VR システム CompleXcope
プログラミングガイド

陰山 聡[1]、佐藤 哲也[2]

文部省 核融合科学研究所　理論・シミュレーション研究センター

509–5292 岐阜県土岐市下石町 322–6

1) kage@toki.theory.nifs.ac.jp
2) sato@toki.theory.nifs.ac.jp

目次

1 CompleXcope の概要 **5**

1.1 バーチャルリアリティとビジュアリゼーション 5
1.2 CompleXcope のハードウエア 5
1.3 CompleXcope プログラミング 6
1.4 このガイドの目的と構成 7

2 OpenGL 入門 **9**

2.1 OpenGL の役割 9
2.2 テキストと Web ページについて 10
2.3 OpenGL を補助するライブラリ 10
2.4 OpenGL プログラムの基本構造 11
2.5 OpenGL の名前規則と関数の接尾子 13
2.6 OpenGL のステート 13
2.7 点、線、多角形の描画 14
2.8 プリミティブの種類 17
2.9 構成した物体を立体的に見せるために 20
2.10 隠面処理とデプステスト 21
2.11 法線ベクトルの指定方法 23
2.12 照明処理 24
2.12.1 照明のモード 24
2.12.2 照明モデル 25
2.12.3 環境光、拡散光、鏡面光 26
2.12.4 照明の使い方 27
2.13 材質の定義 29
2.14 法線ベクトルと照明 30
2.15 ここまでのまとめ 31
2.16 aux ライブラリによる様々な立体の表示 31
2.17 glu library による球、円筒、円盤の描画 33
2.18 ビューイング 34
2.19 モデルビュー変換 35
2.20 射影変換 38
2.21 サンプルプログラムのビューイングについて 39
2.22 サンプルプログラム (平行移動１) 40
2.23 aux library によるウインドウの制御 41
2.24 サンプルプログラム (平行移動２) 42
2.25 サンプルプログラム (回転変換) 46
2.26 サンプルプログラム (平行移動と回転の組合せ) 47
2.27 行列のスタック 47
2.28 サンプルプログラム (スタックの使い方) 49
2.29 アニメーション 51
2.30 アニメーション２ 54
2.31 ディスプレイリスト 57

2.32 テクスチャマッピング 60

3 CAVE ライブラリの使い方 65
3.1 CompleXcope のハードウエアシステム 65
3.2 ワンドについて 65
3.3 CAVE ファイル 65
3.4 CAVE 座標系 66
3.5 プロセス 66
3.6 Configuration File 67
3.7 CompleXcope プログラムの流れ 67
3.8 CAVE シミュレーター 68
3.9 サンプルプログラム 1 69
3.10 サンプルプログラム 2 72
3.11 サンプルプログラム 3 (アニメーション) 74
3.12 サンプルプログラム 4 76
3.13 サンプルプログラム 5 78
3.14 クリッピングプレーン 81
3.15 共有メモリの解放 82
3.16 ワンドによるコントロール 82
3.17 位置の検出 83
3.18 方向の検出 84
3.19 ワンドボタン状態変化の検出 84
3.20 ワンドボタン現在の状態の検出 85
3.21 ジョイスティックの傾きの検出 85
3.22 ワンドを使った相互作用のサンプルプログラム 86
3.23 ナビゲーション 90
3.24 CompleXcope プログラミングのまとめ 94

A CAVE システム診断コマンド 95

B OpenGL の画像をファイルに保存する方法 97

C C 言語入門 (`diff f c`) **101**
C.1 C 言語の基本的構造 101
C.2 コメント 102
C.3 変数の基本型と宣言 103
C.4 演算子 103
C.5 関係演算子 104
C.6 if 文 104
C.7 関数 104
C.8 for 文 106
C.9 while 文 106
C.10 配列 107
C.11 ポインタ 108
C.12 構造体 110

D テクスチャの作成法 113
D.1 任意の画像ファイルからテクスチャを生成するコマンド 113
D.2 x2texture コマンドのソースコード 113

第 1 章

CompleXcope の概要

1.1 バーチャルリアリティとビジュアリゼーション

様々な科学研究分野の中でも特にプラズマ物理学の研究では、スーパーコンピュータによる大規模な計算機シミュレーションが主要な研究手段の一つとなっています。従って、大量に生成される数値データを解析する手段としての可視化 (ビジュアリゼーション) には多くの努力が払われています。これまでその道具は主に高速グラフィックワークテーションと AVS 等のビジュアリゼーションツールとの組合せでしたが、このような２次元的な解析手法ではデータに潜む複雑な３次元構造や時間発展を把握するには不十分である場合が多く、我々は全く別の技術に基づいた革新的なビジュアリゼーション装置の必要性を痛感していました。

幸いなことに、近年バーチャルリアリティ (VR) 技術が急速に進歩し、既に一部では商業化されるほどのレベルに達しています。 VR の特徴として

広い視野角での立体視 — 対象がすべて立体的に見え、その視野角が１８０度近くある。

リアルタイム — 体験者が頭を回転させたり歩いたりしても画像がその動きに応じてリアルタイムで反応する。

対話性 — 仮想現実空間の物体をその場で動かしたり、作り出したり変形したりできる。

などが挙げられます。これらの条件は体験者の没入感 — 体験者が仮想現実世界に完全に没入している感覚 — を高めるのに必須と考えられており、同時にサイエンティフィックビジュアリゼーションの立場から見て理想的な装置の条件でもあります。そこで我々は最先端の VR 技術を全く新しいビジュアリゼーションシステムとして活用する方法を模索してきました。

VR には、大きく分けてヘッドマウンテッドディスプレイ (Head ounted Display, H D) を使う方式と、大きなスクリーンにプロジェクターでステレオ映像を投影するスクリーン投影方式の二つの方式があります。後者のスクリーン投影方式は、広い視野角を得るために、スクリーンに囲まれた部屋の中に体験者を入れてしまうという、ある意味では全く単純なアイディアに基づくシステムですが、その効果は劇的と言っていいほどのものがあります。この方式は大きなスペースを必要とすることと、比較的コストが高いという欠点があるものの、高い没入感という点ではずっと優れています。特に、イリノイ大学シカゴ校 (Univ. Illinois, Chicago) Electronic Visualization Laboratory (EVL) [1]において開発された CAVE と呼ばれる VR システムは、スクリーン投影方式の VR システムのパイオニアであり、かつ最も洗練されたものです。 1996 年、このガイドの著者の一人 (佐藤) は、 CAVE システムについての情報を得るとすぐ、それをデータビジュアリゼーションに応用する可能性を検討するためにその実物を体験してみました。そしてその結果、 CAVE システムの完成度は非常に高く、我々の目的には既に十分な程のレベルに達していると判断しました。このような経緯から、核融合研、理論・シミュレーション研究センターは、 1997 年９月、CAVE 方式に基づく VR システムを導入し、複雑 (complex) なデータを可視化・解析するための道具という意味を込めて CompleXcope と名付けました。

1.2 CompleXcope のハードウエア

CompleXcope システムの中心部は、一辺が 10 フィート (約３メートル) の大きな立方体の部屋です。部屋の正面、右、左の壁面、及び床は全てスクリーンになっています。正面と右側の壁面スクリーンには背後から、床に対しては天井からステレオプロジェクターで映像が投影されます。 (現在のところ左の壁面のスクリーンには投影されま

[1]http://www.evl.uic.edu/EVL/index.html

せん。) そして、これら正面、右、床の３つのスクリーンに投影された画像は滑らかに、切れ間なくつながっています。体験者は液晶シャッター眼鏡をつけ、後で説明するワンドと呼ばれる小さなコントローラを手に持ってこの部屋の中に立ちます。スクリーンの境目では画像は十分滑らかにつながって表示され、ほとんどどちらを向いても画像が見えます。しかもそれが距離感を伴った立体画像であるため、体験者はすぐにスクリーンの存在を忘れ、目の前に本当に物体が浮いているような感覚を覚えます。 CompleXcope を初めて体験した人の多くが声を上げて驚き、手を伸ばして目の前に浮いている物体に触ろうとする程の実在感があります。

CAVE システムのハードウエアは次のような要素から構成されています。

- グラフィックワークステーション
- ステレオ画像プロジェクター
- スクリーン (10 フィート × 10 フィート、壁 3 面 + 床 1 面)
- 液晶シャッター眼鏡 (赤外線同期方式)
- トラッカーシステム (磁気による位置、方位センサー)
- ワンド (インターフェース用の小型コントロール装置)

ワンドとは長さ２０ cm ほどのコントロール装置です。 CompleXcope 内部に立つ体験者はこれを片手に持ちます。ワンドにはいくつかボタンがついていて、どれもワンドを片手で持ったまま操作出来ます。これらのボタンを利用して体験者は CompleXcope 内部に立ったまま、仮想現実物体を変形したり、新たに作り出したり、消去したりすることが出来ます。つまり仮想現実世界をリアルタイムで、対話的に操作できるのです。トラッカーシステムとは、体験者がかけている液晶シャッター眼鏡や手に持っているワンドの位置と方向を検知するシステムです。これらのデータを利用して、 CAVE システムは体験者の視点情報をリアルタイムに更新することが可能になります。つまり、体験者が CompleXcope 内部を歩き回ったり、しゃがみ込んだり、まわりを見渡してみたりしても、スクリーンに投影される画像が自動的に映像が更新されるため、常にそこから見えるべき景色が見えるのです。また、トラッカーシステムからのデータは、アプリーケーションプログラムから取り込むことも可能です。

1.3 CompleXcope プログラミング

CAVE のハードウエアとのインターフェースには、 CAVE ライブラリと呼ばれる EVL で開発されたライブラリを利用します。 CAVE ライブラリは、必要な UNIX プロセスの生成、スクリーンどうしの映像を滑らかにつなげること、上に述べた体験者の視点位置からの適切な射影変換の実行、ステレオ表示に必要なダブルバッファの切替え等、 VR の構成に必須で、しかし最も面倒な部分を自動的に処理してくれる優れたライブラリです。従って CompleXcope をビジュアリゼーションに利用する研究者は、自分が作り出したい仮想現実世界の構成だけに集中すればよいのです。あとは CAVE ライブラリが全て自動的に処理してくれます。仮想現実世界の構成とは、以下の情報を指定することを意味します。

1. 対象とする 3 次元物体の形状
2. その仮想空間での位置 (座標値)
3. その材質がもつ光学特性 (色、反射率等)
4. 光源の位置、色、背景の色

これらは、あらゆる 3 次元コンピュータグラフィックスに共通する基本的な作業です。このような目的のために開発されたグラフィックスライブラリはたくさんありますが、 CompleXcope プログラミングでは主に OpenGL とよばれるグラフィックスライブラリを利用します[2]。 OpenGL は現在事実上の世界標準となっている 3 次元グラフィクスライブラリです。従って CompleXcope でビジュアリゼーションを行うプログラムを書くには OpenGL と CAVE ライブラリという二つのライブラリを知っておく必要があります。このガイドでは第 2 章で OpenGL を、第 3 章では CAVE ライブラリについて解説します。結局、我々がこれから勉強する CompleXcope プログラミングとは、簡単にいえば

[2] その他、Iris GL、 Open Inventor, Iris Performer が利用可能

OpenGL を使って定義した仮想世界を CAVE ライブラリを使って CompleXcope 内に “現実化” する

ことだと言えるでしょう。

1.4 このガイドの目的と構成

このガイドが対象とする読者は、これから CompleXcope システムを用いてデータのビジュアリゼーションを行うあらゆる研究者です。上で述べたように CompleXcope を利用するには OpenGL で CG プログラムを書く方法を知っておく必要があります。しかし CompleXcope で必要となるのは、 OpenGL が持つ多彩な機能のうちのほんの一部です。その理由は、我々の目的とするサイエンティフックビジュアリゼーションが、数値データに潜んでいる構造の把握・解析なので、景観シミュレーション等のように現実の風景と区別がつかない程リアルな画像を作る必要がないためです。 OpenGL にはそのような高品位の CG 作成を可能にする様々な機能が用意されていますが、我々の目的にはそれらは必要ありません。そこでこのガイドでは CompleXcope プログラミングで必要となる OpenGL の基本概念と関数だけをなるべく丁寧に解説します (第 2 章)。この第 2 章では OpenGL で作成した CG をワークステーションのモニターに表示させる方法も解説します。従って、この章は CompleXcope とは独立な “サイエンティフィックビジュアリゼーションのための OpenGL 入門” としても読むことが出来ます。実際、 AVS 等のビジュアリゼーションツールはその中で OpenGL を呼んでいるのです。この章を読めば AVS 等ではできないような細かい設定のビジュアリゼーションプログラムを自分で直接作ることが出来るようになるでしょう。

第 3 章では CAVE library の基本的な機能と使い方について解説します。 OpenGL とは違い、 CAVE library については市販されている解説書などは (今のところ) 存在しません。 Web を通じて得られるマニュアルがほとんど唯一のドキュメントです。しかしながら、 CAVE library は非常に上手く設計されたライブラリなので、そのプログラミング方法をマスターすることはとても簡単です。 (OpenGL に比べたら何でもありません。) 実際、 OpenGL プログラムの画像をワークステーションのモニターに表示させるプログラムよりも、同じ映像を CompleXcope に出力させるプログラムの方がむしろ簡単になります。モニターに出力させる場合は、 X のウインドウ制御やイベント処理、 (3 次元空間から 2 次元面への) 射影変換などを行う部分も自分でプログラムしなければいけないのですが、 CompleXcope では CAVE library がそれらを全て自動的に処理してくれるからです。

Appendix A では CAVE システムの診断に利用されるコマンドを、 Appendix B では OpenGL の画像をイメージファイルに保存する方法について紹介します。

CompleXcope プログラミングでは、 C(または C++) 言語でプログラムを書く必要があります。大規模計算機シミュレーションの分野では、 Fortran のエキスパートではあっても C はあまり使わないという研究者が多いと思います。そこで Appendix C では Fortran プログラマのための C 言語の簡単な入門を試みました。第 2 、 3 章のサンプルプログラムを読んで理解するのに必要最小限の事柄をカバーすることを目標にしましたが、もちろんこの短いページ数では C の完全な解説は不可能です。 C については市販の解説書が多数ありますので、必要に応じてそれらを参照してください。 Appendix D ではテクスチャマッピングで貼り付けるテクスチャ (画像) を pict や gif 等、任意フォーマットの画像から生成するコマンドを紹介します。

第 2 章

OpenGL 入門

2.1 OpenGL の役割

OpenGL[1]は現在標準的なグラフィックスライブラリです。歴史的には SGI (Silicon Graphics, Inc.) が開発した GL というライブラリがその起源ですが、現在は OpenGL ARB (Architecture Review Board) という組織が、その仕様の決定とライセンスの発効を行っています。

OpenGL の役割を理解するために、まず 3 次元コンピュータグラフィックス (CG) で行う一般的な作業について考えてみましょう。

(1) 仮想的な 3 次元空間を想定する
(2) そこにさまざまな 3 次元物体を置く
(3) その表面の材質を表現する光学特性 (反射率等) を指定する
(4) 光源の色、位置と光の出る方向を指定する
(5) 視点の位置と方向を指定する
(6) その視点で見えるべき風景を計算 (射影) し、カラーバッファに書き込む
(7) カラーバッファの内容を計算機のモニター画面に表示する

このうち CG プログラマにとって最も重要な部分は (1) から (5) までの作業で、 OpenGL はまさにこの作業を行うためのライブラリです。 (6) は最も大変な計算で、グラフィックスワークステーションではハードウエアを使って高速に処理します。最後の (7) は当然、それぞれの計算機システムに依存する部分です。正確に言えばその計算機で使っているウインドウシステムに依存します。 OpenGL はこの (7) の作業には全く関わりません。つまり

OpenGL はウインドウシステムとは独立

なのです。 OpenGL の関数は約 120 個ありますが、その中にはウインドウを開いたり、ウインドウに画像の表示を指示したりする関数はありません。このため OpenGL は機種に依存しない汎用的なプログラム作成を可能にしています。現在 X Window、 acintosh、 Windows95/NT、 Be など様々なシステムで OpenGL は使用可能ですが、基本的な部分は全て同じソースコードで同じ CG を作成できるわけです。ただしバッファの内容をウインドウに表示する部分だけはシステム依存ですから、このインタフェース部分だけは変える必要があります。我々の場合、最終的には OpenGL で作成した CG を CompleXcope で表示させることになりますが、いきなり CompleXcope に表示させるのは大変なので、まずはモニタ画面の X Window に表示させて OpenGL の練習をしていきましょう。

X Window 上で本格的に OpenGL プログラミングをするには glut library[1] と呼ばれる . Kilgard によって開発されたライブラリを使うのが一般的です。しかし、ここでの我々の目的は OpenGL の基本機能の理解にあるので、 glut library よりも手軽にウインドウ制御が出来る aux library という補助ライブラリを使うことにします[2]。こ

[1]書き方に注意！ “OpenGL” です。 “Open GL” ではありません。
[2]Xlib、 Motif、 Tcl/Tk などでウインドウの制御をすることも出来ます。

れはウインドウを開いたり、画像を表示するなど、必要最小限の機能を備えたライブラリです。また、後で示しますが、球や円筒など基本的な３次元物体を簡単に構成する関数も用意されています。これから多数のサンプルプログラムを見てきますが、そのソースコードの中に “aux” で始まる関数があったらそれは aux library の関数です。

一方、 CompleXcope に画像を表示する場合には上の (7) の作業は CAVE library が行います。 CAVE library の関数名は “CAVE” で始まっているのですぐに分かります。 CompleXcope ではウインドウを直接扱うことは全くありませんので、当然ながらウインドウ制御に関係した aux library の関数を呼ぶことは全くありません。

実は、上記の (5) の作業 (視点の位置と方向の指定) も CompleXcope では必要ありません。メインの (トラッカーにつながっている) 液晶シャッター眼鏡の位置が常に視点になっていなければならないからです。ですから実は OpenGL の機能のうち、視点の設定に関する関数は CompleXcope プログラミングには必要ありません。トラッカーからの情報を元に視点をリアルタイムで更新する作業は CAVE ライブラリが自動的にやってくれます。 CAVE ライブラリの中で、視点を指定する OpenGL 関数を時々刻々呼んでいるのです。

2.2 テキストと Web ページについて

OpenGL に関する公式テキストとして “*OpenGL Programming Guide*” があります。第一版 [2, 3] は 1993 年に、第２版 [4, 5] は 1997 年に発行されています。第一版では aux library を、第二版は glut library を使って解説されています。赤い表紙のこの本は、 OpenGL のバイブルと呼ぶべきもので、 OpenGL に関して必要なことはほとんど全て書いてありますし、分かりやすく書かれた良い本です。ただし日本語の訳文はあまり良くありません。また、 OpenGL の全ての関数について記述したマニュアルとして青い表紙の “*OpenGL Reference Manual*” があります [6]。また OpenGL が使えるグラフィクスワークステーション上では man コマンドによって OpenGL のマニュアルをオンラインで見ることが可能です。赤い (R) 表紙の本と青い (B) 表紙の本があれば当然緑 (G) の表紙の本があると期待されます。それが上で触れた glut の開発者によって書かれた “*OpenGL Programming for the X Window System*”[1] です。日本語で書かれた OpenGL のテキストとしては [7] があります。

OpenGL についての最新情報は OpenGL の公式ページ

http://www.opengl.org/

を見て下さい。日本語で書かれた OpenGL の WEB ページとして OpenGL_Japan という団体によるものがあります。

http://www.kgt.co.jp/opengl/

また、 OpenGL の FAQ については

http://www.sgi.com/Technology/openGL/

その日本語訳は

http://tech.webcity.ne.jp/~andoh/opengl/openglfaq.html

をご覧下さい。

また、 OpenGL とほとんど互換なライブラリとして esa

http://www.ssec.wisc.edu/~brianp/ esa.html

というものがあります。 esa は Avid Technology, Inc に所属する Brian Paul によって開発され、フリーで入手可能です。

2.3 OpenGL を補助するライブラリ

後で詳しく述べますが、 OpenGL では

- ウインドウの制御
- 球、円筒、トーラス等の基本的３次元物体の定義
- (アニメーション作成に必須な) ダブルバッファの切替え
- X フォントの利用

等は全く出来ないか、出来ても大変手間がかかります。そこでこれらのことを可能にするための補助的なライブラリが何種類か用意されています。

aux library	ウインドウ表示に必要最小限な機能を備えたライブラリ。 球などを簡単に描く関数も用意されている。関数は全て 'aux' という接頭子で始まる。
glu library	視界の方向や射影変換などを簡潔に設定する時に使う。 aux library と同様に球などを簡単に描く関数も用意 されている。'glu' という接頭子を持つ。
glX library	X Window に画像を表示するのに必要な関数を集めた ライブラリ。'glX' という接頭子をもつ。
glut library	X Window 及びそれ以外の多くのウインドウシステムで OpenGL プログラムを表示させるのに利用される高機能 な関数を集めたライブラリ。'glut' という接頭子を持つ。

このガイドでは aux library と glu library を中心に利用します。 glX 関数はダブルバッファの切替えだけに、 glut 関数は全く使いません。

2.4 OpenGL プログラムの基本構造

それでは aux library を使った OpenGL の最も単純なプログラムを見てみましょう。この OpenGL プログラムは $z = 0$ の平面に置かれた白い正方形を描くものです。

simple.c

```
 1:  #include <GL/gl.h>
 2:  #include "aux.h"
 3:
 4:  main() {
 5:
 6:      auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
 7:      auxInitPosition (0, 0, 500, 500);
 8:      auxInitWindow ("simple.c");
 9:
10:      glClearColor (0.0, 0.0, 0.0, 0.0);
11:      glClear(GL_COLOR_BUFFER_BIT);
12:      glColor3f(1.0, 1.0, 1.0);
13:      glMatrixMode (GL_PROJECTION);
14:      glLoadIdentity ();
15:      glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
16:      glBegin(GL_POLYGON);
17:          glVertex3f(-0.5, -0.5, 0.0);
18:          glVertex3f(-0.5,  0.5, 0.0);
19:          glVertex3f( 0.5,  0.5, 0.0);
20:          glVertex3f( 0.5, -0.5, 0.0);
21:      glEnd();
22:      glFlush();
23:      sleep (10);
24:  }
```

最初ですので一行づつ見ていきましょう。

第１行では OpenGL プログラムに必要なファイルを include しています。このファイル[3]では OpenGL で使う全てのマクロ、定数の定義、 OpenGL 関数のプロトタイプ宣言などがなされています。 (CompleXcope プログラムも含めて)OpenGL 関数を使う全てのソースコードでは常にこの行を書く必要があります。

第２行は、 aux library の使用に必要なファイルの include です。この章のサンプルプログラムは全て aux library を使用して OpenGL の練習をしますので、この行も必須です。しかし上で述べたように CompleXcope プログラムでは、この include 文はなくなるでしょう。

第４行からが main 関数です。

第６行から８行では aux library の関数を呼んでいます。第６行では、表示を (1) シングルバッファモードで、かつ (2) RGB モードで行うこと、を指示しています。シングルバッファモードに対して、もう一つダブルバッファモードもあります。これはアニメーションを作る時に使います。また、 RGB モードとは色の指定を行う方法の一つで、 Red, Green, Blue の値を直接指定するモードです。もう一つカラーインデックスモードというのもあります。これはカラーテーブルを用意して、一つのインデックスから色を決める方法で、よりメモリを節約する方法ですが、大量のフレームバッファを持つ現在のグラフィックスワークステーションではほとんど使うことはないでしょう。

第７行ではモニター上に表示するウインドウの位置と大きさを指定しています。ウインドウの左上隅の座標値が (0,0) ピクセル、右下隅が (500, 500) ピクセルと引数で指定しています。ここで座標系は、 (X Window のルールに従い) 原点はモニターの左上隅で、 +x 方向は右、 +y 方向は下です。

