしておけば、VCSSL コンソールは不要なので、hide() 命令で隠してしまっても問題ありません。

■ ボタンイベント処理

ボタンに関するイベントを処理するには、以下のイベントハンドラ関数を使用します。

イベントハンドラ関数	引数	イベント
onButtonClick	int id, string text	ボタンがクリックされた際に呼び出されます。

引数の id には、イベントが発生したボタンの GUI コンポーネント ID が渡されます。また、引数の text には、たボタンの文字列が渡されます。

■ セレクトフィールドイベント処理

セレクトフィールドに関するイベントを処理するには、以下のイベントハンドラ関数を使用します。

イベントハンドラ関数	引数	イベント
onSelectFieldClick	int id, string text	セレクトフィールドがクリックされ、項目が選択された際に呼び出されます。

引数の id には、イベントが発生したセレクトフィールドの GUI コンポーネント ID が渡されます。また、引数の text には、選択された項目名が渡されます。

■ チェックボックスイベント処理

チェックボックスに関するイベントを処理するには、以下のイベントハンドラ関数を使用します。

イベントハンドラ関数	引数	イベント
onCheckBoxClick	int id, bool state	チェックボックスがクリックされた際に呼び出されます。

引数の id には、イベントが発生したチェックボックスの GUI コンポーネント ID が渡されます。また、引数の state には、チェックボックスの ON/OFF 状態が渡されます。

■ キーイベント処理

GUI コンポーネント上でキーボードのキーを操作した際には、キーイベントが発生します。

ただし、キーイベントを利用するには注意が必要です。それは、キー操作がどの GUI コンポーネントに対して行われるかという点に関してです。キーイベントを利用するには、ウィンドウ上に、テキストフィールドなどのキーボード入力を扱う GUI コンポーネントを配置してはいけません。また、ボタンなどの特定のキー操作 (ENTER など) に反応する GUI コンポーネントの存在も考慮が必要です。

キーイベントを利用する際は、ウィンドウ上に配置するのはなるべくテキストラベルか画像ラベルのみに限定する事が推奨されます。

キーイベントを処理するには、以下のイベントハンドラ関数を使用します。

イベントハンドラ関数	引数	イベント
onKeyDown	int id, string key	キーが押された際に呼び出されます。 引数 key には、キーラベルの文字列が格納されています。記号のキーは、記号の文字列に変換されます。
onKeyUp	int id, string key	キーが離された際に呼び出されます。 引数 key には、キーラベルの文字列が格納されています。記号のキーは、記号の文字列に変換されます。
onKeyDown	int id, int keyCode	キーが押された際に呼び出されます。 引数 keyCode には、キーを区別するため、GUI ライブラリに 1 対 1 で定義された値が格納されています。
onKeyUp	int id, int keyCode	キーが離された際に呼び出されます。 引数 keyCode には、キーを区別するため、GUI ライブラリに 1 対 1 で定義された値が格納されています。

引数の id には、イベント発生対象の GUI コンポーネント ID が渡されます。

なお、2 つめの引数は、string 型と int 型の 2 通りがあります。前者は、キーの表面に印字された (キーラベルの) 文字列が格納されます。例えば「A」のキーを押した際は "A" が、「@」のキーを押した際は "@" が格納されます。それに対して後者は、キーに 1 対 1 で割り振られた整数値が格納され、例えば「@」のキーを押した際は KEY_AT の値が格納されます。

前者と後者にはそれぞれメリットとデメリットがあります。前者は比較的手軽に扱える上、連続したキー入力内容を string 変数末尾に追記していくような処理に便利です。反面、キー判別に key == "DOWN" といったリテラルの文字列比較を多用するのはミスタイプに弱く、厳格性に欠けるため、大きなプログラムでは好ましくありません。それに対して後者は、厳格な記述が可能です。小さなプログラムでは記述が面倒になる場合があります。

▼キーラベル文字列と、キーコードとの対応関係 (一部)

文字列	キーコード	文字列	キーコード	文字列	キーコード
A~Z	KEY_A~ KEY_Z	0 ~ 9	KEY_0 ~ KEY_9	;	KEY_SEMICOLON
				:	KEY_COLON
F1~F12	KEY_F1~ KEY_F12	@	KEY_AT	[KEY_LEFT_SQUARE_BRACKET
		/	KEY_SLASH]	KEY_RIGHT_SQUARE_BRACKET
ENTER	KEY_ENTER	+	KEY_PLUS	^	KEY_CIRCUMFLEX
SPACE	KEY_SPACE	-	KEY_MINUS	UP	KEY_UP
TAB	KEY_TAB	,	KEY_COMMA	DOWN	KEY_DOWN
SHIFT	KEY_SHIFT	.	KEY_PERIOD	LEFT	KEY_LEFT
CONTROL	KEY_CONTROL	ALT	KEY_ALT	RIGHT	KEY_RIGHT

■ マウスイベント処理

GUI コンポーネント上でマウスを動かしたり、クリックしたりすると、マウスイベントが発生します。マウスイベントを処理するには、以下のイベントハンドラ関数を使用します。

マウス操作は行えるアクションが多彩なので、それに応じてイベントハンドラも様々なものが用意されています。

イベントハンドラ関数	引数	イベント
onMouseOver	int id, int x, int y	マウスがコンポーネント領域内に入った際に呼び出されます。引数 x, y には、コンポーネントの左上を原点とするマウスカーソル座標が格納されます。
onMouseOut	int id, int x, int y	マウスがコンポーネント領域外に出た際に呼び出されます。引数 x, y には、コンポーネントの左上を原点とするマウスカーソル座標が格納されます。
onMouseMove	Int id, Int x, Int y	マウスがコンポーネント領域内で動いた際に呼び出されます。引数 x, y には、コンポーネントの左上を原点とするマウスカーソル座標が格納されます。
onMouseCkick	int id, int x, int y, int button, int count	<p>マウスのボタンがクリックされた際に呼び出されます。具体的には、環境に設定された短い時間内で、マウスボタンが「押し離し」された場合に呼ばれます。</p> <p>引数 button には、押されたボタンを区別するため、GUI ライブラリに定義された値が格納されます。定数値は、右, 左, 中央ボタンがそれぞれ MOUSE_RIGHT, MOUSE_LEFT, MOUSE_MIDDLE という名称です。</p> <p>最後の引数 count は、連続して押し離しされた回数が格納されます。つまりシングルクリックなら 1、ダブルクリックなら 2 となります。この値は、GUI ライブラリにもそれぞれ MOUSE_SINGLE, MOUSE_DOUBLE と定義されており、比較などで用いるとミスタイプを防げます。</p>
onMouseDown	int id, int x, int y, int button	<p>マウスのボタンが押された際に呼び出されます。</p> <p>引数 button には、押されたボタンを区別するため、GUI ライブラリに定義された値が格納されます。定数値は、右, 左, 中央ボタンがそれぞれ MOUSE_RIGHT, MOUSE_LEFT, MOUSE_MIDDLE という名称です。</p>

onMouseUp	int id, int x, int y int button	マウスのボタンが離された際に呼び出されます。 引数 button には、押されたボタンを区別するため、GUI ライブラリに定義された値が格納されます。定数値は、右, 左, 中央ボタンがそれぞれ MOUSE_RIGHT, MOUSE_LEFT, MOUSE_MIDDLE という名称です。
onMouseDown	int id, int x, int y int button	マウスのいずれかのボタンがドラッグされた際に呼び出されます。 引数 button には、押されたボタンを区別するため、GUI ライブラリに定義された値が格納されます。定数値は、右, 左, 中央ボタンがそれぞれ MOUSE_RIGHT, MOUSE_LEFT, MOUSE_MIDDLE という名称です。
onMouseScroll	Int id, Int degree	マウスホイールが操作された際に呼び出されます。引数 degree にはホイール回転量が渡されます。

引数の id には、イベント発生対象の GUI コンポーネント ID が渡されます。また、引数の x,y には、GUI コンポーネント左上頂点から見たマウスカーソルの位置座標が渡されます。

■ プログラム例

実際に各種イベントハンドラ関数を使用してみましょう。以下のように記述し、実行してみてください。

```
import GUI ;
import Graphics ;

// =====
// GUI コンポーネントの配置 ここから
// =====

// ( 300, 20 ) の位置に 800×600 のウィンドウを生成
int windowID = newWindow( 300, 20, 360, 480, "Hello GUI" );
```

```

// ( 10, 10 ) の位置に 100×50 のボタンを配置
int buttonID = newButton ( 10, 10, 100, 50, "PUSH" );
mountComponent( buttonID, windowID );

// ( 10, 70 ) の位置に 100×50 サイズのテキストフィールドを配置
int textFieldD = newTextField( 10, 70, 100, 30, "INPUT" );
mountComponent( textFieldD, windowID );

// ( 120, 10 ) の位置に 200×200 サイズのテキストエリアを配置
int textAreaID = newTextArea( 120, 10, 200, 200, "INPUT" );
mountComponent( textAreaID, windowID );

// ( 10, 110 ) の位置に 100×20 サイズのセレクトフィールドを配置
string text[ 3 ];
text[ 0 ] = "TYPE-A";
text[ 1 ] = "TYPE-B";
text[ 2 ] = "TYPE-C";
int selectFieldID = newSelectField( 10, 110, 100, 20, text );
mountComponent( selectFieldID, windowID );

// ( 10, 140 ) の位置に 100×20 サイズのチェックボックスを作成
int checkBoxID = newCheckBox( 10, 140, 100, 20, "TURBO", true );
mountComponent( checkBoxID, windowID );

// ( 10, 170 ) の位置に 100×20 サイズのテキストラベルを配置
int textLabelID = newTextLabel( 10, 170, 100, 20, "Hello GUI ! " );
mountComponent( textLabelID, windowID );

// 「Test.png」という名前の PNG 形式画像ファイルを読み込み
int graphicsID = newGraphics( "Test.png" );
// ( 10, 220 ) の位置に 310×200 サイズの画像ラベルを配置
int imageLabelID = newImageLabel( 10, 220, 310, 200, graphicsID );
mountComponent( imageLabelID, windowID );
// =====
// GUI コンポーネントの配置 ここまで
// =====

```



```

// =====
// イベントハンドラ ここから
// =====
// ウィンドウを閉じた際に呼び出される
void onWindowClose( int id ){
    alert( "プログラムを終了します。" );
    exit( );
}

// ボタンクリック時に呼び出される
void onButtonClick( int id, string text ){
    alert( "ボタン " + text + " がクリックされました。" );
}

// セレクトフィールド選択時に呼び出される
void onSelectFieldClick( int id, string text ){
    alert( text + " が選択されました。" );
}

// チェックボックス選択時に呼び出される
void onCheckBoxClick( int id, bool state ){
    alert( "選択状態が " + state + " になりました。" );
}

// マウスドラッグ時に呼び出される
void onMouseDrag( int id, int x, int y, int button ){
    // ドラッグされたコンポーネントの判別
    if( id == imageLabelID ){
        // マウスボタンの判別
        if( button == MOUSE_RIGHT ){
            println( "右ドラッグ X=" + x + " Y=" + y );
        }else if( button == MOUSE_LEFT ){
            println( "左ドラッグ X=" + x + " Y=" + y );
        }
    }
}
}

```

```

// マウスクリック時に呼び出される
void onMouseClick( int id, int x, int y, int button, int count ){
    // クリックされたコンポーネントの判別
    if( id == imageLabelID ){

        // マウスボタンの判別
        if( button == MOUSE_RIGHT ){
            println( "右クリック X=" + x + " Y=" + y );
        }else if( button == MOUSE_LEFT ){
            println( "左クリック X=" + x + " Y=" + y );
        }

        // ダブルクリックの判別
        if( count == MOUSE_DOUBLE ){
            alert( "ダブルクリックされました。" );
        }

    }
}

// キー入力時に呼び出される(キーラベル文字列取得)
void onKeyDown( int id, string key ){
    println( "キー入力 =" + key );
}

// キー入力時に呼び出される(キーコード取得)
void onKeyDown( int id, int keyCode ){
    if( keyCode == KEY_SPACE ){
        alert( "スペースキーが押されました。" );
    }
}

// =====
// イベントハンドラ ここまで
// =====

```

このプログラムを実行すると、ウィンドウが表示され、その上に様々な GUI コンポーネントが表示されます。各 GUI コンポーネントを操作すると、それに応じた処理が実行されます。

▼実行結果

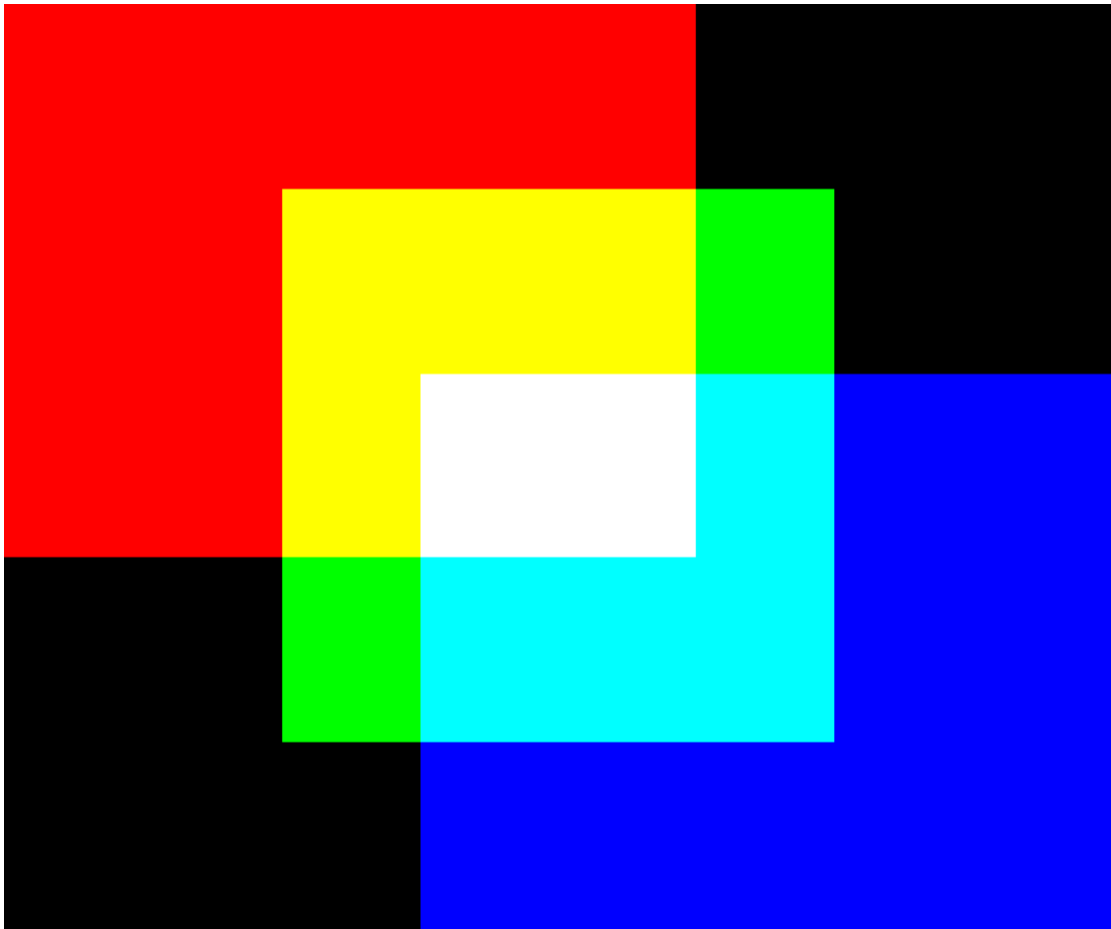


※この画像は模式図です。実際の GUI デザインは、オペレーティングシステムの種類や、VCSSL 処理システムのバージョン、その他環境によって異なります。

第 3 部

2DCG

- 2 次元コンピューターグラフィックス -



この部では、2次元コンピューターグラフィックスを用いたVCSSLプログラムを開発する方法を解説します。この文書の内容を扱うには、VCSSLの基本的な文法をすでに習得している必要があります。

2 次元コンピューターグラフィックス

■ 2 次元コンピューターグラフィックス（2DCG）とは

コンピューターグラフィックス、いわゆる CG とは、コンピューター上の計算処理によって画像や映像を作成する技術の事です。

CG には、平面的な表現を扱う 2 次元コンピューターグラフィックス — 2DCG と、立体的な表現を扱う 3 次元コンピューターグラフィックス — 3DCG が存在します。かつては PC などでの描画処理は 2DCG が主流でしたが、近年のコンピューター性能や 3D 関連技術の目覚ましい進歩により、現在では 3DCG も一般的なものとなりました。

VCSSL では、2DCG と 3DCG の両方を標準ライブラリでサポートしていますが、本文書では 2DCG を中心に扱います。2DCG の制御は、画像の加工や模式図の描画、2D アニメーション、2D ゲーム開発などで必要となります。

■ VCSSL プログラムで 2DCG を扱うには

VCSSL プログラムで 2DCG を扱うには、VCSSL 標準ライブラリの中から、Graphics ライブラリと Graphics2D ライブラリをインポートする必要があります。また、グラフィックスはウィンドウ上に表示する事が多いので、同時に GUI ライブラリをインポートしておく事を推奨します（画像をファイルに保存する場合など、ウィンドウなどを利用しない場合は不要です）。

これらのライブラリをインポートするには、プログラムの先頭行に以下のように記述します。

```
import Graphics ;  
import Graphics2D ;  
import GUI ;
```

これで、プログラム中から 2DCG を扱うための関数が利用可能になります。

レンダラーの生成

2DCG における基本的な作業は、「レンダラー（描画エンジン）」というものを操作し、「グラフィックスデータ」を加工していくといった流れの繰り返しになります。ここでは、これら 2 つの重要な概念について解説します。

■ グラフィックスデータとレンダラー

VCSSL で 2DCG を扱うには、描画内容を扱うデータ「グラフィックスデータ」と、「レンダラー（描画エンジン）」というものについて理解しておく必要があります。しかし、あまり難しいものではありません。

・描画内容を扱うデータ - グラフィックスデータ

例えば現実世界で絵を描くには、絵の内容を保持するための画用紙や、描くためのペンをはじめ、様々なものがが必要です。プログラミングによる描画でもこれらに相当するものがなくて、言語や環境によって異なりますが、イメージ、バッファ、グラフィックスコンテキストといったものが 필요합니다。VCSSL では、これらを一つにまとめて扱います。これを「グラフィックスデータ」と呼びます。つまり、絵を描くために必要なものが付属した、すぐに使える画用紙のようなものです。

・レンダラー（描画エンジン）

レンダラー（描画エンジン）とは、グラフィックスデータに対して、高度な絵を描き込むためのものです。原理的にはグラフィックスデータだけでも描画は可能ですが、レンダラーを使用したほうが高度な描画が可能です。つまり、「絵を描く人」のようなものです。レンダラーには 2DCG 用と 3DCG 用が存在します。平面的な絵が上手い人と、立体的な絵が上手い人が居るわけです。

■ グラフィックスデータの生成

まず、2DCG で描画するグラフィックスデータを生成します。これには Graphics ライブラリの newGraphics 関数を使用します。

```
int newGraphics()
```

この関数は、引数を何も指定しなかった場合、何も描かれていない空白の内容を保持するメモリー領域を確保し、グラフィックスデータ ID を返します。グラフィックスデータ ID とは、グラフィックスデータに割り振られる固有の識別番号です。

■ レンダラー(描画エンジン)の生成

上の newGraphics 関数で生成したグラフィックスデータは、空白で何も絵がありません。そこで、このグラフィックスデータに 2DCG を描画するためのレンダラー(描画エンジン)を生成します。これには newGraphics2DRenderer 関数を使用します。

```
int newGraphics2DRenderer ( int width, int height, int graphicsID )
```

引数の width と height で描画する絵の大きさを、graphicsID で描画対象のグラフィックスデータ ID を指定します。この graphicsID には、newGraphics 関数で確保したものを指定します。

■ プログラム例

それでは実際にレンダラーを生成してみましょう。以下のようにプログラムを記述し、実行してみてください。

```
import Graphics ;
import Graphics2D ;

// グラフィックスデータの生成
int graphicsID = newGraphics( ) ;

// レンダラーの生成
int rendererID = newGraphics2DRenderer( 800, 600, graphicsID ) ;
```


このプログラムを実行すると、レンダラーやグラフィックスデータは準備されますが、まだ画面には何も表示されません。それは、グラフィックスデータの内容を表示する画面をまだ作成していないからです。引き続き、表示画面の準備を行きましょう。

表示画面の生成

描画したグラフィックスデータは、表示画面やファイルに出力しなければ意味がありません。ここでは、グラフィックスの画面出力処理を扱います。

■ 表示画面の生成

実際にプログラム中で 2DCG を活用するためには、描画したグラフィックスデータを表示する画面が必要になります。そこで、GUI ライブラリの関数を用いて表示画面を生成します。

・ウィンドウの生成

まず、ウィンドウを生成するには、GUI ライブラリの `newWindow` 関数を使用します。この関数は以下の仕様を持っています。

```
int newWindow( int x, int y, int width, int height, string title )
```

引数の `x` と `y` でウィンドウ左上頂点の座標を指定し、`width` と `height` でウィンドウの幅と高さを指定します。また、引数 `title` でウィンドウのタイトルバー表示文字列を指定します。

この関数をコールすると、ウィンドウが生成され、それに割り当てられたコンポーネント ID が `int` 型で返されます。なお、コンポーネント ID とは、全ての GUI 部品に割り当てられる識別番号の事です。

・画像ラベルの生成

ウィンドウ上には、グラフィックスデータの内容を表示するための GUI 部品である、画像ラベルを配置します。これには `newImageLabel` 関数を使用します。この関数は以下の仕様を持っています。

```
int newImageLabel ( int x, int y, int width, int height, int graphicsID )
```

引数には、`x` と `y` で画像ラベル左上頂点の座標を指定し、`width` と `height` で画像ラベルの幅と高さを指定します。最後の引数 `graphicsID` には、表示対象のグラフィックスデータの ID を指定しま

す。

この関数をコールすると、画像ラベルが生成され、それに割り当てられたコンポーネント ID が int 型で返されます。

■ 画面の再描画

GUI の表示内容を変更するには、GUI の再描画(リペイント)を行う必要があります。画面に表示すべき内容(グラフィックスデータなど)を変更した際は、それを表示する GUI(画像ラベルやウィンドウなど)も再描画しなければ、画面表示が変わらないままになってしまいます。ご注意ください。

GUI の再描画には、paintComponent 関数を使用します。

```
int paintComponent ( int componentID )
```

引数の componentID には、再描画したい GUI のコンポーネント ID (これは newWindow 関数や newImageLabel 関数などが返す値) を指定します。

2DCG を扱う際は、レンダラーを一通り操作した後に、必ずこの paintComponent 関数をコールするようにしてください。

■ プログラム例

それでは実際にウィンドウを生成し、グラフィックスデータを画面に表示してみましょう。以下のよう
にプログラムを記述し、実行してみてください。

```
import Graphics ;  
import Graphics2D ;  
import GUI ;  
  
// グラフィックスデータの生成  
int graphicsID = newGraphics( ) ;  
  
// レンダラーの生成
```

```
int rendererID = newGraphics2DRenderer( 800, 600, graphicsID );

// 800×600 サイズのウィンドウの生成
int windowID = newWindow( 0, 0, 800, 600, " Hello , 2DCG ! " );

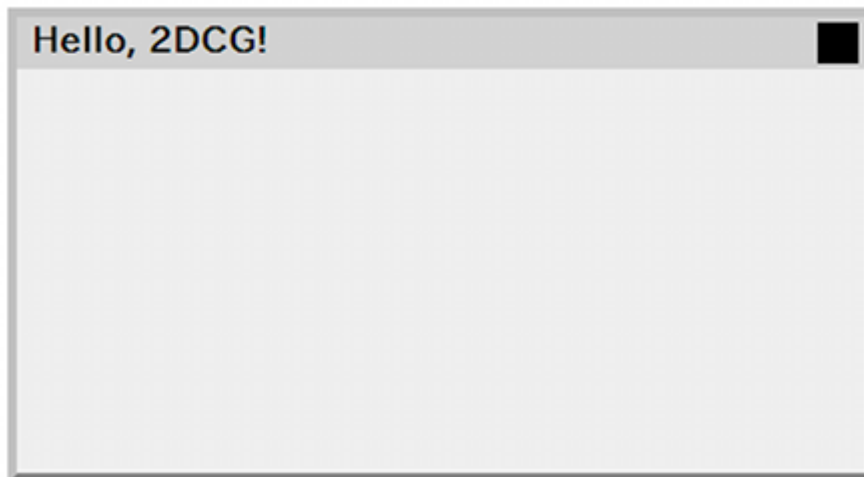
// 画像ラベルの生成
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID );

// 画像ラベルをウィンドウに配置
mountComponent( labelID, windowID );

// GUI の描画
paintComponent( labelID );
paintComponent( windowID );
```

このプログラムを実行すると、800×600 の大きさで、「 Hello , 2DCG ! 」というタイトルのウィンドウが立ち上がります。そのウィンドウには、生成したグラフィックスが表示されています。ただし、グラフィックスデータにはまだ何も描画していないので、ただ空白が表示されるだけです。

▼実行結果



※画像は模式図です。実際のウィンドウのデザインは、オペレーションシステムの種類や、実行環境のバージョン、及びその他環境などにより異なります。

グラフィックスのファイル出力

描画したグラフィックスデータは、表示画面やファイルに出力しなければ意味がありません。ここでは、グラフィックスのファイル出力処理を扱います。

■ グラフィックスのファイル出力

描画したグラフィックスデータは、画面に表示するだけでなく、Graphics ライブラリの `exportGraphics` 関数を使用し、BMP/PNG/JPEG 形式の画像ファイルに出力する事も可能です。

```
void exportGraphics( int graphicsID, string fileName, string format )
```

最初の引数 `graphicsID` には出力するグラフィックスデータの ID を、続いて `fileName` にファイル名、`format` にファイル形式を指定します。ファイル形式には "BMP", "PNG", "JPEG" のどれかを指定します。

なお JPEG 形式では、以下のように末尾に引数をもう一つ追加し、画質をパーセンテージで指定する事が可能です。

```
void exportGraphics( int graphicsID, string fileName, string format, float quality )
```

追加された末尾の引数 `quality` には、画質をパーセンテージで指定します。

■ プログラム例

例として、まだ何も描画していない画像を、PNG 形式で出力してみましょう。

```
import Graphics ;  
import Graphics2D ;
```

```
import GUI ;

// グラフィックスデータの生成
int graphicsID = newGraphics( ) ;

// レンダラーの生成
int rendererID = newGraphics2DRenderer( 800, 600, graphicsID ) ;

/* 画像ファイルを出力 */
exportGraphics( graphicsID, "Test.png", "PNG" ) ;
```

このプログラムを実行しても画面上は何も起こらないので、起動したら数秒だけ待ってコンソールを閉じてください。その後、プログラムのあるフォルダに「 Test.png 」という画像ファイルが生成されているので、画像ビューアソフトなどで開いてください。

上のプログラムでは何も描画していないので、透明な画像が表示されるはずです。

内容のクリアと背景色設定

ここまでで、2DCG を扱う準備が整いました。ここからはレンダラーを操作し、2DCG を描画していく事になります。その最初の一步として、内容のクリアと、背景色の設定を行ってみましょう。

■ 内容のクリア

何らかのものが描画されたグラフィックスデータの内容をクリアするには、clearGraphics2D 関数を使用します。

```
void clearGraphics2D ( int rendererID )
```

最初の rendererID 引数では、操作対象のレンダラーID を指定します。この関数をコールすると、レンダラーは、グラフィックスデータの内容をすべて背景色一色にクリアします。

未設定状態の背景色は透明であり、そのままではウィンドウの基盤面が透けて見える状態となります。ウィンドウの基盤面の色は、動作環境によって白であったりグレーであったりするので、グラフィックスデータをウィンドウに表示して使用する場合、背景色を不透明な色に設定しておく事が推奨されます。

背景色は、以下に述べるように、任意の色に設定する事ができます。

■ 背景色の設定

背景色を設定するには、setGraphics2DColor 関数を使用します。

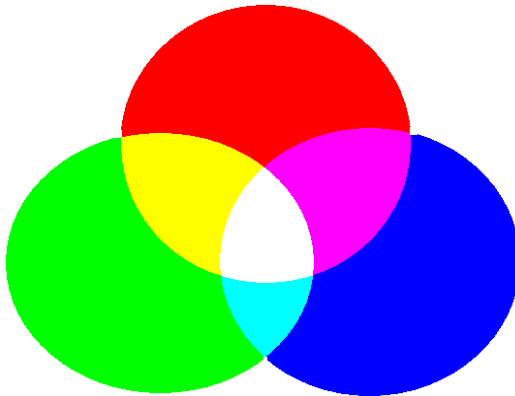
```
void setGraphics2DColor (
    int rendererID,
    int red, int green, int blue, int alpha
)
```

最初の `rendererID` 引数では、設定対象のレンダラーID を指定します。続く引数では、背景の色成分を指定します。色成分はそれぞれ 0～255 の範囲で指定します。色成分の形式は、RGBA 形式をサポートしています。

・RGBA 形式とは

RGBA 形式とは、色の三原色である赤 (Red)、緑 (Green)、青 (Blue) の色成分に、アルファ値 (Alpha) を加えた形式です。アルファ値は色の透明度を表す数値で、0 で完全透明になり、最大にすると不透明になります。

それぞれの色成分からの色の合成は、加法混色によって行われます。これは光の重ね合わせと同じ混色方式であり、絵の具の混ぜ合わせ (減法混色) では無い事にご注意ください。例として、(赤, 緑, 青) = (255, 255, 255) は、黒ではなく白になります。また、(255, 255, 0) は黄色に、(0, 255, 255) は水色に、(255, 0, 255) はマゼンタになります。



■ プログラム例

実際に、背景色を青に設定してみましょう。以下のように記述し、実行してみてください。

```
import Graphics ;
import Graphics2D ;
import GUI ;

// グラフィックスデータとレンダラーの生成
int graphicsID = newGraphics( ) ;
int rendererID = newGraphics2DRenderer( 800, 600, graphicsID ) ;
```



```
// 表示画面の生成
int windowID = newWindow( 0, 0, 800, 600, " Hello 2DCG ! " );
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID );
mountComponent( labelID, windowID );

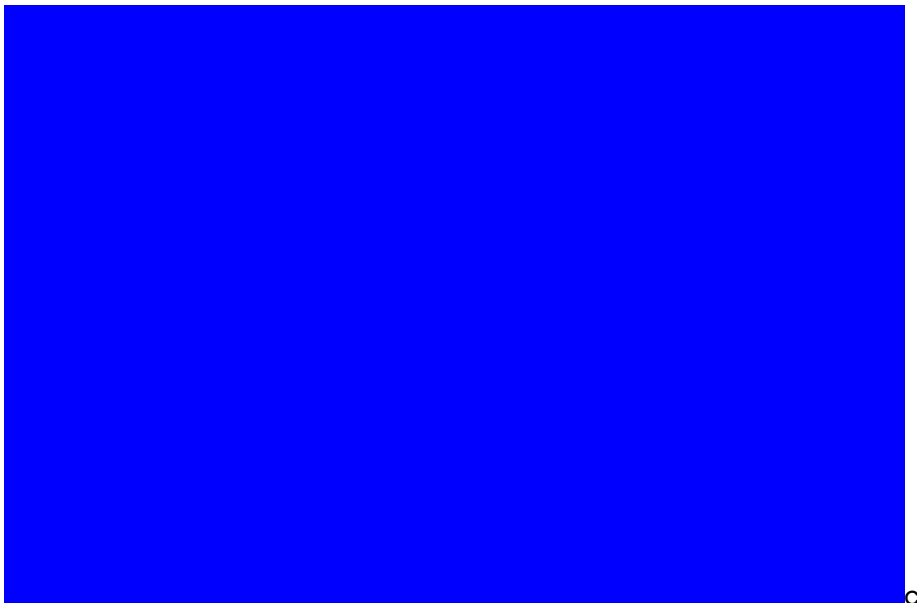
// 背景色を青に設定
setGraphics2DColor( rendererID, 0, 0, 255, 255 );

// グラフィックスを背景色でクリア
clearGraphics2D( rendererID );

// GUI の描画
paintComponent( labelID );
paintComponent( windowID );
```

このプログラムを実行すると、画面に真っ青なウィンドウが表示されます。

▼実行結果



直接描画

レンダラーが描画を行う方法には、直接描画とスプライト描画の 2 通りが存在します。ここでは、直接描画を扱います。

■ 直接描画

直接描画とは、グラフィックスデータに線や幾何学図形などを直接描画していく、単純な描画方法です。直接描画では、描画関数をコールするたびに、グラフィックスデータ上に図形が描画されていき、すでに描画されていた箇所と重なると上描きされます。ちょうど画用紙の上にペンで絵を書いていくような手順に似ています。

直接描画は、メモリー使用量が少なく済むので、描画するものが比較的少なく、かつ静的な内容を描画するのに向いています。使用方法も手軽で直感的です。

直接描画の弱点として、アニメーションなどで時間あたりの描画頻度が多い場合、描画速度が低下する事が挙げられます。この弱点は、特に描画するものが非常に多い場合に顕著となります。これは、描画するものの個数だけ描画関数をコールしなければならないため、関数コールの負荷が効いてくるためです。その場合は、後の章で扱うスプライト描画を使用した方が有利となります。

■ 描画色の設定

直接描画を行う前に、まず描画する色を指定します。これには `setDrawColor` 関数を使用します。

```
void setDrawColor (  
    int rendererID,  
    int red, int green, int blue, int alpha  
)
```

最初の引数 `rendererID` でレンダラーの ID を、続く引数で描画色の色成分を指定します。色成分は RGBA 形式で、それぞれ 0~255 の範囲の値を指定します。