第８行の auxInitWindow() でウインドウを開きます。引数にあたえた文字列がウインドウのタイトルバーに表示されます。

第１０行以降、 “gl” で始まる関数がいよいよ OpenGL 関数です。第１０行の glClearColor() では、バッファをクリアした後に設定する色を指定します。画像の背景色だと考えて構いません。４つの引数はそれぞれ R, G, B, A の値で、０から１の値をとる実数です。 A は Alpha 値を表し、これについての説明はここでは省略します。

第１１行ではバッファをクリアしています。これでバッファ RGBA が全て 0、すなわち黒にセットされます。正確に言えばここでクリアしたのは color buffer です。バッファには他に depth buffer、 stencil buffer などがあります。

第１２行では、描画に用いる色の指定をしています。ここでは $(R, G, B) = (1.0, 1.0, 1.0)$ つまり白を指定しています。これ以降、何かのオブジェクトを描けばそれは白になります。

第１３行と１４行では射影行列を単位行列に初期化しています。その意味については後で詳しく説明します。

第１５行では３次元世界を２次元画像に写すときに正射影を用いることを指定しています。正射影とは視点が無限遠にある場合の射影です。 glOrtho() の６つの引数は射影する３次元領域の (左、右、下、上、手前、向こう) のそれぞれの面の位置を表しています。

第１６行から第２１行まで (glBegin から glEnd まで) が一つの描画単位になっていて、この部分で正方形を描いています。ここがプログラムの最も重要な部分です。第１６行の glBegin() の引数に `GL_POLYGON` が書かれています。これはこれ以降描くものが多角形の面であることを指定しています。

第１７行では正方形の左上の頂点の x,y,z 座標 (-0.5, -0.5, 0.0) を指定しています。

第１８行から２０行までも同様でそれぞれ左下、右下、右上の座標を指定しています。

第２１行の glEnd() で多角形の描画が終了します。

第２２行の glFlush() は、それまで行った描画指令が何らかの理由で滞っている場合でも[4]、強制的に描画を実行させる関数です。必須ではありませんが、描画作業が終了したら常にこの関数を呼ぶようにします。

第２３行は (OpenGL とは関係ない) C の関数で、その機能は「１０秒間なにもしない」というものです。これをとってしまうとプログラムは描画の直後に (glFlush の後) 終了 (正常終了！) します。従って、ウインドウは一瞬表示されてすぐに消えてしまうため、結果の確認が出来ません。そこで sleep(10) を呼ぶことにより 10 秒間ウインドウを表示し続けさせているのです。この章の後半で示すサンプルプログラムでは、 sleep を使わずにウインドウをずっと表示し続ける方法を紹介します。

---

[3] /usr/local/include/GL/gl.h

[4] たとえば OpenGL のクライアントとサーバがネットワークと通じて通信している場合、パケットがいっぱいになるまでクライアントが待っているときなど。

2.5 OpenGL の名前規則と関数の接尾子

OpenGL 関数は必ず “gl” で始まり、関数名を構成する単語の先頭文字を大文字にします。(例えば “glClearColor”)。定数は “GL_” で始まり、全て大文字で表記して単語間には下線を加えます。

また、例えば “glColor3f()” のように、関数名の最後に数字を含む文字 (この場合 “3f”) が含まれている場合があります。このうち、数字の 3 は、この関数が引数を 3 つとることを意味します。色を指定する関数には “glColor4f()” というものもあり、これは引数を 4 つとります。また、 “3f” の “f” は、 float 即ち実数を引数にとることを意味します。つまり関数名の接尾子 “3f” は、「3 つの実数型の引数をとる」ということを表しているわけです。主な接尾子のアルファベット部の種類と、それが意味する引数のデータ型を以下にまとめます。

接尾子	データ型	OpenGL の型定義
b	8 ビット整数	GLbyte
i	32 ビット整数	GLint
f	32 ビット実数	GLfloat
d	64 ビット実数	GLdouble

ここで、各型のメモリサイズが決まっていることに注意して下さい。 C 言語の int や float 等のサイズはシステム依存です。 GLint や GLfloat 等を使うことでシステムに依存しない汎用的な OpenGL プログラムを書くことが可能になります。さて、 OpenGL の関数名には末尾に “v” を付けたものが多くあります。この “v” はベクトル (配列) を意味します。この場合、関数はその引数として、いくつかの引数を別々に受け取るのではなく、数値の配列（つまりベクトル）のポインタをとります。複数の引数を持つ OpenGL の関数のほとんど全てが、ベクトル受け取りと非ベクトル（個別）受け取りのバージョンを持っています。次の例を見てみましょう。

```
float color_array[] = {0.0, 0.0, 1.0};  /* 青 */

glColor3f(1.0, 1.0, 0.0);       /*  現在のカラーを黄に設定  */
glColor3fv(color_array);        /*  現在のカラーを青に設定  */
```

2.6 OpenGL のステート

OpenGL には様々な状態（ステート）、あるいはモードがたくさんあります。これまで出てきたものとしては、“現在のカラー” というのがステートの一つです。ステートの例としては他に、 “現在の点の大きさ”、 “現在の線の太さ”、 “現在のカラーバッファの消去値” （上で背景色として説明したもの）、 “現在の照明の色” などがあります。ステートに関して重要なことは、

ステートは、一度決めたら変更されるまで有効

ということです。再び、色を指定する関数 “glColor3f” を使い、この点を見てみましょう。上記の “simple.c” のうち、正方形の描画に関係する部分（第 16 行 glBegin() から第 21 行 glEnd() まで）を抜き出して、関数 “draw_rectangle()” を定義したとしましょう。

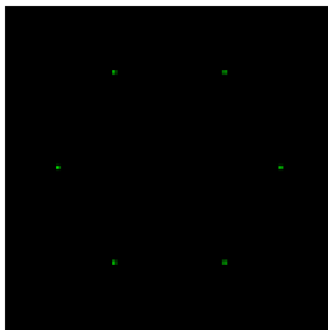
```
glColor3f(1.0, 1.0, 1.0);
draw_rectangle();          /* 白い正方形を描く */

glColor3f(1.0, 0.0, 0.0);
draw_rectangle();          /* 赤い正方形を描く */
draw_rectangle();          /* 赤い正方形をもう一つ描く */

glColor3f(1.0, 1.0, 0.0);
draw_rectangle();          /* 黄色い正方形を描く */
```

このように、glColor3f() で “現在のカラー” を設定すると、それ以降、再び “現在のカラー” ステートが変更されるまで、それ以降はその色で描かれます。（上のプログラムは、このままでは４つの正方形が重なってしまいますが、今はこの点は考えないことにしましょう。）

2.7　点、線、多角形の描画

OpenGL ではどんなに複雑なオブジェクトも、点と線と多角形の組み合わせとして構成します。実際にはほとんどの３次元物体は多角形の集まりとして構成されます。（磁力線などは当然線で定義しますが。）そして、この三種類の基本図形（プリミティブと呼ばれます）は全て頂点 (vertex) の位置を指定することで定義されます。

まず点プリミティブのサンプルを見てみましょう。次のプログラムでは、$z = 0$ の平面上の正６角形の頂点に置かれた６つの点を描きます。



simple2.c

```
#include <GL/gl.h>
#include <math.h>   /* 数学ライブラリ (sin, cos) を使うため */
#include "aux.h"

main() {

    float pi;
    float x0, x1, x2, x3, x4, x5;
    float y0, y1, y2, y3, y4, y5;

    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
```

```
    auxInitWindow ("simple2.c");

    glClearColor (0.0, 0.0, 0.0, 0.0); /* 背景が黒 */
    glClear (GL_COLOR_BUFFER_BIT);     /* カラーバッファのクリア */
    glColor3f(0.0, 1.0, 0.0);          /* 緑色 */
    glPointSize (5.0);                 /* 点の大きさ 5 ピクセル */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (-1.5, 1.5, -1.5, 1.5, -1.5, 1.5); /* 射影領域 */

    pi = 3.141593;
    x0 = cos(0*pi/3); y0 = sin(0*pi/3);
    x1 = cos(1*pi/3); y1 = sin(1*pi/3);
    x2 = cos(2*pi/3); y2 = sin(2*pi/3);
    x3 = cos(3*pi/3); y3 = sin(3*pi/3);
    x4 = cos(4*pi/3); y4 = sin(4*pi/3);
    x5 = cos(5*pi/3); y5 = sin(5*pi/3);

    glBegin(GL_POINTS);
        glVertex3f(x0, y0, 0.0);   /*     o    o      */
        glVertex3f(x1, y1, 0.0);   /*                 */
        glVertex3f(x2, y2, 0.0);   /*  o          o   */
        glVertex3f(x3, y3, 0.0);   /*                 */
        glVertex3f(x4, y4, 0.0);   /*     o    o      */
        glVertex3f(x5, y5, 0.0);   /*    正六角形      */
    glEnd();

    glFlush();
    sleep (10);
}
```

\ end of simple2.c /

このプログラムを走らせると、画面上に６つの緑色の点が現れるはずです。（そして１０秒後にウインドウが自動的に消去されます。）オブジェクトの定義のためには、要するに “glBegin” から “glEnd” の間に必要な回数だけ（必要な引数と共に） “glVertex()” をコールすればいいわけです。従って上のプログラムのうち、glBegin からglEnd までの部分は下のように簡潔に書くことが可能です。

```
    glBegin(GL_POINTS);
        for (i=0; i<6; i++) {
            x = cos(i*pi/3); y = sin(i*pi/3);
            glVertex3f(x, y, 0.0);
        }
    glEnd();
```

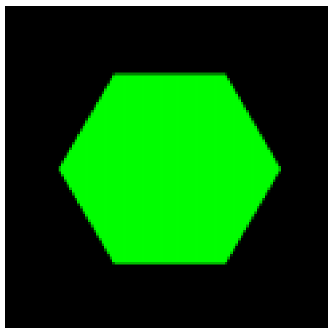
念のため、この方法で書いた完全なプログラムを以下に書きます。

simple3.c

```
#include <GL/gl.h>
#include <math.h>   /* 数学ライブラリ (sin, cos) */
#include "aux.h"

main() {

    float pi, x, y;
    int   i;

    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("simple3.c");

    glClearColor (0.0, 0.0, 0.0, 0.0); /* 背景が黒 */
    glClear (GL_COLOR_BUFFER_BIT);     /* カラーバッファのクリア */
    glColor3f(0.0, 1.0, 0.0);          /* 緑色 */
    glPointSize (5.0);                 /* 点のサイズ 5 ピクセル */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (-1.5, 1.5, -1.5, 1.5, -1.5, 1.5); /* 射影領域 */

    pi = 3.141593;

    glBegin(GL_POINTS);
        for (i=0; i<6; i++) {
            x = cos(i*pi/3); y = sin(i*pi/3);
            glVertex3f(x, y, 0.0);
        }
    glEnd();

    glFlush();
    sleep (10);
}
```

\ end of simple3.c /

上のプログラムでは点プリミティブを描きましたが、これを少し変更して多角形プリミティブを描くように変更してみましょう。そのためには、上の “glBegin()” の引数、 “GL_POINTS” を “GL_POLYGON” に変えるだけです。



simple4.c

```
#include <GL/gl.h>
#include <math.h>
#include "aux.h"

main() {

    int   i;
    float pi, x, y;

    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("simple4.c");

    glClearColor (0.0, 0.0, 0.0, 0.0); /* 背景が黒 */
    glClear (GL_COLOR_BUFFER_BIT);     /* カラーバッファのクリア */
    glColor3f(0.0, 1.0, 0.0);          /* 緑色 */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (-1.5, 1.5, -1.5, 1.5, -1.5, 1.5); /* 射影領域 */

    pi = 3.141593;

    glBegin(GL_POLYGON);     /*   <---- ここを変更    */
        for (i=0; i<6; i++) {
            x = cos(i*pi/3); y = sin(i*pi/3);
            glVertex3f(x, y, 0.0);
        }
    glEnd();

    glFlush();
    sleep (10);
}
```

\ end of simple4.c /

このプログラムを走らせると中が緑色に塗りつぶされた６角形が表示されます。GL_POLYGON を使う時に注意すべきことがあります。

GL_POLYGON で描くのは同一平面上にある凸多角形

だということです。凹多角形は複数の凸多角形に分割して構成します。

2.8 プリミティブの種類

これまで “glBegin()” の引数に指定するプリミティブの種類として “GL_POINTS” と “GL_POLYGON” の二つが出てきました。“GL_POLYGON” を使えば一つの多角形が定義でき、原理的にはどんなに複雑な３次元オブジェクトも、多角形の集まりとして構成できますから “GL_POLYGON” だけで十分だともいえます。

例えばさいころのような６面体を構成することを考えてみましょう。まず８つある頂点
v0, v1, ..., v7 の x、 y、 z 座標を、配列を使ってそれぞれ

```
float v0[] = {-1.0, -1.0, -1.0};
float v1[] = { 1.0, -1.0, -1.0};
   .
   .
   .
```

などと定義します。そしてこれらの配列を使い、次のようにすればいいわけです。

```
                        /*       v4      v5       */
glBegin(GL_POLYGON);    /*        +------+        */
    glVertex3fv(v0);    /*       /      /|        */
    glVertex3fv(v1);    /*    v2+------+ | v7     */
    glVertex3fv(v3);    /*      |    v3| /        */
    glVertex3fv(v2);    /*      |      |/         */
glEnd();                /*      +------+          */
                        /*     v0      v1         */
glBegin(GL_POLYGON);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v5);
    glVertex3fv(v4);
glEnd();

  .
  .    /* glBegin() と glEnd() のペアをあと４回呼ぶ */
  .
```

しかしながら、これはとても効率の悪いプログラムだといえます。なぜなら各頂点の座標データがそれぞれ３回づつ渡されているからです。

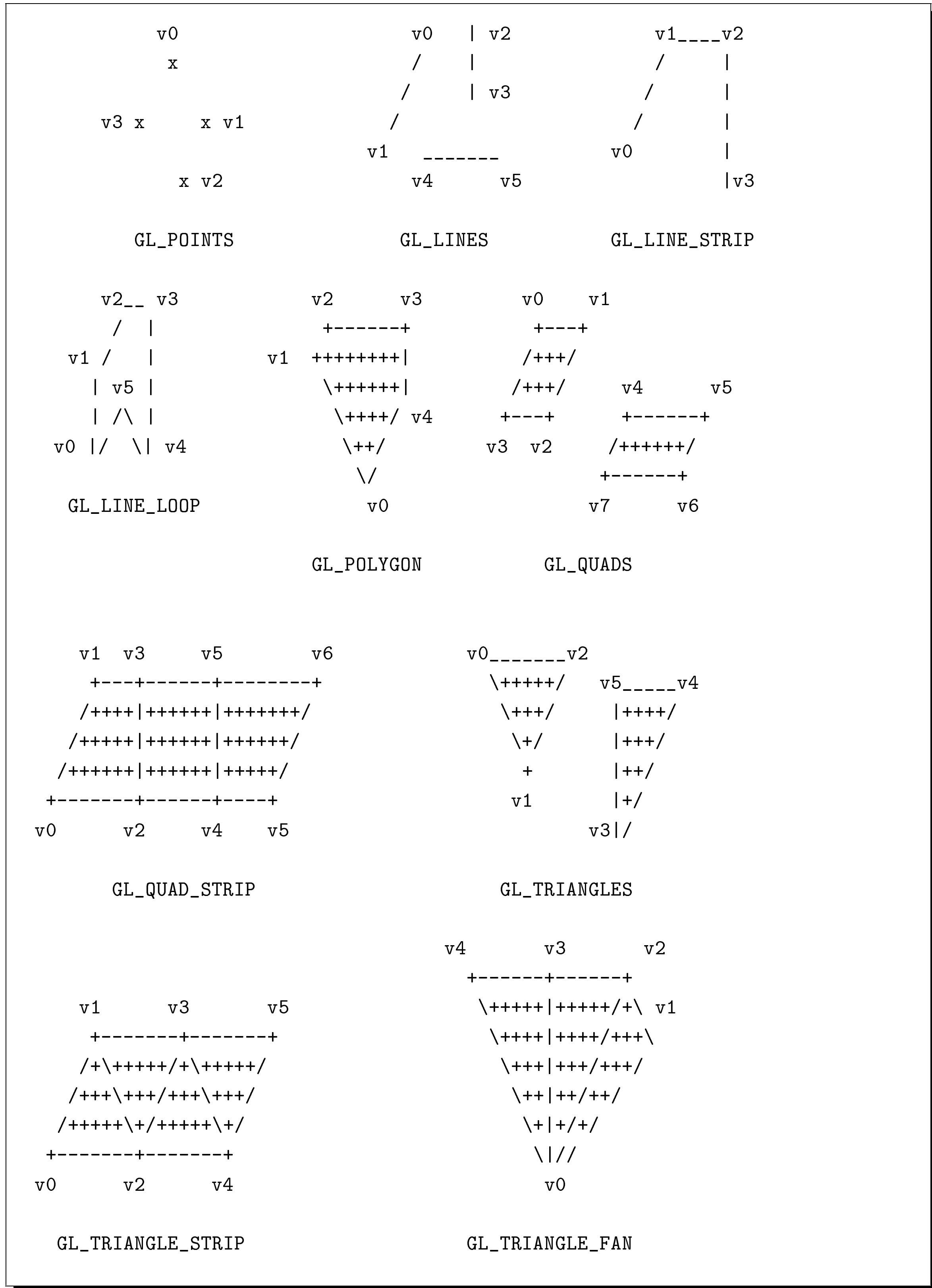
このような場合、より効率的に３次元オブジェクトを定義することが可能です。それは “GL_POLYGON” 以外にも、次のような描画プリミティブが用意されているからです。

描画プリミティブの種類とその意味

GL_POINTS	独立した点
GL_LINES	独立した線分
GL_POLYGON	凸多角形
GL_TRIANGLES	独立した三角形
GL_QUADS	独立した四角形
GL_LINE_STRIP	連結した線分
GL_LINE_LOOP	連結した線分（始点と終点も結ぶ）
GL_TRIANGLE_STRIP	連結した三角形
GL_TRIANGLE_FAN	扇形に連結した三角形
GL_QUAD_STRIP	連結した四角形

それぞれのプリミティブを引数としてglBegin() を呼んで生成される図形は次の通りです[5]。

[5] このガイドで、解説のために示す図の多くは “文字絵” で描いたものです。これはこのガイドのほとんどを携帯端末のエディターで書いたためです。この方がむしろ “味” があっていいのではないかと思うのですが . . .



それぞれの図形は全て`glBegin()` と`glEnd()` の間で`glVertex3f` を`glVertex3fv(v0)`, `glVertex3fv(v1)`, `glVertex3fv(v2)`, `glVertex3fv(v3)`, $\cdots$ の順番に呼んだ時に描かれるものです。`glBegin()` の引き数に入れるプリミティブの名前によって描かれる図形が違うわけです。これらの描画プリミティブを使えば、６面体は次のように簡潔に定義できます。

```
glBegin(GL_QUAD_STRIP);
    glVertex3fv(v0);  glVertex3fv(v1);
    glVertex3fv(v2);  glVertex3fv(v3);
    glVertex3fv(v4);  glVertex3fv(v5);
    glVertex3fv(v6);  glVertex3fv(v7);
    glVertex3fv(v0);  glVertex3fv(v1);
glEnd();

glBegin(GL_QUADS);
    glVertex3fv(v0);  glVertex3fv(v2);
    glVertex3fv(v4);  glVertex3fv(v6);
    glVertex3fv(v1);  glVertex3fv(v3);
    glVertex3fv(v5);  glVertex3fv(v7);
glEnd();
```

もちろん他のプリミティブを使って定義することも可能です。

描画プリミティブを組み合わせればどんなに複雑な3次元物体もモデルすることが出来ますが、それだけではリアリティのあるコンピュータグラフィックスを作ることは出来ません。このままでは隠面処理や照明処理がされていないので映像に立体感が全くないためです。

2.9 構成した物体を立体的に見せるために

立体感のあるリアルな3次元CGを作るためには次のような作業が必要です。

1．隠面処理
2．照明の設定（光源の指定と物体表面での反射、発光の指定）
3．法線ベクトルの設定

隠面処理とは、視線方向に複数の面が重なって存在する時に、一番手前の面だけを表示することです。これを行わないと当然、非現実的な映像が生成されてしまいます。OpenGLの場合、

デフォルトでは隠面処理は行わない

ので、それを行うように指定する必要があります。ただし隠面処理の複雑な作業自体はプログラマが行う必要はなく、ただ隠面処理を行うようにハードウエアに指示するOpenGL関数の一つを呼ぶだけなので簡単です。

次に、上の2にある照明の設定について簡単に触れておきましょう。CGで照明と言う時、日常生活で用いる照明という言葉とほとんど同じ意味と考えて大丈夫です[6]。つまり、一つまたは複数の光源で物体を照らし出すことを意味します。これまでのサンプルプログラムでは照明に関係するOpenGL関数は一つも出てきませんでした。実は、これまでのプログラムでは照明は全く使っていなかったのです。つまり、OpenGLでは

デフォルトでは照明処理を行わない

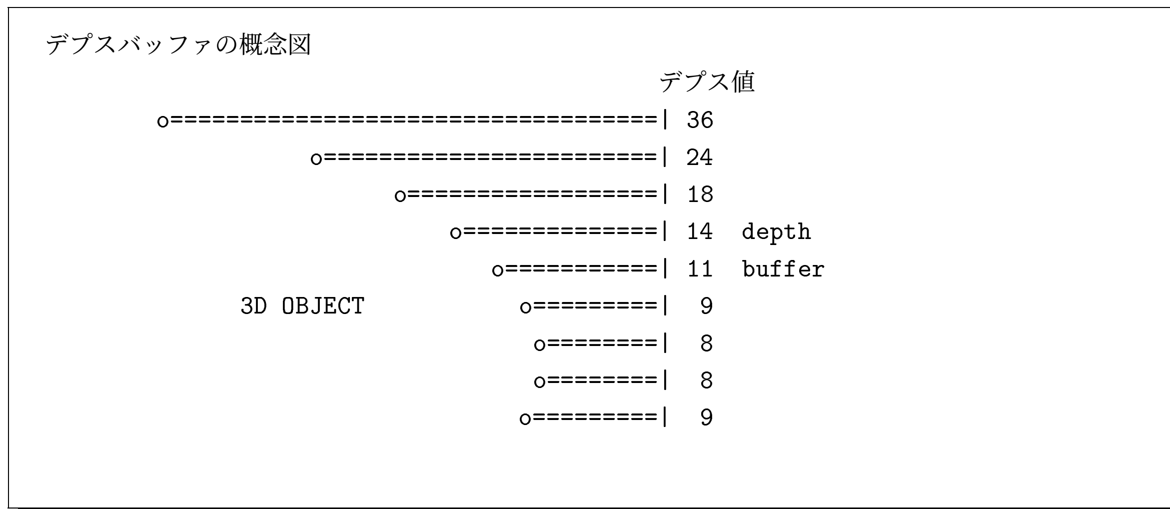
[6]ただし後述するように全く同じではありません。

のです。照明がないと、３次元物体に陰影[7]がつきません。従って、リアルな CG 画像を作成するためには照明が必須です。照明を行うように設定するには、プログラム中で OpenGL 関数の二つほど呼ぶだけなので簡単です。

しかし照明を行う場合、プログラマーにとっては少々面倒なことが生じます、それが上記の３ (法線ベクトルの設定) です。実生活で、物を照らし出してそれを見る状況を考えてみれば明らかなように、照明下で３次元物体の表面にできる陰影は、光源の位置、視点の位置、そして物体表面の各点での法線の向きで決まります。従って、照明を用いるときには３次元物体を構成する頂点各点での法線ベクトルを指定する必要があります。

2.10 隠面処理とデプステスト

上で説明したように OpenGL のデフォルトでは隠面処理[8]を行いません。隠面処理を行うように設定する関数名を紹介する前に、グラフィックワークステーションでは隠面処理をどういう方法で実現しているか理解しておきましょう。そのためにはデプスバッファというものを知っておく必要があります。



グラフィックワークステーションには、デプスバッファ (depth buffer)[9]という特別なバッファが用意されています。このバッファには、投影された２次元画像の各ピクセル位置での、「視点から一番手前の物体までの距離 (= デプス)」が保存されています。プログラムで、３次元物体 (OBJ-X とします) を新たに定義したとき、投影された２次元画像にこのOBJ-X が書き加わる可能性が出来るわけですが、そのとき次のようなテスト (デプステスト) を行い、それに合格したOBJ-X の部分だけを表示するのです[10]。

```
デプステスト
  (1) OBJ-X のある一点 (P とします) と視点との距離 D を計算する。
  (2) 点 P が射影される２次元画像中のピクセル位置 (I,J) を求める。
  (3) 現在デプスバッファに保存されているピクセル位置 (I,J) での
      デプス値と D とを比較し、 D の方が小さければ (つまり
      点 P の方が視点に近ければ) 点 P を表示する。そしてデプス値を D
      に更新する。
```

図で描けば次のようになるでしょう。

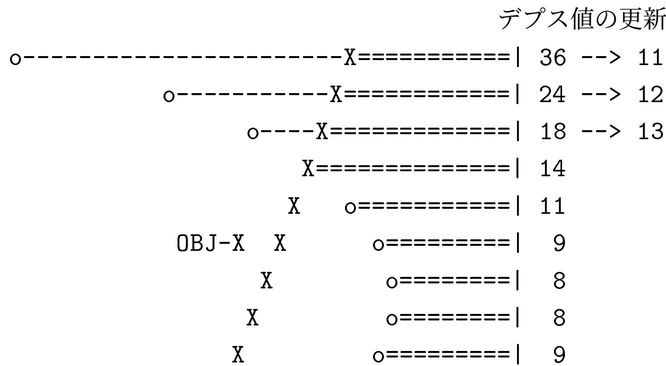
[7] “影” という意味ではありません。光が強くあたるところは明るく、弱いところは暗く見えるという効果です。

[8] ここでいう隠面処理とは、面が面に隠れるようにするだけでなく、例えば、一本の曲線が、曲面を手前から向こう側に向かって貫いているとき、曲面の向こう側にある曲線はちゃんと隠すような処理です。また、線が線に隠れるという効果も含みます。

[9] Z バッファと呼ばれることもあります。

[10] 実際の計算機ではこの通りに処理されてはいませんが、基本的なアルゴリズムはこのように理解できます。

デプステストの概念図



OpenGL で隠面処理を実現するには、プログラムの始めところ (3 次元物体の構成を行う前に)、次の関数でデプステストを行うように設定します。

```
glEnable(GL_DEPTH_TEST);
```

この関数呼び出しは、(デフォルトでは OFF であった)`GL_DEPTH_TEST` を ON にする、と解釈されます。この `glEnable()` という関数は、デプステストのような様々な OpenGL の機能を ON にするときに用いられます[11]。

デプスバッファに関する上の説明からお分かりかと思いますが、隠面処理を行う時に、もう一つ忘れてはならないことがあります。それは、画像を最初に描く時 (正確には画像を更新したとき) には必ずデプスバッファをクリアする必要がある、ということです。デプスバッファのクリアには`glClear(GL_DEPTH_BUFFER_BIT)` を呼べばいいのですが、実際のプログラムでは、デプスバッファをクリアすると同時に、カラーバッファのクリア (つまり背景色での画像の塗りつぶし) をした方がいいという場合がほとんどですから、実際のプログラムでは常に (二行続けて)

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

となるでしょう。実は、上記の二つの関数呼び出しは次のようにすれば、同じ作業をより速く行うことができるので、次のような書き方をするようにしましょう。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

具体的にどのタイミングで`glEnable()` と`glClear()` を呼ぶかは、後に出て来るサンプルプログラムを参考にして下さい。また、 aux library を使っている時に隠面処理を行う場合、`auxInitDisplayMode` (サンプルプログラム simple.c (p.11) の第 6 行参照) の引き数を
`(AUX_SINGLE | AUX_RGB | AUX_DEPTH)` として下さい。

[11] 例えば`glEnable(GL_FOG)` で、遠方の映像が白く霞んだ、まるで霧のかかったような (フォグと呼ばれる) 効果が ON になります。また、`glEnable(GL_LIGHTING)` という照明処理を ON にする例が後で出て来ます (p.25)。

2.11 法線ベクトルの指定方法

照明を ON にする時には、法線ベクトルの指定が必要です。そこで、照明について説明する前に、ここで法線ベクトルの指定方法を紹介しておきましょう。

法線ベクトルは、関数

```
glNormal3f();
```

または (配列で渡す場合)

```
glNormal3fv();
```

で指定します。法線ベクトルについて注意すべきポイントは次の 3 つです。(ただし、(2) に関しては必須ではありません。)

(1) 物体の各頂点がそれぞれ法線ベクトルをもつ
(2) 法線ベクトルは長さを 1 に規格化しておく
(3) 法線ベクトルを指定してから頂点座標を指定する

例えば 3 角形が一つあるとき、3 つの頂点全てに法線ベクトルを指定する必要があります。簡単な例を見てみましょう。

```
float n0[] = {0.0, 0.0, 1.0};    /* 頂点 1 の法線ベクトル */
float n1[] = {0.0, 0.0, 1.0};    /* 頂点 2 の法線ベクトル */
float n2[] = {0.0, 0.0, 1.0};    /* 頂点 3 の法線ベクトル */
float v0[] = {1.0, 0.0, 0.0};    /* 頂点 1 の座標 */
float v1[] = {0.0, 0.0, 0.0};    /* 頂点 2 の座標 */
float v2[] = {0.0, 1.5, 0.0};    /* 頂点 3 の座標 */

glBegin(GL_POYLGON);     /*               n1               */
    glNormal3fv(n0);     /*    n0          |               */
    glVertex3fv(v0);     /*     |          |               */
    glNormal3fv(n1);     /*     |          +         n2    */
    glVertex3fv(v1);     /*     +                     |    */
    glNormal3fv(n2);     /*                           |    */
    glVertex3fv(v2);     /*                           +    */
glEnd();
```

この例の場合は法線ベクトルは長さ 1 であることが自明ですが、そうでない場合は、平方根関数 sqrt() を使って規格化してから`glNormal` に渡すようにして下さい。

ここで一つ重要なことがあります。 OpenGL では

全ての面要素に表と裏がある

という点です。もちろん、それは法線ベクトルの向きで定義されます。

上の例では、３つの頂点に与える法線ベクトルは全て同じですから、法線ベクトルを各頂点に与えるのは、一見無意味で、無駄の様に感じますが、この様な場合の方が例外で、通常は３つの頂点には全て異なる法線ベクトルを指定します。その意義は、次の章で照明について勉強した後に明らかになります。

2.12 照明処理

2.12.1 照明のモード

既に説明したように、これまで見てきたプログラムでは照明処理を行っていませんでした。照明処理に関して重要なのは、OpenGL では

> 照明を使う状態と、照明を使わない状態の二つのモード

があり、後者がデフォルトだということです。この二つのモード (ステート) は、OpenGL の他のステートと同様、一つの OpenGL プログラムが走っている途中に交互に切替えることも可能です。実際、最終的に我々が開発する CompleXcope プログラムでは、この照明の使用／非使用のモード切替えが毎秒何度も行われるでしょう。

照明処理について説明する前に、照明を使わないモードはどういう場合に使うのか、そのとき物体はどういう色で表示されるのか、について説明しておく必要があるでしょう。簡単に言えば、

> 照明を使わないモードは、
> - (磁力線など) 曲線や、点を描く時
> - メニューなどのユーザインターフェース要素の表示
> - テクスチャマッピング

などの表示を行う時に使います。これらに共通する性質は何でしょう？それはまさに「照明処理が必要ない」ということです。例えば磁力線を緑色で描くと決めたら、光源がどこにあろうが、またその光源が何色の光を発していようが、磁力線は緑色で、かつ一定の明るさで見えるようにしたいわけです。従ってこのときは照明を OFF にして磁力線を描きます。そして、この緑という色は、前回説明したように、緑を引数にした`glColor` 関数を呼んで指定します。

また、仮想３次元世界にユーザインターフェースの目的で各種のボタンやメニューを置く時、光源がたまたまそれらの裏側にあるからといって、そのボタンやメニューが暗くて見えなくなっては困ります。それらはインターフェースのために表示するだけで、リアルな外見にするため照明効果を計算する必要は別にないわけです。このような場合も照明処理は行わないステートの下でそれらのメニューを構成します。

最後に示したテクスチャマッピングとは、２次元の画像データを３次元物体の表面に張り付けて、少ない計算負荷でリアルな画像を実現する方法です。例えば、CompleXcope 内に “仮想和室” を作り出す場合を考えましょう。掛け軸や襖の輪郭を描画プリミティブで構成しなければならないのは当然ですが、その表面に描かれた絵柄までプリミティブで構成することは (もちろん可能ですが) 非常に大変ですし、それができても大量のポリゴンで構成されるため、そのレンダリングは恐ろしく時間がかかってしまうでしょう。こういうときには絵柄を２次元画像データ (テクスチャ) としてファイルに保存しておき、掛け軸や襖に “張り付け” てやるのです。これがテクスチャマッピングです[12]。光源がどこにあろうが、どこに視点があろうが、掛け軸や襖の面には張り付けられた画像が表示されるので、これも照明処理とは無関係です。従ってテクスチャマッピングも照明を行わないモードで行います[13]。テクスチャマッピングの実現方法は第 2.32章で解説します。

以上、大事なことは、

[12]掛け軸や襖の輪郭ポリゴンを動かすと、テクスチャも自動的にそれにくっついて移動します。

[13]CompleXcope ではテクスチャマッピングをメッセージ (“*Welcome to CompleXcope*” 等) やメニューの表示に利用します。

照明を使わないモードでは
- glColor 関数で、(テクスチャマッピングではテクスチャイメージで) 指定した色がそのまま表示される。
- 光源の位置や色、および (もし指定している場合でも) 法線ベクトルとは無関係である。

ということです。

2.12.2 照明モデル

さて、それでは照明処理について詳しく見てみましょう。照明処理を始める OpenGL 関数は、

```
glEnable(GL_LIGTHTING);
glEnable(GL_LIGTHT0);
```

です。`glEnable()` 関数はデプステスト開始の指定の時にも出てきました。この関数は、引数で指定した OpenGL の種々の機能を ON にするというものです。上の 1 行目の`glEnable()` は、照明を ON にするという指定です。

OpenGL では光源は複数指定することが可能です。光源には光源 0、光源 1、光源 2、... と、番号づけがされています。上の`glEnable(GL_LIGHT0)` という関数呼び出しは、 0 番の光源を使うという意味です。光源を二つ置く場合には、もう一行加えて

```
glEnable(GL_LIGTHTING);
glEnable(GL_LIGTHT0);
glEnable(GL_LIGTHT1);
```

とします。後で説明するように、プログラムの他の部分で、それぞれの光源の特性 (位置や色等) を指定しますが、光源はそれぞれ固有の名前 (`GL_LIGHT0` 等) で区別されます。

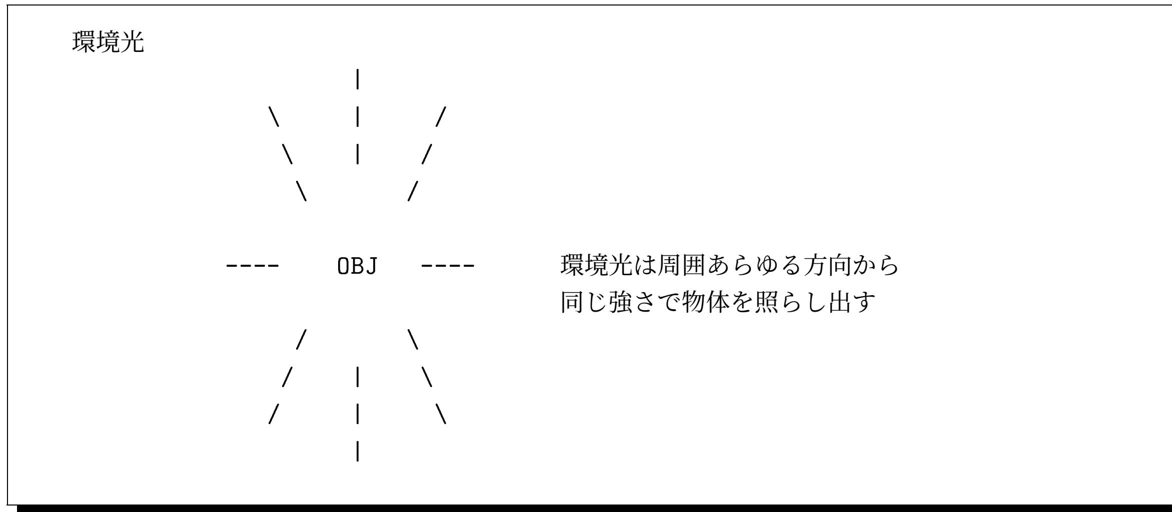
ここで現実世界での照明について考えてみましょう。例えば天井と机の上に蛍光灯があり、窓のない部屋を考えます。現実の光源は有限の大きさを持ち、そのスペクトルも複雑です。光源から出た光は部屋の壁、床、天井、様々な物体に何度も反射と吸収を繰り返して最後に人間の目に入ってきます。もちろん、壁、床、天井、物体がそれぞれ固有の (光の波長の関数としての) 吸収率と反射率を持っており、最終的に目に入ってくる光のスペクトルはそれら全てに依存しています。計算機でこのような過程を全て正確に再現するのはもちろん非常に大変なので、 OpenGL では照明を次の様にモデルします。

- 光源は点である
- 光源は RGB の 3 成分を独立に放射する
- 照明光は一度しか反射されない (視点に入る光路は一度しか折れない)
- 光源から視点に光が直接入射することはない
- 照明光には環境光、拡散光、鏡面光の独立な 3 種類があると考える (それぞれに RGB の 3 成分がある)
- 物体は、上記の 3 種類の光それぞれに、互いに独立な反射係数を持つ
- 物体に当たった 3 種類の光の各 RGB 成分が全て独立に反射される
- 光源が複数ある場合、それぞれに上の計算を行い、最後に上記 3 種類の光を足して RGB の値を決める

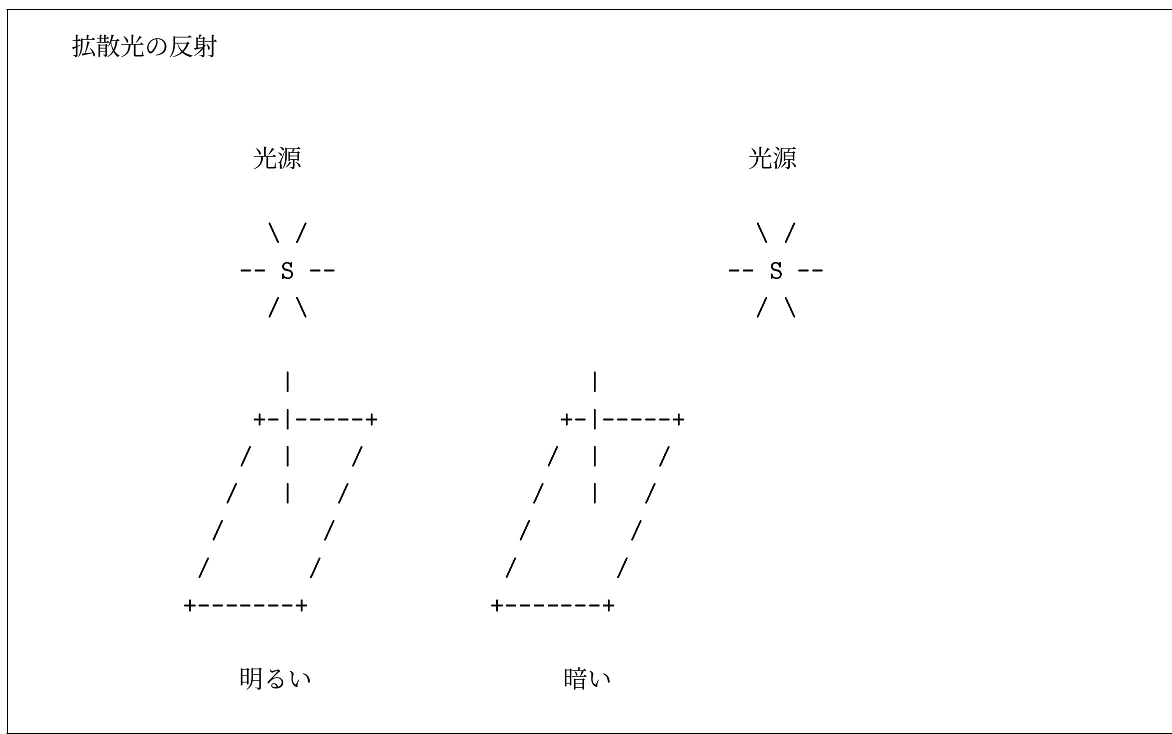
これは大変優れた照明モデルで、これほど簡単であるにもかかわらず、現実の映像とかなり近いリアルな CG が得られます。以下では上記の “3 つの光”、環境光、拡散光、鏡面光について説明しましょう。

2.12.3 環境光、拡散光、鏡面光

環境光 (ambient light) とは、部屋の中を何度も反射を繰り返し、飽和した光に対応します[14]。物体の周囲をあらゆる角度から照らし出す光だと言えます。従って環境光の場合、光源の位置というのは無意味で、ただ、その RGB 各成分の強さだけが重要です。物体の表面に当たった環境光はあらゆる方向に平等に反射されると考えます。つまりどこに視点を置いて見ても常に同じだけ環境光の反射光は目に入ってきます。



拡散光 (diffuse light) とは、環境光とは違い、光源の位置が重要になります。物体を構成するある面要素 (ポリゴン) に拡散光が当たる時、その面要素がどう見えるか (つまり拡散光がどう反射されるか) を見てみましょう。面要素の位置から見た光源の方向ベクトルを $\vec{s}$、面要素の法線ベクトルを $\vec{n}$ とします。ベクトル $\vec{s}$ と $\vec{n}$ が平行の時、その面要素は明るく光ります。これは面要素を真正面から照らし出していることに相当します (下図、左)。しかし、$\vec{s}$ と $\vec{n}$ のなす角度が大きくなると暗くなり (下図、右)、直角にすると全く光を反射しません[15]。

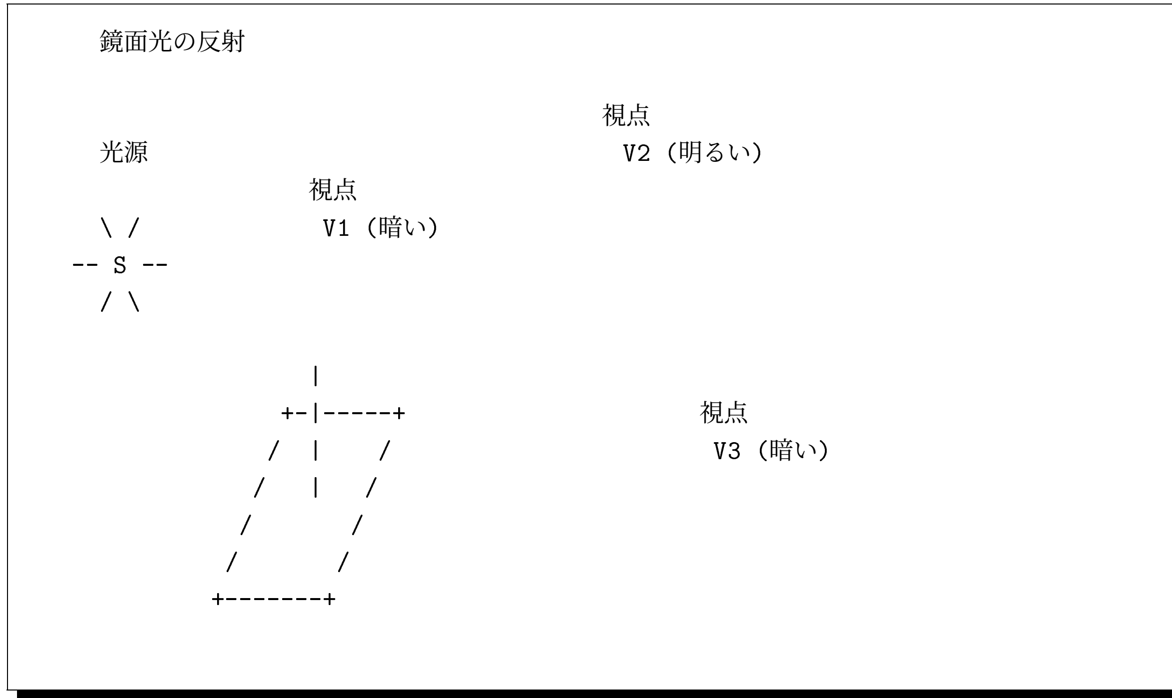


ここで物体に反射される拡散光の強さは、視点の位置 (方向) とは無関係であることにご注意下さい。

[14] 黒体輻射ですね。

[15] つまり面の裏から拡散光で照らされた場合、基本的には面の表側 (これは法線ベクトルの向きで定義されます) は見えません。

最後は鏡面光 (specular light) についてです。鏡面光は、金属や、表面のなめらかなプラスチック等に見られる光沢感を表現するのに用いられます。物体の表面が鏡に近い性質である場合にこのような光沢が見られます。鏡面光は光源から物体の表面に達した時に、そこで鏡面反射する方向には強く光が反射され、それ以外の方向にはほとんど反射されません。従って、鏡面光では視点の位置が重要になります。このことをベクトルで言い替えてみましょう。面要素の位置から見た光源の方向の単位ベクトルを $\vec{s}$、面要素の法線ベクトルを $\vec{n}$ とします。そして、面要素からみた視点の方向の単位ベクトルを $\vec{v}$ としましょう。視点に入る鏡面光は、$\vec{s}+\vec{v}$ が、$\vec{n}$ と平行となる場合は強く、それからずれるに従い弱くなっていきます。次の図を参照して下さい。



2.12.4 照明の使い方

環境光、拡散光、鏡面光の３成分の上手な使い分けがリアルな CG 作成のかぎになります。それでは、それぞれの光の成分が実際の CG でどのような効果を演出するのに関係するのか、説明しましょう。原点 $(0,0,0)$ に置かれた半径１の球を描く場合を考えてみます。宇宙空間に浮かんだ月を思い浮かべて下さい。今、視点 (地球) は $+z$ 方向の無限遠[16]にあるとします。光源を一つ、x 軸上の $+\infty$ に置きます (太陽)[17]。

まず、光源から出る光の拡散光成分と鏡面光成分は完全にゼロとし、(白色の) 環境光成分だけが放射されているものとしましょう。このとき、球はどのように見えるでしょうか？環境光によって球はあらゆる方向から照らし出されていますから、球の存在は確かに “見える” のですが、上で説明したように、環境光はあらゆる方向に光を反射しますので、視点から一番近い球面上の点 $(0,0,1)$ も、それ以外の点も全て同じ明るさで光ることになります。従って環境光だけで見た球は、立体感が全くなく、白色で塗りつぶされた２次元の円にしか見えないことになります。