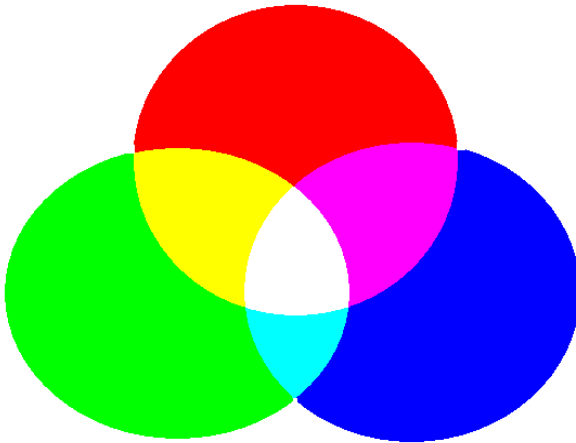
この `setDrawColor` 関数は、描画色を変更したいタイミングでコールします。コール後、この関数

で設定した色で直接描画が行われます。再度コールすると、それ以降は新しい色で直接描画が行われます。

・RGBA 形式とは

RGBA 形式とは、色の三原色である赤 (Red)、緑 (Green)、青 (Blue) の色成分に、アルファ値 (Alpha) を加えた形式です。アルファ値は色の透明度を表す数値で、0 で完全透明になり、最大にすると不透明になります。

それぞれの色成分からの色の合成は、加法混色によって行われます。これは光の重ね合わせと同じ混色方式であり、絵の具の混ぜ合わせ (減法混色) では無い事にご注意ください。例として、(赤, 緑, 青) = (255, 255, 255) は、黒ではなく白になります。また、(255, 255, 0) は黄色に、(0, 255, 255) は水色に、(255, 0, 255) はマゼンタになります。



■ 描画関数のコール

描画色の設定が完了したら、実際にグラフィックスデータに図形などを描画していきます。これには描画関数をコールします。

描画関数は、レンダラーに描画を指示するための関数で、描画する図形に応じて様々な種類のものが存在します。

各種描画関数

ここでは、グラフィックスデータに図形を描画していくための、各種描画関数について扱います。

■ 点の描画

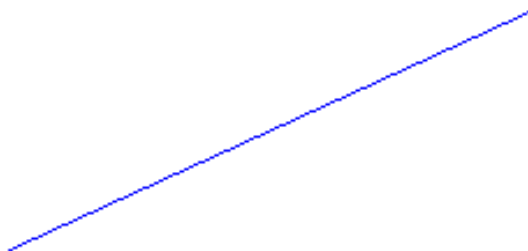


点を描画するには、drawPoint 関数を使用します。

```
void drawPoint(  
    int rendererID,  
    int x, int y, int radius,  
    bool fill  
)
```

最初の引数 rendererID には、レンダラーの ID を指定します。続く引数 x, y には点の中心の座標を、radius には点の半径を指定します。最後の引数 fill は、true の場合に塗りつぶし描画、false の場合に境界線だけの描画となります。

■ 線の描画



線を描画するには、drawLine 関数を使用します。

```
void drawLine(  
    int rendererID,  
    int x1, int y1, int x2, int y2  
)
```

最初の引数 rendererID には、レンダラーの ID を指定します。続く引数 x1, x2 には始点の座標を、x2, y2 には終点の座標を指定します。

■ 長方形の描画



長方形を描画するには、drawRectangle 関数を使用します。

```
void drawRectangle (  
    int rendererID,  
    int x, int y, int width, int height,  
    bool fill  
)
```

最初の引数 rendererID には、レンダラーの ID を指定します。続く引数 x, y には左上頂点の座標を、幅と高さを指定します。最後の引数 fill は、true の場合に塗りつぶし描画、false の場合に境界線のための描画となります。

■ 楕円の描画



楕円(だえん)を描画するには、drawEllipse 関数を使用します。

```
void drawEllipse (  
    int rendererID,  
    int x, int y, int width, int height,  
    bool fill  
)
```

最初の引数 rendererID には、レンダラーの ID を指定します。続く引数 x, y には左上頂点の座標を、幅と高さを指定します。ここで指定した頂点領域に内接する楕円が描画されます。最後の引数 fill は、true の場合に塗りつぶし描画、false の場合に境界線のための描画となります。

■ 多角形の描画

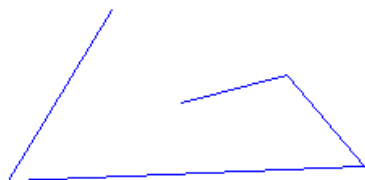


多角形を描画するには、drawPolygon 関数を使用します。

```
void drawPolygon (  
    int rendererID,  
    int x[ ], int y[ ],  
    bool fill  
)
```

最初の引数 `rendererID` には、レンダラーの ID を指定します。続く引数 `x[]`, `y[]` には、多角形を構成する頂点の座標を、配列で指定します。最後の引数 `fill` は、`true` の場合に塗りつぶし描画、`false` の場合に境界線のみ描画となります。

■ 折れ線の描画



折れ線を描画するには、`drawPolyline` 関数を使用します。

```
void drawPolyline (  
    int rendererID,  
    int x[ ], int y[ ]  
)
```

最初の引数 `rendererID` には、レンダラーの ID を指定します。続く引数 `x[]`, `y[]` には、多角形を構成する頂点の座標を、配列で指定します。

■ テキストの描画

テキストを描画するには、`drawText` 関数を用います。

```
void drawText (  
    int rendererID,  
    int x, int y, int lineWidth, int lineHeight  
)
```

最初の引数 `rendererID` には、レンダラーの ID を指定します。続く引数 `x`, `y` には、テキスト始点のアンダーライン(下線)の座標を指定します。座標は画面左上を原点(0, 0)とし、右方向に X 軸、下

方向に Y 軸で指定します。最後の lineWidth と lineHeight には、テキストの行幅と行間隔を指定します。描画時に行幅を超えた長いテキストは、自動で改行されて次の行に描画されます。

テキストの描画に使用するフォントの大きさを変更したい場合は、setDrawFontSize 関数を使用します。

```
void setDrawFontSize ( int rendererID, int fontSize )
```

最初の引数 rendererID には、レンダラーの ID を指定します。続く引数 fontSize には、フォントの大きさを pt (ポイント) 単位で指定します。

pt というのはフォントの大きさを指定する一般的な単位で、画面表示する際に一つの基準となるのは大体 12pt 前後でしょう。10pt だとやや細かい文字、15pt だと少し大きめの文字となります。ここで注意しなければならないのは、1pt が何ピクセルに対応するかというのは環境によって異なるという点です。著しく異なる事はありませんが、一文字あたり数ピクセルずれる事はよくあります。従って、グラフィックス中にテキストを描画する際は、あまりシビアではなく、ある程度ずれる事を想定したレイアウトをする事が推奨されます。

■ 画像の描画

画像を描画するには、drawImage 関数を用います。

```
void drawImage (
    int rendererID,
    int x, int y, int width, int height,
    int graphicsID
)
```

最初の引数 rendererID には、レンダラーの ID を指定します。続く引数 x, y には、画像を描画する左上頂点の座標を、width, height には幅と高さを指定します。最後の引数 graphicsID には、画像の内容を格納するグラフィックスデータの ID を指定します。

引数に指定するグラフィックス ID には、Graphics ライブラリの newGraphics(string fileName)関数で画像ファイルから読み込んだものや、別のレンダラーで描画したものなどが利用できます。

■ プログラム例

それでは、上で扱った各種描画関数を使用してみましょう。以下のように記述し、実行してみてください。

なお、画像の描画を行う場合は、プログラムと同じフォルダに「Test.png」という名前の PNG 形式画像ファイルを置いてから実行してください。

```
import Graphics ;
import Graphics2D ;
import GUI ;

// グラフィックスデータとレンダラーの生成
int graphicsID = newGraphics( ) ;
int rendererID = newGraphics2DRenderer( 800, 600, graphicsID ) ;

// 表示画面の生成
int windowID = newWindow( 0, 0, 800, 600, " Hello 2DCG ! " );
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID ) ;
mountComponent( labelID, windowID );

// 背景色を白に設定してクリア
setGraphics2DColor( rendererID, 255, 255, 255, 255 ) ;
clearGraphics2D( rendererID ) ;

// =====
//      描画処理 ここから
// =====

// (0,0)と(100,100)を結ぶ線を描画(赤色)
setDrawColor( rendererID, 255, 0, 0, 255 ) ;
drawLine( rendererID, 0, 0, 100, 100 ) ;
```

```

// (100,100)を左上とする(500×300)の長方形を描画(青色)
setDrawColor( rendererID, 0, 0, 255, 255 );
drawRectangle( rendererID, 100, 100, 500, 300, true );

// (100,300) を左上とする(300×200)の楕円を描画(マゼンタ)
setDrawColor( rendererID, 255, 0, 255, 255 );
drawEllipse( rendererID, 100, 300, 300, 200, true );

// 多角形、折れ線スプライト用の x 配列、y 配列を用意
int x[ 3 ]; x[ 0 ] = 100 ; x[ 1 ] = 300 ; x[ 2 ] = 300 ;
int y[ 3 ]; y[ 0 ] = 100 ; y[ 1 ] = 100 ; y[ 2 ] = 300 ;

// x 配列、y 配列を頂点とする多角形を描画(黄色)
setDrawColor( rendererID, 255, 255, 0, 255 );
drawPolygon( rendererID, x, y, true );

// x 配列、y 配列を頂点とする折れ線を描画(赤色)
setDrawColor( rendererID, 255, 0, 0, 255 );
drawPolyline( rendererID, x, y );

// (300,50)にテキストを描画(黒色)
setDrawFontSize( rendererID, 30 ); //文字サイズを 30pt に
string text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
setDrawColor( rendererID, 0, 0, 0, 255 );
drawText( rendererID, 300, 50, 250, 35, text );

// 画像ファイル「Test.png」を読み込んで描画
//int testGraphics = newGraphics( "Test.png" );
//drawImage( rendererID, 300, 100, 300, 300, testGraphics );
// ↑「Test.png」を用意してから、コメント記号を外して実行してください

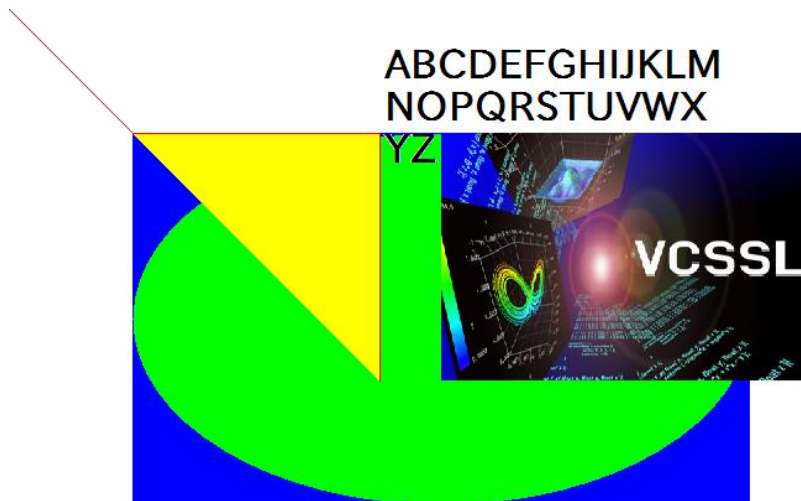
// =====
// 描画処理 ここまで
// =====

```

```
/* 画面の描画 */  
paintComponent( labelID );  
paintComponent( windowID );
```

このプログラムを実行すると、白いウィンドウが表示され、その上に様々な図形が表示されます。

▼ 実行結果



※文字に使用されるフォントは、環境によって異なります。フォントを明示したい場合には
setSpriteFont(string fontName)関数を使用してください。

スプライト描画

レンダラーが描画を行う方法には、これまでに扱ってきた直接描画の他に、スプライト描画という方法が存在します。ここでは、このスプライト描画を扱います。

■ スプライト描画

スプライト描画とは、本来はゲームなどでの 2DCG 全盛期に、ハードウェアで実装されていたスプライト描画という機能を、ソフトウェア的に実装したものです。スプライト描画の概念は、アニメ映像の制作過程における、セル画の概念と非常によく似ています。具体的には、まず描画する図形や画像の数だけ、スプライト(またはスプライト)と呼ばれる「層」をレンダラーに登録しておきます。そして、描画時にそれらのスプライトをレンダラーで重ねて合成し、1 枚の絵を生成します。

スプライト描画が特に威力を発揮するのは、アニメーション処理などを行う時です。例えば、アニメーションの中に、動く図形(や画像)と動かない図形があったとしましょう。この場合、1 回の画面に表示する絵を直接描画で用意すると、まず背景をクリアし、そして全ての図形を draw 系関数で描画しなければなりません。そこでスプライト描画を使用すると、最初に全ての図形をスプライトとして登録しておき、あとは毎回の描画時に、動かしたいスプライトだけを動かせば済むので、プログラムの記述が簡単になりますし、処理速度の面でも有利となります。

スプライト描画と直接描画は、組み合わせて使用する事も可能です。それにはレンダラーを複数用意し、あるレンダラーの描画したグラフィックスデータを、別のレンダラーにスプライトとして登録する、または drawGraphics で描き込むなどの方法を使用します。適材適所でそれぞれの描画方法を組み合わせる事で、効率的なプログラミングが可能となります。

■ スプライトの基本操作

直接描画と異なり、スプライトはプログラム実行中に動かしたり、色を変えたりなど、動的に活用します。そこで、具体的なスプライトの生成を扱う前に、スプライトを動的に活用するための基本操作について述べておきます。

・スプライトの生成

実際のスプライトには様々な種類のものがありますが、それらは全て new~Sprite 関数で生成します。~の部分にはスプライトの種類に固有の名称が入ります。具体的な生成は次章で扱います。

```
void new~Sprite ( ~ )
```

この関数は、スプライトを生成し、そのスプライトに固有の識別番号を返します。

・スプライトの登録

生成したスプライトは、レンダラーに登録しなければ描画されません。レンダラーにスプライトを登録するには、mountSprite 関数を使用します。

```
void mountSprite ( int SpriteID, int rendererID )
```

最初の引数 SpriteID には、登録するスプライトのスプライト ID を指定します。スプライト ID とは、スプライトに割り振られる識別番号で、後に述べるスプライト生成関数が戻り値として返します。続く引数 rendererID には、登録先レンダラーの ID を指定します。

なお、VCSSL3.0 以前の世代では、配置には addSprite 関数を使用していました。しかし、add~ という関数名としては引数の順序が混乱を招くという理由により、VCSSL3.1 以降では、関数名を上記の mountSprite に変えたものが追加されました。つまり addSprite 関数と mountSprite 関数は、名称が異なるだけで全く同一のものです。

・スプライトの描画

レンダラーにすべてのスプライトを登録したら、それらのスプライトを重ね合わせて合成し、1 枚のグラフィックスに描画する必要があります。これには paintGraphics2D 関数を使用します。

```
void paintGraphics2D ( int rendererID )
```

引数の rendererID には、レンダラーの ID を指定します。この関数をコールするまで、スプライトの合成は行われないのでご注意ください。

各種スプライト

スプライトには、線や幾何学図形を描画するもの、テキストを描画するもの、画像を描画するものなど、様々な種類のものが存在します。ここでは、各種スプライトの生成と使用方法を扱います。

■ 点スプライト

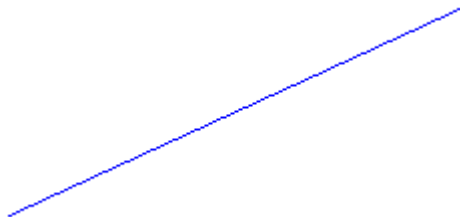


点スプライトを生成するには、newPointSprite 関数を使用します。

```
void newPointSprite ( int x, int y, int radius)
```

引数 x, y には点の中心の座標を、radius には点の半径を指定します。

■ 線スプライト

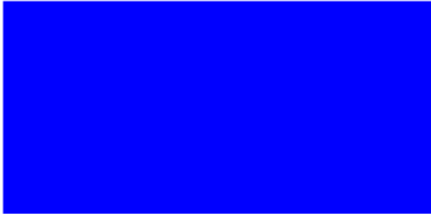


線スプライトを生成するには、newLineSprite 関数を使用します。

```
void newLineSprite ( int x1, int y1, int x2, int y2 )
```

引数 x1, x2 には始点の座標を、x2, y2 には終点の座標を指定します。

■ 長方形スプライト

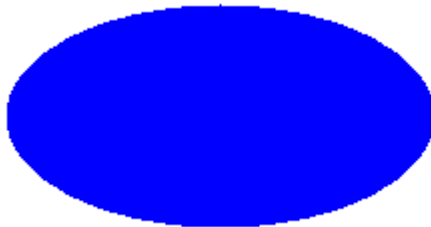


長方形スプライトを生成するには、newRectangleSprite 関数を使用します。

```
void newRectangleSprite ( int x, int y, int width, int height, bool fill )
```

引数 x, y には左上頂点の座標を、幅と高さを指定します。最後の引数 fill は、true の場合に塗りつぶし描画、false の場合に境界線だけの描画となります。

■ 楕円スプライト



楕円スプライトを生成するには、newEllipseSprite 関数を使用します。

```
void newEllipseSprite ( int x, int y, int width, int height, bool fill )
```

引数 x, y には左上頂点の座標を、幅と高さを指定します。ここで指定した頂点領域に内接する楕円が描画されます。最後の引数 fill は、true の場合に塗りつぶし描画、false の場合に境界線だけの描画となります。

■ 多角形スプライト

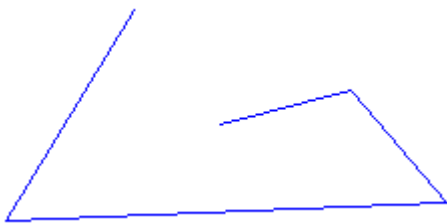


多角形スプライトを生成するには、newPolygonSprite 関数を使用します。

```
void newPolygonSprite ( int x[ ], int y[ ], bool fill )
```

引数 x[], y[] には、多角形を構成する頂点の座標を、配列で指定します。最後の引数 fill は、true の場合に塗りつぶし描画、false の場合に境界線のための描画となります。

■ 折れ線スプライト



折れ線スプライトを生成するには、newPolylineSprite 関数を使用します。

```
void newPolylineSprite ( int x[ ], int y[ ] )
```

引数 x[], y[] には、多角形を構成する頂点の座標を、配列で指定します。

■ テキストスプライト

テキストスプライトを生成するには、newTextSprite 関数を使用します。


```
void newTextSprite ( int x, int y, int lineWidth, int lineHeight )
```

最初の引数 `rendererID` には、レンダラーの ID を指定します。続く引数 `x`, `y` には、テキスト始点のアンダーライン(下線)の座標を指定します。座標は画面左上を原点(0, 0)とし、右方向に X 軸、下方向に Y 軸で指定します。最後の `lineWidth` と `lineHeight` には、テキストの行幅と行間隔を指定します。描画時に行幅を超えた長いテキストは、自動で改行されて次の行に描画されます。

・フォントサイズの設定

テキストの描画に使用するフォントの大きさを変更したい場合は、`setSpriteFontSize` 関数を使用します。

```
void setSpriteFontSize ( int SpriteID, int fontSize )
```

最初の引数 `SpriteID` には、スプライトの ID を指定します。続く引数 `fontSize` には、フォントの大きさを pt(ポイント)単位で指定します。

pt というのはフォントの大きさを指定する一般的な単位で、画面表示する際に一つの基準となるのは大体 12pt 前後でしょう。10pt だとやや細かい文字、15pt だと少し大きめの文字となります。ここで注意しなければならないのは、1pt が何ピクセルに対応するかというのは環境によって異なるという点です。著しく異なる事はありませんが、一文字あたり数ピクセルずれる事はよくあります。従って、テキストを描画する際は、あまりシビアではなく、ある程度ずれる事を想定したレイアウトをする事が推奨されます。

■ 画像スプライト

画像スプライトを生成するには、`newImageSprite` 関数を用います。

```
void newImageSprite ( int x, int y, int width, int height, int graphicsID )
```

引数 `x`, `y` には、画像を描画する左上頂点の座標を、`width`, `height` には幅と高さを指定します。最

後の引数 `graphicsID` には、描画するグラフィックスデータの ID を指定します。

引数に指定するグラフィックス ID には、Graphics ライブラリの `newGraphics(string fileName)`関数で画像ファイルから読み込んだものや、別のレンダラーで描画したものなどが利用できます。

■ プログラム例

それでは、上で扱った各種描画関数を使用してみましょう。以下のように記述し、実行してみてください。

なお、画像の描画を行う場合は、プログラムと同じフォルダに「Test.png」という名前の PNG 形式画像ファイルを置いてから実行してください。

```
import Graphics ;
import Graphics2D ;
import GUI ;

// グラフィックスデータとレンダラーの生成
int graphicsID = newGraphics( ) ;
int rendererID = newGraphics2DRenderer( 800, 600, graphicsID ) ;

// 表示画面の生成
int windowID = newWindow( 0, 0, 800, 600, " Hello 2DCG ! " );
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID ) ;
mountComponent( labelID, windowID );

// 背景色を白に設定してクリア
setGraphics2DColor( rendererID, 255, 255, 255, 255 ) ;
clearGraphics2D( rendererID ) ;

// =====
//      描画処理 ここから
// =====

// (0,0)と(100,100)を結ぶ線スプライトを生成(赤色)
int lineSpriteID = newLineSprite( 0, 0, 100, 100 ) ;
setSpriteColor( lineSpriteID, 255, 0, 0, 255 ) ;
```

```

mountSprite( lineSpriteID, rendererID );

// (100,100)に(500×300)の長方形スプライトを生成(青色)
int rectangleSpriteID = newRectangleSprite( 100, 100, 500, 300, true );
setSpriteColor( rectangleSpriteID, 0, 0, 255, 255 );
mountSprite( rectangleSpriteID, rendererID );

// (100,100)に(500×300)の楕円スプライトを生成(緑色)
int ellipseSpriteID = newEllipseSprite( 100, 100, 500, 300, true );
setSpriteColor( ellipseSpriteID, 0, 255, 0, 255 );
mountSprite( ellipseSpriteID, rendererID );

// 多角形、折れ線スプライト用の x 配列、y 配列を用意
int x[ 3 ]; x[ 0 ] = 100 ; x[ 1 ] = 300 ; x[ 2 ] = 300 ;
int y[ 3 ]; y[ 0 ] = 100 ; y[ 1 ] = 100 ; y[ 2 ] = 300 ;

// x 配列、y 配列を頂点とする多角形スプライトを生成(黄色)
int pgSpriteID = newPolygonSprite( x, y, true );
setSpriteColor( pgSpriteID, 255, 255, 0, 255 );
mountSprite( pgSpriteID, rendererID );

// x 配列、y 配列を頂点とする折れ線スプライトを生成(赤色)
int plSpriteID = newPolylineSprite( x, y );
setSpriteColor( plSpriteID, 255, 0, 0, 255 );
mountSprite( plSpriteID, rendererID );

// (100,100)にテキストスプライトを生成(黒色)
string text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int txtSpriteID = newTextSprite( 300, 50, 250, 35, text );
setSpriteColor( txtSpriteID, 0, 0, 0, 255 );
setSpriteFontSize( txtSpriteID, 30 );
mountSprite( txtSpriteID, rendererID );

```

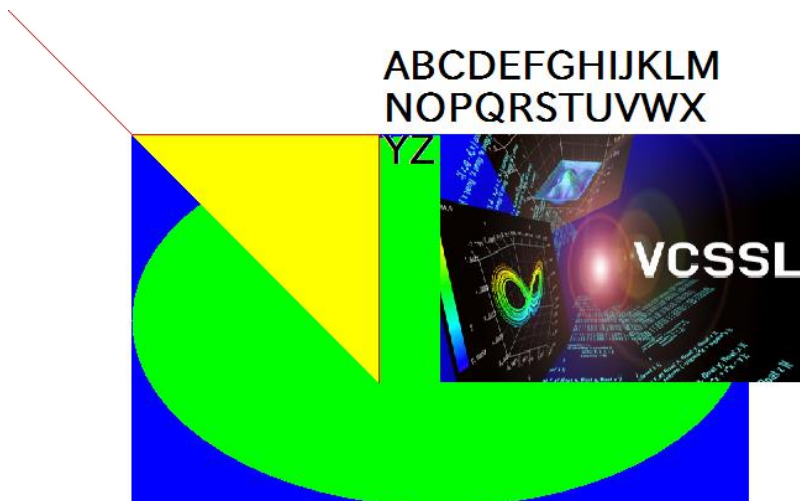
```
// 画像ファイル「Test.png」を読み込み、画像スプライトを生成
// int graphicsID2 = newGraphics( "Test.png" );
// int imageSpriteID = newImageSprite( 350, 100, 300, 200, graphicsID2 );
// mountSprite( imageSpriteID, rendererID );
// ↑「Test.png」を用意してから、コメント記号を外して実行してください

// =====
//      描画処理 ここまで
// =====

// 登録されているスプライトを合成して描画
paintGraphics2D( rendererID );
// 画面の描画
paintComponent( labelID );
paintComponent( windowID );
```

このプログラムを実行すると、白いウィンドウが表示され、その上に様々な図形が表示されます。

▼ 実行結果



※文字に使用されるフォントは、環境によって異なります。フォントを明示したい場合には `setSpriteFont(int SpriteID, string fontName)`関数を使用してください。

スプライトの制御

ここでは、配置後のスプライトの各種設定や制御を扱います。

■ スプライトの色設定

スプライトの色は、生成後に設定します。一度設定した後も自由に変更する事が可能です。スプライトの色設定には `setSpriteColor` 関数を使用します。

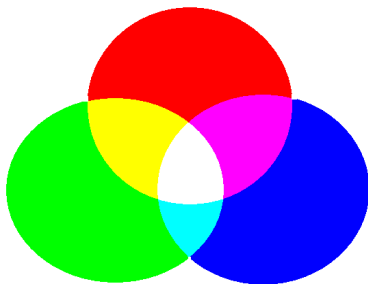
```
void setSpriteColor (  
    int spriteID,  
    int red, int green, int blue, int alpha  
)
```

最初の引数 `spriteID` でスプライトの ID を、続く引数で描画色の色成分を指定します。色成分は RGBA 形式で、それぞれ 0～255 の範囲の値を指定します。

・RGBA 形式とは

RGBA 形式とは、色の三原色である赤 (Red)、緑 (Green)、青 (Blue) の色成分に、アルファ値 (Alpha) を加えた形式です。アルファ値は色の透明度を表す数値で、0 で完全透明になり、最大にすると不透明になります。

それぞれの色成分からの色の合成は、加法混色によって行われます。これは光の重ね合わせと同じ混色方式であり、絵の具の混ぜ合わせ (減法混色) では無い事にご注意ください。例として、(赤, 緑, 青) = (255, 255, 255) は、黒ではなく白になります。また、(255, 255, 0) は黄色に、(0, 255, 255) は水色に、(255, 0, 255) はマゼンタになります。



■ スプライトの深度設定

スプライトは、「深度」という値を持っています。複数のスプライトを重ねて描画する際、深度の大きいほうが奥に隠れ、深度の浅いものが手前に描画されます。この深度を設定するには `setSpriteDepth` 関数を使用します。

```
void setSpriteDepth ( int spriteID, int depth )
```

最初の引数 `spriteID` でスプライトの ID を、続く引数 `depth` でスプライトの深度を指定します。

■ スプライトの位置変更

スプライトの描画位置を変更するには、`setSpriteLocation` 関数を使用します。

```
void setSpriteLocation ( int spriteID, int x, int y )
```

最初の引数 `spriteID` でスプライトの ID を、続く引数 `x, y` でスプライト左上頂点の座標を指定します。なおこの関数は、長方形スプライト、楕円スプライト、テキストスプライト、画像スプライトに対してのみ使用可能です。線スプライト、多角形スプライト、折れ線スプライトに関しては、後述する `setSpriteVector` 関数を使用してください。

■ スプライトのサイズ変更

スプライトの描画サイズを変更するには、`setSpriteSize` 関数を使用します。

```
void setSpriteSize ( int spriteID, int radius )
```

これは点スプライトのためのもので、最初の引数 `spriteID` でスプライトの ID を、続く引数 `radius`

で点の半径を指定します。長方形スプライト、楕円スプライト、テキストスプライト、画像スプライトに対しては以下を使用します。

```
void setSpriteSize ( int spriteID, int width, int height )
```

最初の引数 spriteID でスプライトの ID を、続く引数 radius または width、height でスプライトの幅と高さを指定します。なお、線スプライト、多角形スプライト、折れ線スプライトに関しては、後述する setSpriteVector 関数を使用してください。

ラスタ制御描画

ここでは、最も直接的な描画方法であるラスタ制御描画について扱います。

■ ラスタ制御描画

・ラスタ制御描画の仕組み

これまで扱ってきた直接描画やスプライト描画は、レンダラーに用意されたペイントツールの機能を用いて、図形などを描き込んでいく方式でした。それに対してラスタ制御描画は、そのような高度な機能は使用せず、グラフィックスデータの保持するピクセル情報である「ラスタデータ」を直接的に操作する事により、描画を行う方式です。

・ラスタデータ

ラスタデータとは、グラフィックスデータ上の全てのピクセルにおける色情報を表すデータです。色情報には赤 (Red)、緑 (Green)、青 (Blue)、透明度 (Alpha) の 4 つの要素がありますが、VCSSL ではこれらをそのまま 4 つの独立なラスタデータとして扱います。

VCSSL では、ラスタデータを [高さ] × [幅] の 2 次元配列として扱います。配列のインデックスは [行] [列] となり、行は上から、列は左から数えます。配列の型は int 型で、そのピクセルにおける色の強度を 0~255 の範囲で保持します。

少し具体的な話をすると、青色成分のラスタデータを blue[][] として、blue[0][0] は、グラフィックスデータの左上端のピクセルにおける、青色の強度を表します。同様に blue[10][100] は、グラフィックスデータの上から 10 個目、左から 100 個目のピクセルにおける、青色の強度を表します。

■ ラスタデータの描画

任意のラスタデータの内容を描画するには、レンダラーの setPixel 関数を使用します。

```
void setPixel (
    int rendererID,
    int red[ ][ ], int green[ ][ ], int blue[ ][ ], int alpha[ ][ ]
)
```


最初の引数 `rendererID` でレンダラーの ID を指定します。続く引数 `red`、`green`、`blue`、`alpha` でラスターデータを指定します。

ラスターデータ配列のインデックスは[行][列]でピクセル位置を指定し、値はその位置のピクセルにおける色強度(0~255)を表します。

この関数をコールすると、受け取ったラスターデータの内容に基づいて、レンダラーがグラフィックスデータへの描画を行います。

なお、以下のように、一つの配列で全色成分をまとめて指定する事もできます。

```
void setPixel ( int rendererID, int color[ ][ ][ ] )
```

この場合は、`color` に各ピクセルの色を格納して渡します。配列のインデックスは [行][列][色成分の番号] で、色成分の番号は 赤=0, 緑=1, 青=2, α =3 です。

■ ラスターデータの取得

現在レンダラーが保持しているグラフィックスデータの内容から、各色成分のラスターデータを取得するには、色に応じて以下の 4 つの関数を使用します。

```
int[ ][ ] getPixelRed ( int rendererID )  
int[ ][ ] getPixelGreen ( int rendererID )  
int[ ][ ] getPixelBlue ( int rendererID )  
int[ ][ ] getPixelAlpha ( int rendererID )
```

引数 `rendererID` にはレンダラーの ID を指定します。これらの関数は上から順に、それぞれ赤 (Red)、緑 (Green)、青 (Blue)、透明度 (Alpha) 成分のラスターデータを、`int` 型の 2 次元配列で返します。配列のインデックスは[行][列]となり、行は上から、列は左から数えます。

なお、既存の画像ファイルの内容からラスターデータを取得するには、まずレンダラーの `drawGraphics` 関数などで画像ファイルの内容を描き込んでから、そのレンダラーに対して上の `getPixel`~関数を使用します。

また、以下の `getPixel` 関数を用いて、全色成分をまとめて取得する事もできます。

```
int[ ][ ][ ] getPixel ( int rendererID )
```

引数 `rendererID` にはレンダラーの ID を指定します。戻り値の配列のインデックスは [行][列][色成分の番号] で、色成分の番号は 赤=0, 緑=1, 青=2, α =3 です。

■ プログラム例

実際に簡単な模様のラスターデータを作成し、描画を行って見ましょう。以下のように記述し、実行してみてください。

```
import Graphics ;
import Graphics2D ;
import GUI ;

// グラフィックスデータとレンダラーの生成
int graphicsID = newGraphics( ) ;
int rendererID = newGraphics2DRenderer( 800, 500, graphicsID ) ;

// 表示画面の生成
int windowID = newWindow( 0, 0, 800, 500, "Hello 2DCG !" ) ;
int labelID = newImageLabel( 0, 0, 800, 500, graphicsID ) ;
mountComponent( labelID, windowID );

// ラスターデータを作成
int red[ 500 ][ 800 ] ;
int green[ 500 ][ 800 ] ;
int blue[ 500 ][ 800 ] ;
int alpha[ 500 ][ 800 ] ;

red = 0 ;
green = 0 ;
```

```

blue = 0 ;
alpha = 255 ;

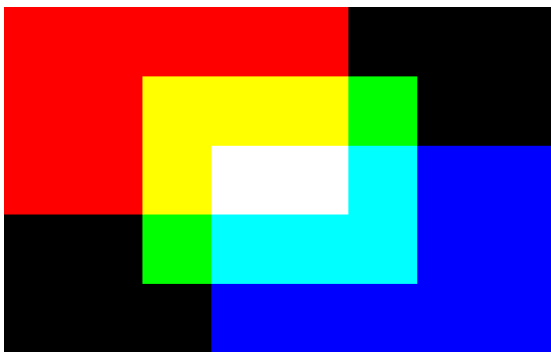
for( int i=0; i<500; i++){
    for( int j=0; j<800; j++){
        if( j < 500 && i < 300 ){
            red[ i ][ j ] = 255 ;
        }
        if( 200 < j && j < 600 && 100 < i && i < 400 ){
            green[ i ][ j ] = 255 ;
        }
        if( 300 < j && 200 < i ){
            blue[ i ][ j ] = 255 ;
        }
    }
}

//ラスターデータの内容を描画
setPixel( rendererID, red, green, blue, alpha ) ;
// GUI の描画
paintComponent( labelID ) ;
paintComponent( windowID ) ;