では、今度は光源から拡散光だけが放射されている場合を考えてみましょう。拡散光は、その光線が光源から出発し、物体の表面に当たる時の面への入射角で明るさが決まります。従って、光源に対向している球上の点 $(1,0,0)$ が最も明るく見え、そこから離れるに従い、球面上の点は暗くなっていきます。そして光源とは反対側にある半球 $(x<0)$ は (各ポリゴンの裏面側に光源があるために) 全く光っていません。つまり真っ暗です (半月ですね)。サイエンティフィックビジュアリゼーションでは、(光源との相対位置が悪いために) 定義した３次元物体の一部が全く見えないようなこのような状況はあまり望ましいものではありません。そこで通常は拡散光に加えて、それよりは弱めの環境光を同時に放射させます。こうすることによって光源の反対側にある半球面もうっすらと光って見え、十分な立体感が得られます。

それでは最後に、光源から鏡面光だけが放射されている場合を考えましょう。この場合、球のわずかな一部分だけが “眩しく” 光って見えるでしょう。(パチンコ玉のような金属性の球を思い浮かべて下さい。) 既に述べた様に、

[16] OpenGL では視点を無限遠に置くことが可能です。
[17] OpenGL では光源を無限遠に置くことも可能です。

鏡面光は、物体の材質が金属的であることを演出したい場合に有効です。また、それだけでなく (むしろそれ以上に) 鏡面光の照明は、物体の立体感を増すという重要な効果も持っています。これは、メビウスの輪のような捻じれたリボン状の 3 次元物体を見る場合を想像してみれば分かります。鏡面光の存在によって、その表面の複数の (互いに離れた独立の) 場所がスポット状に明るく見えるでしょう。これは、(現在の視点から見て) そこでの面要素の法線ベクトルがちょうど鏡面反射条件を満たしているためです。それら複数のスポットを見る観測者は、直感的にその数学的関係を把握し、拡散光だけで見るよりも、より正確なリボンの形状把握が可能になります。

結局、サイエンティフックビジュアリゼーションにおいては (CompleXcope でも)、

> 拡散光と鏡面光を同じ強さで、環境光はそれよりも弱めにする

というのが、光源の設定に対する無難かつ基本的な処方箋と言えるでしょう。 OpenGL のデフォルトでは、 0 番の光源は

> 拡散光と鏡面光を強さ `1`(最大) の白色光、環境光は `0`

となっています。 1 番以上の光源については全て 0 がデフォルトです。

それではここで光源を一つ設定する具体的なコードを見てみましょう。デフォルトの光を使う場合が最も簡単で、必要なのは光源の位置の設定だけです。位置に限らず、光源に関する情報に設定には`glLightf()` または`glLightfv()` を使います。

```
GLfloat light_position[] = {10.0, 5.0, 3.0, 1.0}; /* 光源の位置 */

glLightfv(GL_LIGHT0, GL_POSITION, light_position);

glEnable(GL_LIGHTING);      /* 照明モード ON */
glEnable(GL_LIGHT0);        /* 光源 0 番を ON */

    .
    .
    .
```

座標データ (位置ベクトル `light_position`) には 4 成分あります。最初の 3 つが x, y, z 成分で、第 4 成分を 1.0 にすれば、この x, y, z の位置に光源が置かれます。一般的には、`glLightfv()` の引き数で指定される光源位置のベクトルを (lx, ly, lz, lw) とした時、光源が実際に置かれる位置は、 $(lx/lw, ly/lw, lz/lw)$ となります。従ってこれから予想されるように、

> 無限遠に光源が置くには光源位置の第 4 成分を `0.0` にする。

すると、他の 3 成分 (lx, ly, lz) で指定される方向の無限遠に光源が置かれます。従って OpenGL プログラムでは光源位置の第 4 成分は 0.0 か 1.0 のどちらかにセットするのが普通です。

さて、上で述べたように、デフォルトでは環境光成分は存在しません。全ての成分を設定するには次の様にします。

```
GLfloat light_position[] = {10.0, 5.0, 3.0, 0.0}; /* 無限遠 */
GLfloat light_diffuse[]  = {1.0, 1.0, 1.0, 0.0};  /* 拡散光 */
GLfloat light_ambient[]  = {0.3, 0.3, 0.3, 0.0};  /* 環境光 */
GLfloat light_specular[] = {0.8, 0.8, 0.8, 0.0};  /* 鏡面光 */


glLightfv(GL_LIGHT0, GL_POSITION, light_position); /* 位置   */
glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);  /* 拡散光 */
glLightfv(GL_LIGHT0, GL_SPECURAR, light_specular); /* 鏡面光 */
glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);  /* 環境光 */

glEnable(GL_LIGHTING);     /* 照明モード ON */
glEnable(GL_LIGHT0);       /* 光源０番を ON */

    .
    .
    .
```

2.13 材質の定義

以上の照明の説明から容易に想像できると思いますが、物体の色は、入射光の (R,G,B) のそれぞれに独立な反射係数を与えることで実現します。そして、

> 物体の材質は、環境光、拡散光、鏡面光のそれぞれに対する反射係数を関数
>
> `glMaterialfv()`
>
> を使って設定する

ことで表現します。

以下のコードを見れば、その意味は簡単に分かるでしょう。このプログラムは赤い金属的な質感の球 (1) と緑色の非金属的な質感の球 (2) を描くものです。

```
    .
    .
    .

GLfloat light_position[] = {10.0, 5.0, 3.0, 0.0};    /* 光源関係 */
GLfloat light_diffuse[]  = {1.0, 1.0, 1.0, 0.0};
GLfloat light_ambient[]  = {0.3, 0.3, 0.3, 0.0};
GLfloat light_specular[] = {0.8, 0.8, 0.8, 0.0};

GLfloat ball_1_diffuse[]  = {1.0, 0.0, 0.0, 0.0};    /* 球 1 */
GLfloat ball_1_ambient[]  = {1.0, 0.0, 0.0, 0.0};
GLfloat ball_1_specular[] = {0.8, 0.8, 0.8, 0.0};

GLfloat ball_2_diffuse[]  = {0.0, 1.0, 0.0, 0.0};    /* 球 2 */
```

```
    GLfloat ball_2_ambient[]  = {0.0, 1.0, 0.0, 0.0};
    GLfloat ball_2_specular[] = {0.0, 0.0, 0.0, 0.0};

    glLightfv(GL_LIGHT0, GL_POSITION, light_position);   /* 位置   */
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);    /* 拡散光 */
    glLightfv(GL_LIGHT0, GL_SPECURAR, light_specular);   /* 鏡面光 */
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);    /* 環境光 */

    glEnable(GL_LIGHTING);                               /* 照明モード ON */
    glEnable(GL_LIGHT0);                                 /* 光源 0 番を ON */
    glEnable(GL_DEPTH_TEST);                 /* デプステスト (隠面処理) ON */

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                                        /* バッファの消去 */

/* 球 1 の定義 */
    glMaterialfv(GL_FRONT, GL_DIFFUSE,  ball_1_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  ball_1_amnient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, ball_1_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 110.0);          /* 鏡面度の設定 */

    make_ball_1();                           /* 球 1 の頂点と法線を構成する */

/* 球 2 の定義 */
    glMaterialfv(GL_FRONT, GL_DIFFUSE,  ball_2_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  ball_2_amnient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, ball_2_specular);

    make_ball_2();

         .
         .
         .
```

球 1 の定義の中で GL_SHININESS の設定をするところがあります[18]。これは鏡面度と呼ばれ、 0.0 から 128.0 の値をもち、物体の表面が理想的な鏡に近いほど値が大きくなります。デフォルトは 0 ですので、

> 鏡面光を使う時には鏡面度を必ず設定する

ことを忘れないで下さい。

2.14 法線ベクトルと照明

以上述べたように、照明下での物体の見え方は、

[18] この部分だけ glMaterialfv() ではなく、glMaterialf() が使われていることに注意して下さい。もちろんこれは、ここで引き数として渡しているものが配列へのポインタではなく、実数値そのもの (鏡面度 110.0) であるためです。

(a) 光源から出ている全ての光成分（環境、拡散、鏡面)x(R,G,B)
(b) その時に設定されている物体の反射係数
(c) 視点の位置と各頂点に設定された法線ベクトル

をプログラマが指定すれば、後は OpenGL が自動的に計算してくれます。その自動的な計算というのは次の様なものです。物体を構成する多数のポリゴンのうち、一つのポリゴン (例えば３角形) に対して

(1) ３つの頂点について上記の (a)-(c) を使って RGB 値を計算する
(2) ３角形の内部の各点で、３つの頂点の RGB 値を補間する

従って３つの頂点の法線ベクトルを同じ向きに設定すると、この３角形は平面的に見え、一方、それぞれの法線を別の向きに設定すると、この３角形は曲面的に見えます。(３角形の内部の明るさと色が滑らかに変わっていくので。) これにより、少ないポリゴン数 (従って軽い計算) で滑らかな曲面の外観を表現することが可能となっています。

2.15 ここまでのまとめ

これまで、３次元物体の頂点と法線ベクトルの定義の仕方や、照明と隠面処理の指定方法等を説明してきました。OpenGL を使って３次元 CG プログラムを行なうにはあと二つ重要な項目を知る必要があります。一つは視点の設定や物体の移動・回転など、見ること（ビューイング）に関係すること、もう一つは OpenGL の一連の関数呼び出しをまとめて “マクロ化” することによって、プログラミングを効率化し、同時にレンダリング計算を高速化する方法（ディスプレイリスト）です。また動きのある画像 (アニメーション) を作る方法も紹介します。このガイドのこれ以降の部分では、出来るだけたくさんの完結したサンプルプログラムを使い、実践的な解説をしていこうと思います。これらのサンプルプログラムは、少し修正すれば、そのまま研究に使えるものを目指しました。 AVS 等ではやりにくいような細かい設定の CG も OpenGL を使えば可能です。ぜひこれらのプログラムを X Window 上で試してみて下さい。

第 2.1章で述べたように、このガイドではウインドウを開いたり、ウインドウにレンダリング結果を表示させたりする目的で aux ライブラリと呼ばれる補助ライブラリを利用しています。実は、 aux ライブラリには未だ解説していない便利な関数があり、今後のサンプルプログラムで使えるように、まずそれから紹介しましょう。

2.16 aux ライブラリによる様々な立体の表示

aux ライブラリには、ウインドウの初期設定 (auxInitDisplay　ode 等) や表示 (auxInitWindow) などの機能の他に、様々な３次元物体を一行で構成できるような便利な関数が用意されています。 OpenGL では３次元物体を構成するときには、全ての頂点の座標と法線ベクトルの向きを指定しなければならないことを思い出してください。例えば球[19]を作るためだけでも結構長いコードが必要です。従って、需要が高いと思われる多面体の構成コードが aux ライブラリに登録されています。その一つに球も用意されています。

球
```
auxWireSphere(GLdouble radius);
auxSolidSphere(GLdouble radius);
```

二つの関数は別々の機能を持ち、目的に応じてどちらかを利用します。上のWire の方は頂点どうしを線で結んだ表示（ワイヤフレーム）を行ない、下のSolid の方は法線ベクトルの指定された面で描くため、照光処理を行なう立体的な映像が欲しいときに用います。以下に挙げるその他の関数にもこのようにWire 版とSolid 版が存在します。

[19]もちろん球も多面体で近似します。

立方体
```
auxWireCube(GLdouble size);
auxSolidCube(GLdouble size);
```

直方体
```
auxWireBox(GLdouble width, GLdouble height, GLdouble depth);
auxSolidBox(GLdouble width, GLdouble height, GLdouble depth);
```

トーラス
```
auxWireTorus(GLdouble minorRadius, GLdouble majorRadius);
auxSolidTorus(GLdouble minorRadius, GLdouble majorRadius);
```

円柱
```
auxWireCylinder(GLdouble radius, GLdouble height);
auxSolidCylinder(GLdouble radius, GLdouble height);
```

円錐
```
auxWireCone(GLdouble radius);
auxSolidCone(GLdouble radius);
```

正四面体
```
auxWireTetrahedron(GLdouble radius);
auxSolidTetrahedron(GLdouble radius);
```

正八面体
```
auxWireOctahedron(GLdouble radius);
auxSolidOctahedron(GLdouble radius);
```

正十二面体
```
auxWireDodecahedron(GLdouble radius);
auxSolidDodecahedron(GLdouble radius);
```

正二十面体
```
auxWireIcosahedron(GLdouble radius);
auxSolidIcosahedron(GLdouble radius);
```

```
ティーポット
    auxWireTeapot(GLdouble size);
    auxSolidTeapot(GLdouble size);
```

それぞれの関数の引数の意味は自明でしょう。すべての物体は原点を中心にして構成されます。もちろんこれらの関数を使うときにはプログラムの最初に"aux.h"が include されている必要があります。

2.17 glu library による球、円筒、円盤の描画

球、円筒、円盤を描く時、上記の aux ライブラリを使えばとても手軽にこれらの物体を構成できます。一方その反面、 aux ライブラリの関数では、その物体表面の "滑らかさ" (例えば球を構成するポリゴンの数) が調節出来ないため、サイズを大きくして見た時に表面のポリゴンが見えてしまって気になる場合があります。そのような時には glu ライブラリを利用して下さい。 glu ライブラリにも球、円筒、円盤を描く関数が用意されていて、それらは物体を構成するポリゴンの数を指定することが出来ます。

glu ライブラリで球、円筒、円盤のどれかを描くときには必ず始めに

```
        GLUquadricObj* gluNewQuadric(void);
```

という関数を呼びます。この関数名に含まれる "Quadric" という言葉は、 glu library では球、円筒、円盤を次の 2 次式でモデルすることに由来します。

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yz + a_6zx + a_7x + a_8y + a_9z + a_{10} = 0$$

glu では物体のパラメータ (上の式の a_1 から a_{10}) や属性等をひとつの構造体にまとめます。gluNeqQuadric はその構造体を保存するのに必要なメモリ領域をアロケートし、そのアドレスを返します。その後で、例えば球を描く場合は

```
glu による球
     void gluSphere(GLUquadricObj *qobj, GLdouble radius,
                       GLinit slices, GLint stacks);
```

という関数を呼びます。第 1 引数にはgluNewQuadric でセットしたポインタを、第 2 引数には球の半径をいれます。第 3 引数と第 4 引数が球を構成するポリゴンの細かさを指定する部分です。 slices は経度方向の分割数、 stacks は緯度方向の分割数です。同様に、

```
glu による円筒
     void gluCylinder(GLUquadricObj *qobj, GLdouble baseRadius,
                      GLdouble topRadius, GLdouble height,
                      GLinit slices, GLint stacks);
```

```
glu による円盤
     void gluDisk(GLUquadricObj *qobj, GLdouble innerRadius,
               GLdouble outerRadius, GLinit slices, GLint stacks);
```

も用意されています。それぞれの引数の意味は推測できるでしょう。詳しくは *OpenGL Programming Gluid, Second Edition* [4], p.431 を参照して下さい。また、実際に`gluSphere`を使ったサンプルプログラムが第３章 (Xball.c; p.72) に出て来ます。

2.18 ビューイング

次にビューイングについて説明しましょう。ビューイング (viewing) とは、視点の位置と方向を決めたり、物体を平行移動や回転させたりして望みの映像を作り出す操作で、後でわかるように全て座標変換に関係する作業です。まず日常生活でのカメラによる写真撮影との類推から始めましょう。例えばコップを撮る場合、

- カメラを固定し　（視界設定）
- コップの位置と向きを決め　（モデリング）
- レンズを調節して倍率を決め　（射影）
- 焼き付ける写真の大きさを決める　（ビューポート設定）

という作業が必要です。それぞれの作業に対応する CG での呼び方が上の括弧の中です。 CG におけるこれらの作業全体をビューイングと呼びます。

これらビューイングの設定で、座標変換という数学概念がどのようにして出てくるのでしょうか？ CG では、コップを構成するポリゴンの各頂点の座標をプログラムで与える必要があります。コップが “平ら” に置かれた場合（例えば x–y 平面上のテーブルに置かれた場合）、各座標の指定は、基本的には z 軸に平行な円柱状のものを作るだけですから、それほど難しくないはずです。では、このコップが斜めに傾いて置かれる場合はどうすべきでしょうか？もちろんコップの傾きを考慮に入れて各ポリゴンの頂点の座標を数学的に指定することも (面倒ではありますが) 可能です。しかし、もっといい方法があります。座標系を回転させて、 x–y 面が傾いた新しい座標系を定義し、この新しい座標系のもとでそれからみて “平ら” にコップを置くのです。こうして座標の回転という変換が必要になります。次に、このコップがテーブルの上 x 軸上を速度 v で滑っていくアニメーションを OpenGL でプログラムすることを考えましょう。 (初期 $t = 0$ に原点にあるとして) 時刻 t の映像は、 $x = vt$ を中心にしてポリゴンの各頂点の位置を (glVertex3f 等により）設定するということを次々に行えばアニメーションになります。しかしもっと効率的な方法は、座標系を x 方向に vt だけ平行移動させ、新たな座標系を定義し、この新しい座標の原点にコップを描くのです。こうして座標の平行移動という変換も必要になります。そのほかに座標をあるスケールだけ伸ばしたり縮めたりする変換を使うこともあります。このように座標系を変換してから、新しい座標系のもとで物体を描くという操作の利点は、 “(現在の座標系のもとで) 原点に位置に平らに描く” という機能を持った関数一つだけ用意すれば済むということです。

ここで一つ面白いことがあります。それは

ビューイングは４次元の座標変換である

ということです。 OpenGL ではポリゴンの頂点や光源を設定するときに、その位置座標を (x, y, z, w) の４次元で指定したことを思い出して下さい。この４次元座標は同次座標と呼ばれます。同次座標を使う最大の理由は、頂点の移動、回転、射影が４行４列の行列の掛け算で実行できるためです。頂点の移動や回転だけならばもちろん３行３列の行列で表現できますが、４行４列にすると透視射影や正射影等の射影変換も行列の掛け算[20]で表現できてしまうことがポイントです。グラフィックワークステーションでは、このような４次元の行列計算を行なうハードウエアが用意されているため、レンダリングが高速に処理されるのです。

それではこの座標変換をもう少し詳しく見てみましょう。変換には４種類があり、次の順番に実行されます。

1．モデルビュー変換
2．射影変換
3．透視法除算
4．ビューポート変換

[20] 正確に言えば掛け算＋第４成分 w による割り算。

このうち１ (モデルビュー変換) と、２ (射影変換) は４行４列の行列の掛け算で実行されます。３の透視法除算とは同次座標の第４成分で他の成分を割る演算で、これは OpenGL が自動的に行うのでプログラマは何も考える必要はなりません。そして４のビューポート変換とはウインドウのうちどの部分に実際にフレームバッファの映像を表示させるかを指定するものです。通常はウインドウの利用できるピクセルをめいっぱいに使うので特殊な事情がない限り意識する必要はありません。結局、プログラマが理解しておく必要があるのは、上記の１と２のステップで、どちらも４行４列の行列に関係しており、そのぞれの行列は次のように呼ばれています。

```
モデルビュー変換  <----->  モデルビュー行列

射影変換  <----->  射影行列
```

ポリゴンの各頂点がどのようにして変換されるのかをもう少し詳しく見てみましょう。ある頂点 v(x,y,z,w) はまずモデルビュー行列 M で変換されます。

$$v' = Mv$$

次に v' は射影行列 P で変換されます。

$$v'' = Pv' = PMv$$

後はこの同次座標ベクトル $v''(x, y, z, x)$ の始めの３成分を第４成分で割り（透視法除算）、ビューポート変換と呼ばれる変換によってカラーバッファの各ピクセル位置に写像されて、最終的な CG 画像が得られます[21]。

行列の具体的なイメージを得るためにいくつか例を挙げてみましょう。ただし、後で示すように、プログラム中で行列の各成分を直接指定することは全くありませんので、これらを覚える必要はありません。単にイメージを持ってもらうためだけに示します。座標を (x, y, z) だけ平行移動させる行列は、当然

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

となります。z 軸の回りに角度 φ だけ回転する変換行列は

$$\begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

となります。この例では行列の第４成分の存在意義が全く分かりませんが、射影変換では第４成分が本質的に重要になります。例えば、後のサンプルプログラムで多用することになる`gluPerspective(f,aspect,zNear,zFarr)` という透視射影を行う関数を呼ぶと、

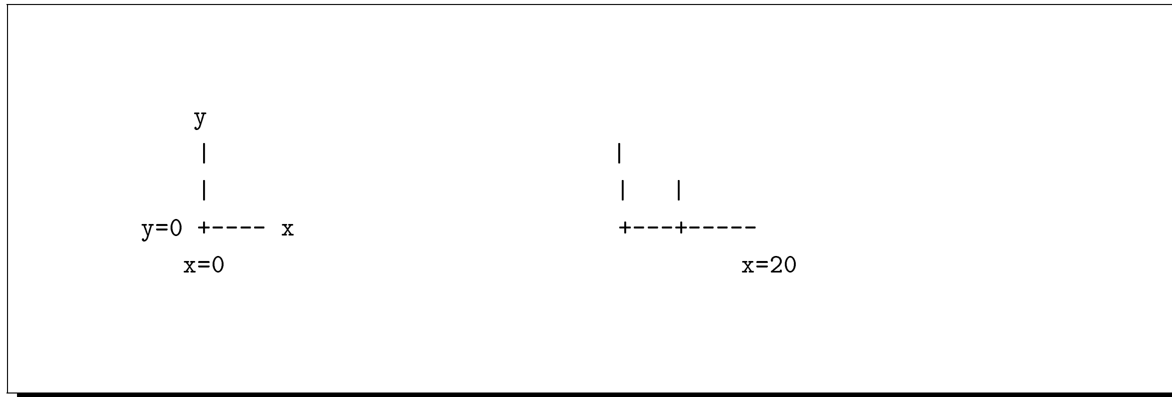
$$\begin{pmatrix} \frac{\cot(f/2)}{aspect} & 0 & 0 & 0 \\ 0 & \cot(f/2) & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2*zFar*zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

という行列が設定されます。

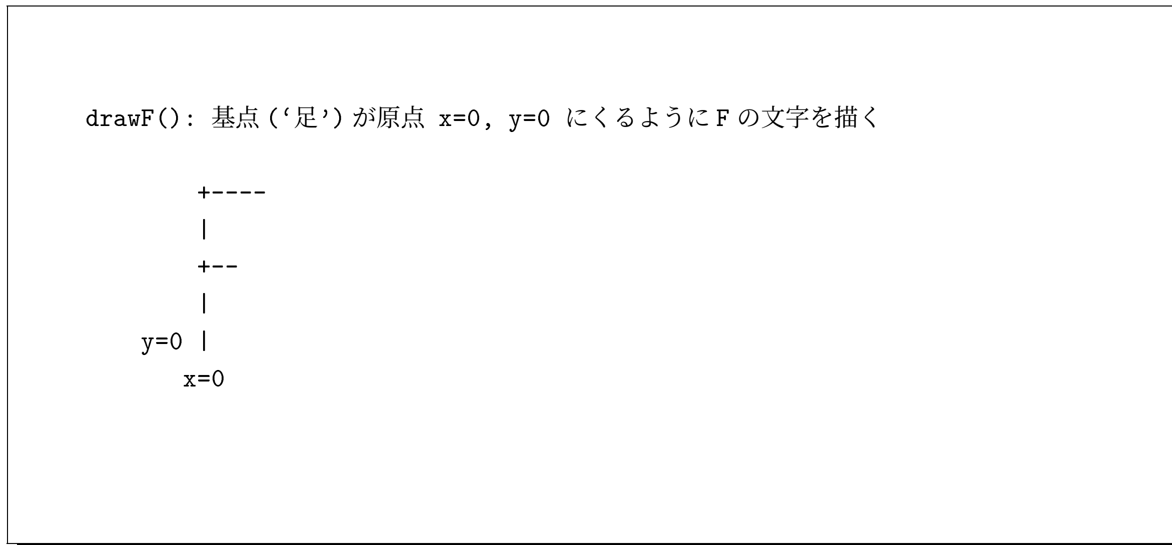
2.19 モデルビュー変換

モデルビュー変換についてもう少し詳しく説明しましょう。簡単のために二次元で考えます。 x–y 平面上に次のような倒れた “F” の文字を描くとしましょう。 F の “足” のところが、 x=20 の位置にあります。