```

このプログラムを実行すると、白いウィンドウが表示され、その上にカラフルな長方形が重なって表示されます。

▼ 実行結果



フレームワークの使用

ここでは、レンダラーや表示画面の生成などを自動化してくれる、フレームワークの使用方法を扱います。

■ 面倒な定型処理を自動で行ってくれるフレームワーク

これまでは、レンダラーやグラフィックスデータ、および表示画面を生成したり、画面の GUI を描画させたりといった処理を、毎回行ってきました。しかし、これらの処理はいつも同じような内容なので、毎回いちいち書くのは面倒です。

そこで VCSSL の標準ライブラリには、これらの処理を自動で行ってくれるフレームワーク「Graphics2DFramework」が用意されています。これを使用すれば、ユーザーは、いくつかのタイミングで自動的に呼び出される関数の中に、描画内容の処理などをただ書くだけで済みます。

フレームワークを使用する方法は非常に簡単で、プログラム内でフレームワークを import するだけです。

```
import graphics2d.Graphics2DFramework ;
```

これだけで、プログラムを実行すると自動でレンダラーやグラフィックスデータが生成され、画面も表示されるようになります。つまり、形の上では、2DCG の描画用プログラムとして一応できあがったものになります。ただし、この段階では画面上は真っ白で、まだ何も描画されません。

■ フレームワークの画面上に描画などを行わせるには、関数を定義する

画面に何らかの内容を描画させるには、プログラム内に onPaint という名前の関数を定義して、その中に行わせたい描画処理を記述します。そうすると、その関数をフレームワーク側が適当なタイミングで実行してくれて、その結果が画面上に表示されるようになります。

これまでのプログラムでは、ユーザーが書いた内容だけが、「上の行から下の行へ」の流れで単純に実行されてきました。しかしフレームワークを使用したプログラムでは、処理の流れを司るのはフレームワークであり、ユーザーはそこに「関数を定義して処理を追加していく」という形で開発を行います。上で述べた onPaint 関数をはじめ、以下のような関数を定義して処理を追加できます：

- ・プログラムの最初に呼び出される関数(各種設定や、画像の読み込み処理などを記述)

```
void onStart ( int rendererID )
```

- ・画面更新周期ごとに呼び出される関数(画面の描画処理などを記述)

```
void onPaint ( int rendererID )
```

- ・画面更新周期ごとに呼び出される関数(アニメーションでの位置の更新処理などを記述)

```
void onUpdate ( int rendererID )
```

- ・画面サイズが変更されたときに呼び出される関数

```
void onResize ( int rendererID )
```

- ・プログラムを終了する時に呼び出される関数

```
void onExit ( int rendererID )
```

引数の「 rendererID 」には、フレームワークから呼び出される際に、自動で用意されている 2DCG レンダラー(描画エンジン)の ID が入っています。

なお、上記の関数を全て定義する必要はありません。使うものだけを定義し、処理を記述してください。単純に画面上に絵を描きたいだけであれば、onPaint 関数だけで十分です。onPaint 関数は毎秒数十回(負荷にもよりますが、大体 1 秒間に 30 回ほど)の頻度で繰り返し実行され続けますが、それと交互に onUpdate 関数も実行されます。アニメーション描画を行いたい場合は、この onUpdate 関数の中に、描画位置などの変数の値を少しずつ変化させるような処理を記述するとよいでしょう。onStart 関数には、画像ファイルの読み込みなど、プログラムの最初に 1 度だけ行えばよい処理を記述します。逆に onExit 関数には、画像ファイルの保存など、最後に 1 度だけ行いたい処理を記述します(画面が閉じられたタイミングで実行されます)。

■ プログラム例

それでは、実際にフレームワークを使用し、画面に単純な内容を描画させてみましょう。以下のよう
に記述し、実行してみてください。

```
import Graphics2D ;
import graphics2d.Graphics2DFramework ;

// 描画処理を記述する関数
// ( 画面更新周期ごとに、毎秒数十回呼び出される )
void onPaint ( int rendererID ) {

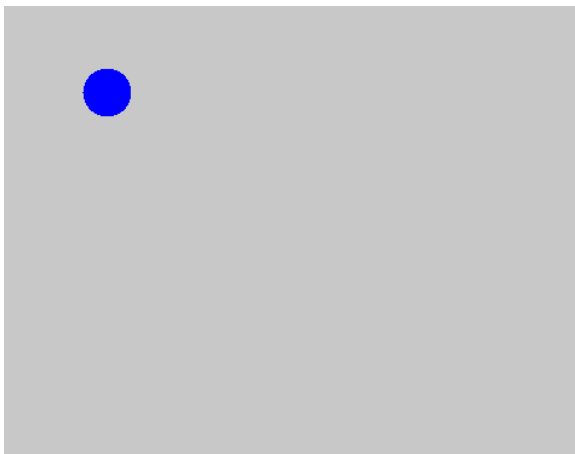
    // 背景色をグレーに設定
    setGraphics2DColor( rendererID, 200, 200, 200, 255 );

    // 描画色を青色に設定
    setDrawColor( rendererID, 0, 0, 255, 255 );

    // 点を描画
    drawPoint( rendererID, 100, 100, 20, true );
}
```

このプログラムを実行すると、グレーのウィンドウが表示され、その上に青い点が表示されます。

▼ 実行結果



続いて、アニメーションを行う例です。以下のように記述し、実行してみてください。

```
import Graphics2D ;
import graphics2d.Graphics2DFramework ;

// 点の描画位置を格納する変数
int x = 0;
int y = 0;

// 描画処理を記述する関数
// ( 画面更新周期ごとに、毎秒数十回呼び出される )
void onPaint ( int rendererID ) {

    // 背景色をグレーに設定
    setGraphics2DColor( rendererID, 200, 200, 200, 255 );

    // 描画色を青色に設定
    setDrawColor( rendererID, 0, 0, 255, 255 );

    // 点を描画
    drawPoint( rendererID, x, y, 20, true );
}

// 更新処理を記述する関数
// ( 画面更新周期ごとに、毎秒数十回呼び出される )
void onUpdate ( int rendererID ) {

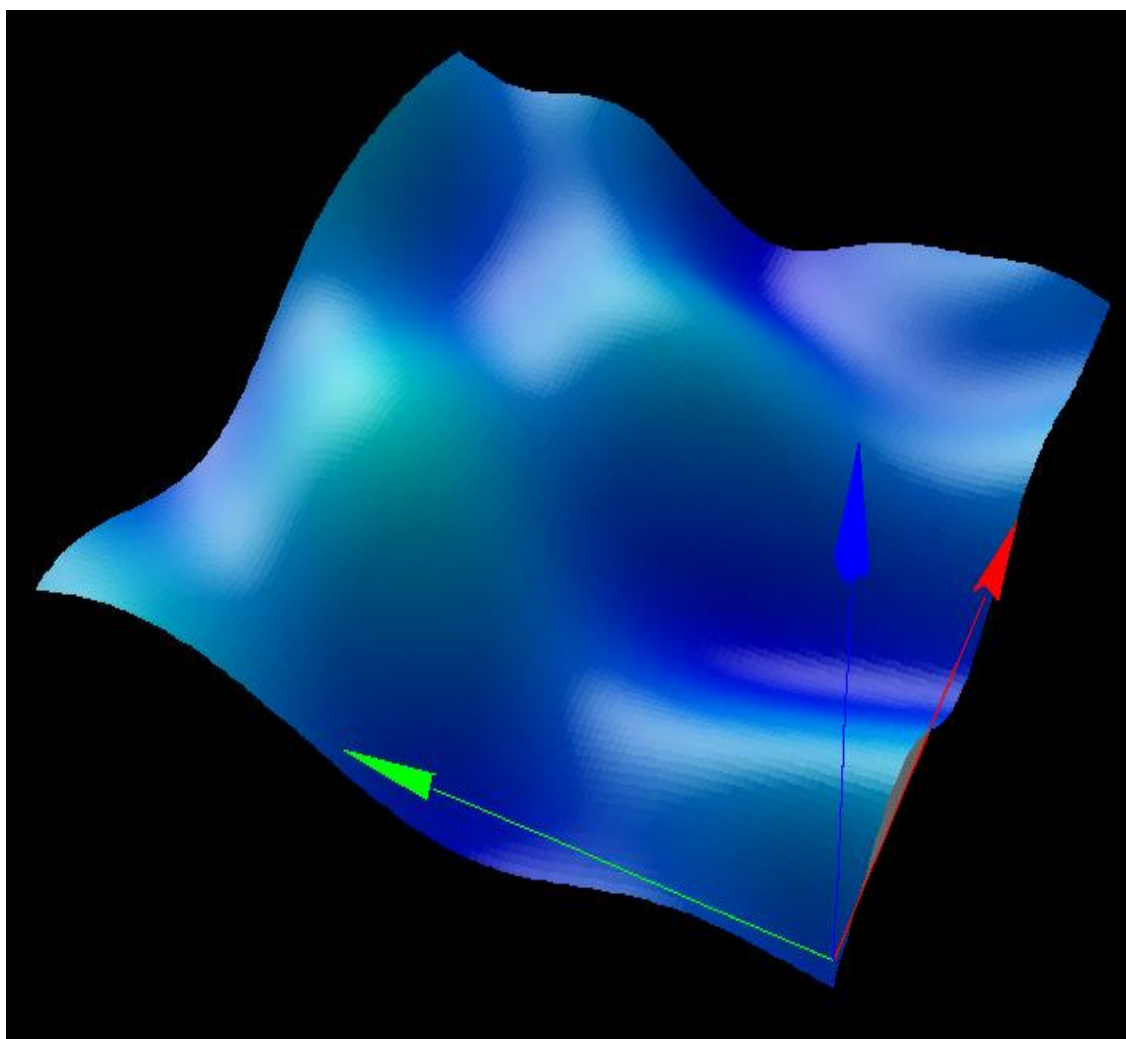
    // 点の描画位置を少しずつ変化させる
    x += 2;
    y += 1;
}
```

このプログラムを実行すると、先ほどと同様、グレーのウィンドウ上に青い点が表示されますが、青い点はゆっくりとアニメーションで移動していきます。

第 4 部

3DCG

- 3 次元コンピューターグラフィックス -



この部では、3次元コンピューターグラフィックスを用いたVCSSLプログラムを開発する方法を解説します。この文書の内容を扱うには、VCSSLの基本的な文法をすでに習得している必要があります。

第一章 基盤

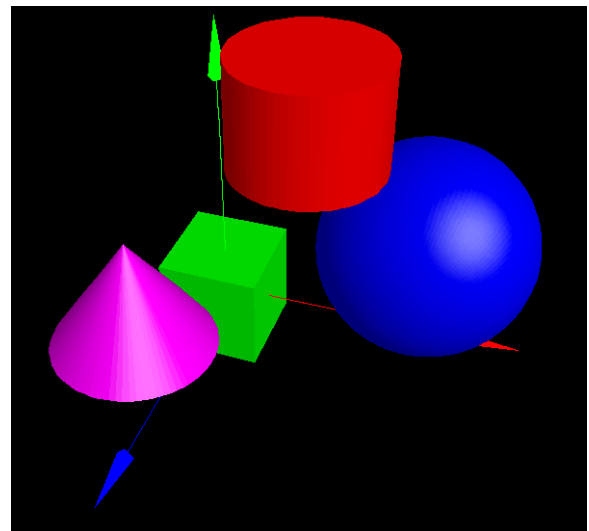
この章では、基礎的な事項の解説をはじめ、ウィンドウや画面の構築、描画、マウス操作時の制御など、基盤に関する事項を扱います。

3 次元コンピューターグラフィックス

■ 3 次元コンピューターグラフィックス（3DCG）とは

コンピューターグラフィックス、いわゆる CG とは、コンピューター上の計算処理によって画像や映像を作成する技術の事です。

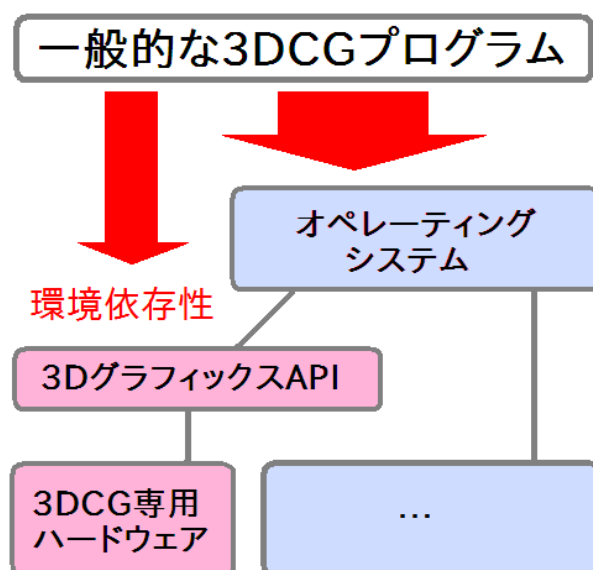
CG には、平面的な表現を扱う 2 次元コンピューターグラフィックス — 2DCG と、立体的な表現を扱う 3 次元コンピューターグラフィックス — 3DCG が存在します。かつては PC などでの描画処理は 2DCG が主流でしたが、近年のコンピューター性能や 3D 関連技術の目覚ましい進歩により、現在では 3DCG も一般的なものとなりました。



■ VCSSL の 3DCG 機能

・言語仕様として 3DCG をサポート、環境に依存しない 3DCG プログラムを開発可能

3DCG を使用した一般的な形態のプログラムは、実行環境への依存性が強い傾向にあり、開発時や実行時には様々な環境依存性を考慮する必要があります。これには、例えばオペレーティングシステムの種類や、グラフィックスボードの世代と性能、加えて下層レイヤーで使用する 3D グラフィックス API の種類やバージョンなど、様々な要因が挙げられます。



しかし VCSSL では、3DCG 機能を正式な言語仕様の一部「 VCSSL Graphics3D 」としてサポートしています。これにより、プログラマは VCSSL よりも下層の差異、つまりオペレーティングシステムや搭載ハードウェアの違いなどによる環境依存性を気にする必要が無く、どこでも動作する 3DCG プログラムを開発する事が可能となっています。

VCSSL 3DCGプログラム

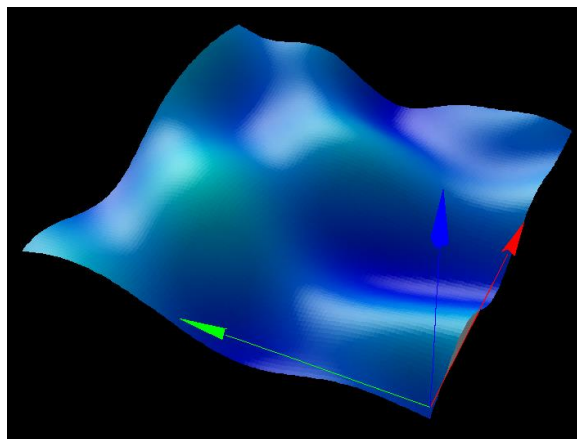
環境依存性なし

VCSSL 処理システム
(実行環境)

このように環境依存性を低減するために、VCSSL の処理系には、3D 描画処理の大半を GPU ではなく CPU で行う、ソフトウェアレンダラー形式の 3D 描画エンジンが内蔵されています。ただし、その反面として描画性能はある程度限定してしまうため、近年の 3D ゲームのように、写真と見間違えるような美麗で高詳細な 3D 映像を描画する事はできません。

・性能は数十万ポリゴン/秒程度

具体的な VCSSL Graphics3D の処理性能は、使用する処理系やコンピューターによっても異なりますが、標準的な場合において概ね 数十万ポリゴン/秒（立体を構成する、三角形や四角形の小さな平面を、1 秒間あたり数十万枚描ける性能）程度のパフォーマンスを発揮します。これは近年における一般的な 3DCG の水準からすると大分見劣りしますが、しかしちょっとした立体を描いたり、数値計算の結果を可視化したりといった用途は十分にこなせます。



・簡単に開発できる

VCSSL は、設計方針として簡易用途での扱いやすさを重視しているため、非常に簡単でシンプルなプログラム開発が可能となっています。これは VCSSL Graphics3D も例外ではありません。

一般的な 3DCG プログラムの開発シーンでは、プログラマには 3DCG の原理やハードウェアの機能、及び数学的な内容などに関する、様々な専門知識などが要求されがちです。

$$V' = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} V$$



これに対して VCSSL Graphics3D では、プログラマがそういった専門的な内容を極力意識する事無く、直感的に 3DCG プログラムの開発を進めていけるように設計されています。

■ VCSSL プログラムで 3DCG を扱うには

VCSSL プログラムで 3DCG を扱うには、VCSSL 標準ライブラリの中から、Graphics ライブラリと Graphics3D ライブラリをインポートする必要があります。また、描画結果はウィンドウ上に表示する事が多いので、同時に GUI ライブラリをインポートしておく事を推奨します（画像をファイルに保存する場合など、ウィンドウなどを利用しない場合は不要）。

これらのライブラリをインポートするには、プログラムの先頭行に以下のように記述します。

```
import Graphics ;  
import Graphics3D ;  
import GUI ;
```

これで、プログラム中から 3DCG を扱うための関数が利用可能になります。

基盤の準備

3DCG プログラミングを行うには、グラフィックスデータやレンダラー（描画エンジン）、表示画面などが最低限必要となります。ここでは、それら必須基盤の準備を一括して行います。

■ グラフィックスデータの生成

まず、3DCG の描画内容を保持しておくデータ領域（イメージ、バッファ、コンテキストなど）が必要です。これらは VCSSL ではまとめて「グラフィックスデータ」として扱います。詳細は 2DCG で述べた通りです。グラフィックスデータは、Graphics ライブラリの newGraphics 関数で生成できます。

```
int newGraphics( )
```

この関数は、空白のグラフィックスデータ領域を確保し、それに固有のグラフィックスデータ ID（識別番号）を割り振って返します。

■ 3DCG レンダラー（描画エンジン）の生成

続いて、グラフィックスデータに 3DCG 映像を描き込むレンダラー（描画エンジン）も生成します。これには newGraphics3DRenderer 関数を使用します。

```
int newGraphics3DRenderer ( int width, int height, int graphicsID )
```

引数の width と height で描画するグラフィックスの大きさを、graphicsID で描画対象のグラフィックス ID を指定します。この graphicsID には、newGraphics 関数で確保したものを指定します。

この関数は、3DCG 描画機能を持つレンダラーを生成し、それに固有のレンダラー ID を返します。これ以降、様々な 3DCG の制御関数を扱いますが、その引数にこのレンダラー ID を指定します。

■ 3DCG レンダラーの各種設定と基本操作

3DCG レンダラーは生成したままの状態でも使用できますが、必要に応じて各種設定を行います。生成後によく行う設定としては、以下のようなものが挙げられます。加えて、毎回必ず使用する基本操作についても挙げておきます。

・背景色の設定

背景色を設定するには、setGraphics3DColor 関数を使用します。

```
void setGraphics3DColor ( int rendererID, int red, int green, int blue, int alpha )
```

最初の引数 rendererID では、設定対象のレンダラーID を指定します。続く引数では、背景の色成分を指定します。色成分はそれぞれ 0～255 の範囲で指定します。色成分の形式は、RGBA 形式※をサポートしています。

・表示倍率の設定

表示倍率を設定するには、setGraphics3DMagnification 関数を使用します。

```
void setGraphics3DMagnification ( int rendererID, float magnification )
```

最初の引数 rendererID では、設定対象のレンダラーID を指定します。続く引数 magnification では、表示倍率を指定します。

なお、表示倍率とカメラ距離の設定はワンセットですが、カメラ距離の設定については、後の章で扱う座標系の操作を用います（ワールド座標系を移動させる）。

・3DCG の描画

レンダラーを駆動させて、グラフィックスデータに 3DCG 映像を描画するには、paintGraphics3D 関数を使用します。

```
void paintGraphics3D ( int rendererID )
```

最初の引数 `rendererID` では、描画を行うレンダラーの ID を指定します。

3DCG の基本的な処理は、まず全ての立体などを配置登録し、続いて位置や角度などを調整し、そして描画を行うといった流れとなります。この描画のタイミングで、この `paintGraphics3D` 関数をコールします。

■ 表示画面の生成

これまでに述べた事項だけで、3DCG の描画は可能です。しかし実用上は、その描画結果を何らかの形で出力する必要があります。最も一般的なのが、表示画面に直接グラフィックスを表示する形態でしょう。

実際に 800×600 サイズの表示画面を生成するには、プログラム中で以下のように記述します。

```
// 表示画面の生成 ( graphicsID は画面表示するグラフィックスデータの ID )  
int windowID = newWindow( 0, 0, 800, 600, " Hello 3DCG ! " );  
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID );  
mountComponent( labelID, windowID );
```

こういった GUI の制御に関する詳しい解説は、ここでは割愛します。詳しくは VCSSL GUI ライブラリのガイドをご参照ください。とりあえず、上記のように即席の表示画面でも、これから扱う 3DCG の内容は十分にこなせます。より高度な表示画面を工夫するのでなければ、これだけで十分です。

■ 画像ファイル出力

グラフィックスデータは、画面に表示するだけでなく、Graphics ライブラリの `exportGraphics` 関数を使用し、BMP/PNG/JPEG 形式の画像ファイルに出力する事も可能です。

```
void exportGraphics( int graphicsID, string fileName, string format )
```

最初の引数 `graphicsID` には出力するグラフィックスデータの ID を、続いて `fileName` にファイル名、

format にファイル形式を指定します。ファイル形式には "BMP", "PNG", "JPEG" のどれかを指定します。

■ プログラム例

それでは実際にウィンドウを生成し、3DCG を画面に表示してみましょう。ついでに、「Test.png」という名前の PNG 形式画像ファイルにも出力してみます。以下のようにプログラムを記述し、実行してみてください。

```
import Graphics ;
import Graphics3D ;
import GUI ;

// グラフィックスデータと 3DCG レンダラーの生成
int graphicsID = newGraphics( ) ;
int rendererID = newGraphics3DRenderer( 800, 600, graphicsID ) ;

// 表示画面の生成
int windowID = newWindow( 0, 0, 800, 600, " Hello 3DCG ! " );
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID ) ;
mountComponent( labelID, windowID );

setGraphics3DColor( rendererID, 0, 0, 0, 255 ) ; // 背景色を黒に設定
paintGraphics3D( rendererID ) ; // 3DCG を描画

// GUI の再描画
paintComponent( labelID ) ;
paintComponent( windowID ) ;

// 画像ファイルに出力
exportGraphics( graphicsID, "Test.png", "PNG" ) ;
```

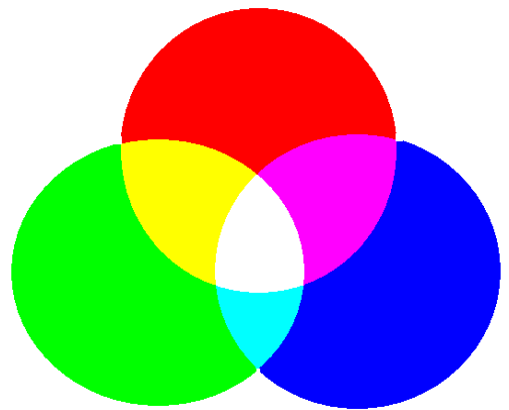


このプログラムを実行すると、ウィンドウが立ち上がり、真っ黒な画面が表示されます。何も表示されていないように見えるのは、単にまだ立体を配置していないからであって、紛れもなく 3D 仮想空間が見えています。

■ ※ RGBA 形式とは

RGBA 形式とは、色の三原色である赤 (Red)、緑 (Green)、青 (Blue) の色成分に、アルファ値 (Alpha) を加えた形式です。アルファ値は色の透明度を表す数値で、0 で完全透明になり、最大にすると不透明になります。

それぞれの色成分からの色の合成は、加法混色によって行われます。これは光の重ね合わせと同じ混色方式であり、絵の具の混ぜ合わせ (減法混色) では無い事にご注意ください。例として、(赤, 緑, 青) = (255, 255, 255) は、黒ではなく白になります。また、(255, 255, 0) は黄色に、(0, 255, 255) は水色に、(255, 0, 255) はマゼンタになります。



マウス操作とアニメーション処理

3DCG は立体を扱うので、見る角度(カメラアングル)によって得られる画像が変化します。そこで、マウス操作で視点を動かせると便利です。ここでは、このような処理の実装方法について扱います。

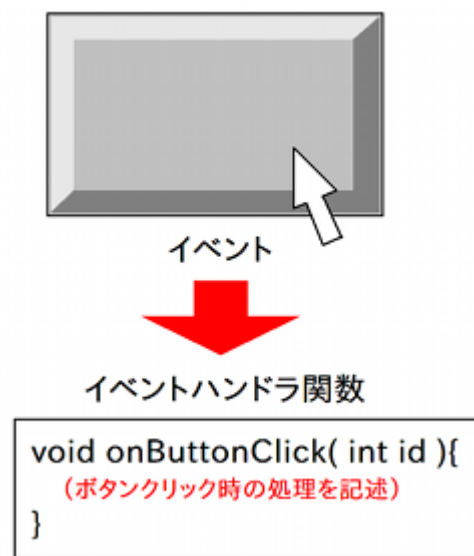
■ イベントハンドラ

ここでは詳しく扱いませんが、GUI コンポーネントに対してマウス操作などのアクションが行われた際(一般に「イベントが発生した」と言う)、システム側からイベントハンドラという特別な関数が呼び出されます。詳しくは GUI ライブラリのガイドをご参照ください。

例えば、マウス操作によってカメラアングルを変化させたい場合、マウス操作に対するイベントハンドラを作成する必要があります。これには標準で用意されているイベントハンドラを使用する方法と、自分で独自にイベントハンドラを記述する方法があります。

前者には、一行で済むので簡単という利点があります。また後者は、細かい部分まですべて思い通りに設計できるという利点があります。ここでは前者の、標準のもので済ませる方法を使用します。

標準のイベントハンドラを利用するには、`setGraphics3DDefaultEventHandler` 関数を使用します。



```
void setGraphics3DDefaultEventHandler ( int rendererID, int componentID )
```

最初の引数 `rendererID` には、レンダラーの ID を指定します。続く引数の `componentID` には、画面表示を行う GUI コンポーネント(画像ラベルなど)の ID を指定します。ここで指定した GUI コンポーネントに対してマウス操作が行われると、それに応じてカメラアングルが自動変更されます。

■ アニメーション

・無限ループ

アニメーション処理を行うには、プログラム起動時から終了時まで、ずっと繰り返し画面描画を行い続ける必要があります。つまり、いわゆる無限ループを設ける必要があります。

無限ループの処理は、while 制御構文の条件式に、true の値を持つ bool 型変数を指定する事で簡単に実現できます。こうすると常にループ条件が真になるので、いつまでもループ内の処理が繰り返さ

れ続けます。アニメーションを終了させる最は、この bool 型変数の値を false にします。

無限ループ

3DCGの描画
GUIコンポーネントの描画
その他の計算処理など
待機(15~30ミリ秒)

```
bool mainLoopState = true ; // アニメーションを終了させる際に false にする
while( mainLoopState ){

    /* この中が無限ループ ここに描画などの処理を記述する */

}
```

・ウィンドウを閉じた際に無限ループを脱出

無限ループを使用する場合は、ウィンドウを閉じた際に無限ループを脱出し、その後 exit 命令をコールして、プログラムを自動的に終了させるようにしましょう。そうしなければ、もし VCSSL コンソールが不可視化されていた場合や、そもそも表示されないような処理システムで実行した場合に、プログラムを終了させる手段が無くなり、いつまでもアニメーション処理が行われ続けてしまいます。

ウィンドウを閉じた際に処理を行うには、イベントハンドラの一つである onWindowClose 関数を作成します。

```
void onWindowClose( int componentID )
```

この関数を作成しておくと、ウィンドウを閉じた際にシステム側から自動で呼び出されます。引数 `componentID` には、閉じられたウィンドウの GUI コンポーネント ID が渡されます。

■ プログラム例

それでは、実際にアニメーション処理を記述してみましょう。ここではカメラアングルの変更を分かりやすくするため、座標軸モデルも配置します。様々なモデルの生成と配置に関しては次章で詳しく扱いますので、とりあえずは下のように記述し、実行してみてください。

```
import Graphics ;
import Graphics3D ;
import GUI ;

// グラフィックスデータと 3DCG レンダラーの生成
int graphicsID = newGraphics( ) ;
int rendererID = newGraphics3DRenderer( 800, 600, graphicsID ) ;

// 表示画面の生成
int windowID = newWindow( 0, 0, 800, 600, " Hello 3DCG ! " );
int labelID = newImageLabel( 0, 0, 800, 600, graphicsID ) ;
mountComponent( labelID, windowID );

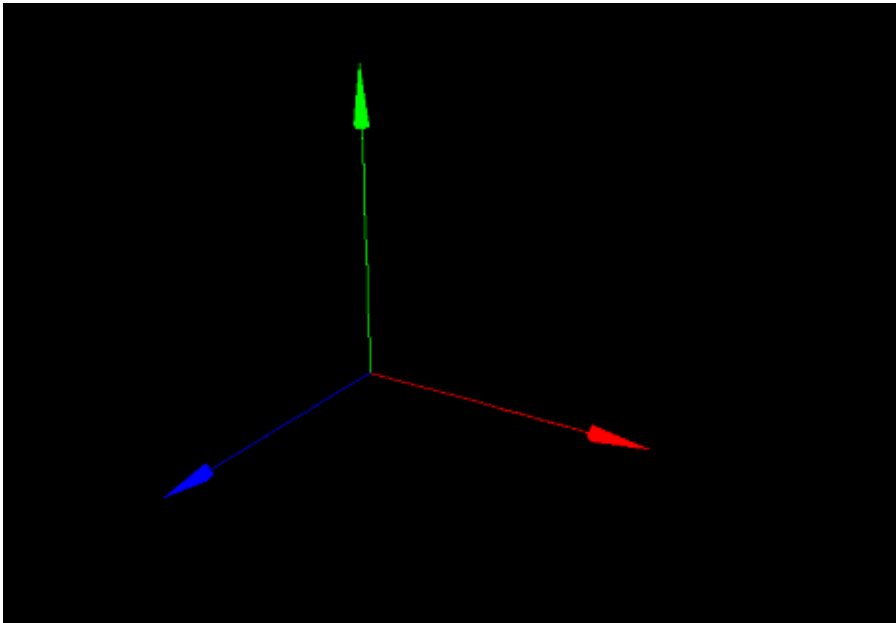
// マウス操作で視点を操作できるようにする
setGraphics3DDefaultEventHandler( rendererID, labelID ) ;

// 背景色を黒に設定
setGraphics3DColor( rendererID, 0, 0, 0, 255 ) ;

//座標軸モデルを生成して配置
int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
mountModel( axis, rendererID ) ;
```

```
// アニメーション処理
bool mainLoopState = true ;
while( mainLoopState ){
    sleep( 30 ); // 30 ミリ秒だけ停止
    paintGraphics3D( rendererID ); // 3DCG を描画
    paintComponent( labelID ); // GUI の再描画
    paintComponent( windowID ); // GUI の再描画
}
exit(); // ループを脱出するとプログラムを終了

// ウィンドウを閉じた際に呼び出される
void onWindowClose( int id ){
    mainLoopState = false ; //ループを脱出
}
```



このプログラムを実行すると、ウィンドウが立ち上がり、黒い背景に座標軸モデルが描画されます。そしてウィンドウ上で左マウスドラッグを行うと、視点が回転します。そして右マウスドラッグを行うと、視点が平行移動します。さらにマウスホイールの回転操作で、表示内容の拡大/縮小を行う事もできます。

フレームワークの使用

ここでは、基盤的な処理を自動化してくれる、簡易フレームワークについて説明します。

■ 標準で使える、3DCG プログラム用のフレームワーク

前回と前々回で扱ったように、3DCG のプログラムでは、表示画面やレンダラー（描画エンジン）の生成、画面描画やアニメーションのためのループ制御、マウス操作に対する処理など、基盤的な処理が必要になります。これらを毎回記述するのは面倒であるため、このような処理を自動で行ってくれるフレームワーク「 Graphics3DFramework 」が標準で用意されています。

■ ユーザーが定義する関数

フレームワークは、単体でも基本構造としてはできあがったプログラムであり、普通に実行する事もできます。ただし、標準では何の立体も配置されておらず、真っ白な画面が表示されるだけです。

そこで、普通は自作のプログラムからフレームワークを読み込んだ上で、立体の配置などの必要な処理だけを追加して使用します。具体的には、決まった名前と引数の関数を定義して、その中にやりたい処理を記述すると、フレームワークがそれ呼び出して実行してくれます。

フレームワークが自動で呼び出してくれる関数には、タイミングに応じて以下 5 つがあります：

・プログラムの最初に呼び出される関数（各種設定や、立体の配置処理などを記述）

```
void onStart ( int rendererID )
```

・画面更新周期ごとに呼び出される関数（画面描画用、3DCG では普通あまり使用しない）

```
void onPaint ( int rendererID )
```

・画面更新周期ごとに呼び出される関数（アニメーションでのモデル移動処理などを記述）

```
void onUpdate ( int rendererID )
```

・画面サイズが変更されたときに呼び出される関数

```
void onResize ( int rendererID )
```

・プログラムを終了する時に呼び出される関数

```
void onExit ( int rendererID )
```

引数の「 rendererID 」には、フレームワークから呼び出される際に、自動で用意されている 3DCG レンダラー（描画エンジン）の ID が入っています。これはモデルの配置などに必要です。

なお、上記の関数を全て定義する必要はありません。使うものだけを定義し、処理を記述してください。よくあるパターンとしては、onStart 関数の中でモデルの生成や配置を行い、onUpdate 関数の中でモデルを少しずつ動かしてアニメーションする、といった具合です。

onUpdate 関数は、負荷にもよりますが、大体 1 秒間に 30 回ほど呼び出されます。それと交互に、2DCG ではよく使用した onPaint 関数も呼び出されますが、3DCG ではあまり使用しません。

■ プログラム例

実際にフレームワークを使用して、座標軸モデルが回転するアニメーションの 3DCG プログラムを作成してみましょう。下のように記述し、実行してみてください。

```
// フレームワークの読み込み
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// 座標軸モデルの ID を控えておく変数
int axis ;
```