[21]実は、 OpenGL のビューイングにはクリッピングという作業もあるのですがここでは説明を省略します。



これを描くためには、まず原点 (x=0) で、倒れていない普通の F の文字を描く関数を考えます。原点で、というのは F の “足” が原点にあるという意味です。いま、この関数を drawF() という名前にしましょう。



drawF() を使って我々の目的である「x=20 にある倒れた F」を描くには、二つのモデルビュー変換を続けて行う必要があります。一つは平行移動、そしてもう一つは回転です。ここでこの二つの変換を行う順番が重要になります。それは、回転してから平行移動して得られる図と、平行移動してから回転して得られた図が全く違うからです。このことは、行列の掛け算が非可換であることに対応しています。（モデルビュー変換は行列の掛け算で実現されることを思い出してください。）平行移動や回転という変換について考えるときには、(1) 座標系を変換するのか、それとも (2) 物体を変換するのか、この二つをはっきりと区別することが重要です。もちろんこの二つはコインの裏表のようなもので、同じ効果（画像）を得るために一連の座標変換をプログラムする時の考え方の違いに過ぎません。OpenGL では (1) の考え方を局所座標系での変換、(2) を固定座標系での変換と呼びます。ここでは局所座標系の変換で考えてみましょう。局所座標では、次のように考えます。

```
局所座標の考え方

  - 視点（カメラ）の位置、向きは動かない。
  - 変換コマンドは座標系に対して適用される。つまり、
  - 座標系を変換（平行移動、回転）し、新たな座標系 K' をつくる
  - これを局所座標系と呼ぶ。
  - 続けて変換を行うときには、また新たな局所座標系 K"をつくる。
  - 描画は、現在の局所座標系に基づいて行う。
```

座標系の平行移動を行うコマンド（関数）は、

```
glTranslatef(GLfloat x, GLfloat y, GLfloat z)
```

です。例えば、 glTranslatef(20.0, 0.0, 0.0) を呼んだとき、もともとの座標系 K の原点にあった局所座標系は x 方向に 20 だけ動き、新しい局所座標系 K' となります。

```
              K                                        K'
       y                                        y'
       |                                        |
       |                                        |
  y=0 +---- x                                   +---- x'
      x=0                                      x=20
```

座標系の回転は

```
glRotatef(GLfloat angle, GLfloat axis_y, GLfloat axis_y, GLfloat axis_z)
```

という関数で行います。引数の意味は容易に推測できるでしょう。例えば、 glRotatef(30.0, 0.0, 0.0, 1.0) は z 軸の回りに局所座標系を 30 度回転させます。その回転向きは、通常の左手系で自然に定義される向き、つまり x-y 面を上から見たときに反時計方向に回す向きです。（逆方向に回転させるには角度を -30.0、あるいは $360 - 30 = 330.0$ とします。）

K' 系を (K 系ではなく) を z' 軸のまわりに $+90°$ 回転させると

```
              K                                        K"
       y                                        x"
       |                                        |
       |                                        |
  y=0 +---- x                           y" -----+
      x=0                                      x=20
```

となります。この座標系を K”系と呼びましょう。

さて、それではこのセクションの始めに述べた「x=20 にある倒れた F」を描く問題に取り掛かりましょう。この場合、上の図で示した (1) K"系の原点を足として (2) K"系の y"軸方向にのびた、 F の字を画けばいいことが分かります。つまり K"系のもとで関数 drawF() を呼べばいいのです。まとめると、

1．局所座標系を x 方向に 20 だけ平行移動し、（座標系 K'）
2．局所座標系を z 軸の回りに +90 度回転させ、（座標系 K"）
3．この局所座標系 K"の原点でまっすぐに立った F を描く

となります。 OpenGL のプログラムでは

```
glPushMatrix();
    glTranslatef(20.0, 0.0, 0.0);
    glRotatef(90.0, 0.0, 0.0, 1.0);
    drawF();
glPopMatrix();
```

と書きます。glPushMatrix と glPopMatrix については後で解説します。ここで重要なことは、

局所座標の考え方でプログラムを書く時には座標の変換と物体
構成の順番通りに (上から下へ) 対応する関数を書いていけば良い

という点です[22]。

2.20 射影変換

射影変換の設定では構成した物体を見る視点の位置と向き、視野領域、及び射影方法を指定します。射影方法には、視点が無限遠にあることに相当する正射影と、透視法に基づいた (遠くの物が小さく見える) 透視法射影の二種類があります。

CompleXcope では射影変換についてはプログラマが考慮する必要はほとんどありません。なぜなら CompleXcope では射影計算の視点の位置は常に液晶シャッター眼鏡の位置で、射影方法は透視法射影と決まっているからです[23]。そして、 CompleXcope プログラムでは射影変換に関しては全て CAVE library が自動的に設定してくれるのです。そこでここでは射影変換について詳しくは解説しません[24]。

正射影には

```
void  gluOrtho(GLdouble left, GLdouble right,
               GLdouble top,  GLdouble bottom,
               GLdouble near, GLdouble far);
```

[22] 固定座標系の解釈でプログラムを書く場合には、関数は下から上に並べていかなければなりません。
[23] そうでなければ CompleXcope 内の体験者の見る映像にはリアリティがなくなってしまいます。
[24] それにサイエンティフィックビジュアリゼーションには、ここで紹介する二つの射影関数だけで充分です。

という関数を使います。6つの引数で決まる直方体の内部に入っている物体を z 軸方向に正射影します。視線の方向は $-z$ 方向を向いています。

透視法射影には

```
void  gluPerspective(GLdouble fovy, GLdouble aspect,
                     GLdouble zNear, GLdouble zFar);
```

という関数が便利です[25]。デフォルトでは視点は原点 (0,0,0)、視線は $-z$ 方向にあります。第1引数の fovy は上下方向の視野角です。海岸に立って水平線を見ているとして、水平線の上の fovy/2 度、下の fovy/2 度の中に入る物体だけが見えます ($0 \leq$ fovy ≤ 180.0)。第2引数の aspect は水平方向 / 垂直方向の視野角の比です。この二つの引数で原点 $(0, 0, 0)$ を頂点とする四角錐が定義されます。しかし、この四角錐の中に入っている物体全てを射影するわけではありません。視点 $(0, 0, 0)$ にあまり近すぎる物体や、遠すぎる物体は射影しないのです。これをクリッピングと呼びます。 第3引数と第4引数で z 軸に垂直な二つの面 (クリッピング面) を決めます。この二つは正の値で、視点から面までの距離を表します。この二つの面に挟まれた四角錐の領域だけが射影されるのです。

`gluPerspective` の fovy と aspect で決まる視角の中に入るように物体を置いても、それが例えば zFar で決まるクリッピング面の向こう側にあったら全く見えなくなってしまいます。だからといって zFar を必要以上に大きくしてもいけません。隠面処理で行われるデプステストはデプスバッファとよばれるバッファを利用して計算されます。もちろんこのデプスバッファの容量は有限です。デプステストは、 zFar と zNear で決まる二つのクリッピング面の間を深さ方向に分割 (量子化) してそれぞれの面の深さを判断するわけですが、その分割数は当然デプスバッファの容量で決まります。むやみに zFar を大きくすると、物体を構成する面どうしのデプステストが “甘く” なってしまい問題です。 zNear と zFar はそれぞれのそれぞれのプログラムで構成する物体の位置と大きさ応じて調節する必要があります。

2.21 サンプルプログラムのビューイングについて

上で説明したように CompleXcope プログラムではビューイングの関して必要なことは全て CAVE library が自動的にしてくれる上、 X Window で OpenGL の CG を表示させる場合でも、我々の目的であるサイエンティフィックビジュアリゼーションにはあまり複雑なビューイングは不必要である、という事情があるのでこの第2章で紹介する OpenGL のサンプルプログラムでは、 OpenGL が持つ多彩なビューイング機能の全てを利用することはせず、

- `gluPerspective` による透視法射影だけを使う。
- 視点は z 軸上、視線は -z 方向（デフォルト）に固定する。
- ビューポート変換はウインドウ全体への最も単純なものだけを行う。

という原則で示します。

最後のビューポート変換については、これまで解説してきませんでした。モデルビュー変換と射影変換を経てフレームバッファに書き込まれた画像データは最終的にはモニター画面のウインドウに表示させなければなりません。そのときウインドウのどの領域にどういう縦横比でフレームバッファの内容を表示させるかを決めるのがビューポート変換です。ウインドウ全体に画像の歪みがないように表示させるのが普通の方法ですし、我々の目的にはこれで充分ですので、ビューポート変換については詳しくは説明しません[26]。ビューポート変換の具体的な指定方法は以下で示すサンプルプログラムにある`glViewPort` 関数の部分を見て下さい。

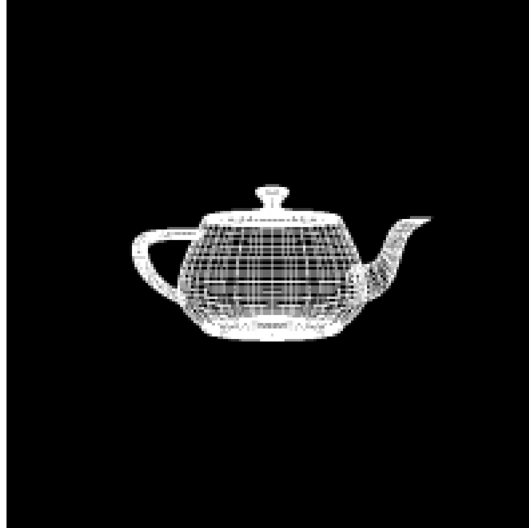
それではいくつかサンプルプログラムを見てみましょう。以下のサンプルプログラムではこれまで説明していない aux library 関数がいくつか出て来ますが、そのほとんどが Window の表示・制御に関係したものです。その機能はそれぞれの関数名から推測できると思いますが、詳しい解説は次のセクションで行います。ひとまずこれらの aux library 関数は X Window 上での OpenGL プログラムを書く場合の “決まり文句” だと考えて下さい。

[25]接頭子からわかるようにこれは glu library 関数です。正射影行列を設定する OpenGL 関数もあるのですが、引数の数が多く、使いにくいのでこの glu 関数を使う方が便利です。

[26]むしろサイエンティフィックビジュアリゼーションにとって歪みのある画像というのはあってはならないものです。

2.22 サンプルプログラム (平行移動１)

以下に示すサンプルプログラムは局所座標系を z 方向に -5 だけ平行移動させ、新しい座標系の原点に auxWireTeapot 関数 (p.31) を呼んでワイヤフレームのティーポットを描くものです。結果としてオリジナルの座標系の原点 $(0, 0.0)$ — そこに視点があります — から $-z$ 方向に 5 だけ遠ざかった点にティーポットは置かれます。OpenGL のデフォルトでは視線は $-z$ 方向に向いていることに注意してください。また、ティーポットがちゃんと見えるように`gluPerspective`でクリッピング面 zNear と zFar をそれぞれ 3.0 と 10.0 に指定しています。



wire_teapot.c

```
/*
 *   wire_teapot.c
\*
     An OpenGL sample program.
        - A wire teapot by the aux library.
        - No lighting.
        - Perspective view by the glu library.
        - No animation.
*/
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void display (void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        auxWireTeapot(1.0);              /* ワイヤーフレームのティーポット */
    glPopMatrix();
    glFlush();
}

void reshape(int width, int height) {
    GLdouble ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;
```

```
    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char** argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

\ end of wire_teapot.c /

2.23 aux library によるウインドウの制御

上のサンプルプログラム wire_teapot.c で出て来たウインドウ制御のための aux library 関数についてここで解説しましょう。

まず main 関数の最初に呼ばれているのは`auxInitDisplayMode` です。これはウインドウに表示するフレームバッファの種類と使用方法を指定するものです。引き数には数多くのパラメータを与えることが可能ですが、我々の目的には次の三通りだけしか使わないでしょう。

隠面処理を行わない時（ワイヤーフレームで描画する時など）

```
auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
```

隠面処理を行なう時（立体感のある CG を作るとき)。 [静止画]

```
auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH);
```

動きのある画像 (アニメーション[27]) を作る時には次のようにします。

隠面処理を行なう時（立体感のある CG を作るとき)。 [アニメーション]

```
auxInitDisplayMode(AUX_DOUBLE | AUX_RGB | AUX_DEPTH);
```

次に`auxInitPosition` が呼ばれています。この関数は、後で呼ぶ`auxInitWindow` で開くウインドウをモニター画面のなかのどの位置に置くかを指定します。

[27] アニメーションについては後 (第 2.29章) で詳しく説明します。

```
auxInitPosition(GLint x, GLint y, GLsize width, GLsize height)
```

ここで (x, y) はウインドウの左上のコーナーの位置 (ピクセル単位)。$(width, height)$ はウインドウのサイズ (ピクセル単位で測った幅と高さ) です。

そしていよいよ次の関数でウインドウを開きます。

```
auxInitWindow(GLbyte *titlestring)
```

引き数にはウンドウのタイトル文字列を入れます。

次の関数は、開いたウインドウに対してマウスを使ってサイズの変更や移動を行ったときに常に呼び出される関数を指定するものです。

```
auxReshapeFunc(void (*function)(GLsizei, GLsizei))
```

auxReshapeFuc の引数は関数 function へのポインタです。 function は二つの引数をもち、その引数には自動的にリサイズ後のウインドウの幅と高さが渡されます。

そして最後に

```
auxMainLoop(void (*displayFunc)(void))
```

が呼ばれています。引数にウインドウの更新 (物体の再描画) が必要な場合に呼ばれる関数 displayFunc を指定します。

2.24 サンプルプログラム (平行移動２)

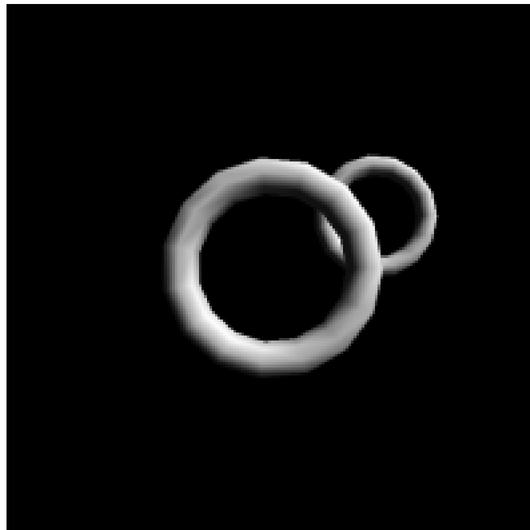
次にワイヤーフレームではなく、照明と隠面処理も入れた立体感のあるプログラムを見てみましょう。

torus.c

```
/*
 *   torus.c
\*
     An OpenGL sample program.
      - A torus by the aux library.
      - With the lighting.
      - Perspective view by the glu library.
      - No animation.
*/

#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void initial(void) {
    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void display (void) {
    GLfloat mat_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        auxSolidTorus(0.2,1.0);                       /* トーラス */
    glPopMatrix();

    glFlush();
}


void reshape(int width, int height) {
```

```
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

\ end of torus.c /



次のプログラムは二つのトーラスを描きます。glTranslate を 2 回続けて呼ぶことにより、二つ目のトーラスを描く直前、局所座標系の原点は $(2, 1, -9)$ に移動しています。この座標系の原点で新たに描かれた二つ目のトーラスが一つ目のトーラスの向こう側に見えます。

torus2.c

```
/*
 *   torus2.c
\*
     An OpenGL sample program.
        - Two tori (see torus.c).
        - No animation.
*/
```

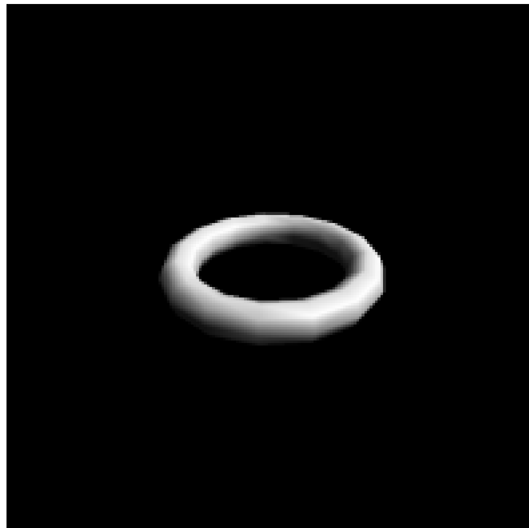
```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void initial(void) {
    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void display (void) {
    GLfloat mat_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        auxSolidTorus(0.2,1.0);
        glTranslatef (2.0, 1.0, -4.0);
        auxSolidTorus(0.2,1.0);
    glPopMatrix();

    glFlush();
}


void reshape(int width, int height) {
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
```

```
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

\ end of torus2.c /

2.25 サンプルプログラム (回転変換)

物体 (あるいは局所座標系) の回転にはglRotate を使います。



torus_rotated.c

torus.c 中、display 関数の中のglPushMatrix からglPopMatrix の部分を次のように変更します。

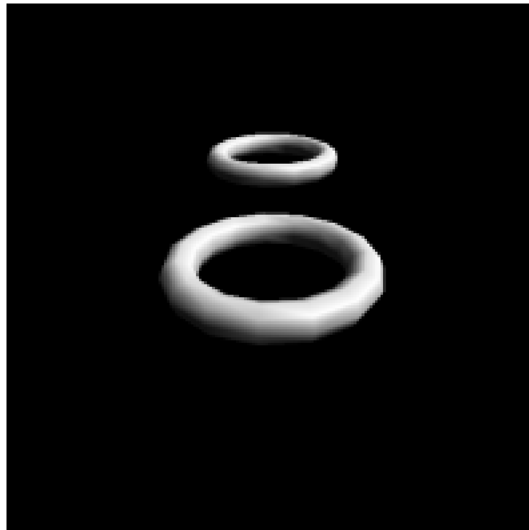
```
        .
        .
    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        glRotatef (-60.0, 1.0, 0.0, 0.0);
        auxSolidTorus(0.2,1.0);
    glPopMatrix();
        .
        .
```

この時glTranslatef と glRotatef を呼ぶ順番に注意して下さい。これを逆に呼ぶと、全く違う画像になってしまいます。

$$(回転変換行列)\times(平行移動行列) \neq (平行移動行列)\times(回転変換行列)$$

だからです。

2.26 サンプルプログラム (平行移動と回転の組合せ)



次のサンプルプログラムでは移動と回転が連続してコールされています。

torus2_rotated.c

`torus.c` 中、`display` 関数の中の`glPushMatrix` から`glPopMatrix` の部分を次のように変更します。

```
    .
    .
    .
glPushMatrix();
    glTranslatef (0.0, 0.0, -5.0);
    glRotatef(-60.0, 1.0, 0.0, 0.0);
    auxSolidTorus(0.2,1.0);
    glTranslatef (0.0, 4.0, 0.0);
    auxSolidTorus(0.2,1.0);
glPopMatrix();
    .
    .
    .
```

局所座標系の考え方ではこのプログラムは、

1. まず (局所) 座標系が $-z$ 方向に移動し、
2. この新しい座標系のもとで x 軸の回りに -60 度座標系が回転され、
3. この座標系のもとで一つ目のトーラスが描かれ、
4. さらに座標系が y 軸方向に 4 だけ移動し、
5. この新しい座標系のもとで二つ目のトーラスが描かれる。

と読みます。

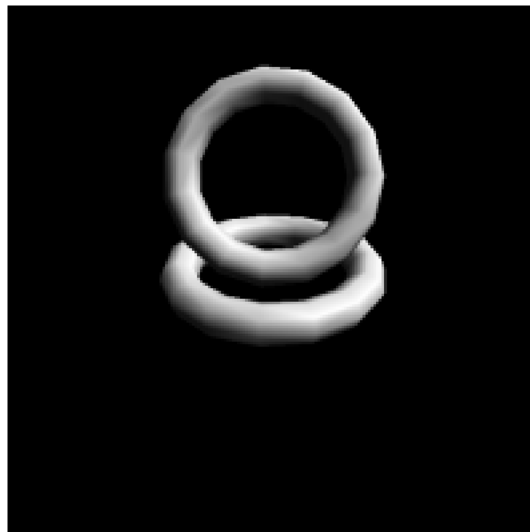
2.27 行列のスタック

実は OpenGL ではモデルビュー変換の行列 (モデルビュー行列) はスタックに保存されています。実行される変換は常にそのスタックの一番上のある行列です。そして、これまで解説せずに使ってきた`glPushMatrix` と`glPopMatix()` はそれぞれスタックへのプッシュとポップを行う関数です。

スタックは計算機の世界ではよく使われる[28]おなじみの考え方です。モデルビュー変換の計算に利用される行列を current matrix と呼びます。この current matrix はスタックの一番上に置かれている行列です。glPush atrix を呼ぶことにより行列を push すると、その時の current matrix をコピーしてスタックの一番上にそのコピーを置きます。この push の後に、座標の平行移動や回転などの操作があれば current matrix はそれぞれの変換に対応する行列の掛け算の結果、値が変わっていきます。そして glPop atrix を呼んで行列を pop すると、スタックの一番上の行列が捨てられて、二番目にある行列が一番上になり、これが current matrix となります。この行列は少し前に glPush atrix を呼んだ時の current matrix そのものですから、そのときの射影行列の状態に戻ったわけです。行列スタックを上手く使うと、複数の要素の位置関係が階層構造で記述されるような複雑な物体の記述が簡単にできます。

ところで、実は OpenGL には行列のスタックが 3 種類あります。一つは上に述べたモデルビュー行列のスタック、もう一つは射影行列のスタック、そして最後がテクスチャマトリックス (これについてはこのガイドでは全く触れません) です。 glPush atrix と glPop atrix はこの 3 種類の行列スタックそれぞれに対して適用することが可能です。 OpenGL のプログラム中で、 glPush atrix や glPop atrix が呼ばれた時、どの行列スタックに対して適用されるのかは “マトリックスモード” 呼ばれる OpenGL のステートの一つで制御されます。デフォルトのマトリックスモードはモデルビュー行列なので何も指定せずに glPush atrix や glPop atrix を呼べばその対象はモデルビュー行列ということになります。既に説明したように、我々の目的には複雑な射影変換は不必要ですし、テクスチャマトリックスを直接使うこともありません。従って、このガイドのサンプルプログラムでは、マトリックスモードを変更することはありません[29]。サンプルプログラムに出て来る行列スタックの push と pop は常にモデルビューマトリックスのスタックに対して行われます。

スタックの機能は、すぐ上で紹介したプログラム torus2_rotated.c とプログラム、及びそれぞれに対応する図を見比べればすぐに分かるでしょう。



torus2_rotated2.c

torus.c 中、display 関数の中のglPushMatrix からglPopMatrix の部分を次のように変更します。

```
      .
      .
   glTranslatef (0.0, 0.0, -5.0);
   glPushMatrix();
       glRotatef(-60.0, 1.0, 0.0, 0.0);
       auxSolidTorus(0.2,1.0);
   glPopMatrix();
   glPushMatrix();
       glTranslatef (0.0, 1.0,  0.0);
```

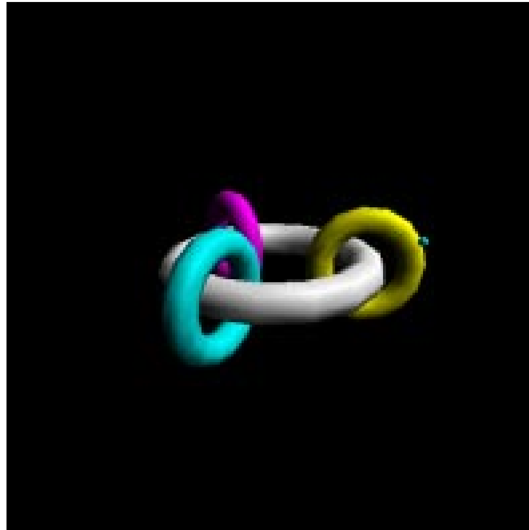
[28]例えばポストスクリプトや UNIX シェルのディレクトリ移動等。

[29]マトリックスモードを変更する関数は glMatrixMode 関数です。

```
        auxSolidTorus(0.2,1.0);
    glPopMatrix();
        .
        .
```

2.28 サンプルプログラム (スタックの使い方)

スタックを使えば複数の物体の組合せで構成される複雑な 3 次元物体も容易に定義できます。次のプログラムは図のような 3 つのトーラスと一つの小さな球を描くものです。行列の push と pop を繰り返すことにより、局所座標系の原点を階層的に移動させて物体を組み合わせる様子が理解できると思います。



link.c

```
/*
 *   link.c
\*
     An OpenGL sample program.
        - Three linked tori (with a ball).
        - No animation.
*/

#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void initial(void) {
    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
```

```
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void set_mat_color(GLfloat *diffuse, GLfloat *ambient, GLfloat *specular) {
        glMaterialfv(GL_FRONT, GL_DIFFUSE,  diffuse);
        glMaterialfv(GL_FRONT, GL_AMBIENT,  ambient);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
}

void display (void) {
    GLfloat diffuse0[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat ambient0[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat specular0[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat diffuse1[] = { 1.0, 1.0, 0.0, 1.0 };
    GLfloat ambient1[] = { 0.1, 0.1, 0.0, 1.0 };
    GLfloat specular1[] = { 0.9, 0.9, 0.0, 1.0 };
    GLfloat diffuse2[] = { 1.0, 0.0, 1.0, 1.0 };
    GLfloat ambient2[] = { 0.1, 0.0, 0.1, 1.0 };
    GLfloat specular2[] = { 0.9, 0.0, 0.9, 1.0 };
    GLfloat diffuse3[] = { 0.0, 1.0, 1.0, 1.0 };
    GLfloat ambient3[] = { 0.0, 0.1, 0.1, 1.0 };
    GLfloat specular3[] = { 0.0, 0.9, 0.9, 1.0 };

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glTranslatef(0.0, 0.0, -5.0);
    glRotatef(-70.0, 1.0, 0.0, 0.0);
    glPushMatrix();
        set_mat_color(diffuse0, ambient0, specular0);
        auxSolidTorus(0.2,1.0);
        glPushMatrix();
            glTranslatef(1.0, 0.0, 0.0);
            glRotatef(90.0, 1.0, 0.0, 0.0);
            set_mat_color(diffuse1, ambient1, specular1);
            auxSolidTorus(0.15,0.5);
            glPushMatrix();
                glRotatef(30.0, 0.0, 0.0, 1.0);
                glTranslatef(0.5, 0.0, 0.0);
                glRotatef(-50.0, 0.0, 1.0, 0.0);
                glTranslatef(0.2, 0.0, 0.0);
                set_mat_color(diffuse3, ambient3, specular3);
                auxSolidSphere(0.05);
            glPopMatrix();
        glPopMatrix();
        glPushMatrix();
            glRotatef(120.0, 0.0, 0.0, 1.0);
            glTranslatef(1.0, 0.0, 0.0);
            glRotatef(90.0, 1.0, 0.0, 0.0);
            set_mat_color(diffuse2, ambient2, specular2);
```

```
            auxSolidTorus(0.15,0.5);
        glPopMatrix();
        glPushMatrix();
            glRotatef(240.0, 0.0, 0.0, 1.0);
            glTranslatef(1.0, 0.0, 0.0);
            glRotatef(90.0, 1.0, 0.0, 0.0);
            set_mat_color(diffuse3, ambient3, specular3);
            auxSolidTorus(0.15,0.5);
        glPopMatrix();
    glPopMatrix();

    glFlush();
}


void reshape(int width, int height) {
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

\ end of link.c /

2.29 アニメーション

アニメーションとは動きのある画像 (動画) です。CG ではダブルバッファを利用してアニメーションを実現します。ダブルバッファとは、ウインドウに表示すべき画像が保存される場所であるカラーバッファが 2 枚用意されていることを意味します。一方のバッファ (これをバッファ A と呼びましょう) に OpenGL でレンダリングされた結果が書き込まれている間に、もう一方のバッファ (バッファ B とします) の内容がモニタに表示されています。そしてバッファ A への書き込みが終了した後、モニタ表示用のバッファを B から A に切替えて、新しい画像 (バッファ A) がモニタに表示されます。そして今度はバッファ B にレンダリング結果を書き込始め、それが終了すると表示バッファを B に切替えます。

OpenGL でアニメーションをつくるのはとても簡単です。プログラムはこれまでに紹介した静止画のためのプログラムを機械的に変更するだけで済みます。変更すべき箇所は、

1. main 関数の auxInitDisplayMode() で AUX_DOUBLE を宣言する
2. main 関数で auxIdleFunc() を追加する
3. auxIdleFunc() に引数で渡される関数と auxMainLoop() に渡される関数の違いに注意する
4. glFlush() のかわりに glXSwapBuffers() をコールする

以上です。その名前から推測できるように、glXSwapBuffers が表示バッファを切替える関数です。glX ライブラリの仕様によれば glXSwapBuffers はその内部で glFlush を呼んでいるので glFlush と glXSwapBuffers を続けてコールする必要はありません。

それではサンプルプログラムを見てみましょう。

このプログラムでは、二つの回転の単純な組合せによりトーラスが面白い動きを見せます。

swing_torus.c

```
/*
 *   swing_torus.c
\*
     An OpenGL sample program.
        - Animation.
        - A rotating torus (see torus.c).
*/

#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

static GLdouble spin=0.0;

void initial(void) {
    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
```

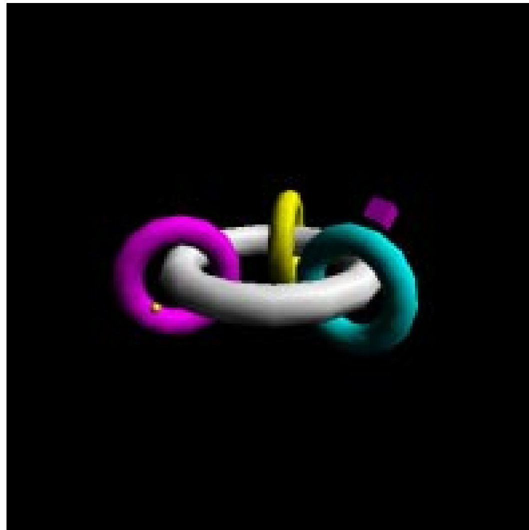
```
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
}

void display (void) {
    GLfloat mat_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        glRotatef (spin, 1.0, 0.0, 0.0);
        glRotatef (spin, 0.0, 1.0, 1.0);
        auxSolidTorus(0.2,1.0);
    glPopMatrix();

    glXSwapBuffers(auxXDisplay(), auxXWindow());
}

void rotation(void) {
    spin += 1.5;
    if (spin > 360.0) spin -= 360.0;
    display();
}

void reshape(int width, int height) {
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGBA | AUX_DEPTH);
```

```
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxIdleFunc(display);
    auxMainLoop(rotation);
}
```

\ end of swing_torus.c /

2.30 アニメーション2

アニメーションのサンプルプログラムをもう一つ見てみましょう。



このプログラムは link.c をもとにしてトーラスや球 (2 つ) と立方体 (一つ) が動くようにしたものです。ちょっとした変更で、簡単に面白いアニメーションが作れるのが分かるでしょう。

link_anime.c

```
/*
 *   link_anmime.c
\*
     An OpenGL sample program.
        - Animation.
*/

#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

static GLdouble toroidal_spin = 0.0;
static GLdouble ball1_spin = 0.0;
static GLdouble ball2_spin = 0.0;
static GLdouble ball3_spin = 0.0;

void initial(void) {
```

```
    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void set_mat_color(GLfloat *diffuse, GLfloat *ambient, GLfloat *specular) {
        glMaterialfv(GL_FRONT, GL_DIFFUSE,  diffuse);
        glMaterialfv(GL_FRONT, GL_AMBIENT,  ambient);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
}

void display (void) {
    GLfloat diffuse0[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat ambient0[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat specular0[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat diffuse1[] = { 1.0, 1.0, 0.0, 1.0 };
    GLfloat ambient1[] = { 0.1, 0.1, 0.0, 1.0 };
    GLfloat specular1[] = { 0.9, 0.9, 0.0, 1.0 };
    GLfloat diffuse2[] = { 1.0, 0.0, 1.0, 1.0 };
    GLfloat ambient2[] = { 0.1, 0.0, 0.1, 1.0 };
    GLfloat specular2[] = { 0.9, 0.0, 0.9, 1.0 };
    GLfloat diffuse3[] = { 0.0, 1.0, 1.0, 1.0 };
    GLfloat ambient3[] = { 0.0, 0.1, 0.1, 1.0 };
    GLfloat specular3[] = { 0.0, 0.9, 0.9, 1.0 };

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glPushMatrix();
      glTranslatef(0.0, 0.0, -5.0);
      glRotatef(-70.0, 1.0, 0.0, 0.0);
      glRotatef(toroidal_spin, 0.0, 0.0, 1.0);
      glPushMatrix();
        set_mat_color(diffuse0, ambient0, specular0);
        auxSolidTorus(0.2,1.0);
        glPushMatrix();
          glTranslatef(1.0, 0.0, 0.0);
          glRotatef(90.0, 1.0, 0.0, 0.0);
          set_mat_color(diffuse1, ambient1, specular1);
          auxSolidTorus(0.15,0.5);
          glPushMatrix();
```

```
          glRotatef(ball1_spin, 0.0, 0.0, 1.0);
          glTranslatef(0.5, 0.0, 0.0);
          glRotatef(ball1_spin*5, 0.0, 1.0, 0.0);
          glTranslatef(0.18, 0.0, 0.0);
          set_mat_color(diffuse3, ambient3, specular3);
          auxSolidSphere(0.05);
        glPopMatrix();
      glPopMatrix();
      glPushMatrix();
        glRotatef(120.0, 0.0, 0.0, 1.0);
        glTranslatef(1.0, 0.0, 0.0);
        glRotatef(90.0, 1.0, 0.0, 0.0);
        set_mat_color(diffuse2, ambient2, specular2);
        auxSolidTorus(0.15,0.5);
        glPushMatrix();
          glRotatef(ball2_spin, 0.0, 0.0, 1.0);
          glTranslatef(0.5, 0.0, 0.0);
          glRotatef(ball2_spin*5, 0.0, 1.0, 0.0);
          glTranslatef(0.18, 0.0, 0.0);
          set_mat_color(diffuse1, ambient1, specular1);
          auxSolidSphere(0.05);
        glPopMatrix();
      glPopMatrix();
      glPushMatrix();
        glRotatef(240.0, 0.0, 0.0, 1.0);
        glTranslatef(1.0, 0.0, 0.0);
        glRotatef(90.0, 1.0, 0.0, 0.0);
        set_mat_color(diffuse3, ambient3, specular3);
        auxSolidTorus(0.15,0.5);
        glPushMatrix();
          glRotatef(ball3_spin, 0.0, 0.0, 1.0);
          glTranslatef(0.5, 0.0, 0.0);
          glRotatef(ball3_spin*5, 0.0, 1.0, 0.0);
          glTranslatef(0.30, 0.0, 0.0);
          set_mat_color(diffuse2, ambient2, specular2);
          auxSolidCube(0.20);
        glPopMatrix();
      glPopMatrix();
    glPopMatrix();
  glPopMatrix();

  glXSwapBuffers(auxXDisplay(), auxXWindow());
}


void rotation(void) {
  toroidal_spin += 1.0;
  ball1_spin += 8.0;
  ball2_spin += 4.0;
  ball3_spin += 1.0;
  display();
```

```
    }

void reshape(int width, int height) {
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGB | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxIdleFunc(display);
    auxMainLoop(rotation);
}
```

\ end of link_anime.c /

2.31 ディスプレイリスト

ディスプレイリストとは複数の OpenGL 関数の呼び出しをまとめて "マクロ" 化し、その実行計算を効率化するものです。ただし、ディスプレイリストを説明するのにマクロという言葉は適当ではないかもしれません。ディスプレイリストは、複数の OpenGL 関数を単にまとめただけのものではなく、(つまり呼び出し時に単に展開されるのではなく) 定義時に "コンパイル" されるのです。たとえばモデルビュー行列の設定 (`glTranslatef` 等の呼び出し) や、射影行列の設定 (`gluPerspective` の呼び出し) をすると 4 行 4 列の行列、及びその逆行列の値の計算が実行されます。複数の回転や移動が続くとそのかけ算が計算されます。これと全く同じ計算を何度も行うことは当然非効率的です。ディスプレイリストを定義すると、一連の演算の最終結果の値だけが保存されるので、その実行は高速になります。照明や材質の設定なども計算負荷が高いので、プログラム中でこれらのモードを何度も切替える必要がある場合もディスプレイリストを利用すべきです。静止画像では、プログラムの簡潔化のためにディスプレイリストを使う場合が多いでしょう。一方、アニメーションプログラムでは、レンダリング速度の向上のためにディスプレイリストの使用が不可欠となります。ディスプレイリストは CompleXcope プログラミングでも自然に多用することになります。

ディスプレイリストで実行が効率化される例の一つとして p.15で紹介した六角形を描くプログラム simple3.c を考えてみましょう。その中心部分は六角形を定義する

```
glBegin(GL_POINTS);
    for (i=0; i<6; i++) {
        x = cos(i*pi/3); y = sin(i*pi/3);
        glVertex3f(x, y, 0.0);
    }
glEnd();
```

のところです。六角形を一つ描くだけならばこのままでも問題はありませんが、同じ六角形を異なる場所に多数描く場合、上のプログラム部分を (サブルーチン化して) 六角形の数だけコールするのは考えものです。プログラムの実行時に、六角形の数だけ三角関数の計算が繰り返されることになるからです。このような場合にディスプレイリストを利用すると大変高速にプログラムが実行されます。

ディスプレイリストでは六角形を描く上の OpenGL プログラムの一群を一つの単位として登録し、非負整数のラベル (指標番号) をつけます。その後、登録した指標番号を呼び出すことだけで、一連の OpenGL プログラムがまとめて実行されます。その際、三角関数の計算は、その計算結果の数値データだけを保持しているので、登録したディスプレイリストを呼び出すたびに三角関数の計算が行われるようなことはありません。一般にディスプレイリストでは、一まとめの OpenGL 関数の実行を常に最適化するように設計されているので、ディスプレイリストを利用することにより、利用しない場合よりも実行速度が遅くなることはありません。

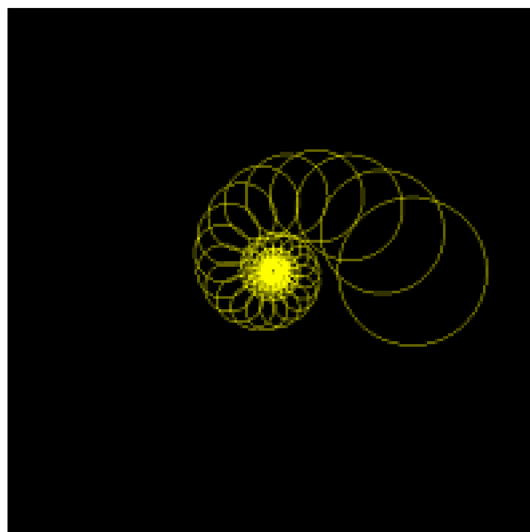
ディスプレイリストを定義するには、以下の様に`glNewList` と`glEndList` でマクロ化したい一連の OpenGL 関数を挟みます。

```
GLuint int list_no = 1;

glNewList(list_no, GL_COMPILE);
        .
        .
        .
        .
glEndList();
```

これで 1 番のディスプレイリストが定義されます。あとはプログラム中の適当な位置で

```
glCallList(1);
```

と呼べば 1 番のディスプレイリストが実行されます。同様に 2 番、3 番... のディスプレイリストも定義、実行が可能です。一般には、ディスプレイリストを利用すべき場面として (1) プログラム中に何度も利用される物体の定義、(2) 行列操作、(3) 照明処理、(4) テクスチャの定義、等があります。それでは、サンプルプログラムを見てみましょう。



このプログラムでは同じ半径の円をたくさん描いています。その位置はすこしずつずれていて、原点は螺旋の上にのっています。透視法射影のため遠くの円が小さく見えます。このようなプログラムをディスプレイリストを使わずに書けば、プログラムは複雑になり、また実行速度が大変遅くなってしまいます。

circles.c

```
/*
 *   circles.c
\*
     An OpenGL sample program.
        - No Lighting.
        - Display List.
*/

#include <math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

#define PI    3.14159265358979
#define TWOPI (2*PI)

GLuint list1 = 1;

void initial(void) {
    int i;
    GLfloat x,y;

    glNewList(list1, GL_COMPILE);
      glColor3f(1.0, 1.0, 0.0);
      glTranslatef(1.5, 0.0, 0.0);
      glBegin(GL_LINE_LOOP);
          for (i=0; i<100; i++) {
            x = 0.8*cos(TWOPI*i/100.0);
            y = 0.8*sin(TWOPI*i/100);
            glVertex3f(x,y,0.0);
          }
      glEnd();
      glTranslatef(-1.5, 0.0, 0.0);
      glRotatef(20.0, 0.0, 0.0, 1.0);
      glTranslatef(0.0, 0.0, -1.0);
    glEndList();

    glShadeModel(GL_FLAT);
}

void display (void) {
    int i;
    glClear(GL_COLOR_BUFFER_BIT);
    glTranslatef (0.0, 0.0, -5.0);
    glPushMatrix();
      for (i=0; i<200; i++)
          glCallList(list1);
    glPopMatrix();
    glFlush();
}
```

```
void reshape(int width, int height) {
    GLdouble ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 1000.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char** argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

\ end of circles.c /

ディスプレイリストの指標番号には非負整数を使います。この番号は上のサンプルプログラムのようにプログラマが自分で決めることが可能ですが、大規模な OpenGL プログラムにおいて多数のディスプレイリストを使用する場合、重複した指標番号を選んでしまう危険があります。そこで OpenGL には未使用の指標番号を自動的に生成してくれる関数が用意されています。

未使用の指標番号を一つ生成する

```
GLuint glGenLists(1);
```

この関数の返値が未使用の指標番号です。未使用の指標番号が一度に n 個欲しい時には

未使用の指標番号を n 個生成する

```
GLuint glGenLists(n);
```

とすると、未使用の指標番号を n 個確保します。その返値 ℓ から $\ell+1, \ell+2, \cdots, \ell+n-1$ までを指標番号として利用することができます。

2.32 テクスチャマッピング

テクスチャマッピングとは、p.24で説明したように、２次元の画像を３次元物体の表面に張り付けることです。OpenGL のテクスチャマッピングでは、複雑な多角形や平らではない物体の表面に画像 (テクスチャ) を張り付けることも可能です。フライトシミュレータや景観シミュレーションのようにリアルな外観が重要な CG では、風景や建

物の壁面模様の表現などにテクスチャマッピングが多用されます。グラフィックワークステーションではテクスチャマッピングのレンダリングを高速に処理するハードウエアが用意されています。

テクスチャのデータフォーマットは何種類かありますが、最も簡単なのは、 RGB の次のような配列

```
GLubyte image[WIDTH][HEIGHT][3]
```

です。これは RGB の値 (0 から 255) が R,G,B の順番に WIDTH × HEIGHT 回並んだものです。重要な点として

> WIDTH と HEIGHT は 2 の冪乗で、かつ 64 以上の整数

でなければなりません。

張り付けるべき画像、つまりテクスチャを作るにはパソコンなどで作成した画像を上のフォーマットに変換するのが一番簡単です。 OpenGL には残念ながら pict や gif などの画像ファイルから自動的にテクスチャデータを生成する機能は用意されていませんので、自分で作らなければなりません。そのヒントとして上記のテクスチャフォーマットが AVS のイメージデータ形式 (*.x) に似ていることを利用します。 AVS のイメージデータフォーマットは、始めに width と height の値が書き込まれ、その後に ARGB の値が 0 から 255 に規格化されてこの順番に並んでいます。したがって自分で作成した画像ファイルを一度 AVS のイメージデータに変換し、そこから width と height の値を読み込んでから、このヘッダ部分 (と A) を除けばテクスチャデータが得られます。 Appendix D に、このような方法でテクスチャファイルを作り出すプログラムを掲載しておきますので参考にして下さい。

サイエンティフィックビジュアリゼーションにおいては、あまり複雑なテクスチャマッピングは必要ありません。ユーザーインターフェース用の文字やメニューを表示させる手段としてテクスチャマッピングを利用する場合がほとんどでしょう。ここでは最も簡単な、 2 次元のテクスチャを 3 次元空間の長方形にマップする方法を示したサンプルプログラムを示すだけにとどめます。ビジュアリゼーションプログラムでインターフェースのためのメニューやメッセージ文字列を表示させる場合は、このサンプルプログラムのうち、読み込むテクスチャファイルの名前 (ここでは “texture.sample.256x256”) と、張り付られる長方形の各頂点の座標値を変更するだけで利用できるでしょう。テクスチャマッピングのより高度な機能やこのサンプルプログラムでコールされているテクスチャマッピング関係の各 OpenGL 関数については *OpenGL Programming Guide* [4, 5] を参照して下さい。このサンプルプログラムでは width と height がどちも 256 ピクセルのテクスチャを使っています。

texture.c

```
/*
 *   texture.c
\*
     An OpenGL sample program.
        - Texuture Mapping.
*/

#include <stdio.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

#define IMAGEWIDTH  256
#define IMAGEHEIGHT 256

char    image[IMAGEWIDTH][IMAGEHEIGHT][3];
GLuint  texidx = 1;

void initial(void) {
    FILE *fp;
    fp = fopen("texture.sample.256x256","r");
    if (fp==NULL) {
        printf(" open err; image file.\n");
        exit(9);
    }

    fread(image, 1, IMAGEWIDTH*IMAGEHEIGHT*3, fp);

    fclose(fp);

    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);


    glNewList(texidx, GL_COMPILE);
        glTexImage2D(GL_TEXTURE_2D, 0, 3,
                     IMAGEWIDTH, IMAGEHEIGHT, 0, GL_RGB,
                     GL_UNSIGNED_BYTE, &image[0][0][0]);
        glPushMatrix();
            glBegin(GL_QUADS);
                glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, -1.0, 0.0);
                glTexCoord2f(1.0, 1.0); glVertex3f( 1.0, -1.0, 0.0);
                glTexCoord2f(1.0, 0.0); glVertex3f( 1.0,  1.0, 0.0);
                glTexCoord2f(0.0, 0.0); glVertex3f(-1.0,  1.0, 0.0);
            glEnd();
        glPopMatrix();
    glEndList();
```

```
    glClearColor(0., 0., 0., 0.);
}


void display (void) {
    int   i;

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glDisable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);

    glTranslatef (0.0, -2.5, -5.0);

    for (i=2; i>=-2; i--) {
        glPushMatrix();
            glRotatef(i*20.0, 0.0, 0.0, 1.0);
            glTranslatef(0.0, 3.0, 0.0);
            glCallList(texidx);
        glPopMatrix();
    }

    glFlush();
}


void reshape(int width, int height) {
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 15.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