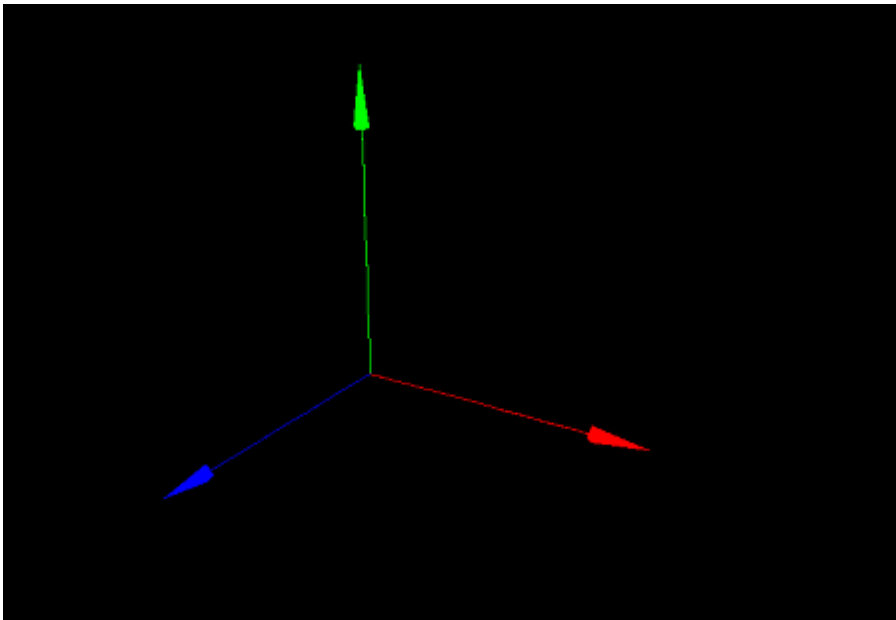
```
// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    axis = newAxisModel( 3.0, 3.0, 3.0, ) ;
    mountModel ( axis, rendererID ) ;
}

// 画面更新周期ごとに、毎秒数十回呼び出される関数
void onUpdate ( int rendererID ) {

    // 座標軸モデルを Z 軸まわりに少しだけ回転させる
    rotModelZ ( axis, 0.03 ) ;
}
```



このプログラムを実行すると、ウィンドウが立ち上がり、座標軸モデルがゆっくりとアニメーションで回り続けます。ウィンドウ上で左マウスドラッグを行うと視点が回転し、右マウスドラッグを行うと、視点が平行移動します。さらにマウスホイールの回転操作で拡大/縮小を行う事もできます。

光源の生成と配置

ここでは、光源の生成と配置について扱います。

■ 球モデルの配置

まずは、球モデルを配置してみましょう。様々なモデルの生成と配置に関しては次章で詳しく扱いますので、とりあえずは下のように記述し、実行してみてください。

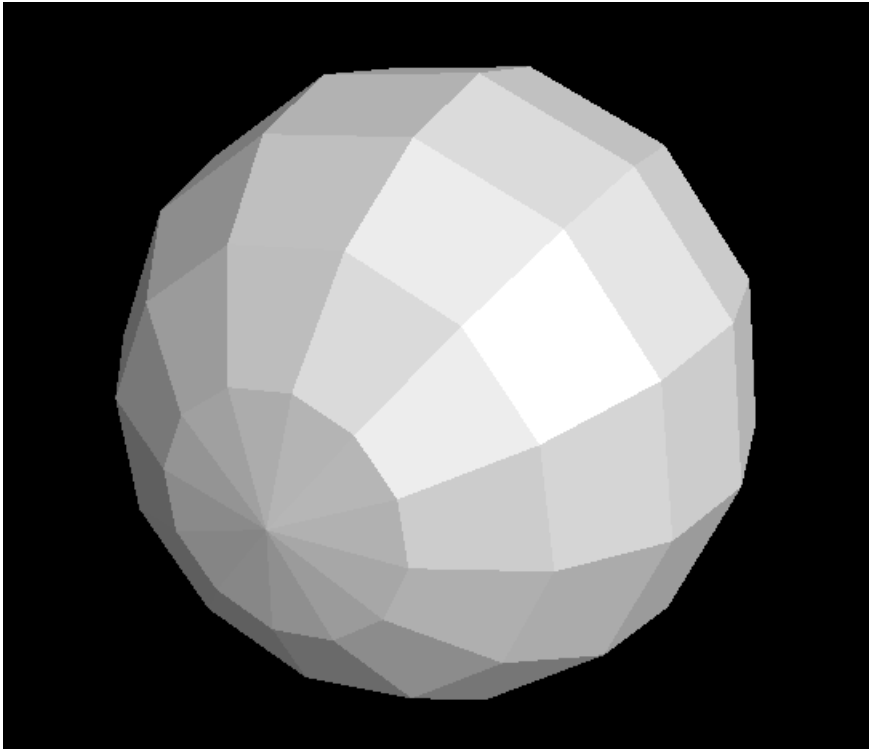
```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 球モデルを生成して配置
    int sphere = newSphereModel( 2.0, 2.0, 2.0, 12, 8 ) ;
    mountModel( sphere, rendererID ) ;

}
```



このプログラムを実行すると、画面中心に球が表示されます。球にはちゃんと陰影が付いて、立体感があるように描画されています。これは、フレームワークの Graphics3DFramework が、標準で光源を用意して配置してくれているためです。

もし光源が無ければ、球には光が当たらずに真っ黒になるはずですが。実際に、標準の光源の明るさを 0 にして、明かりを消してみましょう。上のプログラムの onStart 関数内(どこでも大丈夫です)に、以下のように 3 行だけ追記してみてください：

```
...

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 標準の光源の明るさを 0 にする
    setDirectionalLightBrightness( 0.0 );
    setAmbientLightBrightness( 0.0 );

    ...
}
```



実行すると、このように真っ黒になります。球に光が一切当たらないためです。

■ 光源の生成

光の加減を自分で細かく調整したい場合、上のように標準の光源を消してから、自分で光源を生成して配置する事になります。光源を生成するには、`newLight` 関数、`newPointLight` 関数、`newAmbientLight` 関数の3つの関数を使用します。`newLight` 関数は、全体を一定の方向から照らす平行光源を生成します。`newPointLight` 関数は、特定の点から放射状に照らす点光源を生成します。`newAmbientLight` 関数は、方向に関係無く、モデル全体を一様に照らすアンビエント(環境光)光源を生成します。

```
int newLight ( int x, int y, int z, float power )
```

```
int newPointLight ( int x, int y, int z, float power )
```

```
int newAmbientLight ( int x, int y, int z, float power )
```

引数 x, y, z で光源の存在する方向/位置ベクトル、power で輝度を指定します。輝度は 0.0～1.0 の範囲で指定します。輝度に関しては、全体を一個の光源で照らす場合、大体 0.5 前後が適正値です。

この関数は各種の光源を生成し、その光源の ID を返します。

■ 光源の配置

光源を配置するには、mountLight 関数を使用します。

```
void mountLight ( int lightID, int rendererID )
```

最初の引数 lightID には、光源の ID を指定します。続く引数 rendererID には、レンダラーの ID を指定します。光源は、任意の座標系の上に配置する事も可能です。そのような場合には引数を一つ追加し、配置先座標系を指定します。

```
void mountLight ( int lightID, int rendererID, int coordinateID )
```

最初の引数 lightID には、光源の ID を指定します。続く引数 rendererID には、レンダラーの ID を指定します。最後の引数 coordinateID に、座標系の ID を指定します。

■ プログラム例

それでは、先程のプログラムで光源を配置してみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

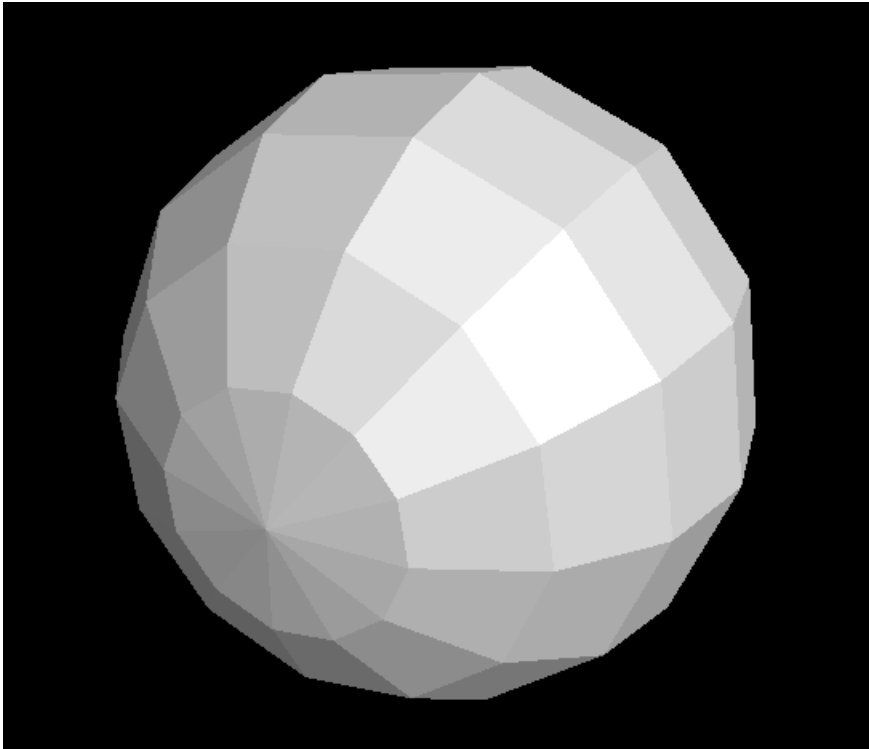
    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 標準の光源の明るさを 0 にする
    setDirectionalLightBrightness( 0.0 );
    setAmbientLightBrightness( 0.0 );

    // (1.2, 1.5, 1.0) の位置に、輝度 0.5 の光源を配置
    int light = newLight( 1.2, 1.5, 1.0, 0.5 ) ;
    mountLight( light, rendererID ) ;

    // 原点に、輝度 0.5 のアンビエント光源を配置
    int ambientLight = newAmbientLight( 0.0, 0.0, 0.0, 0.5 ) ;
    mountLight( ambientLight, rendererID ) ;

    // 球モデルを生成して配置
    int sphere = newSphereModel( 2.0, 2.0, 2.0, 12, 8 ) ;
    mountModel( sphere, rendererID ) ;
}
```



このプログラムを実行すると、ちゃんと光に照らされた球が表示されます。最初の実行結果と同じですが、最初は標準で用意されている光源で照らされていたのに対し、今度は自分で生成・配置した光源で照らされています。

なお、光の明るさは、`newLight` 関数で生成した光源の輝度によって調整できます。影の場所の明るさは、`newAmbientLight` 関数で生成した、アンビエント光源の輝度によって調整できます。

光源の各種設定

ここでは、色や明るさ(輝度)など、光源の各種設定を扱います。

■ 光源の色設定

光源の色を設定するには、setLightColor 関数を使用します。

```
void setLightColor (
    int lightID,
    int red, int green, int blue, int alpha
)
```

最初の引数 lightID では、設定対象の光源の ID を指定します。続く引数では、背景の色成分を指定します。色成分はそれぞれ 0~255 の範囲で指定します。色成分の形式は、RGBA 形式をサポートしています。

色の付いた光でモデルを照らすと、光の色が持つ色成分が、モデルの色が持つ色成分をそれぞれ照らします。つまり光の青成分は、モデルの青成分のみを照らし、赤成分には影響を与えません。

具体的には、白いモデルを青い光で照らすと、普通の青色になります。しかし、緑のモデルや赤いモデルを青い光で照らしても、全体的に暗いまです。

■ 光源の位置設定

光源の位置を変更するには、setLightLocation 関数を使用します。

```
void setLightLocation ( int lightID, float x, float y, float z )
```

引数 x、y、z で光源の存在する方向/位置ベクトルを指定します。

■ 光源の輝度設定

光源の輝度を変更するには、setLightBrightness 関数を使用します。

```
void setLightBrightness ( int lightID, float power )
```

引数 power で光源の輝度を指定します。輝度は 0.0～1.0 の範囲で指定します。全体を一個の光源で照らす場合、大体 0.5 前後の輝度が適正值です。

光源が多数存在する場合は、その分だけ一個当たりの輝度を下げてやる必要があります。しかし輝度の等しい光源が様々な方向に存在すると、陰影が一様化してしまい、立体感が無くなってしまいがちです。そのような場合には、各光源の輝度をある程度ばらけた値に設定する事が有効です。

第二章 モデリング

この章では、標準モデルの配置や移動をはじめ、ポリゴンを用いた独自モデルの作成、マテリアルの設定など、3次元空間の舞台を形づくる「モデリング」の内容を扱います。

モデルの生成と配置

ここでは、モデルの生成と配置について扱います。

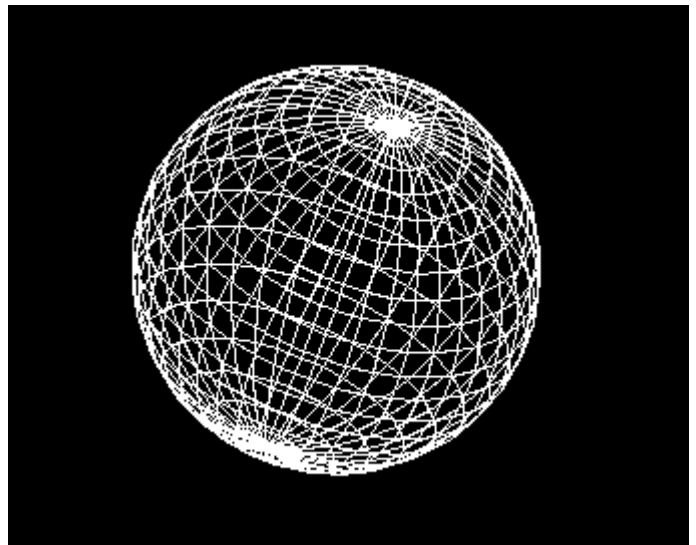
■ モデルとは

モデルとは、3 次元の形状を表現するもので、ポリゴンという微小な多角形の集合で構成されています。ペーパークラフトのようなものを想像するとよいかもしれません。

例えば球モデルは、ちょうど地球儀の経線と緯線で囲まれる四角形の集合のように、規則正しく並んだ大量の四角形ポリゴンで構成されています。

VCSSL では、自分で独自にポリゴンの集合を生成してモデルにまとめる事もできますが、基本的な形状を表現するモデルは標準で用意されています。

これからしばらくは、この標準モデルを中心に扱っていきます。



■ モデルの生成

・標準モデルの生成

標準モデルを生成するには、new〜Model 関数を使用します。〜の部分には標準モデル固有の名称が入ります。標準モデルには様々な種類が存在しますが、各モデルの具体的な生成に関しては次章で扱います。

```
int new〜Model ( 〜 )
```

この関数は標準モデルを生成し、そのモデルの ID を返します。

・独自モデルの生成

自分で生成したポリゴンの集合からモデルを生成するには、newModel 関数を使用します。なお、ポリゴンの生成や制御に関しては、また後の章で扱います。

```
int newModel ( int polygonID[ ] )
```

引数 polygonID には、モデルを構成するポリゴン集合の ID を配列で指定します。この関数は、受け取ったポリゴンの集合を一つのモデルにまとめ、そのモデルの ID を返します。ポリゴン集合をモデルにまとめる利点は、平行移動や回転などを一括して行える事です。

・コピーモデルの生成

newModel 関数では、すでに生成したモデルをコピーし、同型の新しいモデルを作る事も可能です。

```
int newModel ( int copyModelID )
```

引数に copyModelID には、コピーしたいモデルのモデル ID を指定します。

■ モデルの配置

生成したモデルを配置するには、mountModel 関数を使用します。

```
int mountModel ( int modelID, int rendererID )
```

引数 modelID には配置するモデルの ID を、続く引数 rendererID にはレンダラーの ID を指定します。

なお、次のように座標系を指定して配置する事も可能です。座標系に関しては、本文書の後半で扱います。

```
int mountModel ( int modelID, int rendererID, int coordinateID )
```

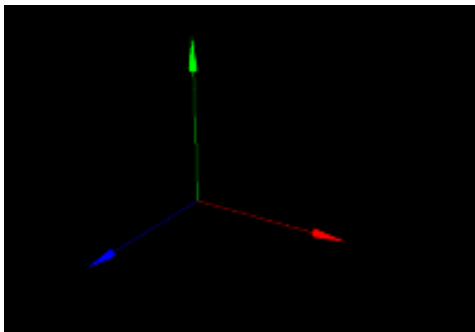
引数 modelID には配置するモデルの ID を、続く引数 rendererID にはレンダラーの ID、最後の引数 coordinateID には座標系の ID を指定します。

なお、VCSSL3.0 以前の世代では、配置には addModel 関数を使用していました。しかし、add～という関数名としては引数の順序が混乱を招くという理由により、VCSSL3.1 以降では、関数名を上記の mountModel に変えたものが追加されました。つまり addModel 関数と mountModel 関数は、名称が異なるだけで全く同一のものです。

各種標準モデル

ここでは、様々な種類の標準モデルについて扱います。

■ 座標軸モデル — AxisModel



座標軸モデル(アキシス)は、3次元空間の方向を指し示すモデルです。3DCGの舞台設計は、最初にこの座標軸モデルを配置する事から始まるとも言える、非常に重要なモデルです。

座標軸モデルを生成するには、newAxisModel 関数を使用します。

```
int newAxisModel ( float lx, float ly, float lz )
```

引数 lx, ly, lz には、それぞれ X 軸、Y 軸、Z 軸方向の長さを指定します。

■ 球モデル — SphereModel



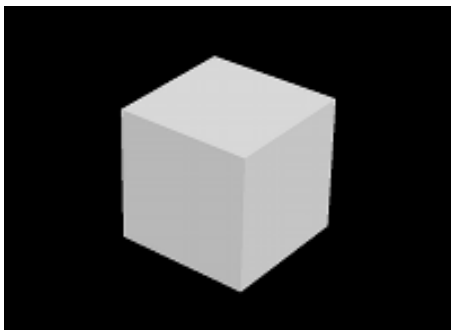
球モデル(スフィア)を生成するには、newSphereModel 関数を使用します。

```
int newSphereModel (
    float rx, float ry, float rz,
    int n1, int n2
)
```

引数 rx、ry、rz には、底面の X、Y、Z 方向半径を指定します。続く引数 n1 には緯度方向のポリゴン数、n2 には経度方向のポリゴン数を指定します。モデル全体のポリゴン数は $n1 \times n2$ となります。

ポリゴン数とは、曲面を表現するのに使用するポリゴンの数です。ポリゴン数を上げるほど曲面がなめらかに、下げるほどカクカクになります。

■ 直方体モデル — BoxModel

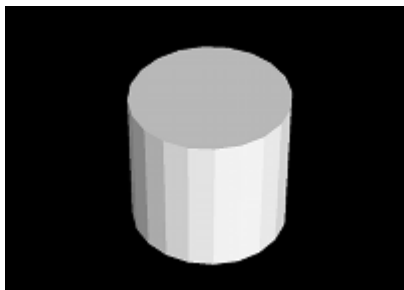


直方体モデル (ボックス) を生成するには、newBoxModel 関数を使用します。

```
int newBoxModel ( float lx, float ly, float lz )
```

引数 lx、ly、lz には、それぞれ X 軸、Y 軸、Z 軸方向の長さを指定します。なお、中心 (重心) が原点に一致するように配置されます。

■ 円柱モデル — CylinderModel

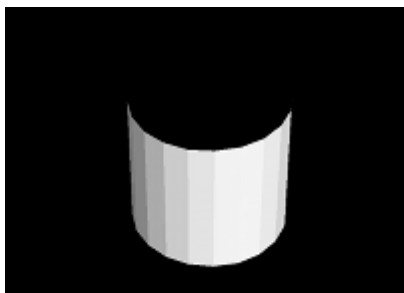


円柱モデル(シリンダー)を生成するには、newCylinderModel 関数を使用します。

```
int newCylinderModel (  
    float rx, float ry, float rz,  
    int n1, int n2  
)
```

引数 lx、ly、lz には、それぞれ X 軸方向半径、Y 軸方向半径、高さを指定します。なお、底面の中心が原点に一致するように配置されます。

■ 底面なしの円筒モデル — TubeModel



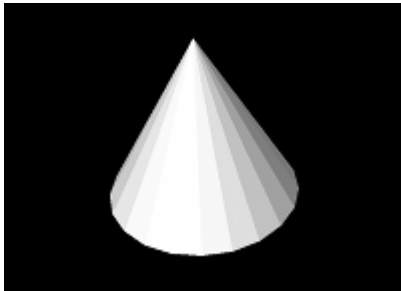
底面なしの円筒モデル(チューブ)を生成するには、newTubeModel 関数を使用します。

```
int newTubeModel (  
    float rx, float ry, float rz,  
    int n1, int n2  
)
```


引数 lx, ly, lz には、それぞれ X 軸方向半径、Y 軸方向半径、高さを指定します。なお、底面の中心が原点に一致するよう配置されます。

続く引数 n1 には緯度方向のポリゴン数、n2 には高さ方向のポリゴン数を指定します。モデル全体のポリゴン数は $n1 \times n2$ となります。

■ 円錐モデル — ConeModel



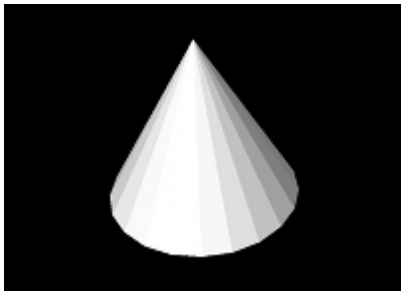
円錐モデル(コーン)を生成するには、newConeModel 関数を使用します。

```
int newConeModel (  
    float rx, float ry, float rz,  
    int n1, int n2  
)
```

引数 lx, ly, lz には、それぞれ X 軸方向半径、Y 軸方向半径、高さを指定します。なお、底面の中心が原点に一致するよう配置されます。

続く引数 n1 には緯度方向のポリゴン数、n2 には高さ方向のポリゴン数を指定します。モデル全体のポリゴン数は $n1 \times n2$ となります。

■ 底面なしの円錐モデル — ShadeModel



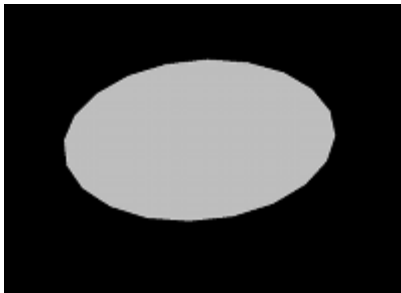
底面なしの円錐モデル(シェード)を生成するには、newShadeModel 関数を使用します。

```
int newShadeModel (  
    float rx, float ry, float rz,  
    int n1, int n2  
)
```

引数 lx, ly, lz には、それぞれ X 軸方向半径、Y 軸方向半径、高さを指定します。なお、底面の中心が原点に一致するよう配置されます。

続く引数 n1 には緯度方向のポリゴン数、n2 には高さ方向のポリゴン数を指定します。モデル全体のポリゴン数は $n1 \times n2$ となります。

■ 円盤モデル — DiskModel



円盤モデル(ディスク)を生成するには、newDiskModel 関数を使用します。

```
int newDiskModel ( float rx, float ry, int n )
```

引数 lx, ly には、それぞれ X 軸方向半径、Y 軸方向半径を指定します。なお、中心(重心)が原点に一致するよう配置されます。

続く引数 n にはポリゴン数を指定します。ポリゴン数とは、曲面を表現するのに使用するポリゴンの数です。ポリゴン数を上げるほど曲面がなめらかに、下げるほどカクカクになります。

・面の向きと、裏面の描画省略

モデルには、表面と裏面があります。例えば円盤モデルは、生成時に Z 軸の上の方が表面となり、通常は表面しか描画されません。つまり裏面から見ても、なにも存在しないように見えます。見えるようにするには、`setModelCull(modelID, false, false)` で裏面の描画省略を無効化します。

■ プログラム例

それでは、上で扱った各種標準モデルを実際に使用してみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis, rendererID ) ;

    // 球モデルを生成して配置
    int sphere = newSphereModel( 0.5, 0.5, 0.5, 10, 7 ) ;
    mountModel( sphere, rendererID ) ;

    // 直方体モデルを生成して配置
    int box = newBoxModel( 1.0, 1.0, 1.0 ) ;
    mountModel( box, rendererID ) ;
    moveModel( box, 1.25, 0.0, 0.0 ) ; // 平行移動
```

```
// 円柱モデルを生成して配置
int cylinder = newCylinderModel( 0.5, 0.5, 1.0, 20, 1 );
mountModel( cylinder, rendererID );
moveModel( cylinder, 2.5, 0.0, 0.0 ); // 平行移動

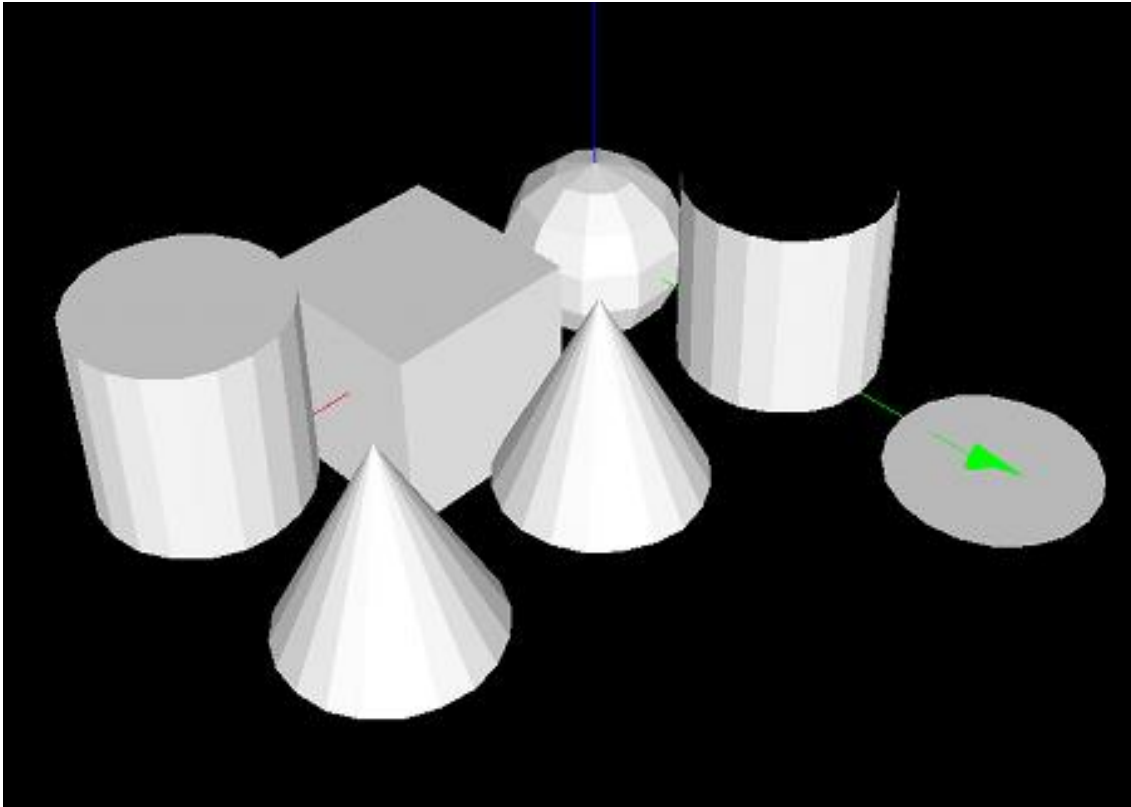
// 底面なしの円筒モデルを生成して配置
int tube = newTubeModel( 0.5, 0.5, 1.0, 20, 1 );
mountModel( tube, rendererID );
moveModel( tube, 0.0, 1.25, 0.0 ); // 平行移動

// 円錐モデルを生成して配置
int cone = newConeModel( 0.5, 0.5, 1.0, 20, 1 );
mountModel( cone, rendererID );
moveModel( cone, 1.25, 1.25, 0.0 ); // 平行移動

// 底面なしの円錐モデルを生成して配置
int shade = newShadeModel( 0.5, 0.5, 1.0, 20, 1 );
mountModel( shade, rendererID );
moveModel( shade, 2.5, 1.25, 0.0 ); // 平行移動

// 円盤モデルを生成して配置
int disk = newDiskModel( 0.5, 0.5, 20 );
mountModel( disk, rendererID );
moveModel( disk, 0.0, 2.5, 0.0 ); // 平行移動

}
```



このプログラムを実行すると、黒い背景に様々な標準モデルが表示されます。

円筒の内側は裏面なので、標準では描画されません。描画するには円筒モデル生成後に `setModelCull(tube, false, false) ;` と追記し、裏面の描画省略を無効化してください。底面なしの円錐モデルなどについても同様です。

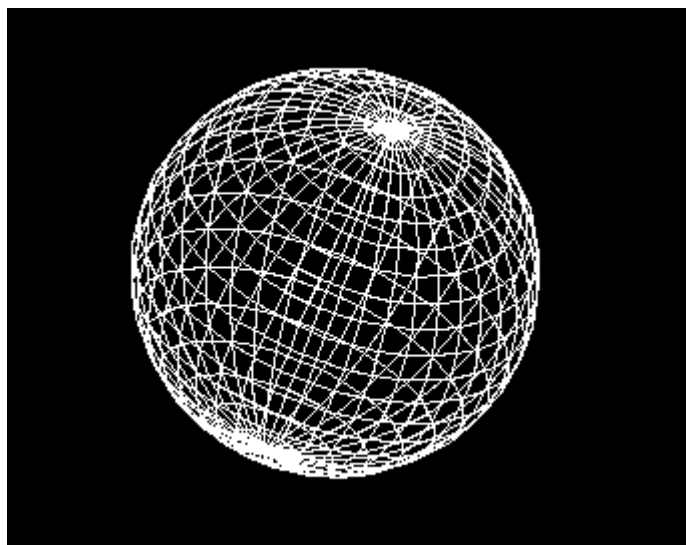
ポリゴンの生成と配置

ここでは、ポリゴンの生成と配置について扱います。

■ ポリゴンとは

ポリゴンとは、3 次元の微小な面などを表現するもので、3 次元形状を構成する最も細かい基本要素となるものです。一般に、モデルは複数のポリゴンによって構成されており、特に曲面を持つモデルでは大量のポリゴンを使用します。

なお、ポリゴンは英語で多角形という意味であるため、厳密な意味でのポリゴンは三角形や四角形などに限定されます。しかし VCSSL Graphics3D では少し意味を拡張し、「ベクトルの集合で構成される描画基本要素」の事をすべてポリゴンと呼びます。従って三角形や四角形はもちろん、点や線、加えて文字や画像などの描画基本要素もポリゴンと呼びます。



■ ポリゴンの生成

ポリゴンを生成するには、new〜Polygon 関数を使用します。〜の部分にはポリゴンの種類に固有の名称が入ります。ポリゴンには様々な種類が存在しますが、各ポリゴンの具体的な生成に関しては次章で扱います。

```
int new〜Polygon ( 〜 )
```

この関数はポリゴンを生成し、そのモデルの ID を返します。

なお、ポリゴンの集合は、newModel 関数を用いてモデルにまとめる事が可能です。

```
int newModel ( int polygonID[ ] )
```

引数 polygonID には、モデルを構成するポリゴン集合の ID を配列で指定します。この関数は、受け取ったポリゴンの集合を一つのモデルにまとめ、そのモデルの ID を返します。ポリゴン集合をモデルにまとめる利点は、平行移動や回転などを一括して行える事です。

■ ポリゴンの配置

生成したポリゴンをモデルにまとめず、直接配置するには、mountPolygon 関数を使用します。

```
int mountPolygon ( int polygonID, int rendererID )
```

引数 polygonID には配置するポリゴンの ID を、続く引数 rendererID にはレンダラーの ID を指定します。

なお、次のように座標系を指定して配置する事も可能です。座標系に関しては、本文書の後半で扱います。

```
int mountPolygon ( int polygonID, int rendererID, int coordinateID )
```

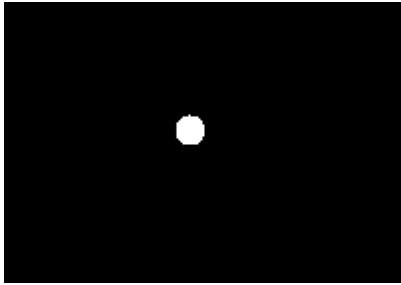
引数 polygonID には配置するポリゴンの ID を、続く引数 rendererID にはレンダラーの ID、最後の引数 coordinateID には座標系の ID を指定します。

なお、VCSSL3.0 以前の世代では、配置には addPolygon 関数を使用していました。しかし、add ~ という関数名としては引数の順序が混乱を招くという理由により、VCSSL3.1 以降では、関数名を上記の mountPolygon に変えたものが追加されました。つまり addPolygon 関数と mountPolygon 関数は、名称が異なるだけで全く同一のものです。

各種ポリゴン

ここでは、様々な種類のポリゴンを扱います。

■ 点ポリゴン — PointPolygon



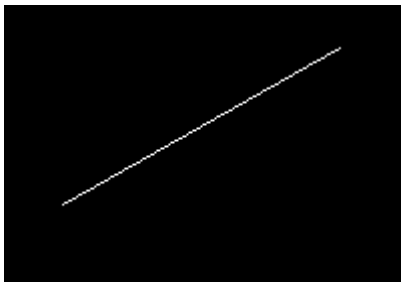
点ポリゴンは、点を表現するためのポリゴンです。本来は小さな点を表現するために使うものですが、球を大きな点ポリゴンで代用するといった使い方もできます。その場合立体感は無くなりますが、描画負荷を大幅に軽減する事が可能です。

点ポリゴンを生成するには、newPointPolygon 関数を使用します。

```
int newPointPolygon (  
    float x, float y, float z,  
    float size  
)
```

引数 (x, y, z) で中心の位置を、続く引数 size で半径を指定します。

■ 線ポリゴン — LinePolygon



線ポリゴンは、直線を表現するためのポリゴンです。1 ピクセルの細い線のほか、遠近感のある

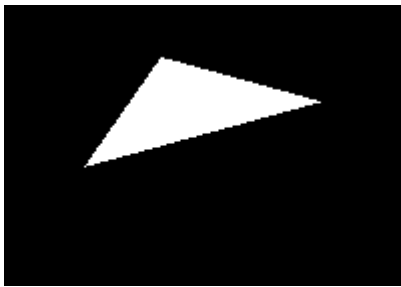
太い線も扱えます。

線ポリゴンを生成するには、newLinePolygon 関数を使用します。

```
int newLinePolygon (  
    float x1, float y1, float z1,  
    float x2, float y2, float z2  
)
```