\ end of texture.c /

第 3 章

CAVE ライブラリの使い方

3.1 CompleXcope のハードウエアシステム

この章では CompleXcope に OpenGL で構成した仮想現実世界を表示させるプログラムを解説します。 CAVE システムは現在世界各地にありますが、そのハードウエアシステムの構成は全て同じではありません。我々の CAVE システムのハードウエア構成を見てみましょう。

```
1．背面投影型スクリーン（壁用）３枚
2．床面スクリーン　１枚
       スクリーンサイズ ＝ 10 フィート x 10 フィート
3．液晶ステレオプロジェクター　３台
4．SGI ONYX, 4 x CPU R4400, Main mem. 512MB, 3 x Reality Engine II
5．反射鏡 ３枚
6．液晶シャッター眼鏡
7．ワンド
8．ヘッド・ワンドトラッキング Ascention Technology 社 "Flock of Bird"
9．オーディオシステム　無し
```

3.2 ワンドについて

ワンドには３つのボタンと一つのジョイスティックがついており、 CompleXcope 内の人間が VR 空間の物体と相互作用するために用いられます。ワンドは各ボタンが「押された」「放された」という信号や、ジョイスティックが前後、左右、ななめに倒された角度と方向を ONYX に送ります。プログラム内でワンドの各機能を使うように指定しない限り、それらの情報は使われません。ワンドの使い方は第 3.16章で解説します。

3.3 CAVE ファイル

ハードウエアに加えて CAVE システムの基本的な柱は CAVE ライブラリとよばれるライブラリ及びいくつかの補助的なコマンドです。 CAVE 関係のファイルは全て

```
/usr/local/CAVE
```

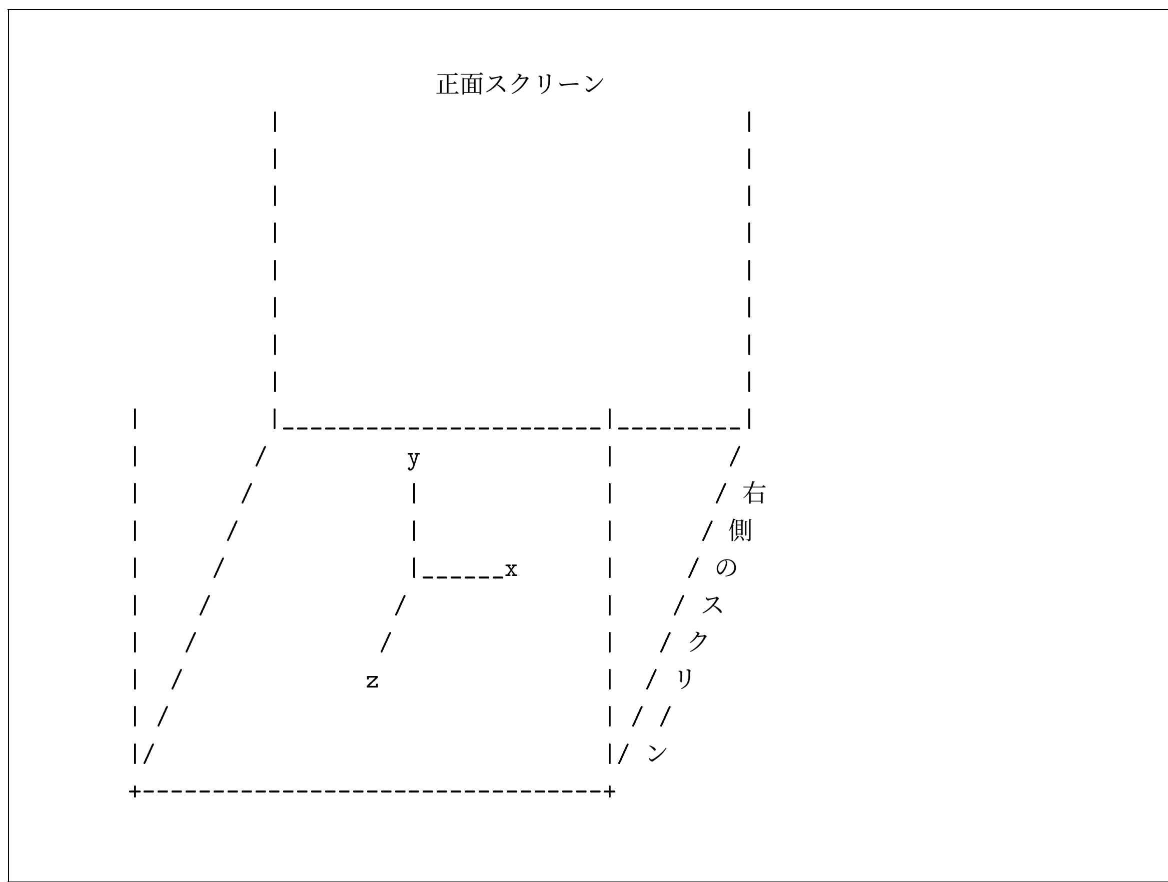
にあります。その下には

```
/usr/local/CAVE/include   ヘッダファイル群
/usr/local/CAVE/lib       ライブラリ群
/usr/local/CAVE/bin       診断用のコマンド群
```

等のディレクトリがあります。

3.4 CAVE 座標系

CAVE の座標系は、床面上の中央が原点、向かって正面が +x、鉛直上方が +y、入り口方向が +z と定義されています。



従って CompleXcope の部屋は

```
-5 <= x <= +5
 0 <= y <= +10
-5 <= z <= +5
```

となります。距離の単位はデフォルトではフィートです。

3.5 プロセス

一つの CompleXcope アプリケーションを動かしているときには、同時に 5 つのプロセスが走っています。

```
1. display process (1)
2. display process (2)
3. display process (3)
4. application computation process
5. tracking process
```

display process というのは文字通り画像を表示するプロセスです。 CompleXcope ではスクリーンが全部で 3 面あるので、 display process は 3 つあります。 application computation process というのは、 CAVE プログラムのメインループで、これがユーザからの終了信号を待っています。また、時間変化のある映像を表示する場合には、このプロセスが時間発展を計算します。 tracking process というのは、トラッカーが検知した液晶シャッター眼鏡とワンドの位置、方向のデータを取り込むプロセスです。以上のプロセスは ONYX 上の共有メモリ (shared memorry) を利用して通信します。

3.6 Configuration File

CAVE には様々な実行モードがあります。（例えばステレオ表示モードとモノラル表示モードなど。）それらのモードは .caverc という名前の configuration file で変更します。

```
# sample of .caverc
simulator y
DisplayMode mono
```

この configuration file では二つのモードをデフォルトから変更しています。このファイルをユーザのホームディレクトリ、またはアプリケーション実行時のカレントディレクトリにおいておけば、後で説明する CAVE simulator が on になり、 VR 画像は CompleXcope のスクリーンではなく、ワークステーション ONYX のモニタに表示されます。また Display ode mono というのは画像をステレオではなくモノラルで表示させるオプションです。プレゼンテーション等の目的で CompleXcope の映像をカメラで撮影するときにはこのオプションが便利です。 CAVE configuration file は次の順番に読まれ、この順番に上書きされます[1]。 (存在しない場合は無視されます。)

```
/usr/local/CAVE/etc/cave.config
/usr/local/CAVE/etc/aso.config
~/.caverc
./.caverc
```

通常 (CompleXcope にステレオで表示する場合) はユーザが個別に設定するファイル ~/.caverc または ./.caverc は全てコメント (# で始まる行) にしておくか、ファイルを消去しておきます。

3.7 CompleXcope プログラムの流れ

ここで CompleXcope プログラムの構造について説明しましょう。全ての CompleXcope のアプリケーションプログラムは基本的に全く同じ構造を持っています。まず、

(1) (Optional) アプリケーションに固有な初期化

を行います。例えば、使用可能な共有メモリの大きさを CAVE のデフォルトの値から変更する場合などは、ここで必要な関数をコールします。 (これに関しては後で説明します。) そして次に

[1] 2 行目にある aso というのは CompleXcope で使っている ONYX の host name です。

(2) (Required) CAVE configuration の設定

を行います。これは上で説明した CAVE configuration file を読み込み、各パラメータを設定するものです。 configuration file からだけではなく、各アプリケーションのコマンドラインオプションからも与えることが可能です。そして

(3) (Optional) 共有メモリの確保と初期化

を行います。次に、

(4) (Required) CAVE ライブラリの初期化

を行います。この時にスクリーンの数と同じだけの display process と一つの application computation process、及び一つの tracking process を fork します。そして、ある CAVE ライブラリ関数の引数として渡す形で

(5) (Optional) 各アプリケーションに固有な OpenGL プログラムの初期化

を行う関数を指定します。ここでは様々な OpenGL ステートの初期化 (例えば背景色の設定) や数値データの設定を行います。プログラム中のこの部分は既に fork されているので、この関数は (スクリーンの数だけある) 複数の display process それぞれで実行されます。次に、やはりある CAVE ライブラリ関数の引数として渡す形で

(6) (Required) display 関数の指定

を行います。この関数は 3 次元物体や照明のなど OpenGL による仮想世界を構成する部分、つまりプログラムの中心部分といえます。当然この関数も display process の数だけコピーされています。そして、別の CAVE ライブラリ関数の引数として渡す形で

(7) (Optional) computation 関数の指定

を行います。例えばアニメーションプログラムの場合、時間発展に対応して変化する量の計算を行う関数をここで指定します。そして最後に

(8) (Required) CAVE ライブラリ終了処理

を行う関数をコールします。

3.8 CAVE シミュレーター

プログラムを作成する過程では、完成までにプログラムを何度も走らせてテストするのが普通で、これは CompleXcope プログラミングでも同様です。しかしテストする度に CompleXcope の中に靴を脱いで入って、液晶シャッター眼鏡をつけて・・・と繰り返すのは大変面倒です。 CAVE ライブラリを用いたプログラムのこのようなテストのために CAVE シミュレータが用意されています。これは CAVE での映像、操作を X Window 上でシミュレートするものです。 CAVE configuration file で simulator を y と設定してからプログラムを走らせて下さい[2]。モニタ画面にウインドウが表示されます。 CAVE シミュレータでは CompleXcope 内部にいる人間の頭の位置・向き、ワンドの位置・向きと、プログラムで構成された 3 次元物体が同時に表示されます。

CAVE シミュレータでは体験者の頭の動き、ワンドの移動やボタンの操作などをキーボードとマウスでシミュレートします。

[2]モノラル表示モードも on にした方が見やすいでしょう。

```
CAVE シミュレータでの頭の動きのコントロール方法

    left arrow          move left
    right arrow         move right
    Up arrow            move forward
    Down arrow          move backward
    Shift + up arrow    move up
    Shift + donw arrow  move down
    Alt + left arrow    rotate left
    Alt + right arrow   rotate right
    Alt + up arrow      rotate up
    Alt + down arrow    rotate down
    p                   reset head and wand to initial
```

```
CAVE シミュレータでのワンドの動きのコントロール方法

   Cntl + mouse movement   move wnad left/right/forward/back
   Shift + mouse movement  move wand left/right/up/down
   Alt + mouse movement    rotate wand/left/right/up/down
   < and >                 rotate wand about Z
   Home                    reset wand to be in front of user
   F1/F2/F3/...            select wand 1/2/3/... as the current wand
```

```
CAVE シミュレータでのワンドのボタン・ジョイスティックのコントロール方法

   mouse left button       wand left button
   mouse middle button     wand middle button
   mouse right button      wand right button
   Space + mouse movement  joystick
```

```
CAVE シミュレータでの表示方法のコントロール方法

    1       CAVE 内部で人が見る映像を表示
    2       CAVE を「外から見た」映像を表示
    w       ワンド（赤い棒）の表示 / 非表示
    u       人の頭の表示 / 非表示
    INSERT  CAVE の枠の表示 / 非表示
```

3.9 サンプルプログラム１

それでは早速サンプルプログラムを見てみましょう。次のプログラムは p.42で解説した OpenGL のサンプルプログラム torus.c (トーラスを一つ描く) を CompleXcope 用に変更したものです。たったこれだけの短いプログラムで VR が作れるのは驚くべきことではないでしょうか。

Xtorus.c

```
/*
 *   Xtorus.c
\*
     A CompleXcope sample program.
          - A torus by aux lib; 4 feet off the floor.
          - Copied and changed from /usr/local/CAVE/src/ogl/ball.c
          - No animation.
          - No interaction.
          - No navigation.
*/

#include <cave_ogl.h>

/* init_gl - GL initialization function. This function will be called
  exactly once by each of the drawing processes, at the beginning of
  the next frame after the pointer to it is passed to CAVEInitApplication.
  It defines and binds the light and material data for the rendering. */

void init_gl(void) {
    GLfloat mat_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };

    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glClearColor(0., 0., 0., 0.);

    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  mat_diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  mat_ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
    glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 64.0);

    glEnable(GL_LIGHT0);
    }

/* draw_torus - the display function. This function is called by the
  CAVE library in the rendering processes' display loop. It draws a
  torus 4 feet off the floor */

void draw_torus(void) {
    glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
```

```
    glEnable(GL_LIGHTING);
    glPushMatrix();
        glTranslatef(0.0, 4.0, -2.0);
        auxSolidTorus(0.2,1.0);
    glPopMatrix();
    glDisable(GL_LIGHTING);
}

main(int argc,char **argv) {
    /* Initialize the CAVE */
    CAVEConfigure(&argc,argv,NULL);
    CAVEInit();
    /* Give the library a pointer to the GL initialization function */
    CAVEInitApplication(init_gl,0);
    /* Give the library a pointer to the drawing function */
    CAVEDisplay(draw_torus,0);
    /* Wait for the escape key to be hit */
    while (!CAVEgetbutton(CAVE_ESCKEY))
        sginap(10);    /* Nap so that this busy loop doesn't waste CPU time */
    /* Clean up & exit */
    CAVEExit();
}
```

\ end of Xtorus.c /

"CAVE" で始まる関数はすべて CAVE ライブラリの関数です。重要な関数について解説しましょう。

`CAVEConfigure()`	Configuration データを読み込み、設定する。

`CAVEInit()`	CAVE 初期化。そしてプロセスの fork

`CAVEInitApplication()`	OpenGL 関係の初期化を行う関数の指定

`init_gl()`	（ユーザ定義）OpenGL の初期化処理

`CAVEDisplay()`	ディスプレイ関数の指定

`draw_torus()`	(ユーザ定義)OpenGL 表示処理

CAVEgetbutton(CAVE_ESCKEY)	ESC キー hit の検知

sginap()	プロセスのサスペンド

CAVEExit()	子プロセスを停止させて、プログラムを終了する

3.10 サンプルプログラム２

次に、球を描くプログラムを見てみましょう。上の Xtours.c の “auxSolidTorus” の代わりに “auxSolidSphere” とすれば球を描けますが、この aux ライブラリ関数では球の滑らかさを調節できず不便です。そこで glu ライブラリの関数を用いて球を描いてみます (p.33参照)。

Xball.c

```
/*
 *   Xball.c
\*
     A CompleXcope sample program.
          - A ball by the glu lib; 1 feet radius, 4 feet off the floor.
          - Copied and changed from /usr/local/CAVE/src/ogl/ball.c
          - No animation.
          - No interaction.
          - No navigation.
*/

#include <cave_ogl.h>
#include <GL/glu.h>

static GLUquadricObj *sphereObj;

/* init_gl - GL initialization function. This function will be called
    exactly once by each of the drawing processes, at the beginning of
    the next frame after the pointer to it is passed to CAVEInitApplication.
    It defines and binds the light and material data for the rendering,
    and creates a quadric object to use when drawing the sphere. */
void init_gl(void) {
    GLfloat mat_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };

    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
```

```
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glClearColor(0., 0., 0., 0.);

    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glEnable(GL_LIGHT0);

    sphereObj = gluNewQuadric();
}

/* draw_ball - the display function. This function is called by the
    CAVE library in the rendering processes' display loop. It draws a
    ball 1 foot in radius, 4 feet off the floor, and 2 foot in front
    of the front wall (assuming a 10' CAVE). */
void draw_ball(void) {
    glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
    glEnable(GL_LIGHTING);
    glPushMatrix();
        glTranslatef(0.0, 4.0, -3.0);
        gluSphere(sphereObj, 1.0, 16, 16);
    glPopMatrix();
    glDisable(GL_LIGHTING);
}

main(int argc,char **argv) {
/* Initialize the CAVE */
    CAVEConfigure(&argc,argv,NULL);
    CAVEInit();
/* Give the library a pointer to the GL initialization function */
    CAVEInitApplication(init_gl,0);
/* Give the library a pointer to the drawing function */
    CAVEDisplay(draw_ball,0);
/* Wait for the escape key to be hit */
    while (!CAVEgetbutton(CAVE_ESCKEY))
        sginap(10);    /* Nap so that this busy loop doesn't waste CPU time */
/* Clean up & exit */
    CAVEExit();
}
```

\ end of Xball.c /

3.11 サンプルプログラム 3 (アニメーション)

それでは次に動きのある映像を作って見ましょう。次のプログラムは p.52で解説したトーラスが回転する swing_torus.c の CompleXcope 版です。

Xswing_torus.c

```
/*
 *   Xswing_torus.c
\*
     A CompleXcope sample program.
          - Copied and changed from /usr/local/CAVE/src/ogl/bounce.c
          - A rotating torus.
          - Calculation of their movements are performed in the main process.
          - It is communicated to the three drawing processes through
            the shared memory.
          - No interaction.
          - No navigation.
*/

#include <cave_ogl.h>
#include <GL/glu.h>

/* The data that will be shared between processes */
struct _torusdata {
        float angle1;
        float angle2;
        };

void init_gl(void),draw(struct _torusdata *);
struct _torusdata *init_shmem(void);
void compute(struct _torusdata *);

main(int argc, char **argv) {
    struct _torusdata *torus;
    CAVEConfigure(&argc,argv,NULL);
    torus = init_shmem();
    CAVEInit();
    CAVEInitApplication(init_gl,0);
    CAVEDisplay(draw,1,torus);
    while (!CAVEgetbutton(CAVE_ESCKEY)) {
        compute(torus);
        sginap(1);
    }
    CAVEExit();
}

struct _torusdata *init_shmem(void) {
    struct _torusdata *torus;
    torus = CAVEMalloc(sizeof(struct _torusdata));
    bzero(torus,sizeof(struct _torusdata));
    return torus;
```

```
}

void compute(struct _torusdata *torus) {
    float t = CAVEGetTime();
    torus->angle1 = t*50;
    torus->angle2 = t*73;
}

void init_gl(void) {
    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glEnable(GL_LIGHT0);
    glClearColor(0., 0., 0., 0.);

    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
}

void draw(struct _torusdata *torus) {
    glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
    glEnable(GL_LIGHTING);

    glTranslatef(0.0, 4.0, -3.0);
    glPushMatrix();
        glRotatef(torus->angle1, 1.0, 0.0, 0.0);
        glRotatef(torus->angle2, 0.0, 1.0, 1.0);
        auxSolidTorus(0.2, 1.0);
    glPopMatrix();

    glDisable(GL_LIGHTING);
}
```

\ end of Xswing_torus.c /

このプログラムが、動きのない Xtorus.c や Xball.c と比べて大きく違うのは、 main の while() のループ中で compute() を呼んでいるところです。ここで時間発展の計算を行います。具体的にはトーラスの回転角度 (angle1 と angle2) を時間に比例して増やしていきます。この角度データを用いて、ディスプレイプロセスがそれだけ回転させたトーラスを描くわけです。ディスプレイプロセスは 3 つあり、それらは comupte() を行っているプロセス (application computation process) とは別なプロセスであることを思い出して下さい。 application computation process が更新した値を 3 つのディスプレイプロセスは知っていなければなりません。 CAVE のプログラムでは、この場合のようにプロセス間で常にデータを引き渡しています。そして、この通信のために shared memory を用いています。今の場合、角度のデータは torus という名前の構造体にまとめられ、その torus は shared memory 上に置かれています。 shared memory への割り当てを行うのが init_shmem() 関数内での CAVE alloc() です。

ここで重要な点をひとつ。

```
CAVE で利用できる shared memory はデフォルトで 8MB である。

それ以上使う場合は

     CAVESetOption(CAVE_SHMEM_SIZE, size_in_bytes)

を CAVEConfigure() の前で呼ぶ。
```

3.12 サンプルプログラム 4

もう一つアニメーションプログラムを見てみましょう。赤と緑の二つのボールが床で跳ね返ります。

Xbounce.c

```
/*
 *   Xbounce.c
\*
     A CompleXcope sample program.
          - Copied and changed from /usr/local/CAVE/src/ogl/bounce.c
          - Two bouncing balls.
          - Calculation of their movements are performed in the main process.
          - It is communicated to the three drawing processes through
            the shared memory.
          - No interaction.
          - No navigation.
*/

#include <cave_ogl.h>
#include <GL/glu.h>

/* The data that will be shared between processes */
struct _balldata {
        float y;
        };

void init_gl(void),draw_balls(struct _balldata *);
struct _balldata *init_shmem(void);
void compute(struct _balldata *);

main(int argc, char **argv) {
    struct _balldata *ball;
    CAVEConfigure(&argc,argv,NULL);
/* Initialize shared memory */
    ball = init_shmem();
    CAVEInit();
    CAVEInitApplication(init_gl,0);
/* Give the library a pointer to the drawing function, plus one argument */
    CAVEDisplay(draw_balls,1,ball);
```

```
    while (!CAVEgetbutton(CAVE_ESCKEY)) {
/* Update the balls' positions */
        compute(ball);
        sginap(1);
    }
    CAVEExit();
}

/* init_shmem - initializes shared memory. The data is allocated from a
    shared memory arena, and so will be common to all processes forked after
    this is called. */
struct _balldata *init_shmem(void) {
    struct _balldata *ball;
    ball = CAVEMalloc(2*sizeof(struct _balldata));
    bzero(ball,2*sizeof(struct _balldata));
    return ball;
}

/* compute - compute new positions for the balls. The height of the balls
    is a function of the current CAVE time. */
void compute(struct _balldata *ball) {
    float t = CAVEGetTime();
    ball[0].y = fabs(sin(t)) * 5 + 1;
    ball[1].y = fabs(sin(t*1.2)) * 3 + 1;
}

static GLuint redMat, blueMat;
static GLUquadricObj *sphereObj;

/* init_gl - initialize GL lighting & materials */
void init_gl(void) {
    float redMaterial[] = { 1, 0, 0, 1 };
    float blueMaterial[] = { 0, 0, 1, 1 };

    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glClearColor(0., 0., 0., 0.);

    glEnable(GL_LIGHT0);

    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);


    /* 1st display list */
    redMat = glGenLists(1);
```

```
    glNewList(redMat, GL_COMPILE);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, redMaterial);
    glEndList();
    /* 2nd display list */
    blueMat = glGenLists(1);
    glNewList(blueMat, GL_COMPILE);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, blueMaterial);
    glEndList();

    sphereObj = gluNewQuadric();
}

/* draw_balls - draw the two balls, using the shared data for their
  y coordinates */
void draw_balls(struct _balldata *ball) {
    glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
    glEnable(GL_LIGHTING);

    glCallList(redMat);
    glPushMatrix();
        glTranslatef(3.0, ball[0].y, 0.0);
        gluSphere(sphereObj, 1.0, 16, 16);
    glPopMatrix();

    glCallList(blueMat);
    glPushMatrix();
        glTranslatef(0.0, ball[1].y, -3.0);
        gluSphere(sphereObj, 1.0, 16, 16);
    glPopMatrix();

    glDisable(GL_LIGHTING);
}
```

\ end of Xbounce.c /

3.13 サンプルプログラム 5

これまでのプログラムでは glEnable(GL_LIGHTING) を呼び、照明効果のある OpenGL 映像を表示していました。最後に、照明処理を行わないプログラムを見てみましょう。基本的な構造は全く同じで、単に GL_LIGHTING を off にして通常の OpenGL プログラムを書くだけです。このプログラムを走らせると CompleXcope 内が冬になります。