引数 (x1, y1, z1) で始点の位置を指定し、(x2, y2, z2) で終点の位置を指定します。最後に float の引数を追加して太さを指定する事も可能で、その場合は遠近感のある太い線になります。

■ 三角形ポリゴン — TrianglePolygon



三角形ポリゴンは、微小な三角形を表現するためのポリゴンです。

三角形ポリゴンを生成するには、newTrianglePolygon 関数を使用します。

```
int newTrianglePolygon (  
    float x1, float y1, float z1,  
    float x2, float y2, float z2,  
    float x3, float y3, float z3  
)
```

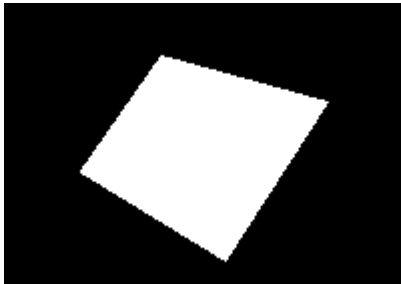
引数 (x1, y1, z1) で頂点 1 の位置を、(x2, y2, z2) で頂点 2 の位置を、(x3, y3, z3) で頂点 3 の位置を指定します。

三角形ポリゴンには、裏表の向きが存在します。頂点1、2、3が反時計回りに見える方向から見ると、それが表面となります。別の言い方をすると、頂点1、2、3の回転方向へ右ネジを回すと進む向

きが、表面の方向ベクトルとなります。

三角形ポリゴンは、表側から見た場合のみ描画され、裏側から見てもなにも存在しないように見えます。

■ 四角形ポリゴン — QuadranglePolygon



四角形ポリゴンは、微小な四角形を表現するためのポリゴンです。

四角形ポリゴンを生成するには、newQuadranglePolygon 関数を使用します。

```
int newQuadranglePolygon (  
    float x1, float y1, float z1,  
    float x2, float y2, float z2,  
    float x3, float y3, float z3,  
    float x4, float y4, float z4  
)
```

引数 (x1, y1, z1) で頂点 1 の位置を、(x2, y2, z2) で頂点 2 の位置を、(x3, y3, z3) で頂点 3 の位置を、(x4, y4, z4) で頂点 4 の位置を指定します。

四角形ポリゴンには、裏表の向きが存在します。頂点 1、2、3、4 が反時計回りに見える方向から見ると、それが表面となります。別の言い方をすると、頂点 1、2、3、4 の回転方向へ右ネジを回すと進む向きが、表面の方向ベクトルとなります。

四角形ポリゴンは、表側から見た場合のみ描画され、裏側から見てもなにも存在しないように見えます。

■ 画像ポリゴン — ImagePolygon

画像ポリゴンは、画像ファイルの内容や、別のレンダラーでの描画結果など、グラフィックスデータ

の内容を表示するためのポリゴンです。画像ポリゴンで表現される画像は、いわゆるテクスチャマッピングでは無く、常に画面の方向を向いている画像として扱われます。

グラフィックスポリゴンを生成するには、newImagePolygon 関数を使用します。

```
int newImagePolygon (  
    float x, float y, float z,  
    float lx, float ly,  
    int graphicsID,  
)
```

引数 (x, y, z) で表示位置を、続く引数 lx, ly で表示する際の大きさ(幅と高さ)を、最後の引数 graphicsID で画像のグラフィックスデータの ID を指定します。

画像の位置は、画像の左下が (x, y, z) に一致するように表示されます。左上では無い事にご注意ください。また、画像の大きさ(幅と高さ)は、画面上のピクセル数やポイント数では無く、3次元空間における距離単位で扱われます。

・オフセット設定

画像ポリゴンは、画像の左下が指定位置に一致するように描画されます。しかし、これを画像中心に一致するようにしたい場合などもあります。そのような場合には、オフセット設定を行います。オフセット設定とは、テキストポリゴンや画像ポリゴンを、描画時に上下左右方向にずらす設定です。

オフセット設定を行うには、setPolygonOffset 関数を使用します。

```
int setPolygonOffset (  
    int polygonID,  
    float dx, float dy,  
)
```

最初の引数で設定対象のポリゴン ID を、続く引数 dx で右方向へのオフセット量を、dy で上方向へのオフセット量を指定します。すると、描画時に画面上下左右方向へオフセットされます。なお、オフセット量は、画面上のピクセル数では無く、3次元空間における距離単位で扱われます。

■ テキストポリゴン — TextPolygon

テキストポリゴンは、文字列を表現するためのポリゴンです。テキストポリゴンで表現される文字列は、いわゆる立体文字ではなく、常に画面の方向を向いている文字として扱われます。モデルの横にコメントなどを添えるのに使用します。

テキストポリゴンを生成するには、newTextPolygon 関数を使用します。

```
int newTextPolygon (  
    float x, float y, float z,  
    float size,  
    string text,  
)
```

引数 (x, y, z) で表示位置を、続く引数 size で文字の大きさ(高さ)を、最後の引数 text で文字を指定します。text の内容が単一の文字でなく文字列の場合は、画面上にも文字列が表示されます。

文字の位置は、text に指定した最初の文字の左下が (x, y, z) に一致するように表示されます。また、文字の大きさ(高さ)は、画面上のピクセル数やポイント数ではなく、3次元空間における距離単位で扱われます。

・フォントの指定

テキストポリゴンのフォントを指定するには、setPolygonFont 関数を使用します。ただし、どのようなフォントが使用できるかは、環境に依存しますのでご注意ください。

```
int setPolygonFont ( int polygonID, string fontName )
```

最初の引数 polygonID で設定対象のポリゴンの ID を、続く引数 fontName でフォント名を指定します。

・オフセット設定

テキストポリゴンは、文字の左下が指定位置に一致するように描画されます。しかし、これを中心に一致するようにしたい場合などもあります。そのような場合には、オフセット設定を行います。オフセット設定とは、テキストポリゴンやグラフィックスポリゴンを、描画時に上下左右方向にずらす設定です。

オフセット設定を行うには、setPolygonOffset 関数を使用します。

```
int setPolygonOffset (
    int polygonID,
    float dx, float dy,
)
```

最初の引数で設定対象のポリゴン ID を、続く引数 dx で右方向へのオフセット量を、最後の引数で上方向へのオフセット量を指定します。すると、描画時に画面上下左右方向へオフセットされます。

なお、オフセット量は、画面上のピクセル数やポイント数ではなく、3 次元空間における距離単位で扱われます。

■ プログラム例

それでは、これまでに扱った各種ポリゴン（グラフィックスとテキストを除く）を、実際に使用してみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定（省略可能）
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;
```

```

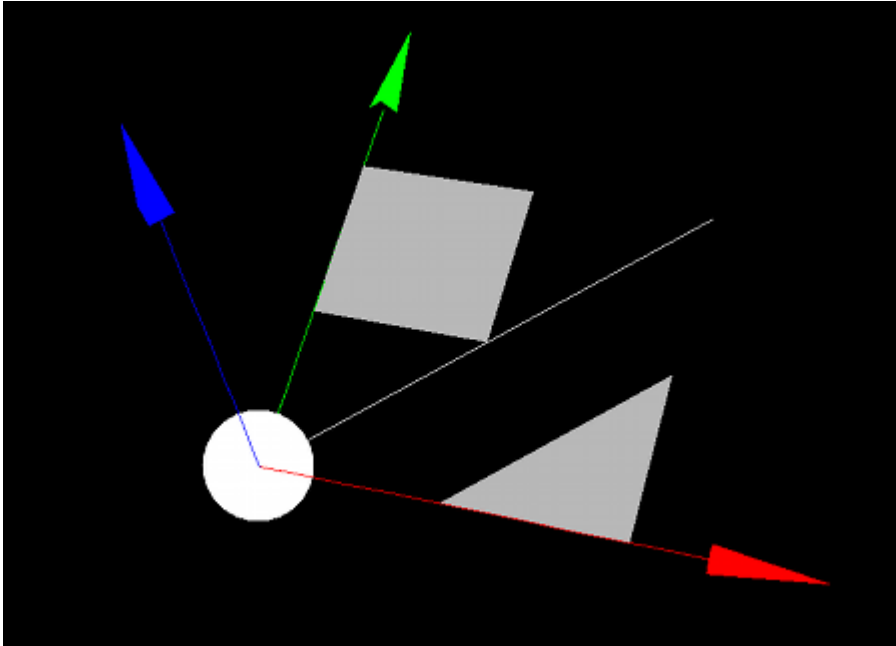
// 点ポリゴンを生成して配置
int point = newPointPolygon( 0.0,0.0,0.0, 0.3 );
mountPolygon( point, rendererID );

// 線ポリゴンを生成して配置
int line = newLinePolygon( 0.0,0.0,0.0, 2.0,2.0,0.0 );
mountPolygon( line, rendererID );

// 三角形ポリゴンを生成して配置
int triangle = newTrianglePolygon(
    0.0,0.0,0.0,
    1.0,0.0,0.0,
    1.0,1.0,0.0
);
mountPolygon( triangle, rendererID );
movePolygon( triangle, 1.0, 0.0, 0.0 ); // 平行移動

// 四角形ポリゴンを生成して配置
int quad = newQuadranglePolygon(
    0.0,0.0,0.0,
    1.0,0.0,0.0,
    1.0,1.0,0.0,
    0.0,1.0,0.0
);
mountPolygon( quad, rendererID );
movePolygon( quad, 0.0, 1.0, 0.0 ); // 平行移動
}

```



このプログラムを実行すると、黒い背景に様々なポリゴンが表示されます。

立体の移動

ここでは、モデルやポリゴンの移動を扱います。

■ モデルの移動

アニメーションなどで、モデルを動的に移動させ続ける場合には、後の章で扱う座標系を使用します。しかし、一度だけ移動させるだけといった場合には、ここで扱う moveModel 関数で簡単に行う事ができます。

モデルの移動を行うには、moveModel 関数を使用します。

```
void moveModel (  
    int modelID,  
    float dx, float dy, float dz  
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数 dx、dy、dz では、それぞれ X、Y、Z 方向の平行移動距離を指定します。

■ ポリゴンの移動

ポリゴンの移動を行うには、movePolygon 関数を使用します。

```
void movePolygon (  
    int polygonID,  
    float dx, float dy, float dz  
)
```

最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 dx、dy、dz では、

それぞれ X、Y、Z 方向の移動距離を指定します。

■ プログラム例

実際に球モデルを配置し、Z 軸方向へ移動させてみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

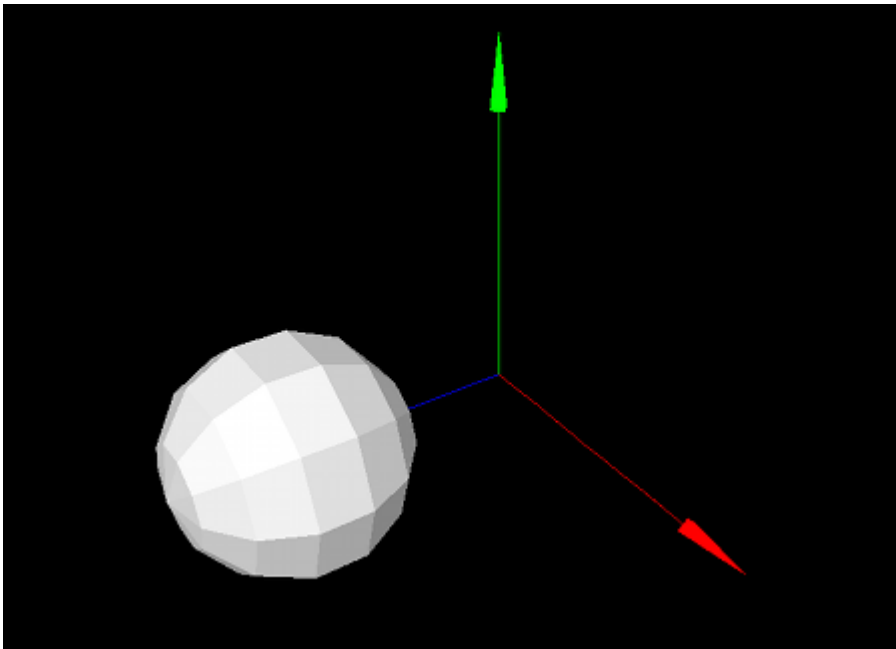
    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis, rendererID ) ;

    // 球モデルを生成して配置
    int sphere = newSphereModel( 1.0, 1.0, 1.0, 10, 7 ) ;
    mountModel( sphere, rendererID ) ;

    // Z 軸方向に移動
    moveModel( sphere, 0.0, 0.0, 2.0 ) ;

}
```



このプログラムを実行すると、黒い画面に白い球が表示されます。白い球は、Z 軸方向に 2 だけ移動されています。

アニメーションで移動させたい場合などは、以下のように onUpdate 関数内で少しずつ移動させます。onUpdate 関数は、フレームワークから 1 秒間に数十回程度、自動で呼び出されます。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// モデルの ID を控えておく変数
int axis, sphere ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

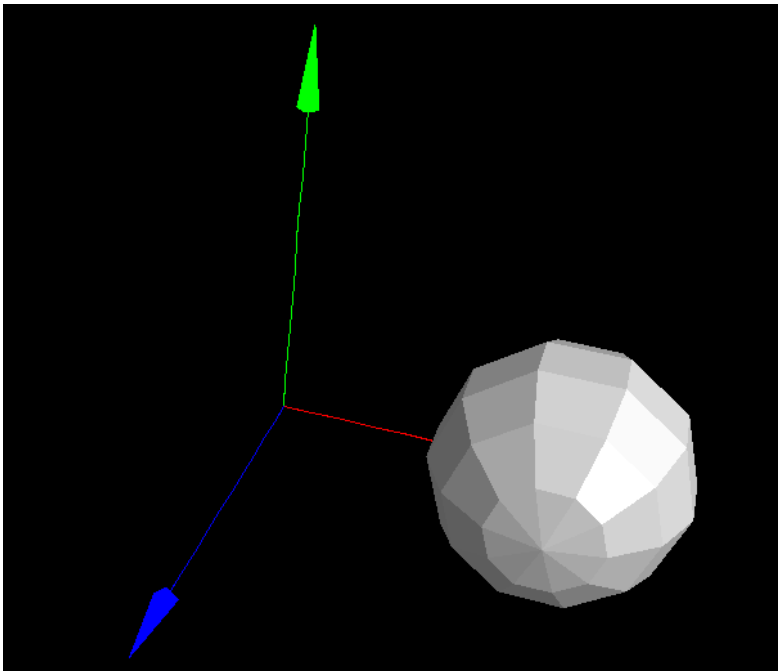
    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;
```

```
// 座標軸モデルを生成して配置
axis = newAxisModel( 3.0, 3.0, 3.0 );
mountModel( axis, rendererID );

// 球モデルを生成して配置
sphere = newSphereModel( 1.0, 1.0, 1.0, 10, 7 );
mountModel( sphere, rendererID );
}

// 画面更新周期ごとに、毎秒数十回呼び出される関数
void onUpdate (int renderer) {

    // X 軸方向に少しずつ移動
    moveModel( sphere, 0.02, 0.0, 0.0 );
}
```



このプログラムを実行すると、白い球が X 軸方向に、アニメーション的にゆっくりと動いていきます。

立体の回転

ここでは、立体の回転を扱います。

■ モデルの回転

アニメーションなどで、モデルを動的に回転させ続ける場合には、後の章で扱う座標系を使用します。しかし、一度だけ回転させるだけといった場合には、ここで扱う `rotModel` などの関数で簡単に行う事ができます。

平行移動などとは異なり、回転には複数の関数が存在します。用途に応じて使い分けてください。

・座標軸まわりの回転

座標軸まわりの回転を行うには、`rotModelX`、`rotModelY`、`rotModelZ`、関数を使用します。

```
void rotModelX ( int modelID, float angle )  
void rotModelY ( int modelID, float angle )  
void rotModelZ ( int modelID, float angle )
```

3つの関数がありますが、それぞれ X、Y、Z 方向の回転を扱います。最初の引数 `modelID` では、設定対象のモデルの ID を指定します。続く引数 `angle` では、軸まわりの回転角度を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアン（後述）を使用します。

・任意方向ベクトルまわりの回転

回転軸を任意の方向ベクトルとするには、`rotModel` 関数を使用します。

```
void rotModel (
    int modelID,
    float angle,
    float vx, float vy, float vz
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。残りの引数 vx、vy、vz では、回転軸のベクトル成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアン（後述）を使用します。

・任意の原点と方向を持つベクトルまわりの回転

回転軸を、任意の原点と方向を持つベクトルとするには、rotModel 関数の引数を増やして使用します。

```
void rotModel (
    int modelID,
    float angle,
    float vx, float vy, float vz,
    float px, float py, float pz
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。次の引数 angle では、軸まわりの回転角度を指定します。続く引数 vx、vy、vz では、回転軸の方向成分を指定します。残りの引数 px、py、pz では、回転軸の原点成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアン（後述）を使用します。

■ ポリゴンの回転

ポリゴンの回転も、モデルと同様に様々な種類が用意されています。

・座標軸まわりの回転

座標軸まわりの回転を行うには、rotPolygonX、rotPolygonY、rotPolygonZ、関数を使用します。

```
void rotPolygonX ( int polygonID, float angle )  
void rotPolygonY ( int poygonID, float angle )  
void rotPolygonZ ( int polygonID, float angle )
```

3つの関数がありますが、それぞれ X、Y、Z 方向の回転を扱います。最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアン（後述）を使用します。

・任意方向ベクトルまわりの回転

回転軸を任意の方向ベクトルとするには、rotPolygon 関数を使用します。

```
void rotPolygon (  
    int polygonID,  
    float angle,  
    float vx, float vy, float vz  
)
```

最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。残りの引数 vx、vy、vz では、回転軸のベクトル成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアン（後述）を使用します。

・任意の原点と方向を持つベクトルまわりの回転

回転軸を、任意の原点と方向を持つベクトルとするには、rotPolygon 関数の引数を増やして使用します。

```
void rotPolygon (  
    int polygonID,  
    float angle,  
    float vx, float vy, float vz,  
    float px, float py, float pz  
)
```

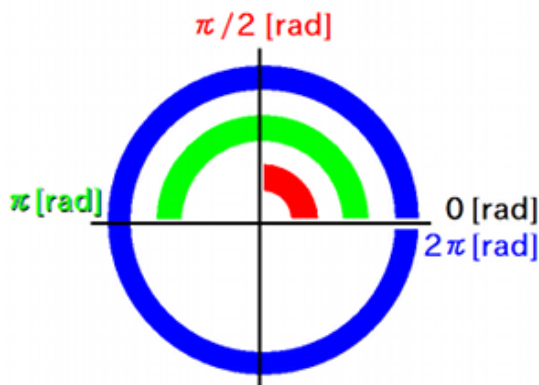
最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。次の引数 angle では、軸まわりの回転角度を指定します。続く引数 vx、vy、vz では、回転軸の方向成分を指定します。残りの引数 px、py、pz では、回転軸の原点成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアン（後述）を使用します。

■ ラジアン [rad] とは

角度の指定には、ラジアンという単位を使用します。これは主に理工学系の分野で多用される単位で、180 度がちょうど π (円周率) ラジアンとなるような単位です。つまり 90 度なら $\pi/2$ ラジアン、45 度なら $\pi/4$ ラジアンとなります。

なお、円周率の値は Math ライブラリに、PI という名前の float 型定数として用意されています。



■ プログラム例

実際に箱型モデルを配置し、Z 軸まわりに 45 度 = $\pi/4$ ラジアン回転させてみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math; // 円周率(Pi)の値を使うため

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

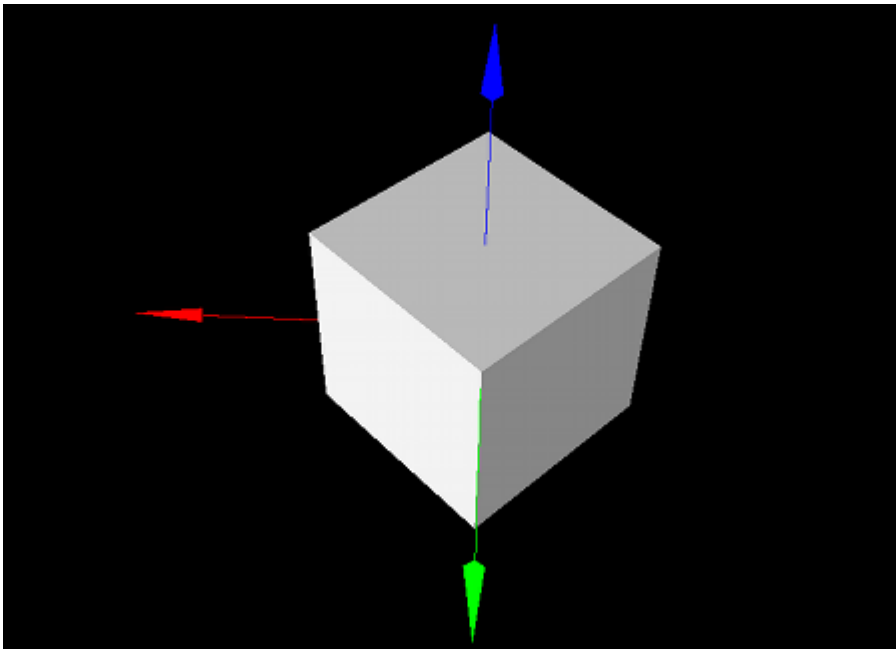
    // 画面サイズや背景色の設定(省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis, rendererID ) ;

    // 箱型モデルを生成して配置
    int box = newBoxModel( 2.0, 2.0, 2.0 ) ;
    mountModel( box, rendererID ) ;

    // Z 軸まわりに 45 度回転
    rotModelZ( box, PI/4.0 );

}
```

このプログラムを実行すると、黒い画面に白い箱が表示されます。箱は、Z 軸を回転軸として 45 度 $=\pi/4$ ラジアンだけ回転されています。

アニメーションで回転させたい場合などは、以下のように onUpdate 関数内で少しずつ回転させます。onUpdate 関数は、フレームワークから 1 秒間に数十回程度、自動で呼び出されます。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math; // 円周率(Pi)の値を使うため

// モデルの ID を控えておく変数
int axis, box ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

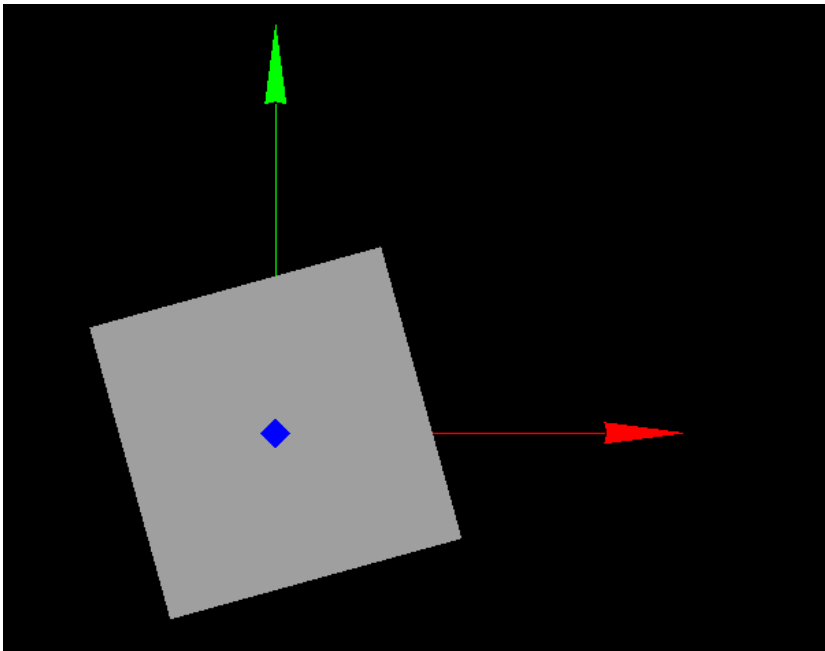
    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;
```

```
// 座標軸モデルを生成して配置
axis = newAxisModel( 3.0, 3.0, 3.0 );
mountModel( axis, rendererID );

// 箱型モデルを生成して配置
box = newBoxModel( 2.0, 2.0, 2.0 );
mountModel( box, rendererID );
}

// 画面更新周期ごとに、毎秒数十回呼び出される関数
void onUpdate (int renderer) {

    // Z 軸まわりに少しずつ回転
    rotModelZ( box, 0.02 );
}
```



このプログラムを実行すると、白い箱がアニメーション的に、ゆっくりと回転し続けます。

立体の拡大と縮小

ここでは、モデルやポリゴンの拡大と縮小を扱います。

■ モデルの拡大と縮小

モデルの拡大と縮小を行うには、scaleModel 関数を使用します。

```
void scaleModel (  
    int modelID,  
    float sx, float sy, float sz  
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数 sx、sy、sz では、それぞれ X、Y、Z 方向のスケーリング倍率を指定します。倍率を 1.0 より大きくすると拡大に、小さくすると縮小になります。倍率を変えたくない方向には 1.0 を指定します。

■ ポリゴンの拡大と縮小

ポリゴンの拡大と縮小を行うには、scalePolygon 関数を使用します。

```
void scalePolygon (  
    int polygonID,  
    float sx, float sy, float sz  
)
```

最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 sx、sy、sz では、それぞれ X、Y、Z 方向のスケーリング倍率を指定します。倍率を 1.0 より大きくすると拡大に、小さくすると縮小になります。倍率を変えたくない方向には 1.0 を指定します。

■ プログラム例

実際に球モデルを配置し、Z 軸方向へ 2 倍に引き伸ばしてみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

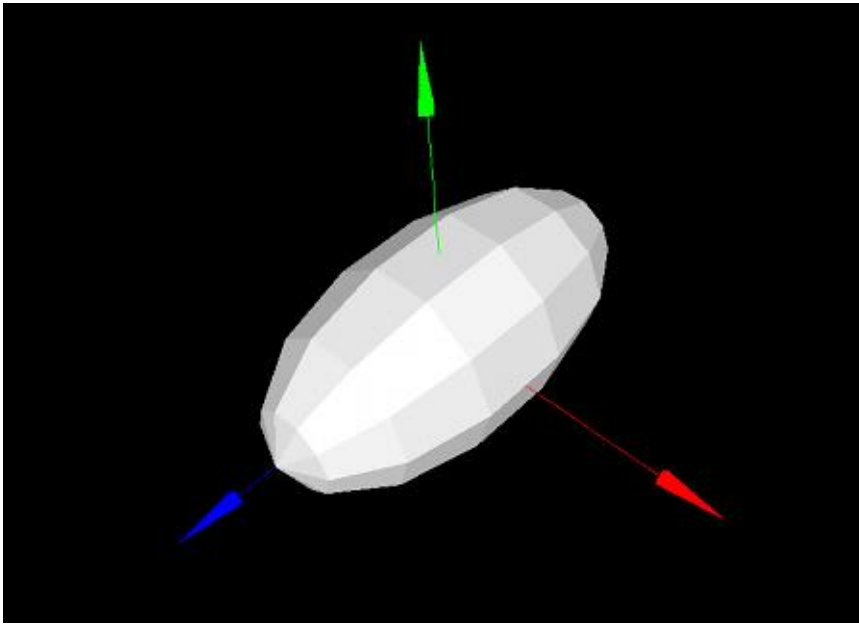
    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis, rendererID ) ;

    // 球モデルを生成して配置
    int sphere = newSphereModel( 1.0, 1.0, 1.0, 10, 7 ) ;
    mountModel( sphere, rendererID ) ;

    // Z 軸方向に 2 倍引き伸ばす
    scaleModel( sphere, 1.0, 1.0, 2.0 ) ;

}
```



このプログラムを実行すると、黒い画面に白い球が表示されます。白い球は、Z 軸方向に 2 倍引き伸ばされています。

立体の反転

ここでは、モデルやポリゴンの反転を扱います。立体の反転には、面の向きを裏表で反転させる「リバース反転」と、形状を鏡映しに反転させる「ミラー反転」が存在します。

■ モデルのリバース反転

モデルの面には、裏表の向きが存在します。面は表側から見た場合のみ描画され、裏側から見ても何も存在しないように見えます。標準モデルでは、通常は外側が表面となっており、従って内側からは何も見えません。

しかし、例えばゲーム制作などにおいて、天球を作成するような場合や、トンネル状の通路を作成するような場合など、モデルの内側を表面にしたい場合がしばしばあります。このような場合には、リバース反転を行う事で、モデルの裏表を反転させる事ができます。

モデルのリバース反転を行うには、reverseModel 関数を使用します。

```
void reverseModel ( int modelID )
```

引数 modelID では、設定対象のモデルの ID を指定します。

■ ポリゴンのリバース反転

ポリゴンにも、三角形ポリゴンと四角形ポリゴンには裏表が存在します。これらポリゴンに対してリバース反転を行うには、reversePolygon 関数を使用します。

```
void reversePolygon ( int polygonID )
```

引数 polygonID では、設定対象のポリゴンの ID を指定します。

■ モデルのミラー反転

モデルを X 軸などの方向へ、鏡映しに反転させたいといった場合には、ミラー反転を行います。モデルのミラー反転を行うには、mirrorModel 関数を使用します。

```
void mirrorModel (  
    int modelID,  
    bool x, bool y, bool z  
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数 x、y、z では、それぞれ X、Y、Z 方向のミラー反転の有無を指定します。

■ ポリゴンのミラー反転

モデルと同様、ポリゴンのミラー反転を行うには、mirrorPolygon 関数を使用します。

```
void mirrorPolygon (  
    int polygonID,  
    bool x, bool y, bool z  
)
```

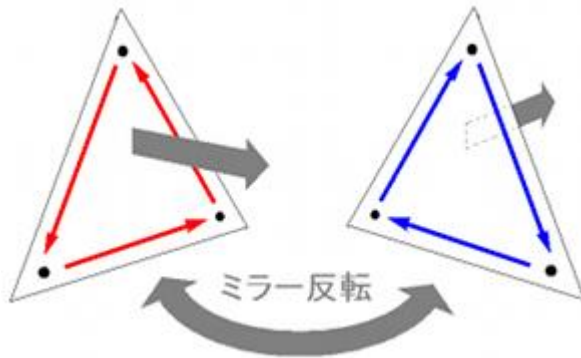
最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 x、y、z では、それぞれ X、Y、Z 方向のミラー反転の有無を指定します。

■ ミラー反転では、面の裏表も反転してしまう

ミラー反転を行う際には、注意しておく点があります。それは、ミラー反転を行うと、同時に面の裏表も反転してしまうという点です。従って、モデルやポリゴンに対してミラー反転を一度行う度に、reverseModel 関数や reversePolygon 関数も同時に呼び出して、面の裏表を正しくしてやる必要

があります。

このようになる理由は、時計回りを鏡映反転すると反時計回りになる事に由来しています。面の裏表の向きは、ポリゴンを構成する頂点が反時計回りに並ぶ場合に表と判定されていますが、この向きはミラー反転すると逆転します。従って、こちら側が表に見えていたポリゴンをミラー反転すると、向こう側が表になってしまうのです。



■ プログラム例

実際に円筒モデルを配置し、Z 方向にミラー反転させてみましょう。一回のミラー反転では面の裏表が逆転するため、同時にリバース反転も行います。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis, rendererID ) ;
```

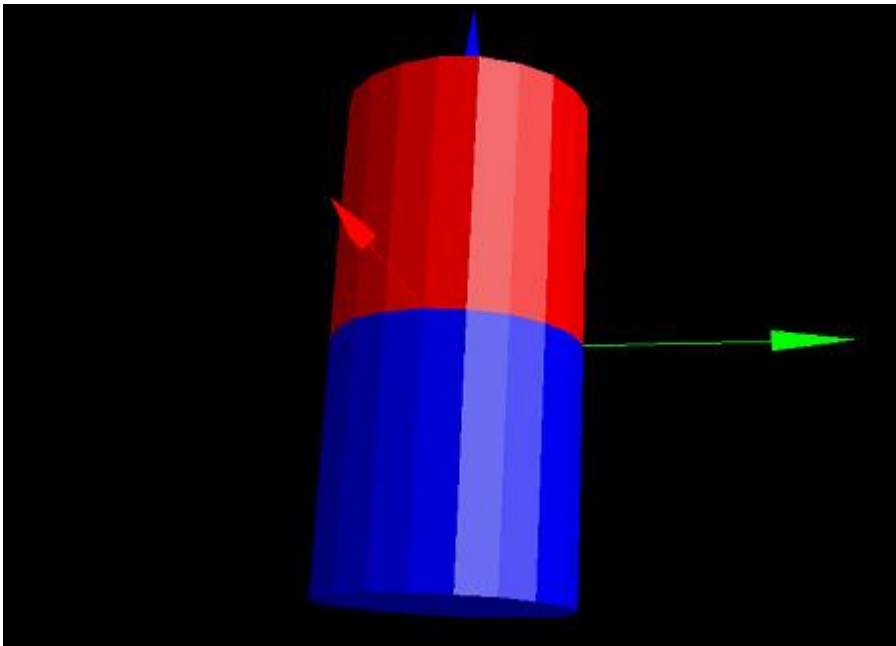


```
// 円筒モデル 1 を生成して配置
int cylinder1 = newCylinderModel( 1.0, 1.0, 2.0, 20, 1 );
setModelColor( cylinder1, 255, 0, 0, 255 ); // 赤色に設定
mountModel( cylinder1, rendererID );

// 円筒モデル 2 を生成して配置
int cylinder2 = newCylinderModel( 1.0, 1.0, 2.0, 20, 1 );
setModelColor( cylinder2, 0, 0, 255, 255 ); // 青色に設定
mountModel( cylinder2, rendererID );

// モデル 2 を Z 軸方向にミラー反転
mirrorModel( cylinder2, false, false, true );
reverseModel( cylinder2 ); // 裏表の反転を補正

}
```



このプログラムを実行すると、黒い画面に赤色と青色の円筒が表示されます。青色の円筒は、ちょうど赤色の円筒を Z 軸方向に反転した形状となっています。

立体の色設定

ここでは、立体の色設定を扱います。

■ モデルの色設定

モデルの色を変更するには、setModelColor 関数を使用します。

```
void setModelColor (  
    int modelID,  
    int red, int green, int blue, int alpha  
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数では、モデルの色成分を指定します。色成分はそれぞれ 0～255 の範囲で指定します。色成分の形式は、RGBA 形式をサポートしています。

■ ポリゴンの色設定

ポリゴンの色を変更するには、setPolygonColor 関数を使用します。

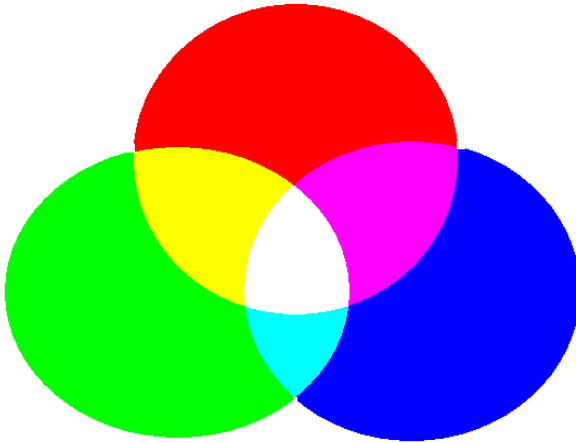
```
void setPolygonColor (  
    int polygonID,  
    int red, int green, int blue, int alpha  
)
```

最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数では、ポリゴンの色成分を指定します。色成分はそれぞれ 0～255 の範囲で指定します。色成分の形式は、RGBA 形式をサポートしています。

■ RGBA 形式とは

RGBA 形式とは、色の三原色である赤 (Red)、緑 (Green)、青 (Blue) の色成分に、アルファ値 (Alpha) を加えた形式です。アルファ値は色の透明度を表す数値で、0 で完全透明になり、最大にすると不透明になります。

それぞれの色成分からの色の合成は、加法混色によって行われます。これは光の重ね合わせと同じ混色方式であり、絵の具の混ぜ合わせ (減法混色) では無い事にご注意ください。例として、(赤, 緑, 青) = (255, 255, 255) は、黒ではなく白になります。また、(255, 255, 0) は黄色に、(0, 255, 255) は水色に、(255, 0, 255) はマゼンタになります。



■ プログラム例

実際に球モデルを配置し、青色に設定してみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {
```

```
// 画面サイズや背景色の設定(省略可能)
setWindowSize(800, 600);
setBackgroundColor(0, 0, 0, 255);

// 球モデルを生成して配置
int sphere = newSphereModel( 1.0, 1.0, 1.0, 10, 7 );
mountModel( sphere, rendererID );

// 球モデルを青色に設定
setModelColor( sphere, 0, 0, 255, 255 );

}
```



このプログラムを実行すると、黒い背景に青い球が描画されます。

立体の形状設定

ここでは、標準モデルやポリゴンの形状設定を扱います。形状設定により、すでに生成したモデルやポリゴンの形状を変更する事ができます。この設定は、アニメーションなど、立体を動的に変形させたい場合などに行います。

■ 標準モデルの形状設定

標準モデルの形状を変更するには、setModelSize 関数を使用します。

```
void setModelSize (  
    int modelID,  
    float l1, float l2, ...  
)
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数 l1, l2, ... では、モデルの形状を特徴付ける長さを指定します。引数 l1, l2, ... の個数は標準モデルの種類によって異なり、その標準モデルを生成する new~Model 関数のものと同様の個数・意味を持ちます。例えば円筒モデル (CylinderModel) なら、l1・l2 で底面の X・Y 方向半径を、l3 で高さを指定します。

・標準モデルのみ使用可能

setModelSize 関数は、標準モデル専用の関数です。ポリゴン集合から生成した独自モデルに対しては使用できません。

■ ポリゴンの形状(頂点座標)設定

ポリゴンの頂点座標を変更するには、setPolygonVertex 関数を使用します。

```
void setPolygonVertex (
    int polygonID,
    float x1, float y1, float z1,
    float x2, float y2, float z2,
    ...
)
```

最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 x1, y1, ... では、ポリゴンの頂点位置を指定します。引数 x1, y1, ... の個数はポリゴンの種類によって異なり、そのポリゴンを生成する new〜Polygon 関数の引数における、座標値指定部分と同様の個数・意味を持ちます。

例えば点ポリゴン(PointModel)なら3つで、(x1, y1, z1)で中心位置を指定し、線ポリゴン(LinePolygon)なら6つで、(x1, y1, z1)と(x2, y2, z2)で線の始点と終点位置を指定します。同様に三角形ポリゴンなら9つ、四角形ポリゴンなら12個の座標値を指定します。

■ プログラム例

実際に球モデルを配置し、時間に伴って変形させてみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // sin 関数を使うため

// モデルの IQ を控えておく変数
int axis, sphere;

// 時刻のカウンタ(画面更新周期単位)
int t = 0 ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {
```

```

// 画面サイズや背景色の設定(省略可能)
setWindowSize(800, 600);
setBackgroundColor(0, 0, 0, 255);

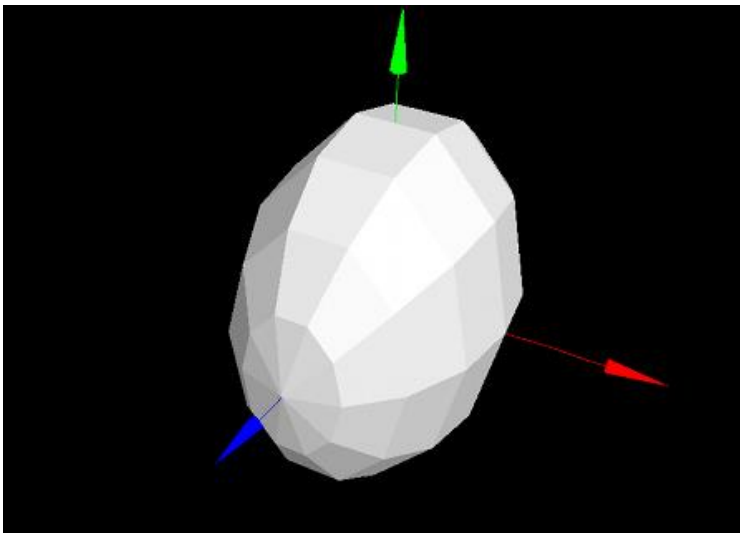
// 座標軸モデルを生成して配置
axis = newAxisModel( 3.0, 3.0, 3.0 );
mountModel( axis, rendererID );

// 球モデルを生成して配置
sphere = newSphereModel( 2.0, 2.0, 2.0, 10, 8 );
mountModel( sphere, rendererID );
}

// 画面更新周期ごとに、毎秒数十回呼び出される関数
void onUpdate (int rendererID) {

    // 球モデルを変形し、時刻カウンタを加算
    setModelSize( sphere, 1.0+sin( 0.1*t ), 2.0, 2.0 );
    t++;
}

```



このプログラムを実行すると、黒い背景に白い球が描画され、アニメーションで潰れたり膨らんだりを繰り返します。

立体のフィル設定

ここでは、標準モデルやポリゴンのフィル設定を扱います。フィル設定により、ポリゴン塗りつぶさずに、境界線のみを描画する事が可能になります。これを利用して、モデルのワイヤーフレーム描画などを行う事も可能です。

■ フィル設定

フィル設定とは、立体を構成するポリゴンを描画する際に、ポリゴン境界線の中を塗りつぶすかどうかの設定です。

フィル設定は、標準状態では有効になっており、立体は平面の集合で構成されているように描画されます。これを無効にする事で、立体が線の集合で構成されているように描画されるようになります(いわゆるワイヤーフレーム描画)。

■ モデルのフィル設定

モデルのフィル設定を行うには、setModelFill 関数を使用します。

```
void setModelFill ( int modelID, bool fill )
```

最初の引数 modelID では、設定対象のモデルの ID を指定します。続く引数 fill では、モデルを構成する全てのポリゴンに対し、描画時に塗りつぶすかどうかを指定します。

■ ポリゴンのフィル設定

ポリゴンのフィル設定を行うには、setPolygonFill 関数を使用します。

```
void setPolygonFill ( int polygonID, bool fill )
```


最初の引数 polygonID では、設定対象のポリゴンの ID を指定します。続く引数 fill では、ポリゴンを描画時に塗りつぶすかどうかを指定します。

■ プログラム例

実際に球モデルを配置し、フィル設定を無効にして、ワイヤーフレーム描画を行ってみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

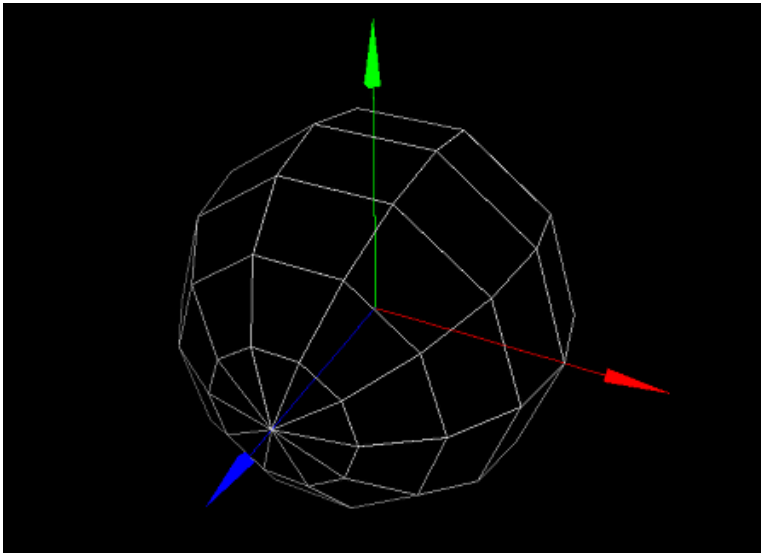
    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 座標軸モデルを生成して配置
    int axis = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis, rendererID ) ;

    // 球モデルを生成して配置
    int sphere = newSphereModel( 2.0, 2.0, 2.0, 10, 8 ) ;
    mountModel( sphere, rendererID ) ;

    // 球モデルのフィル設定を無効に
    setModelFill( sphere, false ) ;

}
```



このプログラムを実行すると、黒い背景に白い球がワイヤーフレームで描画されます。

なお、モデルの内側の面は、普通は見えない部分であるため、標準では描画されないようになっています。しかし上の例のように、ワイヤーフレーム描画を行う場合は、内側が描画されない事で不自然に見えるかもしれません。内側も描画させたい場合は、上のプログラムの例では球モデル生成後に `setModelCull(sphere, false, false)` ; と追記し、ポリゴン裏面の描画省略機能を無効化してください。

立体の材質設定

ここでは、立体の材質設定を扱います。ここで言う「材質」とは、表面の模様などではなく、光の反射に関する特性の事を意味します。

■ 材質パラメータ(マテリアルパラメータ)

・材質と光の反射

実際の物体は、材質によって見え方が異なります。例えば単なる板でも、それが紙なら「サラサラ」に見え、プラスチックならば「ピカピカ」に見えます。そのように見え方が異なる理由は、厳密には様々なものがありますが、恐らく最も大きいのは光の反射効果でしょう。

・様々な反射

一言で光の反射と言っても、それはいくつかの種類に分けられます。

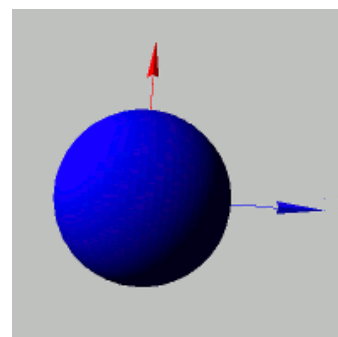
例えばプラスチックの板を例に挙げてみましょう。全体的に一様な明るさで見えている部分は、光がプラスチックに当たり、それが様々な方向に拡散的に反射した光が、我々の目に入って見えているわけです。スポット的に明るい部分(ハイライト)は、光源からまっすぐ飛んできた光がプラスチック表面で全反射したものが見えているわけです。さらに、光に対して裏側にある部分は、光が空気中で散乱されたものが板の裏側まで回りこんで照らしているわけです。

このように、光の反射にも様々なものが存在します。3DCG では、こういった各種の反射のバランスを調整する事で、材質を表現します。このバランスを調整するパラメータの事を材質パラメータ(マテリアルパラメータ)と呼びます。

■ 拡散反射パラメータ(Diffuse)

拡散反射パラメータは、光が立体に当たり、その後様々な方向へ拡散的に反射する度合いを調整するパラメータです。例えるなら、紙に光が当たっている場合に、その明るい面における光の反射の度合いを調整する感じです。

拡散反射パラメータは、立体表面のさらさらな質感に大きく影響します。布や紙では大きめに指定すると良いでしょう。逆に金属などでは小さめに抑えます。



・モデルに対して設定する

モデルに対して拡散反射パラメータを設定するには、setModelDiffuse 関数を使用します。

```
void setModelDiffuse ( int modelID, float ref )
```

最初の引数 modelID には、設定対象のモデルの ID を指定します。続く引数 ref には、拡散反射の強さを 0.0～1.0 の範囲で指定します。

・ポリゴンに対して設定する

ポリゴンに対して拡散反射パラメータを設定するには、setPolygonDiffuse 関数を使用します。

```
void setPolygonDiffuse ( int polygonID, float ref )
```

最初の引数 polygonID には、設定対象のポリゴンの ID を指定します。続く引数 ref には、拡散反射の強さを 0.0～1.0 の範囲で指定します。

■ 回折反射パラメータ (Diffractive)

拡散反射パラメータは、光が立体付近の空気中で散乱され、立体裏側まで回りこむ度合いを調整するパラメータです。この効果が無いと、宇宙空間に浮かぶの天体のように、立体の裏側が完全な真っ暗になってしまいます。

このパラメータは、後で扱う環境光反射パラメータと同じような使い方をします。しかし、環境光反射とは異なり、回折反射では光源方向との角度に応じてグラデーションで照らされます。

・モデルに対して設定する

モデルに対して回折反射パラメータを設定するには、setModelDiffractive 関数を使用します。

```
void setModelDiffractive ( int modelID, float ref )
```

最初の引数 modelID には、設定対象のモデルの ID を指定します。続く引数 ref には、回折反射の

強さを 0.0～1.0 の範囲で指定します。

・ポリゴンに対して設定する

ポリゴンに対して回折反射パラメータを設定するには、setPolygonDiffractive 関数を使用します。

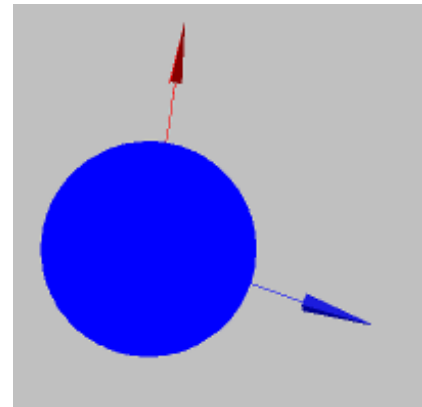
```
void setPolygonDiffractive ( int polygonID, float ref )
```

最初の引数 polygonID には、設定対象のポリゴンの ID を指定します。続く引数 ref には、回折反射の強さを 0.0～1.0 の範囲で指定します。

■ 環境光反射パラメータ (Ambient)

環境光反射パラメータは、立体の周囲から一様に飛んでくる光の効果を調整するパラメータです。現実の光は、立体に向かって直進してくるものだけではありません。まず部屋の壁など、別の方向へ飛んでから散乱され、その後間接的に立体を照らす光も存在します。環境光とは、こういった周囲からの光の事を指します。

このパラメータは、すでに扱った回折反射パラメータと同じような使い方をします。しかし、回折反射とは微妙に異なり、照らし方に角度依存性が全くありません。従って立体全体に一様な色が付くようになります。



・モデルに対して設定する

モデルに対して環境光反射パラメータを設定するには、setModelAmbient 関数を使用します。

```
void setModelAmbient ( int modelID, float ref )
```

最初の引数 modelID には、設定対象のモデルの ID を指定します。続く引数 ref には、環境光反射の強さを 0.0～1.0 の範囲で指定します。

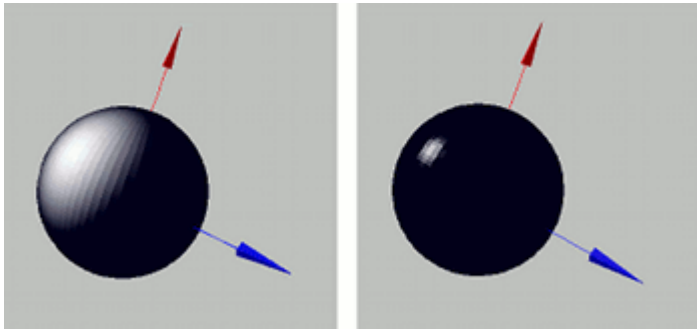
・ポリゴンに対して設定する

ポリゴンに対して環境光反射パラメータを設定するには、setPolygonDiffraction 関数を使用します。

```
void setPolygonAmbient ( int polygonID, float ref )
```

最初の引数 polygonID には、設定対象のポリゴンの ID を指定します。続く引数 ref には、環境光反射の強さを 0.0～1.0 の範囲で指定します。

■ 鏡面反射パラメータ (Specular)



鏡面反射パラメータは、光源から立体へまっすぐ向かってきた光が、立体表面で散乱されずに、そのまま鏡面反射する度合いを調整するパラメータです。

例えばプラスチックなどのつるつるした材質において、光源が白く移りこむ箇所（ハイライト）は、この鏡面反射によるものです。

このパラメータにより、立体表面の滑らかさを表現する事ができます。

・モデルに対して設定する

モデルに対して鏡面反射パラメータを設定するには、setModelSpecular 関数を使用します。

```
void setModelSpecular ( int modelID, float ref, float angle )
```

最初の引数 modelID には、設定対象のモデルの ID を指定します。続く引数 ref には、鏡面反射の強さを 0.0～1.0 の範囲で指定します。最後の引数 angle には、ハイライトの広がり角を 0.0～1.5 までの範囲で指定します。

・ポリゴンに対して設定する

ポリゴンに対して鏡面反射パラメータを設定するには、setPolygonSpecular 関数を使用します。

```
void setPolygonSpecular ( int polygonID, float ref, float angle )
```

最初の引数 polygonID には、設定対象のポリゴンの ID を指定します。続く引数 ref には、鏡面反射の強さを 0.0～1.0 の範囲で指定します。最後の引数 angle には、ハイライトの広がり角を 0.0～1.5 までの範囲で指定します。

■ 自発光パラメータ (Emissive)

自発光パラメータは、一様に発光する度合いを調整するパラメータです。

見え方の効果は環境光反射とほぼ同じですが、こちらは光源の輝度に全く影響されません。

・モデルに対して設定する

モデルに対して自発光パラメータを設定するには、setModelEmissive 関数を使用します。

```
void setModelEmissive ( int modelID, float ref )
```

最初の引数 modelID には、設定対象のモデルの ID を指定します。続く引数 ref には、自発光の強さを 0.0～1.0 の範囲で指定します。

・ポリゴンに対して設定する

ポリゴンに対して自発光パラメータを設定するには、setPolygonEmissive 関数を使用します。

```
void setPolygonEmissive ( int polygonID, float ref )
```

最初の引数 polygonID には、設定対象のポリゴンの ID を指定します。続く引数 ref には、自発光の強さを 0.0～1.0 の範囲で指定します。

第三章 座標変換

この章では、座標系を使用して3次元の舞台に動きをつけたり、アクションゲームのような高度なカメラワークを実現したりするための、「座標変換」について扱います。

座標系

ここでは、動きのある3D表現を扱う上で欠かせない概念である、「座標系」について解説します。「座標系」と言うといかにも数学的なイメージで難しく感じるかもしれませんが、VCSSL Graphics3D では座標系を直感的かつ簡単に扱えるように工夫されています。具体的な利用方法は後に解説するとして、ここでは座標系そのものの意味、また利点などについて解説します。

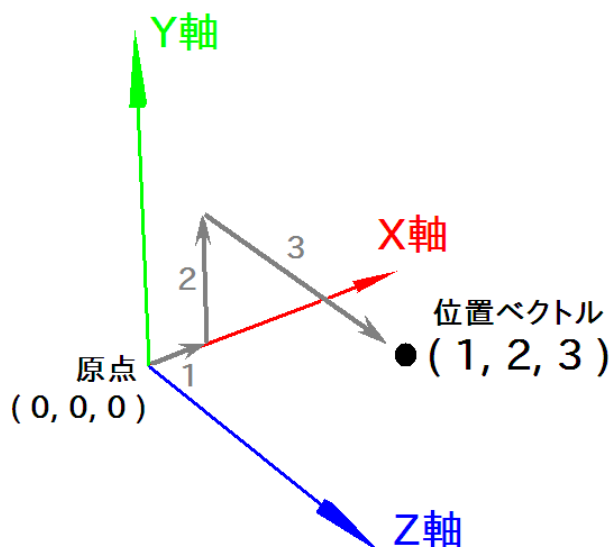
■ 座標系とは

・座標系と位置ベクトル

これまで、立体の移動やポリゴンの頂点位置指定などにおいて、空間中の一点の位置を指定するのに、 (X, Y, Z) の3つの数値の組を使ってきました。このような数値の組を位置ベクトルと呼びます。3つの数値はそれぞれX成分、Y成分、Z成分と呼びます。

実は位置ベクトルだけでは、空間中の位置を指定するのには不十分です。というのも、位置ベクトルは、「座標系」という基準があって、はじめて意味を持つからです。

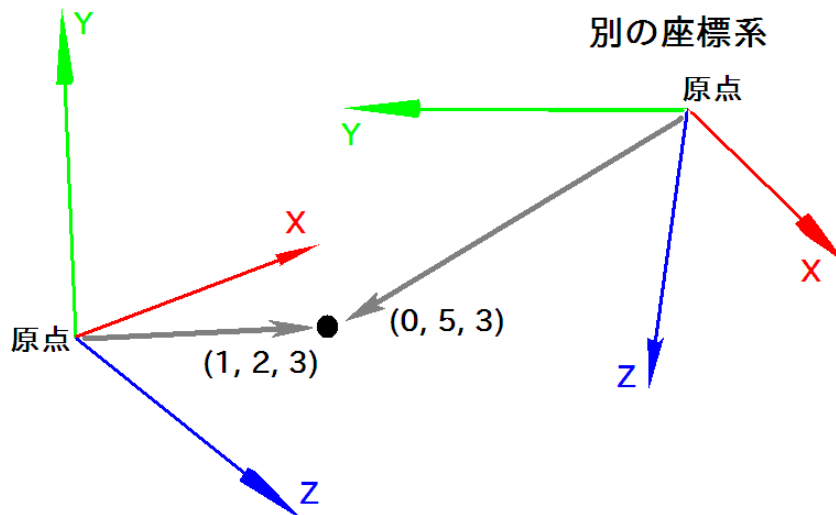
例えば位置ベクトル $(1, 2, 3)$ の指し示す位置は、原点つまり $(0, 0, 0)$ に対応する位置から、X軸方向へ1移動し、Y軸方向へ2移動し、そしてZ軸方向へ3移動した位置です。このように、空間中の位置を指定するには、位置ベクトルのX,Y,Z成分の他に、基準として「原点の位置」と「X,Y,Z軸の方向」が必要になります。これら2つの基準要素をまとめて、「座標系」と呼びます。



・位置ベクトルの成分は座標系によって異なる

空間中のある位置を指し示す位置ベクトルは、座標系が変わると、成分の値も変わります。

例として、ある座標系から見て、位置ベクトル $(1, 2, 3)$ が指し示す位置を考えてみましょう。この位置を別の座標系から見ると、その位置ベクトルは、下図のように $(0, 5, 3)$ となるかもしれません。

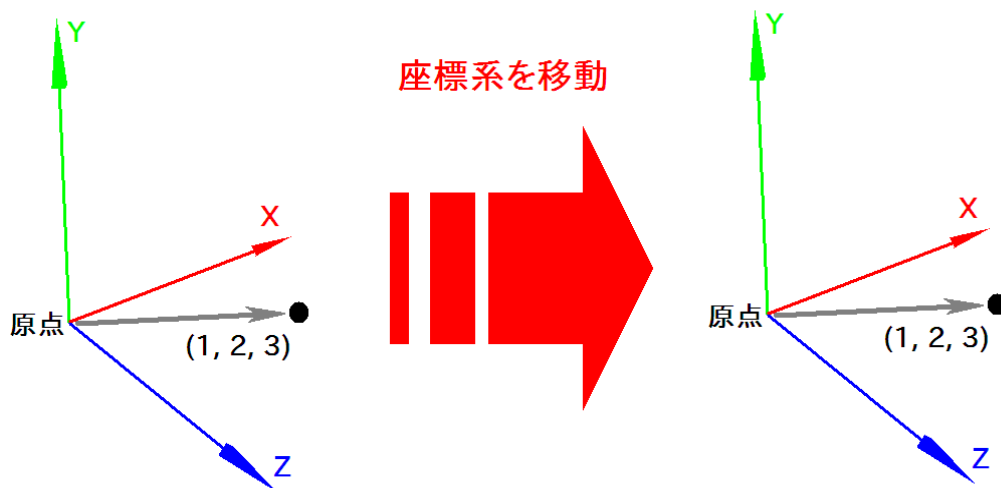


このような不一致が起こる理由は 2 つあります。第一に、両者は原点の位置が違います。つまり両者にとって $(0, 0, 0)$ が指し示す位置がすでに異なるのです。そして第二に、両者の X,Y,Z 各軸が向いている方向も異なりますから、仮に原点の位置が同じであったとしても、位置ベクトルの X,Y,Z 成分は全く異なる値になってしまいます。

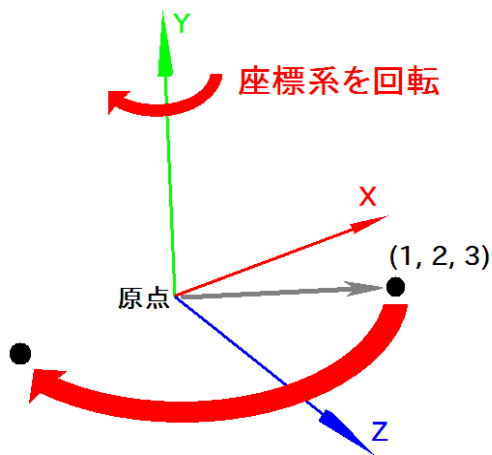
■ 立体を動かす仕組み

座標系は、立体を動的に動かしたい場合に使用します。

例えば、まず適当な座標系を用意し、立体をその座標系の $(1, 2, 3)$ に配置します。この立体を動かしたい場合、座標系のほうを動かします。すると、座標系に配置された立体も一緒に動いて見えます。なぜなら座標系が動くと、その座標系を基準にした $(1, 2, 3)$ が示す場所も動くからです。



同様に、立体を回転させたい場合は、座標系を回転させます。すると座標系のX,Y,Z各軸の位置関係が回転しますから、(1, 2, 3) の示す場所も回転し、結果として立体も一緒に回転します。



これらの操作は、要するに、立体を動かすにあたって「 立体の位置情報を書き換える代わりに、位置の基準(=座標系)を動かした 」という事です。

座標系を動かさずに、立体の位置である (1, 2, 3) の成分そのものを書き換えても、移動や回転は可能です。実際にこれまでに用いてきた moveModel / movePolygon 関数や、rotModel / rotPolygon 関数は、内部でそのような処理を行っています。

しかし、複数のモデルを一緒に動かしたい場合などには、座標系を動かすほうが遥かに簡単に処理を行えます。例えば「自動車」を動かしたい場合には、それを構成するタイヤモデルやボディモデル、運転手モデルなどを一つずつ動かすのはとても面倒です。そこで適当な座標系を用意し、自動車の部品を全てそこに配置してやれば、あとは座標系だけを動かせば済みます。

■ 座標系の種類

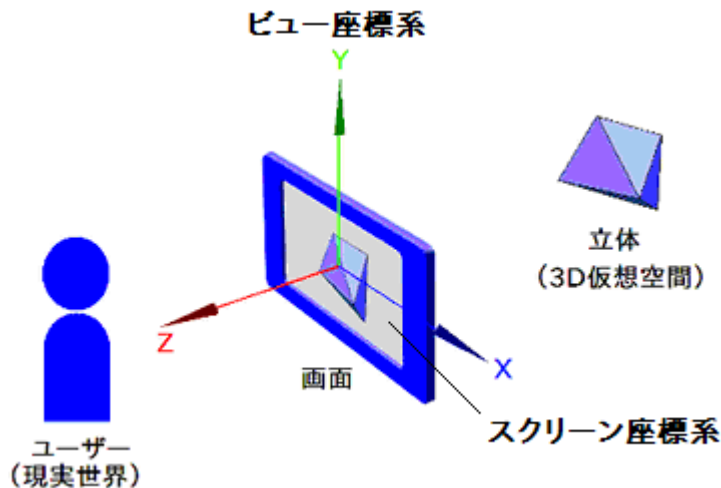
座標系には、大別して以下の 3 つの種類があります。

・ビュー座標系 / スクリーン座標系

ビュー座標系は、パソコンの画面に張り付いた形で存在する、特別な座標系です。一般には、画面右方向に X 軸、画面上方向に Y 軸、画面手前方向に Z 軸が取られます（画面奥に Z 軸を取るものも存在します）。3D 仮想空間を動かしても、この向きは常に変わりません。

つまり、ビュー座標系の X-Y 平面はパソコンの画面に一致しており、ゆえに現実世界と仮想世界を繋ぐ座標系と言えるでしょう。また、3D 仮想空間の視点（ビュー、カメラ）に張り付いた座標系と言う事もできます。ビュー座標系は、カメラ座標系と呼ばれる事もあります。

パソコン画面に描画される 3D コンピュータグラフィックス映像は、3D 仮想空間世界の全ての立体を、ビュー座標系の X-Y 平面に、（遠近感を付けた上で）投射したものとと言えます。この、投射されて 2 次元になった X-Y 平面の事を、スクリーン座標系と呼ぶ事があります。



・ワールド座標系

ビュー座標系に加えて、もう一つ特別な座標系が存在します。それがワールド座標系です。ワールド座標系は、ビュー座標系上に、1 個だけ配置されています（逆に、ワールド座標系上にビュー座標系が配置されていると見る場合もあります）。

そして、立体モデルは通常、ビュー座標系ではなくワールド座標系上に配置されます。このようにワールド座標系を介する事で、視点を変更する際の処理がワールド座標系を動かすだけで済み、非常に簡単になります。

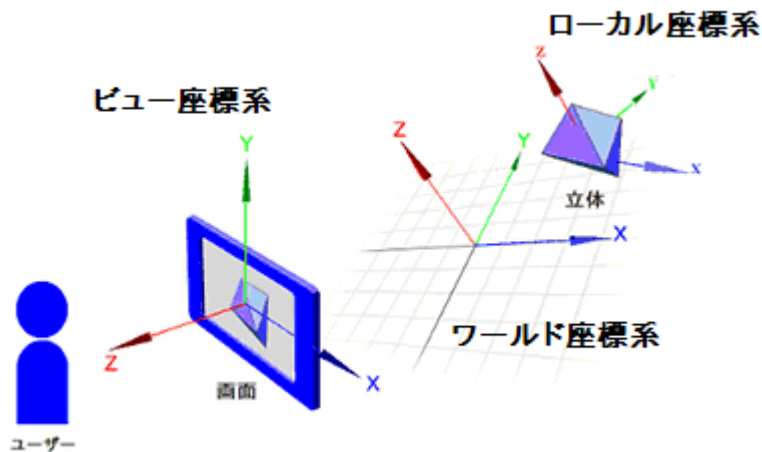
加えて、3D 仮想空間上において視点に左右されない絶対的な座標が定義できるという利点も

あります。例えば3Dシューティングゲームなど、広大な舞台において複雑に運動する物体を多数制御しなければならないような場合など、地面に固定された絶対的な座標系を基準として処理を記述するのが好ましい事がよくあります。このような場合に、ワールド座標系は良い基準として機能します。

・ローカル座標系

さて、ビュー座標系でもワールド座標系でも無い、プログラマが自由な用途で使用する座標系は、一般にローカル座標系と呼ばれます。ローカル座標系は必須では無いので、プログラマが必要に応じて必要な数だけ宣言し、用意します。

例えば「動く街」を表現したいなら、まず野原や山などの動かない立体はワールド座標系上に配置し、電車やバス・飛行機などの動く立体にはローカル座標系をそれぞれに用意して、その上に配置します。



座標系の生成と配置

ここでは、座標系の生成と配置について扱います。

■ 座標系の生成

・ 一般的な座標系 (ローカル座標系)

一般的な座標系 (ローカル座標系) を生成するには、`newCoordinate` 関数を使用します。

```
int newCoordinate ( )
```

この関数は座標系を生成し、その座標系に固有の ID を割り振って返します。

・ 特別な座標系 (ワールド座標系、ビュー座標系)

ワールド座標系やビュー座標系などの特別な座標系は、レンダラー側で自動的に生成され、確保されています。これらの座標系にもやはり ID が割り振られており、`getWorldCoordinate` 関数や `getViewCoordinate` 関数でその ID を取得する事ができます。

```
int getWorldCoordinate ( int rendererID )  
int getViewCoordinate ( int rendererID )
```

引数 `rendererID` には、レンダラーの ID を指定します。関数が2つありますが、上の関数はワールド座標系の ID を、下の関数はビュー座標系の ID を返します。

■ 座標系の配置

座標系は、他の座標系の上に配置して使用します。この配置先となる座標系を親座標系と呼び

ます。

座標系は何階層でも多重配置が可能ですが、座標系の親をたどっていくと必ずビュー座標系またはワールド座標系まで繋がっていなければなりません。

座標系を他の座標系の上に配置するには、`mountCoordinate` 関数を使用します。特にワールド座標系の上に配置する場合には、以下のように引数を指定します。

```
int mountCoordinate ( int childID, int rendererID )
```

引数 `childID` には、ワールド座標系上に配置したい座標系(子座標系)の ID を指定します。続く引数 `rendererID` には、レンダラーの ID を指定します。

座標系を、ワールド以外の座標系に配置するには、もう一つ引数を追加し、配置先座標系(親座標系)の ID を指定します。

```
int mountCoordinate ( int childID, int rendererID, int parentID )
```

引数 `childID` に配置したい座標系(子座標系)の ID を、`parentID` に配置先の座標系(親座標系)の ID を指定します。つまり `childID` の座標系を、`parentID` の座標系の上に配置します。真ん中の引数 `rendererID` には、レンダラーの ID を指定します。

なお、VCSSL3.0 以前の世代では、配置には `addCoordinate` 関数を使用していました。しかし、`add~` という関数名としては引数の順序が混乱を招くという理由により、VCSSL3.1 以降では、関数名を上記の `mountCoordinate` に変えたものが追加されました。つまり `addCoordinate` 関数と `mountCoordinate` 関数は、名称が異なるだけで全く同一のものです。

■ プログラム例

実際にローカル座標系を生成し、ワールド座標系の上に配置してみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

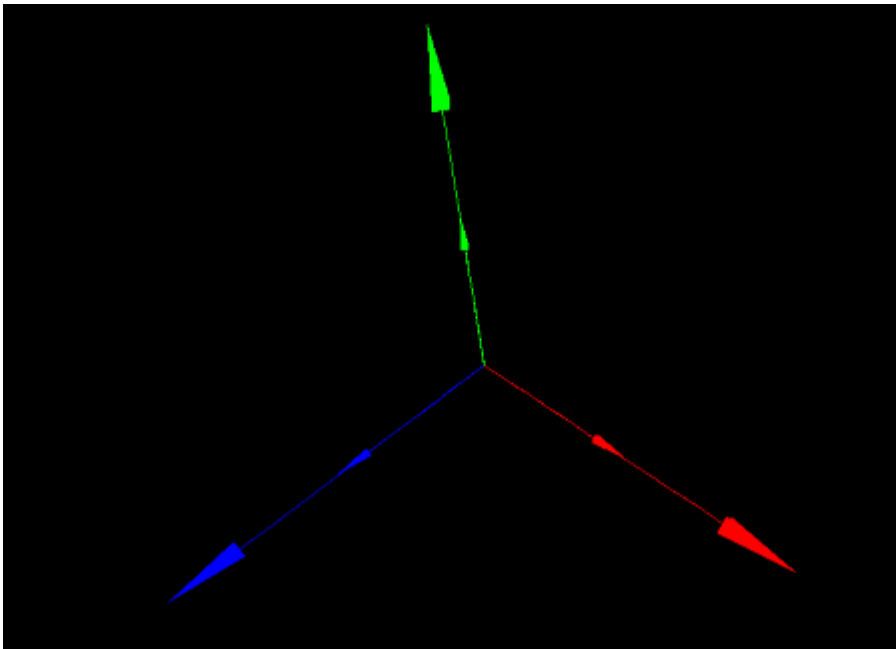
    // ローカル座標系を生成
    int coord = newCoordinate() ;

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID ) ;

    // ローカル座標系上に座標軸モデルを配置
    int axis1 = newAxisModel( 1.5, 1.5, 1.5 ) ;
    mountModel( axis1, rendererID, coord ) ;

    // ワールド座標系上に座標軸モデルを配置
    int axis2 = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis2, rendererID ) ;

}
```

このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものと小さいものが重なっており、大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

この状態で、ローカル座標系を移動させたり回転させたりすると、その上に配置された、小さい方の座標軸モデルの位置や向きも一緒に変化します。座標系の移動や回転については、これから扱います。

座標系の移動

ここでは、座標系の移動について扱います。ここで扱う移動は、親座標系の座標軸を基準としたものです。自身の座標軸を基準とした移動は、次の「座標系の歩行」で扱います。

■ 座標系の移動

座標系の移動を行うには、moveCoordinate 関数を使用します。

```
int moveCoordinate (  
    int coordinateID,  
    float dx, float dy, float dz  
)
```

最初の引数 coordinateID では、設定対象の座標系の ID を指定します。続く引数 dx、dy、dz では、それぞれ X、Y、Z 方向の平行移動距離を指定します。

■ プログラム例

実際にローカル座標系をワールド座標系の上に配置し、移動させてみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。

なお、次で扱う歩行との違いを分かりやすくするため、ローカル座標系を Z 軸方向に微妙に回転させてから移動させます。

以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;  
import Graphics3D ;
```

```
// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // ローカル座標系を生成
    int coord = newCoordinate() ;

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID ) ;

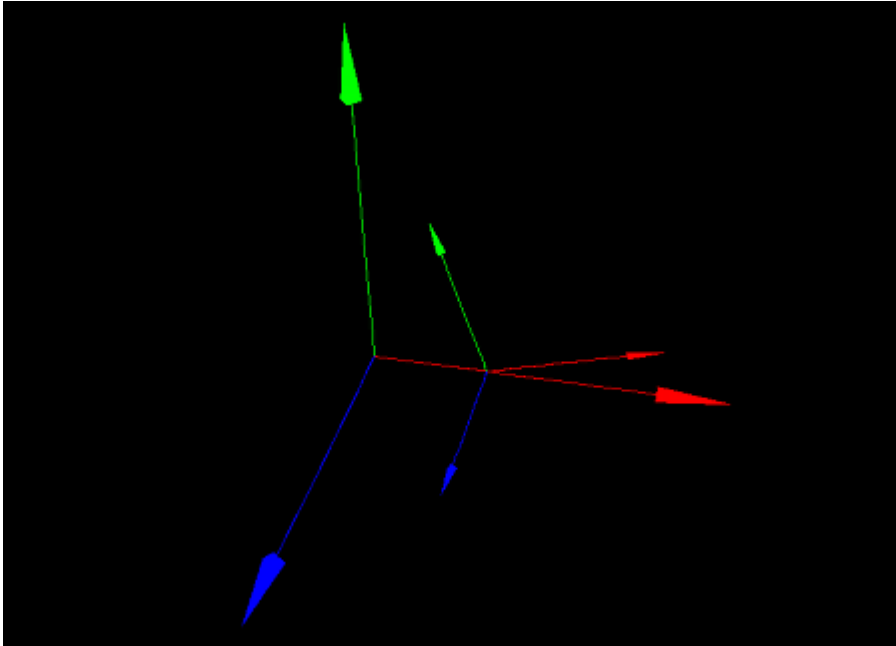
    // ローカル座標系を Z 軸まわりに回転
    rotCoordinateZ( coord, 0.3 ) ;

    // ローカル座標系を移動
    moveCoordinate( coord, 1.0, 0.0, 0.0 ) ;

    // ローカル座標系上に座標軸モデルを配置
    int axis1 = newAxisModel( 1.5, 1.5, 1.5 ) ;
    mountModel( axis1, rendererID, coord ) ;

    // ワールド座標系上に座標軸モデルを配置
    int axis2 = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis2, rendererID ) ;

}
```



このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

ローカル座標系は、親座標系であるローカル座標系の座標軸から見て、(1.0, 0.0, 0.0)だけ移動した位置に表示されます。

座標系の歩行

ここでは、座標系の歩行について扱います。ここで言う「歩行」とは移動の一種ですが、親座標系の座標軸を基準とした通常の移動ではなく、自身の座標軸を基準とした移動の事を意味します。

■ 座標系の歩行

座標系を歩行させるには、walkCoordinate 関数を使用します。

```
int walkCoordinate (
    int coordinateID,
    float dx, float dy, float dz
)
```

最初の引数 coordinateID では、設定対象の座標系の ID を指定します。続く引数 dx、dy、dz では、それぞれ X、Y、Z 方向の平行移動距離を指定します。

■ プログラム例

実際にローカル座標系をワールド座標系の上に配置し、歩行させてみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。また、歩行と移動の違いを分かりやすくするため、ローカル座標系をZ軸方向に微妙に回転させてから歩行させます。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
```

```
// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600);
    setBackgroundColor(0, 0, 0, 255);

    // ローカル座標系を生成
    int coord = newCoordinate();

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID );

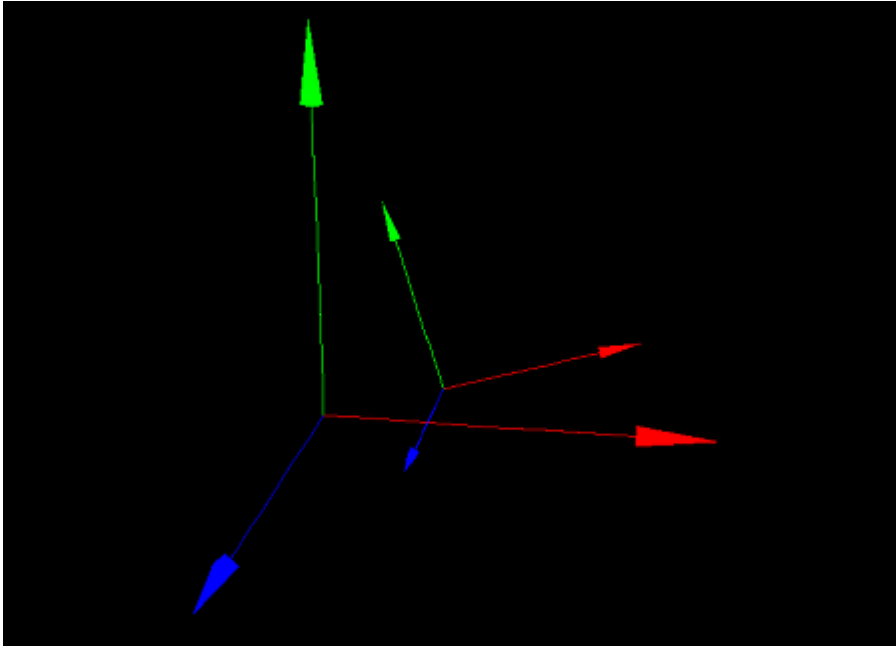
    // ローカル座標系を Z 軸まわりに回転
    rotCoordinateZ( coord, 0.3 );

    // ローカル座標系を歩行させる
    walkCoordinate( coord, 1.0, 0.0, 0.0 );

    // ローカル座標系上に座標軸モデルを配置
    int axis1 = newAxisModel( 1.5, 1.5, 1.5 );
    mountModel( axis1, rendererID, coord );

    // ワールド座標系上に座標軸モデルを配置
    int axis2 = newAxisModel( 3.0, 3.0, 3.0 );
    mountModel( axis2, rendererID );

}
```



このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

ローカル座標系は、自身の座標軸から見て、(1.0, 0.0, 0.0)だけ移動した位置に表示されます。

座標系の原点位置制御

これまで扱ってきた座標系の移動や歩行は、現在の座標系の位置を基準に、追加的に移動する方法でした。しかし場合によっては、座標系の原点位置を直接的に制御したい場合もあります。ここでは、そのような座標系の位置制御について扱います。

■ 座標系の原点位置制御

・原点位置の指定

座標系の原点位置を直接的に指定するには、setCoordinateLocation 関数を使用します。

```
int setCoordinateLocation (  
    int coordinateID,  
    float x, float y, float z  
)
```

最初の引数 coordinateID では、設定対象の座標系の ID を指定します。続く引数 x、y、z では、それぞれ原点位置の X、Y、Z 方向の成分を指定します。

なお、setCoordinateLocation 関数は比較的新しい関数であり、VCSSL3.1 以前では setCoordinateOrigin 関数が使用されていました。機能的には同じもので、現在はどちらでも使えますが、特に理由が無ければ、新しいプログラムでは setCoordinateLocation 関数の使用が推奨されます。

・原点位置の取得

座標系の原点位置を取得するには、getCoordinateLocation 関数を使用します。

```
int[ ] getCoordinateLocation ( int coordinateID )
```


戻り値には、[0]、[1]、[2]にそれぞれ原点位置の X、Y、Z 成分が返されます。引数 coordinateID には、取得対象の座標系の ID を指定します。

なお、この `getCoordinateLocation` 関数も比較的新しい関数であり、VCSSL3.1 以前では `getCoordinateOrigin` 関数が使用されていました。こちらも現在はどちらも使えますが、特に理由が無ければ、新しいプログラムでは `getCoordinateLocation` 関数の使用が推奨されます。

■ プログラム例

実際にローカル座標系をワールド座標系の上に配置し、原点位置の指定によってアニメーション的に移動させてみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // sin 関数を使用するため

// 座標系の ID を控えておく変数
int coord ;

// 時刻カウンタ(画面更新周期単位)
int t = 0 ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定(省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // ローカル座標系を生成
    coord = newCoordinate() ;

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID ) ;
```

```

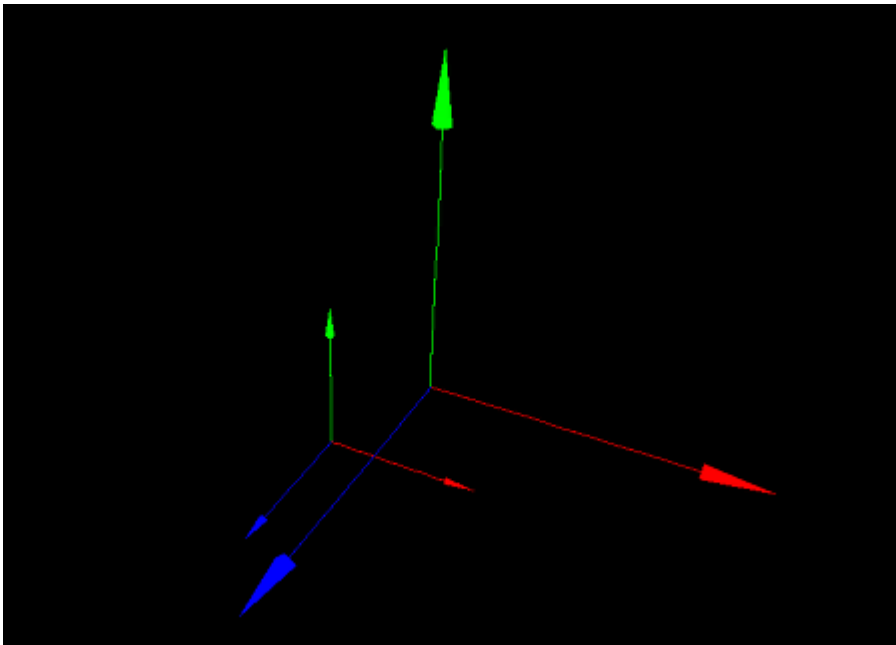
// ローカル座標系上に座標軸モデルを配置
int axis1 = newAxisModel( 1.5, 1.5, 1.5 );
mountModel( axis1, rendererID, coord );

// ワールド座標系上に座標軸モデルを配置
int axis2 = newAxisModel( 3.0, 3.0, 3.0 );
mountModel( axis2, rendererID );
}

// 画面更新周期ごとに、毎秒数十回呼び出される関数
void onUpdate ( int rendererID ) {

    // 原点位置指定でリサージュ曲線移動し、時刻カウンタを加算
    setCoordinateLocation( coord, sin(0.1*t), sin(0.5*t), 0.0 );
    t++;
}

```



このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

ローカル座標系は、リサージュ曲線を描きながら複雑に運動します。

座標系の回転

ここでは、座標系の回転を扱います。ここで扱う回転は、親座標系の座標軸を基準としたものです。自身の座標軸を基準とした回転については、次の「座標系の自転」で扱います。

■ 座標系の回転

・座標軸まわりの回転

親座標系の座標軸まわりの回転を行うには、rotCoordinateX、rotCoordinateY、rotCoordinateZ 関数を使用します。

```
void rotCoordinateX ( int coordID, float angle )  
void rotCoordinateY ( int coordID, float angle )  
void rotCoordinateZ ( int coordID, float angle )
```

3つの関数がありますが、それぞれ X、Y、Z 方向の回転を扱います。

最初の引数 coordID では、設定対象の座標系の ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアンを使用します。

・任意方向ベクトルまわりの回転

回転軸を、親座標系から見た任意の方向ベクトルとするには、rotCoordinate 関数を使用します。

```
void rotCoordinate (  
    int coordID,  
    float angle,  
    float vx, float vy, float vz  
)
```

最初の引数 coordID では、設定対象の座標系の ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。残りの引数 vx、vy、vz では、回転軸のベクトル成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアンを使用します。

・任意の原点と方向を持つベクトルまわりの回転

回転軸を、親座標系から見た任意の原点・方向を持つベクトルとするには、roCoordinate 関数の引数を増やして使用します。

```
void rotCoordinate (
    int coordID,
    float angle,
    float vx, float vy, float vz,
    float px, float py, float pz
)
```

最初の引数 coordID では、設定対象の座標系の ID を指定します。次の引数 angle では、軸まわりの回転角度を指定します。続く引数 vx、vy、vz では、回転軸の方向成分を指定します。残りの引数 px、py、pz では、回転軸の原点成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアンを使用します。

■ プログラム例

実際にローカル座標系をワールド座標系の上に配置し、45 度 = $\pi/4$ ラジアンだけ回転させてみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。また、次で扱う自転との違いを分かりやすくするため、最初にローカル座標系を X 方向に少し移動させてから回転させます。以下のように記述し、実行してみてください。

```

import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // 円周率 (PI) の値を使うため

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // ローカル座標系を生成
    int coord = newCoordinate() ;

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID ) ;

    // ローカル座標系を X 軸方向へ移動させる
    moveCoordinate( coord, 1.0, 0.0, 0.0 ) ;

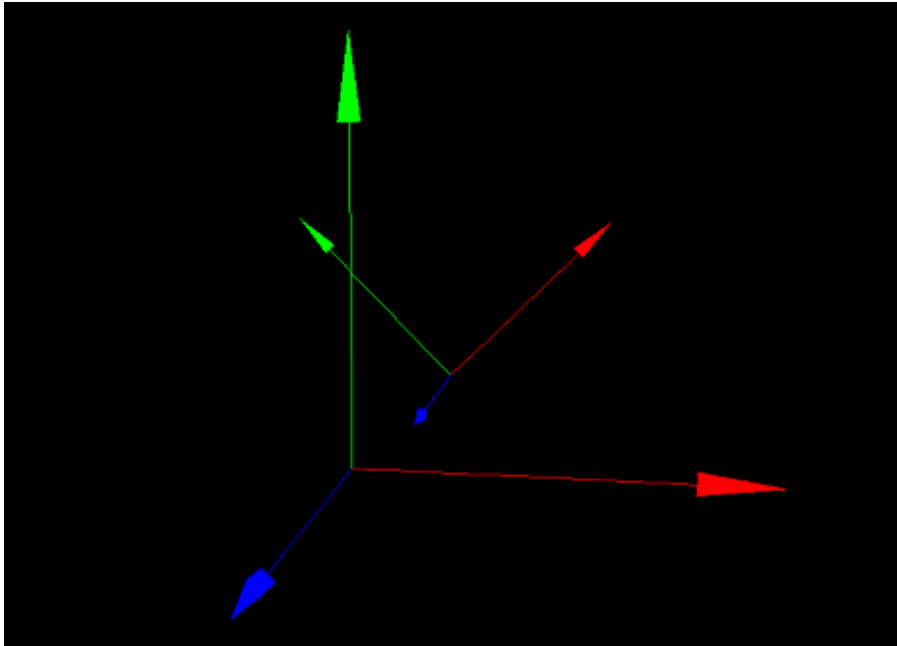
    // ローカル座標系を Z 軸まわりに 45 度回転
    rotCoordinateZ( coord, PI/4.0) ;

    // ローカル座標系上に座標軸モデルを配置
    int axis1 = newAxisModel( 1.5, 1.5, 1.5 ) ;
    mountModel( axis1, rendererID, coord ) ;

    // ワールド座標系上に座標軸モデルを配置
    int axis2 = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis2, rendererID ) ;

}

```



このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

ローカル座標系は、親座標系の X 軸の方向へ 1.0 だけ移動してから、親座標系の Z 軸まわりに 45 度だけ回転した状態となっています。

座標系の自転

ここでは、座標系の自転を扱います。ここで扱う自転とは、自身の座標軸を基準とした回転の事を意味します。

■ 座標系の自転

・座標軸まわりの自転

自身の座標軸まわりの自転を行うには、spinCoordinateX、spinCoordinateY、spinCoordinateZ、関数を使用します。

```
void spinCoordinateX ( int coordID, float angle )  
void spinCoordinateY ( int coordID, float angle )  
void spinCoordinateZ ( int coordID, float angle )
```

3つの関数がありますが、それぞれ X、Y、Z 方向の回転を扱います。

最初の引数 coordID では、設定対象の座標系の ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアンを使用します。

・任意方向ベクトルまわりの自転

自転軸を、自身から見た任意の方向ベクトルとするには、spinCoordinate 関数を使用します。

```
void spinCoordinate (  
    int coordID,  
    float angle,  
    float vx, float vy, float vz  
)
```

最初の引数 coordID では、設定対象の座標系の ID を指定します。続く引数 angle では、軸まわりの回転角度を指定します。残りの引数 vx、vy、vz では、回転軸のベクトル成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアンを使用します。

・任意の原点と方向を持つベクトルまわりの回転

自転軸を、自身から見た任意の原点・方向を持つベクトルとするには、roCoordinate 関数の引数を増やして使用します。

```
void spinCoordinate (  
    int coordID,  
    float angle,  
    float vx, float vy, float vz,  
    float px, float py, float pz  
)
```

最初の引数 coordID では、設定対象の座標系の ID を指定します。次の引数 angle では、軸まわりの回転角度を指定します。続く引数 vx、vy、vz では、回転軸の方向成分を指定します。残りの引数 px、py、pz では、回転軸の原点成分を指定します。

回転角度は、座標軸の向きに右ネジを進める向きを正とします。また、角度の単位にはラジアンを使用します。

■ プログラム例

実際にローカル座標系をワールド座標系の上に配置し、45 度 = $\pi/4$ ラジアンだけ自転させてみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。また、自転と回転との違いを分かりやすくするため、最初にローカル座標系を X 方向に少し移動させてから自転させます。以下のように記述し、実行してみてください。


```

import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // 円周率 (PI) の値を使うため

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // ローカル座標系を生成
    int coord = newCoordinate() ;

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID ) ;

    // ローカル座標系を X 軸方向へ移動させる
    moveCoordinate( coord, 1.0, 0.0, 0.0 ) ;

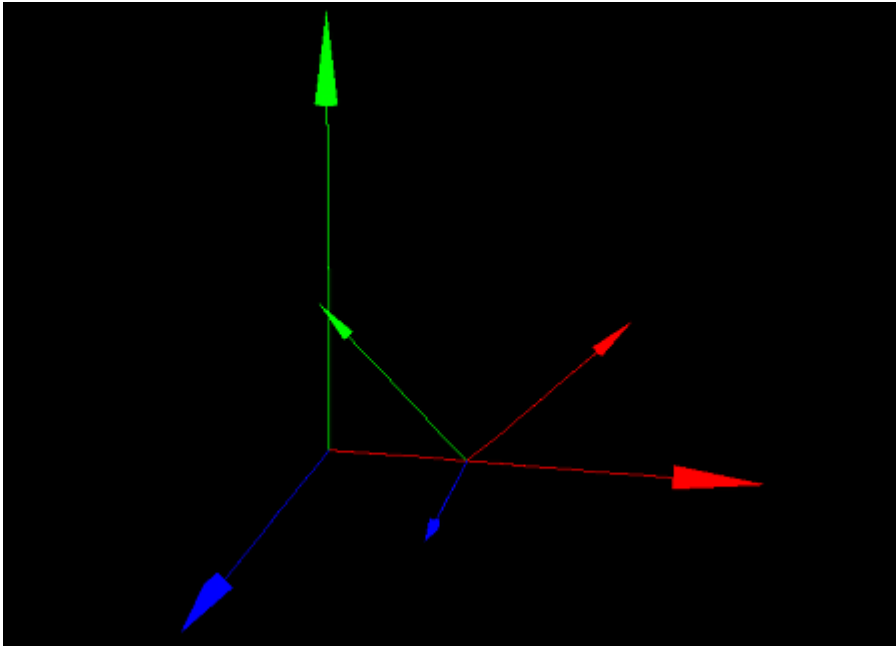
    // ローカル座標系を Z 軸まわりに 45 度自転
    spinCoordinateZ( coord, PI/4.0 ) ;

    // ローカル座標系上に座標軸モデルを配置
    int axis1 = newAxisModel( 1.5, 1.5, 1.5 ) ;
    mountModel( axis1, rendererID, coord ) ;

    // ワールド座標系上に座標軸モデルを配置
    int axis2 = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis2, rendererID ) ;

}

```



このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

ローカル座標系は、親座標系の X 軸の方向へ 1.0 だけ移動してから、親座標系の Z 軸まわりに 45 度だけ自転した状態となっています。

座標系のオイラー角姿勢制御

これまで扱ってきた座標系の回転や自転は、現在の座標系の姿勢を基準に、追加的に回転する方法でした。しかし場合によっては、座標系の姿勢を直接的に制御したい場合もあります。ここでは、そのような座標系の姿勢制御について扱います。

■ Z-X-Z 系オイラー角

・Z-X-Z 系オイラー角

3次元座標系の姿勢を表現するには、オイラー角の概念が必要です。オイラー角は、座標系の姿勢を3つの特別な角度で表現する方法です。オイラー角の扱いや解釈に関しては、分野や書籍によって様々なものがありますが、ここでは工学分野などで一般的な Z-X-Z 系オイラー角について解説します。

オイラー角で用いる3つの角度を、それぞれ α 、 β 、 γ とします。このとき、Z-X-Z 系オイラー角では、座標系の姿勢は次のようにして決定されます。最初に、座標系を Z 軸のまわりに α だけ自転させます。続いて、X 軸のまわりに β だけ自転させます。最後に、また Z 軸のまわりに γ だけ自転させます。これで座標系の姿勢が決まります。

・オイラー角の弱点

α 、 β 、 γ の値を決定すると、座標系の姿勢はただ一通りに定まります。しかし、必ずしもその逆は言えない事に注意が必要です。つまりある座標系の姿勢に対して、対応する α 、 β 、 γ の組が無数に存在する場合があります。つまりオイラー角にとっては苦手な姿勢というものが存在し、それがしばしば思わぬ不具合の原因となってしまう場合があります。例えば非常に有名な問題として、ジンバルロックというもの知られています。

こうした弱点を回避するためには、オイラー角のみを使用するのではなく、適材適所で `rotCoordinate` 関数や `spinCoordinate` 関数など、任意軸まわりの回転を使用する事が有効です。

■ 座標系の姿勢制御

・オイラー角による姿勢の指定

座標系の姿勢をオイラー角で指定するには、setCoordinateAngle 関数を使用します。

```
void setCoordinateAngle (
    int coordinateID,
    float alpha, float beta, float gamma
)
```

最初の引数 coordinateID では、設定対象の座標系の ID を指定します。続く引数 alpha で Z-X-Z 系オイラー角の第一角 α を、同様に beta で第二角 β を、gamma で第三角 γ を指定します。

・オイラー角の取得

座標系の姿勢からオイラー角を取得するには、getCoordinateAngle 関数を使用します。

```
float[ ] getCoordinateAngle ( int coordinateID )
```

この関数は、座標系の Z-X-Z 系オイラー角を配列で返します。配列には [0] に第一角 α 、[1] に第二角 β 、[2] に第三角 γ が格納されています。引数 coordinateID には、取得対象の座標系の ID を指定します。

なお、上でも述べた通り、同じ姿勢を現すオイラー角の組は複数存在します。この関数は、あくまでその組の内のどれかを返すという事に注意する必要があります。

■ プログラム例

実際にローカル座標系をワールド座標系の上に配置し、こまのようにアニメーション的に回転させてみましょう。区別しやすくするために、ローカル座標系には小さめの座標軸モデルを、ワールド座標系には大きめの座標軸モデルを配置します。

```

import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // sin 関数を使用するため

// 座標系の ID を控えておく変数
int coord ;

// 時刻カウンタ(画面更新周期単位)
int t = 0 ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定(省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // ローカル座標系を生成
    coord = newCoordinate( ) ;

    // ローカル座標系をワールド座標系に配置
    mountCoordinate( coord, rendererID ) ;

    // ローカル座標系上に座標軸モデルを配置
    int axis1 = newAxisModel( 1.5, 1.5, 1.5 ) ;
    mountModel( axis1, rendererID, coord ) ;

    // ワールド座標系上に座標軸モデルを配置
    int axis2 = newAxisModel( 3.0, 3.0, 3.0 ) ;
    mountModel( axis2, rendererID ) ;

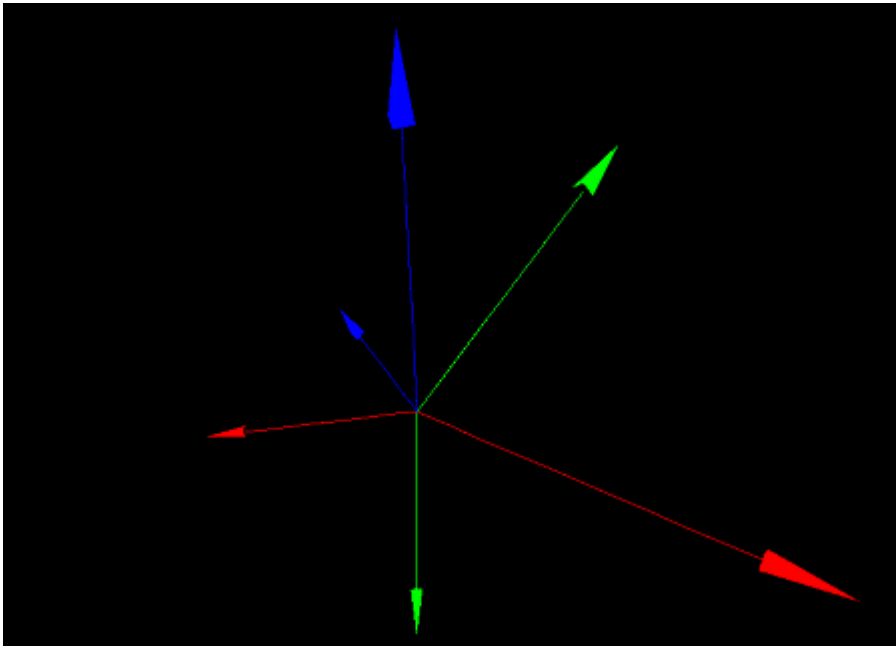
}

```

```
// 画面更新周期ごとに毎秒数十回呼び出される関数
void onUpdate (int renderer) {

    // オイラー角による姿勢制御
    setCoordinateAngle(
        coord,
        0.08*t, 0.5*sin(0.03*t), 0.3*t
    );

    // 時刻カウンタを加算
    t++;
}
}
```



このプログラムを実行すると、黒い画面に座標軸モデルが表示されます。座標軸モデルは大きいものがワールド座標系の上に、小さいものがローカル座標系の上に配置されています。

ローカル座標系は、止まりかけのこまのような、複雑な動きをします。Z 軸をゆっくりと公転させているのが第一角 α の効果、Z 軸をゆらゆらと振動させているのが第二角 β の効果、座標系を高速にスピンさせているのが第三角 γ の効果です。

カメラワーク

ここでは、実践的なカメラワークを実現する方法について扱います。

■ カメラワーク

一般的なカメラワークには、以下の2つが挙げられます。

・地球儀方式(※)のカメラワーク

最も単純なカメラワークは、ビュー座標系上のワールド座標系を、マウス操作に伴ってくるくると自転させる方式のものです。例えるなら地球儀のような方式なので、ここでは地球儀方式(※これは別に一般的な呼称ではありません)と呼びましょう。この方式は、ある程度小さい立体を、色々な角度から眺めるような場合に向いています。これまでに使用してきた、マウス操作で視点を操作するための `setGraphics3DEventSource` 関数でも、この方式のカメラワークを採用しています。

地球儀方式のカメラワークは非常に簡単に実装できるため(ワールド座標系を、普通に移動させたり回転させたりするだけ)、ここではあまり深く扱いません。

・カメラ移動方式のカメラワーク

もう一つ、地球儀方式と共によく用いられるのが、ワールド座標系上でカメラを動かすようなカメラワークです。例えば 3D のアクションゲームなどでは、主人公の背後から見下ろすような視点で、主人公が移動するとカメラも後ろから追従していきます。このようなカメラ移動方式のカメラワークは、広大な立体を移動しながら眺めるような場合に向いています。

ここでは、このカメラ移動方式のカメラワークを中心に扱います。

■ ビュー変換

ワールド座標系からビュー座標系への座標変換は、特別にビュー変換と呼ばれます。カメラワークを制御する事は、つまりはこのビュー変換を制御する事です。

・座標変換の処理順序

一般に、立体の座標変換順序は

ローカル座標系

→ (ローカル変換) → ローカル座標系

→ (ワールド変換) → ワールド座標系

→ (ビュー変換) → ビュー座標系

という流れで処理されます。

・地球儀方式のビュー変換

地球儀方式のカメラワークの場合は、**座標系の配置階層の順序は、そのまま座標変換処理の順序に一致します**。即ち、ローカル座標系がワールド座標系上に配置されており、ワールド座標系がビュー座標系上に配置されている(と見なせる)状態です。

そのため、ビュー変換は、通常のローカル変換やワールド変換と全く変わりません。従って、カメラワークの制御も通常と変わらず、ワールド座標系を `moveCoordinate` 関数で移動させたり、`rotCoordinate` 関数で回転させたりする事で制御します。

・カメラ移動方式のビュー変換

カメラ移動方式のカメラワークでは、ワールド座標系の上にビュー座標系を配置し、ビュー座標系を動かすような制御を行う必要があります。従ってこの場合、**ビュー変換において、座標系の配置階層の順序と、座標変換処理の順序が逆転します**。そのため、ビュー変換は、ローカル変換やワールド変換とは少し異なる変換を行う必要が生じます。

しかし VCSSL Graphics3D では、このようなカメラ移動方式のビュー変換を制御するための関数が標準で用意されています。プログラマは、通常の `moveCoordinate` 関数や `rotCoordinate` 関数の代わりに、`moveView` 関数や `rotView` 関数を使うだけで、通常の座標系操作と全く同じ感覚で、視点の操作を行う事ができます。

■ 視点制御関数の引数

これから解説する、視点を制御するための関数は、1 つめの引数が座標系の ID ではなく、レンダラーの ID となっています。この点は混同しないように注意が必要です。

■ ビュー座標系の移動

ワールド座標系の座標軸を基準とした、ビュー座標系の移動には、moveView 関数を使用します。

```
void moveView ( int rendererID, float dx, float dy, float dz )
```

最初の引数 rendererID でレンダラーの ID を、続く引数 dx、dy、dz でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。

■ ビュー座標系の歩行

ビュー座標系の座標軸を基準とした、ビュー座標系の動き、つまりビュー座標系の歩行には、walkView 関数を使用します。

```
void walkView ( int rendererID, float dx, float dy, float dz )
```

最初の引数 rendererID でレンダラーの ID を、続く引数 dx、dy、dz でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。

■ ビュー座標系の回転

ワールド座標系の座標軸を基準とした、ビュー座標系の回転には、rotViewX、rotViewY、rotViewZ 関数や、rotView 関数を使用します。

```
void rotViewX ( int rendererID, float angle )  
void rotViewY ( int rendererID, float angle )  
void rotViewZ ( int rendererID, float angle )
```

これら 3 つの関数は、それぞれ X、Y、Z 軸まわりの回転を行います。最初の引数 `rendererID` でレンダラーの ID を、続く引数 `dx`、`dy`、`dz` でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。

任意方向の回転軸まわりで回転させるには、`rotView` 関数を使用します。

```
void rotView (  
    int rendererID, float angle,  
    float vx, float vy, float vz  
)
```

最初の引数 `rendererID` でレンダラーの ID を、続く引数 `dx`、`dy`、`dz` でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。残りの引数 (`vx`, `vy`, `vz`) で、ワールド座標系から見た回転軸の方向ベクトルを指定します。

回転軸の方向と原点位置を指定するには、`rotView` 関数に引数を追加して使用します。

```
void rotView (  
    int rendererID, float angle,  
    float vx, float vy, float vz  
    float px, float py, float pz  
)
```

最初の引数 `rendererID` でレンダラーの ID を、続く引数 `dx`、`dy`、`dz` でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。残りの引数 (`vx`, `vy`, `vz`) と (`px`, `py`, `pz`) で、ワールド座標系から見た回転軸の方向ベクトルと原点位置ベクトルを指定します。

■ ビュー座標系の自転

ビュー座標系の座標軸を基準とした、ビュー座標系の回りこみ、つまりビュー座標系の自転には、`spinViewX`、`spinViewY`、`spinViewZ` 関数や、`spinView` 関数を使用します。

```
void spinViewX ( int rendererID, float angle )  
void spinViewY ( int rendererID, float angle )  
void spinViewZ ( int rendererID, float angle )
```

これら 3 つの関数は、それぞれ X、Y、Z 軸まわりの自転を行います。最初の引数 rendererID でレンダラーの ID を、続く引数 dx、dy、dz でそれぞれ X、Y、Z 方向の移動距離を指定します。

任意方向の回転軸まわりで自転させるには、spinView 関数を使用します。

```
void spinView (  
    int rendererID, float angle,  
    float vx, float vy, float vz  
)
```

最初の引数 rendererID でレンダラーの ID を、続く引数 dx、dy、dz でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。残りの引数 (vx, vy, vz) で、ワールド座標系から見た自転軸の方向ベクトルを指定します。

自転軸の方向と原点位置を指定するには、spinView 関数に引数を追加して使用します。

```
void spinView (  
    int rendererID, float angle,  
    float vx, float vy, float vz  
    float px, float py, float pz  
)
```

最初の引数 rendererID でレンダラーの ID を、続く引数 dx、dy、dz でそれぞれ X、Y、Z 方向のカメラの移動距離を指定します。残りの引数 (vx, vy, vz) と (px, py, pz) で、ワールド座標系から見た自転軸の方向ベクトルと原点位置ベクトルを指定します。

■ ビュー座標系の位置制御

ワールド座標系の座標軸を基準とした、ビュー座標系の位置制御（原点位置の指定）には、`setViewLocation` 関数を使用します。

```
void setViewLocation ( int rendererID, float x, float y, float z )
```

最初の引数 `rendererID` でレンダラーの ID を、続く引数 `x`、`y`、`z` でそれぞれビュー座標系原点の `X`、`Y`、`Z` 座標を指定します。

なお、VCSSL3.1 以前は `setViewOrigin` 関数が使用されており、現在はどちらでも使用できますが、特に理由が無ければ、新しいプログラムでは `setViewLocation` 関数の使用が推奨されます、

■ ビュー座標系の姿勢制御

ワールド座標系を基準とした、ビュー座標系の姿勢制御（`X-Z-X` オイラー角指定）には、`setViewAngle` 関数を使用します。

```
void setViewAngle (
    int rendererID,
    float alpha, float beta, float gamma
)
```

最初の引数 `rerendererID` でレンダラーの ID を指定します。続く引数 `alpha` で `Z-X-Z` 系オイラー角の第一角 α を、同様に `beta` で第二角 β を、`gamma` で第三角 γ を指定します。

■ プログラム例

実際にビュー座標系を操作し、3D 舞台の中をゲームのように歩いてみましょう。キー入力イベントハンドラを作成し、キーボードの上下左右キーによるビュー座標系の移動を実装します。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600) ;
    setBackgroundColor(0, 0, 0, 255) ;

    // 以下、カラフルなモデルで 3D 舞台を作成
    int box1 = newBoxModel( 1.0, 3.0, 1.0 );
    mountModel( box1, rendererID );
    setModelColor( box1, 255, 0, 0, 255 );

    int box2 = newBoxModel( 1.0, 3.0, 1.0 );
    mountModel( box2, rendererID );
    setModelColor( box2, 0, 255, 0, 255 );
    moveModel( box2, 5.0, 0.0, 5.0 );

    int box3 = newBoxModel( 1.0, 3.0, 1.0 );
    mountModel( box3, rendererID );
    setModelColor( box3, 0, 0, 255, 255 );
    moveModel( box3, -5.0, 0.0, 5.0 );

    int box4 = newBoxModel( 1.0, 3.0, 1.0 );
    mountModel( box4, rendererID );
    setModelColor( box4, 255, 255, 0, 255 );
    moveModel( box4, 5.0, 0.0, -5.0 );

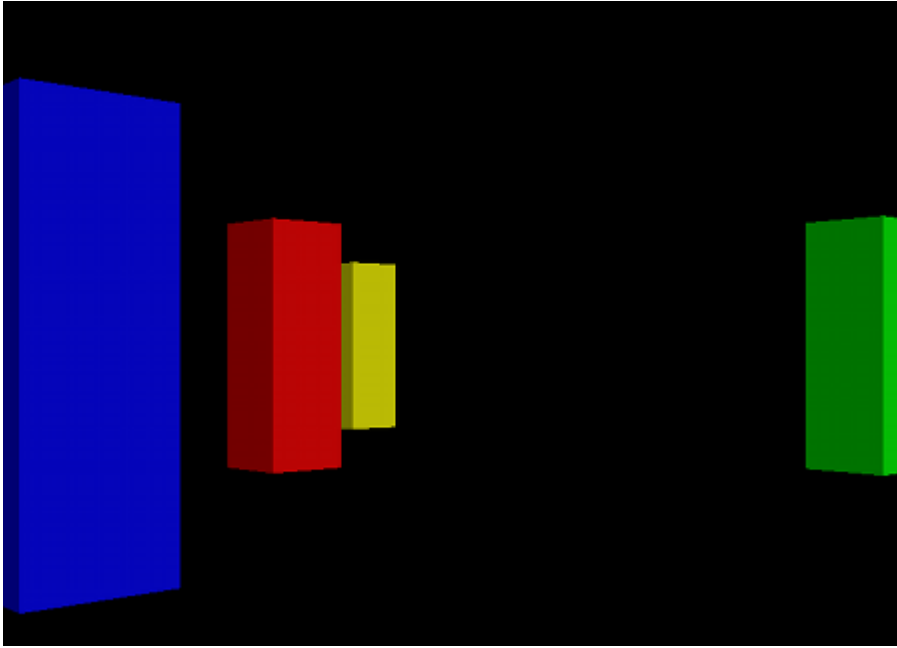
    int box5 = newBoxModel( 1.0, 3.0, 1.0 );
    mountModel( box5, rendererID );
    setModelColor( box5, 0, 255, 255, 255 );
    moveModel( box5, -5.0, 0.0, -5.0 );
}
```

```
// キーが入力された際に呼び出される ( GUI ライブラリのイベントハンドラ )
void onKeyDown( int id, string key ){

    // レンダラーの ID を取得
    int rendererID = getRenderer();

    // 上下キーでカメラの前進後退
    if( key == "UP" ){
        walkView( rendererID, 0.0, 0.0, -0.3 );
    }
    if( key == "DOWN" ){
        walkView ( rendererID, 0.0, 0.0, 0.3 );
    }

    // 左右キーでカメラの方向転換
    if( key == "RIGHT" ){
        spinViewY ( rendererID, -0.05 );
    }
    if( key == "LEFT" ){
        spinViewY ( rendererID, 0.05 );
    }
}
```



このプログラムを実行すると、黒い背景にカラフルな建物が表示されます。キーボードの上下左右キー入力により、建物の間を歩いて移動する事ができます。

第四章 ベクトル/ジオメトリ演算

この章では、3次元空間において高度な位置制御を行うためのベクトル演算や、立体同士の接触判定などを行うためのジオメトリ演算を扱います。この章の内容は、グラフィックスの描画には必要ありません。しかし、3DCGの舞台を自由に歩き回るようなものを開発する場合には、必要不可欠な内容となります。

ベクトルの生成と配置

ここでは、ベクトルの生成と配置について扱います。VCSSL では、ベクトルの座標変換や内積・外積などの基本演算などを簡単に扱えるようにするため、ベクトルを生成して座標系上に配置できるようになっています。

■ ベクトルとは

3 次元空間に関する作業では、 (X,Y,Z) のように 3 つの値を一組にして扱うのが便利です。例えば空間上の位置はまさしく (X,Y,Z) の組で決まりますし、空間上の方向も 3 つの方向成分 (V_x,V_y,V_z) で決まります。動く物体が単位時間に移動する変位も同様に (D_x,D_y,D_z) というように、3 つの方向成分で表せます。そこで、このような 3 つの方向成分を一組にまとめたものが、3 次元におけるベクトル (3 次元ユークリッド空間ベクトル) です。

■ ベクトルの生成

ベクトルを生成するには、newVector 関数を使用します。

```
int newVector ( float x, float y, float z )
```

引数 x, y, z に、それぞれベクトルの X, Y, Z 成分を指定します。この関数は引数を成分にもつベクトルを生成し、そのベクトルに固有の ID を割り振って返します。

引数を以下のようにして、すでに存在しているベクトルのコピーを生成する事もできます。

```
int newVector ( int copyID )
```

引数にはコピーしたいベクトルの ID を指定します。

■ ベクトルの配置

生成したベクトルは、モデルやポリゴンと同様、座標系の上に配置して使用します。ベクトルを配置するには `mountVector` 関数を指定します。

```
int mountVector ( int vectorID, int rendererID )  
int mountVector ( int vectorID, int rendererID, int CoordinateID )
```

2 つの関数がありますが、上の関数ではワールド座標系上に、下の関数では任意の座標系上に配置する事ができます。引数 `vectorID` には配置するベクトルの ID を、続く引数 `rendererID` にはレンダラーの ID を、最後の `CoordinateID` に配置先座標系の ID (省略するとワールド座標系) を指定します。

ベクトルの操作と演算

ここでは、ベクトルの基本的な操作と演算について扱います。

■ ベクトルの成分設定

ベクトルの成分を設定するには、setVector 関数を使用します。

```
void setVector ( int vectorID, float x, float y, float z )
```

引数 vectorID に設定対象のベクトルの ID を、引数 x、y、z に、それぞれベクトルの X、Y、Z 成分を指定します。この関数で、すでに存在しているベクトルの成分を書き換える事ができます。

なお、次のように、ベクトルの X、Y、Z 成分を独立に設定する関数も使用できます。

```
void setVectorX ( int vectorID, float value )  
void setVectorY ( int vectorID, float value )  
void setVectorZ ( int vectorID, float value )
```

3 つの関数がありますが、それぞれベクトルの X、Y、Z 成分を設定します。

■ ベクトルの成分取得

ベクトルの成分を取得するには、getVectorX、getVectorY、getVectorZ 関数を使用します。

```
float getVectorX ( int vectorID )  
float getVectorY ( int vectorID )  
float getVectorZ ( int vectorID )
```

3つの関数がありますが、それぞれ X、Y、Z 成分を返します。引数 vectorID に取得対象のベクトルの ID を指定します。

■ 内積

ベクトルの内積を計算するには、getVectorInnerProduct 関数を使用します。

```
float getVectorInnerProduct ( int vectorID1, int vectorID2 )
```

引数 vectorID1 と vectorID2 に、内積を計算する 2 つのベクトルの ID を指定します。この関数は指定されたベクトルの内積を計算して返します。

■ 外積(クロス積)

ベクトルの外積を計算するには、getVectorCrossProduct 関数を使用します。

```
void getVectorCrossProduct ( int vectorID1, int vectorID2, int crossVector )
```

この関数は、引数 vectorID1 と vectorID2 に指定したベクトルの外積を計算し、その結果を引数 crossVector に指定されたベクトルに代入します。crossVector はあらかじめ生成しておく必要があります。

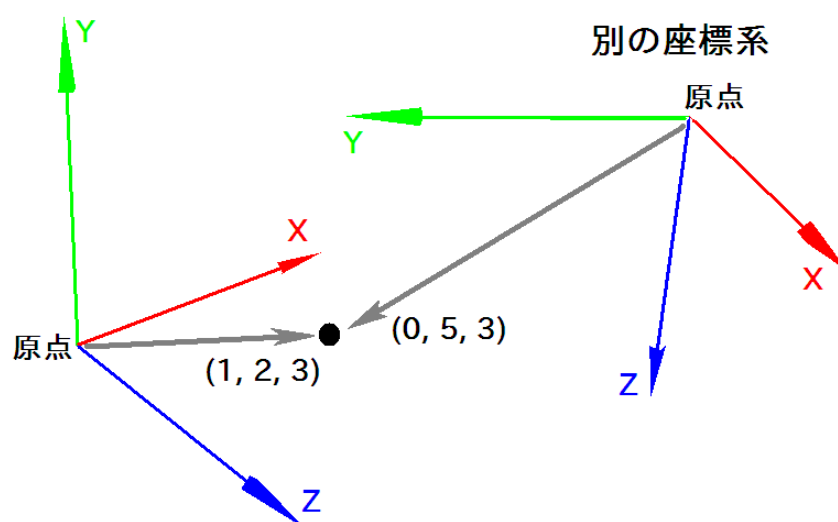
座標変換

ここでは、ベクトルやポリゴン、モデルの座標変換について扱います。

■ ベクトルの成分は座標系とワンセット

これまで扱ってきたベクトルは、主に立体の位置関係を把握するために使用します。例えば、ある位置を別の座標系から見るとどの位置になるかを求めたり、モデルとベクトルとの交点を求めたりするなどの用途が挙げられます。

そうした場合、生成したベクトルの成分値をそのまま使用しても意味がありません。なぜなら、異なる座標系では、一般に原点の位置や座標軸の方向も異なるためです。例えばある座標系から見た $(1, 2, 3)$ は、別の座標系から見た $(0, 5, 3)$ かもしれません。つまりベクトルの成分に格納された値は、そのベクトルが配置されている座標系でしか意味を持ちません。ベクトルの成分と座標系は常にワンセットで考えるべきなのです。



■ 座標変換とは

上述のような事情から、異なる座標系上に配置された2つのベクトル同士の位置関係を、そのまま成分によって把握することはできません。こうした場合には、それら2つのベクトルを共通の座標系へ座標変換してから、成分を使用します。

座標変換とは、「ある座標系上のベクトル(が指し示す空間的な位置)を、別の座標系で見るとどういったベクトルになるか」というのを求める変換です。

■ ベクトルの座標変換

ベクトルを座標変換するには、transformVector 関数を使用します。

```
void transformVector ( int vectorID, int bufferID, int coordinateID )
```

引数 vectorID に座標変換したいベクトルの ID を、続く引数 bufferID に座標変換後の成分を格納するベクトルの ID を、最後の coordinateID に座標変換する先の座標系の ID を指定します。

この関数は座標変換後のベクトルを新規生成するのではなく、引数 bufferID に指定されたベクトルに成分を代入します。この bufferID には、あらかじめ newVector(int copyID)関数などを用い、変換するベクトルのコピーを作っておいて、それを渡します。

■ ポリゴンの座標変換

ポリゴンも、頂点の位置を表すベクトルを内部に保持しています。従ってポリゴンの位置関係を計算するには、ポリゴンを構成するベクトルもまとめて座標変換する必要があります。

ポリゴンを座標変換するには、transformPolygon 関数を使用します。

```
void transformPolygon ( int polygonID, int bufferID, int coordinateID )
```

引数 polygonID に座標変換したいポリゴンの ID を、続く引数 bufferID に座標変換後のベクトル成分を格納するポリゴンの ID を、最後の coordinateID に座標変換する先の座標系の ID を指定します。

この関数は座標変換後のポリゴンを新規生成するのではなく、引数 bufferID に指定されたポリゴンに代入します。この bufferID には、あらかじめ newPolygon(int copyPolygonID)関数などを用い、変換するポリゴンのコピーを作っておいて、それを渡します。特にポリゴンの種類が異なると正常に演算できないため、必ずコピーを作って指定するようにしてください。

■ モデルの座標変換

モデルも多数のポリゴンで構成されているため、大量のベクトルを内部に保持しています。従ってモデルの位置関係を計算するには、モデルを構成するベクトルもまとめて座標変換する必要があります。

モデルを座標変換するには、transformModel 関数を使用します。

```
void transformModel ( int modelID, int bufferID, int coordinateID )
```

引数 modelID に座標変換したいモデルの ID を、続く引数 bufferID に座標変換後のベクトル成分を格納するモデルの ID を、最後の coordinateID に座標変換する先の座標系の ID を指定します。

この関数は座標変換後のモデルを新規生成するのではなく、引数 bufferID に指定されたモデルに代入します。この bufferID には、あらかじめ newModel(int copyModelID)関数などを用い、変換するモデルのコピーを作っておいて、それを渡します。特にモデルのポリゴン数や頂点数が異なると正常に演算できないため、必ずコピーを作って指定するようにしてください。

■ プログラム例

実際にローカル座標系上にベクトルを配置し、ワールド座標系に座標変換してみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // 円周率(PI)の値を使用するため

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {
```

```

// ローカル座標系を生成
int coord = newCoordinate( );

// ローカル座標系をワールド座標系に配置
mountCoordinate( coord, rendererID );

// ローカル座標系を Z 軸まわりに 45 度回転
rotCoordinateZ( coord, PI/4.0 );

// X=1.0 のベクトルを生成
int vector = newVector( 1.0, 0.0, 0.0 );

// ローカル座標系上にベクトルを配置
mountVector( vector, rendererID, coord );

// 座標変換結果の控え用に、vector のコピーを生成
int trans = newVector( vector );

// vector をワールド座標系へ座標変換し、結果を trans に代入
transformVector(
    vector, trans, getWorldCoordinate( rendererID )
);

// 座標変換結果を出力
println(
    getVectorX( trans ), getVectorY( trans ), getVectorZ( trans )
);
}

```

このプログラムでは、Z 軸まわりに 45 度だけ回転したローカル座標系上に (X, Y, Z) = (1.0, 0.0, 0.0) のベクトルを配置し、それをワールド座標系に座標変換した結果を、VCSSL コンソールに出力しています。このプログラムを実際に実行すると、VCSSL コンソールに

「 0.7071067811865476 0.7071067811865475 0.0 」

と表示されます。この座標変換された X と Y の値は $1/\sqrt{2}$ に等しく、正しく座標変換された事がわかります。

画面射影

ここでは、ベクトルの画面射影を扱います。画面射影により、3D 空間上の任意の位置に対応する、グラフィックスデータ上の X、Y 座標 (ウィンドウ座標) を得る事ができます。

■ 画面射影

・ベクトルの画面射影

3D 空間中の位置が、描画領域上のどの位置に対応しているかを知りたい場合には、画面射影を行います。3D 空間に配置したベクトルに対して画面射影を行う事で、描画領域上でそのベクトルに対応する位置の、X 座標と Y 座標 (いわゆるウィンドウ座標) を得る事ができます。

・得られる座標は左下が原点

ここで注意が必要なのが、描画領域上の原点位置です。

3DCG 以外の分野では、描画領域上の左上を原点とするのが一般的です。しかし、右手座標系を採用する一部の 3DCG の分野では、様々な事情により、描画領域の左下を原点とする事がよくあります (もちろん従来通り左上が原点の場合もあります)。VCSSL Graphics3D でも右手座標系を採用しているため、描画領域の左下が原点と見なされます。

結果として、VCSSL GUI や VCSSL Graphics2D では左上が原点であるのに対し、VCSSL Graphics3D でのみ左下が原点となります。従ってこれらを連携させる場合には、この違いに少し注意が必要となります。

■ ベクトルの画面射影

・X 座標を得る

ベクトルに対して画面射影を行い、描画領域上の X 座標を得るには、projectVectorX 関数を使用します。

```
int projectVectorX ( int vectorID, int rendererID )
```

最初の引数 vectorID には、画面射影を行うベクトルの ID を指定します。続く引数 rendererID に

は、レンダラーの ID を指定します。

・Y 座標を得る

ベクトルに対して画面射影を行い、描画領域上の Y 座標を得るには、projectVectorY 関数を使用します。

```
int projectVectorY ( int vectorID, int rendererID )
```

最初の引数 vectorID には、画面射影を行うベクトルの ID を指定します。続く引数 rendererID には、レンダラーの ID を指定します。

■ プログラム例

実際にワールド座標系の原点が、描画領域上のどこに相当するかを取得してみましょう。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 原点ベクトルを生成
    int vector = newVector( 0.0, 0.0, 0.0 ) ;

    // ワールド座標系上にベクトルを配置
    mountVector( vector, rendererID ) ;

    // 画面射影
    int x = projectVectorX( vector, rendererID );
    int y = projectVectorY( vector, rendererID );
```

```
// 結果を出力  
println( "X=" + x + ", " + "Y=" + y );  
  
}
```

このプログラムを実行すると、VCSSL コンソールに「 X=391, Y=277 」などと表示されます（値は環境やバージョン等に依存します）。これはちょうど描画領域の中心における座標になっています。なお、ここで使用しているフレームワーク Graphics3DFramework における、ウィンドウ上の描画領域の GUI コンポーネント ID は、getImageLabel() 関数で取得できます。

衝突判定

ここでは、立体の衝突判定について扱います。

■ 衝突判定とは

例えばゲームなど、3DCG の舞台を動き回るようなプログラムの開発において、非常に重要な役割を担うのが、衝突判定の処理です。

衝突判定とは、その名前の通り、立体同士の衝突を判定するための処理です。例えば 3DCG の舞台を動き回るようなプログラムでは、主人公と地面、または壁などとの衝突判定が必要不可欠となります。主人公が地面と衝突している際はそれ以上落下しないように、また壁と衝突した際は垂直に押し戻されるように、衝突判定を使用して適切な処理を行う必要があります。

■ 直線とポリゴンとの衝突判定

VCSSL Graphics3D において、衝突判定の基本となるのが、直線とポリゴンとの衝突判定です。これは、任意の原点と方向を持つ直線（原点から一方向にのみ無限に伸びる、いわゆる半直線）と、ポリゴンとが交点を持つかどうかという判定です。なお、直線とポリゴンとが交点を持つ場合には、その交点の位置と、交点におけるポリゴンの法線ベクトル（ポリゴンに垂直な方向を持ち、大きさが 1 のベクトル）の情報も得たい場合が多いでしょう。

直線とポリゴンとの交点位置や法線ベクトルは、高校数学の教科書に載っているような公式で独自に求める事もできますが、VCSSL Graphics3D ではあらかじめ用意されている `getPolygonIntersection` 関数で簡単に求める事が可能です。

```
bool getPolygonIntersection (  
    int polygonID,  
    int directionalVectorID, int pointVectorID,  
    int intersectionVectorID, int normalVectorID  
)
```

最初の引数 `polygonID` には判定対象のポリゴンの ID を、続く引数 `directionalVectorID` と

pointVectorID にそれぞれ直線の方角ベクトルと原点位置ベクトルの ID を指定します。残る引数 intersectionVectorID と normalVectorID には、交点の位置と法線ベクトル情報を格納するためのベクトルを用意して、その ID を指定します。なお、ポリゴンや直線のベクトルは、あらかじめ共通の座標系へ座標変換を行ったものを指定する必要があります。

この関数はポリゴンと直線との位置関係を解析し、交点が存在すれば true を、存在しなければ false を返します。交点が存在する場合は、引数に指定された intersectionVectorID のベクトルに交点位置ベクトル情報が、normalVectorID のベクトルに法線ベクトル情報が代入されます。

■ 直線とモデルとの衝突判定

モデルはポリゴンの集合体なので、直線とモデルとの衝突判定は、直線とポリゴンとの衝突判定の繰り返しで行う事ができます。つまりモデルを構成するすべてのポリゴンに対して、直線との衝突判定を行います。交点が複数存在する場合は、直線の原点に最も近い交点を検索します。この処理はあらかじめ関数として用意されています。

モデルをポリゴンとの衝突判定を行うには、getModelIntersection 関数を使用します。

```
bool getModelIntersection (
    int modelID,
    int directionalVectorID, int pointVectorID,
    int intersectionVectorID, int normalVectorID
)
```

最初の引数 modelID には判定対象のモデルの ID を、続く引数 directionalVectorID と pointVectorID にそれぞれ直線の方角ベクトルと原点位置ベクトルの ID を指定します。残る引数 intersectionVectorID と normalVectorID には、交点の位置と法線ベクトル情報を格納するためのベクトルを用意して、その ID を指定します。なお、モデルや直線のベクトルは、あらかじめ共通の座標系へ座標変換を行ったものを指定する必要があります。

この関数はモデルと直線との位置関係を解析し、交点が存在すれば true を、存在しなければ false を返します。交点が存在する場合は、引数に指定された intersectionVectorID のベクトルに交点位置ベクトル情報が、normalVectorID のベクトルに法線ベクトル情報が代入されます。交点が複数存在する場合は、直線の原点位置(pointVectorID で指定)に最も近いものが選択されます。

■ ポリゴン同士の衝突判定

ポリゴン同士の衝突判定には、様々な方法が考えられます。ここでは、直線とポリゴンとの衝突判定を応用した方法を解説します。

簡単のため、ポリゴンを正三角形のポリゴンとします。まず、三角形の 3 つの頂点から、となりの頂点の方向へ半直線を 3 本引き、V1、V2、V3 と名づけます。右回りに引いても左回りに引いても構いません。

続いて V1、V2、V3 と、別のポリゴンとの接触判定を行い、交点の位置を求めます。例えば V1 と別のポリゴンとの間に交点が存在すれば、V1 の原点と交点との距離を求めます。

もしこの距離が、三角形の一辺の長さよりも小さければ、この三角形と別のポリゴンは接触しています。逆に大きい場合や、そもそも交点が存在しない場合には、接触していません。

■ モデル同士の衝突判定

モデル同士の接触判定は、原理上、ポリゴン同士の接触判定をあらゆるポリゴンの組に対して行えば、厳密に行う事が可能です。直方体などの少数ポリゴンモデル同士についてはそれでも良いでしょう。しかし例えば 1000 ポリゴンのモデル同士の場合、ポリゴンの組は $1000 \times 1000 = 100$ 万通りもできてしまい、素直に衝突判定を行っていたのでは処理が追いつきません。そこでモデル同士の場合は、ある程度処理を簡略化する必要が生じてくるでしょう。

・モデル同士の距離で判定(バウンディングスフィア)

最も簡単な方法として挙げられるのが、モデル同士の中心間距離で衝突判定を行う方法(バウンディングスフィア)です。モデルの中心が原点になるように、モデルをそれぞれ座標系に配置し、あとは座標系の原点位置が一定の距離内に接近したら衝突と見なします。この方法は非常に軽い負荷で処理する事が可能なので、大量のモデル同士のおおまかな衝突判定に向いているでしょう。

・適当な箇所から半直線(レイ)を伸ばし、ポリゴンとの交点で判定

もう少し詳細な衝突判定として、半直線(レイ)との交点で衝突判定を行う事も考えられます。まず、モデルの形状を考慮した上で、でっばった箇所の先端など、ある程度衝突しそうな箇所から、衝突しそうな方向へ半直線を伸ばします。そして、この半直線と別のモデルとの交点が、一定以上近い距離に存在する場合に、モデル同士が衝突したと見なします。

・描画用のモデルとは別に、衝突判定用の荒い形状を別に作る

描画用の細かい形状のモデルを囲むように、おおまかな衝突判定用の形状を作るのも有用でしょう。VCSSLでは、これは透明なポリゴンやモデルを作って、描画用モデルに重ねて配置するなどに対応できます。衝突用の形状は、例えば直方体（バウンディングボックス）などが考えられます。

■ プログラム例

実際にワールド座標系上に箱型モデルを配置し、上から点を落下させ、箱型表面ではね返してみよう。これには直線とモデルとの衝突判定を用います。具体的には、落下する点から下向きに半直線を延ばし、それとモデルとの交点が一定以内の距離に来ると、落下速度の向きを反転させてはね返します。以下のように記述し、実行してみてください。

```
import graphics3d.Graphics3DFramework ;
import Graphics3D ;
import Math ; // abs 関数を使用するため

// モデルやポリゴンの ID を控えておく変数
int axis, box, pointPolygon ;

// ベクトルの ID を控えておく変数
int directionVector, pointVector, interVector, normalVector ;

// アニメーションや運動に関する変数
double y = 3.0; // 点の高度
double v = -0.0; // 点の速度
double a = -1.0; // 重力加速度
double dt = 0.05; // 運動のアニメーション時間間隔
```

```

// プログラムの最初に呼び出される関数
void onStart ( int rendererID ) {

    // 画面サイズや背景色の設定 (省略可能)
    setWindowSize(800, 600);
    setBackgroundColor(0, 0, 0, 255);

    // ワールド座標系上に座標軸モデルを配置
    axis = newAxisModel( 3.0, 3.0, 3.0 );
    mountModel( axis, rendererID );

    // 点の運動を見るため、点ポリゴンを配置
    pointPolygon = newPointPolygon( 0.0, 0.0, 0.0, 0.1 );
    mountPolygon( pointPolygon, rendererID );

    // ワールド座標系上に箱形モデルを配置
    box = newBoxModel( 1.0, 1.0, 1.0 );
    mountModel( box, rendererID );

    // 衝突判定直線の方法ベクトルを用意
    directionVector = newVector( 0.0, -1.0, 0.0 );

    // 衝突判定直線の原点ベクトルを用意
    pointVector = newVector( 0.0, 3.0, 0.0 );

    // 交点格納用のベクトルを用意
    interVector = newVector();

    // 法線格納用のベクトルを用意
    normalVector = newVector();
}

```



```

// 画面更新周期ごとに毎秒数十回呼び出される関数
void onUpdate (int renderer) {

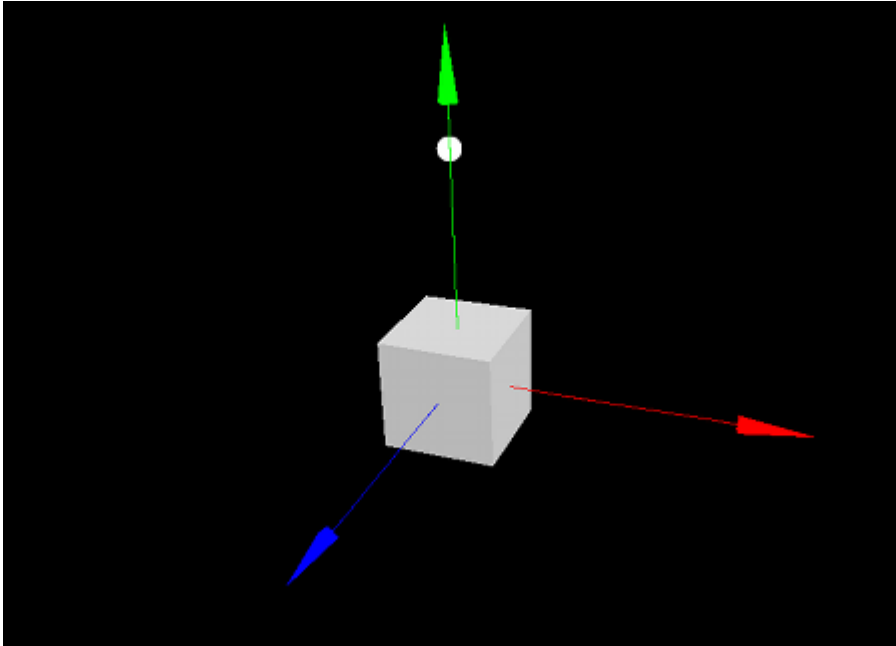
    // 落下する点の位置を計算
    v += a * dt ;
    y += v * dt ;

    // 点ポリゴンの位置と衝突判定直線の原点位置を更新
    setVector( pointVector, 0.0, y, 0.0 ) ;
    setPolygonVector( pointPolygon, 0.0, y, 0.0 ) ;

    // 衝突判定、戻り値は交点の有無
    bool b = getModelIntersection(
        box, directionVector, pointVector, interVector, normalVector
    );

    // 交点が存在し、距離が 0.2 以内なら速度反転して跳ね返す
    if( b ){
        // abs 関数は絶対値を返す関数
        if( abs( getVectorY( interVector ) - y ) < 0.2 ){
            v = abs( v ) ;
        }
    }
}

```



このプログラムを実行すると、画面に白い箱とボールが表示されます。ボールは落下していき、ボールの表面で跳ね返ります。

本文書内の商標について

[1] Oacle と Java は、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。

[2] Windows は米国 Microsoft Corporation の米国およびその他の国における登録商標です。

[3] Linux は、Linus Torvalds 氏の米国およびその他の国における商標または登録商標です。

その他、文中に使用されている商標は、その商標を保持する各社の各国における商標または登録商標です。

プログラミング言語 VCSSL リファレンスガイド 題 22 版

著 者 松井文宏

このガイドの内容は、以下の Web サイトにおいても公開されています。

<https://www.vcssl.org/ja-jp/doc/>