Xsnowfall.c

```
/*
 *   Xsnowfall.c
\*
     A CompleXcope sample program.
          - Snow fall.
          - No lighting.
```

```
          - No interaction.
          - No navigation.
*/

#include <cave_ogl.h>
#include <GL/glu.h>
#include <stdlib.h>

#define NCOMMET 500
#define XMAX 20.0
#define YMAX 20.0
#define ZMAX 10.0
#define XMIN (-20.0)
#define YMIN (-5.0)
#define ZMIN (-20.0)

/* The data that will be shared between processes */
struct _snowdata {
        float xpos;
        float ypos;
        float zpos;
        float xaxis;
        float yaxis;
        float zaxis;
        float spin;
        float vx;
        float vy;
        float vz;
        };

void init_gl(void),draw(struct _snowdata *);
struct _snowdata *init_shmem(void);
void compute(struct _snowdata *);

static GLuint flake_indx;

main(int argc, char **argv) {
    struct _snowdata *snows;
    CAVEConfigure(&argc,argv,NULL);
    snows = init_shmem();
    CAVEInit();
    CAVEInitApplication(init_gl,0);
    CAVEDisplay(draw,1,snows);
    while (!CAVEgetbutton(CAVE_ESCKEY)) {
        compute(snows);
        sginap(1);
    }
    CAVEExit();
}

float ransu(void) {
```

```
    float r = rand() / 32768.0;
    return r;
    }

struct _snowdata *init_shmem(void) {
    int   i;
    struct _snowdata *snows;

    snows = CAVEMalloc(NCOMMET*sizeof(struct _snowdata));
    bzero(snows,NCOMMET*sizeof(struct _snowdata));

    for (i=0; i<NCOMMET; i++) {
        snows[i].xpos = XMIN + (XMAX-XMIN)*ransu();
        snows[i].ypos = YMIN + (YMAX-YMIN)*ransu();
        snows[i].zpos = ZMIN + (ZMAX-ZMIN)*ransu();
        snows[i].xaxis = ransu();
        snows[i].yaxis = ransu();
        snows[i].zaxis = ransu();
    }

    return snows;
}

void compute(struct _snowdata *snows) {
    [illegible]

COMMET; i++)snowyloat
sfor                                                6036907D21740neY(snow);
                                                    snows[i].ypos
```

```
        glVertex3f(-flake_size/2, 0.0, 0.0);
      glEnd();
    glEndList();

    flake_indx = glGenLists(1);
    glNewList(flake_indx, GL_COMPILE);
      glPushMatrix();
        glCallList(tmp_indx);
        glPushMatrix();
          glRotatef(180.0, 1.0, 0.0, 0.0);
          glTranslatef(0.0, -flake_size/sqrt(3.0), 0.0);
          glCallList(tmp_indx);
        glPopMatrix();
    glEndList();
}

void draw(struct _snowdata *snows) {
    int i;

    glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);

    for (i=0; i<NCOMMET; i++) {
        glPushMatrix();
            glTranslatef(snows[i].xpos,
                         snows[i].ypos,
                         snows[i].zpos);
            glRotatef(snows[i].spin,
                      snows[i].xaxis,
                      snows[i].yaxis,
                      snows[i].zaxis);
            glCallList(flake_indx);
        glPopMatrix();
    }

}
```

\ end of Xsnowfall.c /

一つの VR シーンの中で照明処理を行う物体と、行わない物体の 2 種類が混在している場合[3]は、描画ルーチン (上では draw) の中で glEnable(GL_LIGHTING) と glDisable(GL_LIGHTING) を呼び、それぞれの後で 2 種類の物体を別けて描画します。

3.14 クリッピングプレーン

以上のプログラムを見て分かるように、デプステストの on、射影変換などは全て自動的に CAVE ライブラリが行ってくれます。p.39で解説したように、 OpenGL の射影変換では、奥行き方向のクリッピング (ある深さ領域に入っているものだけをレンダリングする操作) を行いますが、 CAVE ライブラリは、それを CAVENear と CAVEFar という global float 変数で決めていて、そのデフォルトは CAVENear = 0.1 (フィート)、 CAVEFar = 100 (フィート) とされています。

[3]例えば磁力線は非照明モードで、等値面は照明モードで描きます。

3.15 共有メモリの解放

CAVE プログラムでは共有メモリを多用します。そのメモリのアロケーションは CAVE alloc() でプログラマが行いますが、解放は (プログラムが正常終了した場合) CAVEExit() が自動的に行うので気にする必要はありません[4]。しかし、(プログラム開発中はよくあることですが) 実行中のプログラムがクラッシュしてしまった場合など、CAVE alloc() でアロケートしたメモリが、終了後も共有メモリに残ってしまっている場合があります。その様な時には、

ipcs –m –b

というコマンドで active になっている共有メモリ領域のサイズとそれをアロケートしたユーザ名が分かるので試してみて下さい。

```
% ipcs -m -b
IPC status from /dev/kmem as of Mon Jul 13 11:52:15 1998
T      ID      KEY         MODE       OWNER    GROUP  SEGSZ
Shared Memory:
m       0 0x53637444 --rw-r--r--     root      sys    136
m       1 0x00002648 --rw-rw-rw-     root      sys    228
m       2 0x00002649 --rw-rw-rw-     root      sys    140
m       3 0x000007c8 --rw-rw-rw-     root      sys      4
m       4 0x000009a4 --rw-rw-rw-    guest    guest  55912
```

この場合、 guest ユーザは 55912 バイトを解放せずに残しています。これをシェルから解放するには、“ipcrm –m <ID>” コマンドを使います。

```
aso% ipcrm -m 4
aso% ipcs -m -b
IPC status from /dev/kmem as of Mon Jul 13 11:52:27 1998
T      ID      KEY         MODE       OWNER    GROUP  SEGSZ
Shared Memory:
m       0 0x53637444 --rw-r--r--     root      sys    136
m       1 0x00002648 --rw-rw-rw-     root      sys    228
m       2 0x00002649 --rw-rw-rw-     root      sys    140
m       3 0x000007c8 --rw-rw-rw-     root      sys      4
```

3.16 ワンドによるコントロール

CompleXcope では仮想現実世界との相互作用のためにワンドを利用します。ワンドの 3 つのボタンには下図のように番号がついています。また、ジョイスティック[5]は前後、左右、斜めに倒せますが、その傾きの向きと大きさは下の様に 2 次元の x-y 座標の値で得られます ($|x| \le 1$, $|y| \le 1$)。

[4]何らかの理由で共有メモリの解放が必要ならば CAVEFree() という関数が用意されています。

[5]スティックと言うよりもむしろボタンに似ていますが、 CAVE の伝統に従い、ジョイスティックと呼びます。

```
ワンド
                       ボタンの番号

                         ---------
                        /1   2   3\
                       / O   O   O \
                      |      ---    |
                      |     / J \   |  J: ジョイスティック
                       \    \___/   /
                        \_________/       その傾きに対する座標系:
                          |     |                y
                          |     |                |
                          |     |                |
                          |     |                +----- x
                          |     |
```

CAVE ライブラリにはワンドに関する次の機能が用意されています。

1. ワンドの位置を検出する。
2. ワンドの３次元的な向き (方向ベクトル) を検出する。
3. ワンドの各ボタンの状態 (push/nonpush) が変更されたかどうかを知る。
4. 現在ワンドのボタンが押されているかどうかを判断する。
5. ジョイスティックの傾き (前後／左右) の角度 (x,y) を検出する。

3.17 位置の検出

CAVE プログラムではトラッカーシステムが検知する液晶シャッター眼鏡とワンドの位置／方向のデータを tracking process が取り込みます。この tracking process そのものは CAVEInit 関数をコールすれば自動的に生成されるのでユーザは意識する必要はありません。 CAVElibrary には、この tracking process から必要な情報を取り出す関数が用意されています。まず、位置を検出する次の関数から説明しましょう。

```
void CAVEGetPosition(CAVEID posid, float position[3])
```

ここで posid は CAVE_HEAD, CAVE_WAND, CAVE_LEFT_EYE, CAVE_RIGHT_EYE, CAVE_HEAD_NAV, CAVE_WAND_NAV, CAVE_LEFT_EYE_NAV, CAVE_RIGHT_EYE_NAV のいずれかが入ります。それぞれの意味は次の通りです。

CAVE_HEAD	:	実空間座標での頭 (液晶シャッター眼鏡) の位置
CAVE_WAND	:	実空間座標でのワンドの位置
CAVE_LEFT_EYE	:	実空間座標での左目の位置
CAVE_RIGHT_EYE	:	実空間座標での右目の位置
CAVE_HEAD_NAV	:	ナビゲーション座標での頭の位置
CAVE_WAND_NAV	:	ナビゲーション座標でのワンドの位置
CAVE_LEFT_EYE_NAV	:	ナビゲーション座標での左目の位置
CAVE_RIGHT_EYE_NAV	:	ナビゲーション座標での右目の位置

ここで、実空間座標とナビゲーション座標という言葉が出て来ました。実空間座標とは、前回説明した CAVE の部屋内に定義されたフィートを単位にした座標系です。従って、この座標系での位置は $-5 \leq x \leq 5$, $0 \leq y \leq 10$, $-5 \leq z \leq 5$ の範囲に必ず入っています。トラッカー装置が検出する各装置の位置・方向はこの座標系での値なので、実空間座標はトラッカー座標とも呼ばれます。

一方、ナビゲーション座標系とは後述するナビゲーション機能を用いている時に使うと便利な座標系で、単位 (feet) は変わりませんが、その値は任意の値を取ることが出来ます。

3.18 方向の検出

頭やワンドの向いている方向を得るには次の関数を用います。

```
void CAVEGetVector(CAVEID vectorid, float vector[3])
```

ここで引き数 vectorid には

```
CAVE_HEAD_FRONT,   CAVE_WAND_FRONT,
CAVE_HEAD_BACK,    CAVE_WAND_BACK,
CAVE_HEAD_LEFT,    CAVE_WAND_LEFT,
CAVE_HEAD_RIGHT,   CAVE_WAND_RIGHT,
CAVE_HEAD_UP,      CAVE_WAND_UP,
CAVE_HEAD_DOWN,    CAVE_WAND_DOWN,
```

または、

```
CAVE_HEAD_FRONT_NAV,   CAVE_WAND_FRONT_NAV,
CAVE_HEAD_BACK_NAV,    CAVE_WAND_BACK_NAV,
CAVE_HEAD_LEFT_NAV,    CAVE_WAND_LEFT_NAV,
CAVE_HEAD_RIGHT_NAV,   CAVE_WAND_RIGHT_NAV,
CAVE_HEAD_UP_NAV,      CAVE_WAND_UP_NAV,
CAVE_HEAD_DOWN_NAV,    CAVE_WAND_DOWN,
```

が入ります。前者は実空間座標の値を取り出し、後者はナビゲーション空間座標の値を取り出します。それぞれの引き数の意味するところは明らかでしょう。 例えば CAVE_WAND_FRONT を第１引数にして CAVEGetVector 関数を呼べば、第２引数 vector にはワンドの指す前方の単位ベクトルが入ります。

3.19 ワンドボタン状態変化の検出

ワンドのボタン (１番から３番) のそれぞれが、最後に調べたときと比べて何か変化があったかどうか (押された、あるいは離された) を調べる関数です。

```
int CAVEButtonChange(int button)       /* button = 1,2, or 3 */
```

この関数を呼んで 0 が返って来た時にはそのボタンは前回と比べて変化がなく、 1 はボタンが押されたこと、 2 はボタンが離されたことを意味します。

3.20 ワンドボタン現在の状態の検出

CompleXcope プログラムにおいては、ある処理をしているその時にワンドのボタンが押されているかどうかを知りたい場合がしばしばあります。その場合には次のマクロのうちどれかを用います。

```
CAVEBUTTON1
CAVEBUTTON2
CAVEBUTTON3
```

それぞれ 1 番、 2 番、 3 番のボタンが現在押されているかどうかを教えてくれます。押されている場合には値 1 がそうでない場合には 0 が帰ってきます。これは例えば次のように使います。

```
if (CAVEBUTTON1) {
    .
    .
    procedure to be done when button 1 (left) is pressed
    .
    .
}

 if (!CAVEBUTTON2) {
    .
    .
    procedure to be done when button 2 (middle) is not pressed
    .
    .
}
```

3.21 ジョイスティックの傾きの検出

ジョイスティックの傾きの向きと大きさの検出には次のマクロを使います。

```
CAVE_JOYSTIC_X
CAVE_JOYSTIC_Y
```

それぞれ上で説明したジョイスティックの x,y 方向への傾きを $[-1.0, 1.0]$ の範囲の float 値で返します。このマクロを使う時には次の点に注意して下さい。それは、指がジョイスティックに触っていない場合でも、ノイズのために上記のマクロは (非常に小さいとはいえ) 必ずしも正確に 0.0 の値を返さないという点です。従って通常は次のような判定を行います。

```
if (fabs(CAVE_JOYSTIC_Y)>0.2) {
    .
    .
    /* procedure to be done when joystick is pushed forward
                                              or backward */
    .
    .
}
```

3.22 ワンドを使った相互作用のサンプルプログラム

それではワンドを使って仮想現実空間と相互作用するプログラムのサンプルを見てみましょう。始めは何も表示されてはいませんが、ワンドの左ボタンを押すと、ワンドの先に半径１フィートの赤い球が現われます。ワンドの真ん中ボタンを押すと、その瞬間にワンドが球の中にあればその球を “捕まえる” ことが出来ます。球を捕まえるとその色が黄色に変わります。そして、真ん中ボタンを押したままワンドを動かすと、黄色くなった球はワンドを持った手の動きについて行きます。また、真ん中ボタンを押したまま、ジョイスティックを前に倒すと球が大きくなり、後ろに倒すと小さくなります。真ん中ボタンを離した瞬間に球は赤に戻ります。そして、ワンドの右ボタンを押すと球は消去されます。

interact.c

```
/*
 *   interact.c
\*
     A CompleXcope sample program.
          - Copied and changed from /usr/local/CAVE/src/ogl/interact.c
          - A sample program for the CAVE interaction procedure.
          - A ball appears when wand left button is pressed at the wand tip.
          - The ball is grabbed when wand middle button is pressed.
          - The ball is destroyed when wand right button is pressed.
          - No navigation.
*/


#include <cave_ogl.h>
#include <GL/glu.h>
#define SQR(x) ((x)*(x))

struct _balldata {
    float x,y,z;
    float radius;
    int   grabbed;
    int   exist;
};

static GLuint mat0, mat1;

void init_gl(void);
```

```
void draw_ball(struct _balldata *);
void check_add(struct _balldata *);
struct _balldata * init_shmem(void);
void update_held(struct _balldata *);
void check_grab(struct _balldata *);

main(int argc,char **argv) {
    struct _balldata  *ball;
    CAVEConfigure(&argc,argv,NULL);
    ball = init_shmem();
    CAVEInit();
    CAVEInitApplication(init_gl, 0);
    CAVEDisplay(draw_ball, 1, ball);
    while (!CAVEgetbutton(CAVE_ESCKEY)) {
        check_add(ball);
        if ( ball->grabbed )
            update_held(ball);
        else
            check_grab(ball);
        sginap(1);
    }
    CAVEExit();
}


/************\
> check_add  <
\************/
void check_add(struct _balldata *ball) {
   float wandPos[3], wandFront[3];
   if (CAVEButtonChange(1) == -1) {         /* left button is released */
        CAVEGetPosition(CAVE_WAND, wandPos);          /* wand position */
        CAVEGetVector(CAVE_WAND_FRONT, wandFront);   /* wand direction */
        ball->x = wandPos[0] + wandFront[0]*0.5;
        ball->y = wandPos[1] + wandFront[1]*0.5;
        ball->z = wandPos[2] + wandFront[2]*0.5;
        ball->radius = 1;
        ball->grabbed = 0;
        ball->exist = 1;
   }
   if (CAVEButtonChange(3) == 1)
        ball->exist = 0;
}

/*************\
> init_shmem  <
\*************/
struct _balldata * init_shmem(void) {
    struct _balldata *ball;
    ball = (struct _balldata *)CAVEMalloc(sizeof(struct _balldata));
    bzero(ball, sizeof(struct _balldata));
```

```
    return ball;
}

/**************\
> update_held  <
\**************/
void update_held(struct _balldata *ball) {
   float wandPos[3];
   if (!CAVEBUTTON2) {
        CAVEGetPosition(CAVE_WAND, wandPos);
        ball->x = wandPos[0];
        ball->y = wandPos[1];
        ball->z = wandPos[2];
        ball->grabbed = 0;
   }
   else {
       if (CAVE_JOYSTICK_Y > .5)             /* Joystick pushed forward */
           ball->radius *= 1.005;
       else if (CAVE_JOYSTICK_Y < -.5)       /* Joystick pushed backward */
           ball->radius /= 1.005;
   }
}


/*************\
> check_grab  <
\*************/
void check_grab(struct _balldata *ball) {
   float wandPos[3];
   float distsq;
   if (CAVEButtonChange(2) == 1) {     /* wand middle button is pressed */
        CAVEGetPosition(CAVE_WAND, wandPos);
        if (ball->exist) {
            distsq = SQR(ball->x - wandPos[0]) +
                     SQR(ball->y - wandPos[1]) +
                     SQR(ball->z - wandPos[2]);
            if (distsq < SQR(ball->radius)) {
                ball->grabbed = 1;
            }
        }
   }
}


static GLUquadricObj *sphereObj;

/**********\
> init_gl  <
\**********/
void init_gl(void) {
    GLfloat diffuse0[]  = { 0.0, 0.0, 1.0, 1.0 };
```

```
    GLfloat ambient0[]  = { 0.0, 0.3, 0.3, 1.0 };
    GLfloat specular0[] = { 0.4, 0.4, 0.4, 1.0 };
    GLfloat diffuse1[]  = { 1.0, 1.0, 0.0, 1.0 };
    GLfloat ambient1[]  = { 0.3, 0.3, 0.0, 1.0 };
    GLfloat specular1[] = { 0.4, 0.4, 0.4, 1.0 };

    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat light_ambient[]  = { 0.3, 0.3, 0.3, 0.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glEnable(GL_LIGHT0);
    glClearColor(0., 0., 0., 0.);
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    sphereObj = gluNewQuadric();

    mat0 = glGenLists(1);
    glNewList(mat0, GL_COMPILE);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  diffuse0);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambient0);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular0);
        glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 100.0);
    glEndList();

    mat1 = glGenLists(1);
    glNewList(mat1, GL_COMPILE);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  diffuse1);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambient1);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular1);
        glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 100.0);
    glEndList();
}

/************\
> draw_ball  <
\************/
void draw_ball(struct _balldata *ball) {
     glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
     glEnable(GL_LIGHTING);

     if (ball->exist) {
       if (ball->grabbed) {
         glPushMatrix();
           CAVEWandTransform();
           glCallList(mat1);
           gluSphere(sphereObj, ball->radius, 32, 32);
         glPopMatrix();
       }
```

```
        else {
            glPushMatrix();
              glTranslatef(ball->x, ball->y, ball->z);
              glCallList(mat0);
              gluSphere(sphereObj, ball->radius, 32, 32);
            glPopMatrix();
        }
    }
    glDisable(GL_LIGHTING);
}
```

\ end of interact.c /

3.23 ナビゲーション

CompleXCope の部屋の大きさは一辺が 10 フィートの立方体です。直径 1 フィートの球を CompleXcope の部屋の真ん中 (x=0,y=5,z=0) に浮かせるプログラムを書いた場合、ボールの内部を覗きたければ CompleXcope の中を歩いていってボールの中に頭を突っ込めばいいわけです。では、そのボールが正面の壁の向こう側 (例えば x=0, y=5, z=-8) にある場合は、どうすればボールの内部が覗けるのでしょうか？無限に広がる VR 空間の中で、我々が歩き回れるのは CompleXcope の大きさ（１０フィート立方の立方体）だけです。このようなときのためにナビゲーションと呼ばれる機能が用意されています。ナビゲーションとは、「ワンド等をもちいて、 VR 空間中に CompleXcope が置かれている位置や向きを動かしていく機能」です。

ナビゲーションの意味と使い方は、次のサンプルプログラムを実行させて、 CompleXcope で実際に体験してみれば簡単に理解できるでしょう。このプログラムは先程の interact.c と似ていますが、ナビゲーション機能を使っている点が違います。また、初期に CompleXcope の部屋が置かれている場所を表すワイヤフレームが表示されます。簡単のため真ん中ボタンで球を捕まえたり大きさを変えたりする機能は省略しました。このプログラムではジョイスティックを (前または後に) 倒せば、ワンドの向いている方向に向かって、 ナビゲーション座標系中を 視点 (CompleXcope が置かれている場所) が (前または後に) 動いて行きます。また、ジョイスティックを左右に倒せばナビゲーション座標系中を視線 (CompleXcope が置かれている方向) が回転します。 interact.c と比べて、 CAVEGetPosition() 等の関数の引き数が、 CAVE_WAND_NAV 等、 “_NAV” 付きのものに変わっていることに注意して下さい。

ナビゲーションを実現する CAVE ライブラリの関数を簡単に解説しておきましょう。 navigate() 関数中の、

```
void CAVENavTranslate(float x,float y,float z)
```

は、座標系の平行移動行列を設定します。また、

```
void CAVENavRot(float angle, char axis)
```

は座標系の回転行列を設定します。これらを呼んでいる関数 navigate() は applicatin computation process にあることに注意して下さい。

これらの行列に基づいた、実際の座標変換は draw_ball 関数 (display process) 内の

```
void CAVENavTransform(void)
```

で行われます。それではサンプルプログラムを見てみましょう。

navigate.c

```
/*
 *   navigate.c
\*
     A CompleXcope sample program.
          - Copied and changed from /usr/local/CAVE/src/ogl/navigate.c
          - A sample program for the CAVE navigation function.
          - A ball appears when wand left button is pressed at the wand tip.
          - The ball is destroyed when wand right button is pressed.
*/


#include <cave_ogl.h>
#include <GL/glu.h>

struct _balldata {
    float x,y,z;
    float radius;
    int   exist;
};

static GLuint mat0, boundary;

void init_gl(void);
void draw_ball(struct _balldata *);
void check_add(struct _balldata *);
struct _balldata * init_shmem(void);
void navigate(void);

main(int argc,char **argv) {
    struct _balldata  *ball;
    CAVEConfigure(&argc,argv,NULL);
    ball = init_shmem();
    CAVEInit();
    CAVEInitApplication(init_gl, 0);
    CAVEDisplay(draw_ball, 1, ball);
    while (!CAVEgetbutton(CAVE_ESCKEY)) {
        navigate();
        check_add(ball);
        sginap(1);
    }
    CAVEExit();
}


#define SPEED 5.0f  /* Max navigation speed in feet per second */

/* navigate - perform the navigation calculations. This checks the joystick
  state and uses that to move and rotate. The Y position of the joystick
  determines the speed of motion in the direction of the wand. The X position
```

```
  of the joystick determines the speed of rotation about the CAVE's Y axis.
  Joystick values in the range -.2 to .2 are ignored; this provides a dead
  zone to eliminate noise. The motion is scaled by dt, the time passed since
  the last call to navigate(), in order to maintain a smooth speed. */
void navigate(void) {
    float jx=CAVE_JOYSTICK_X,jy=CAVE_JOYSTICK_Y,dt,t;
    static float prevtime = 0;
    t = CAVEGetTime();
    dt = t - prevtime;
    prevtime = t;
    if (fabs(jy)>0.2)   {
        float wandFront[3];
        CAVEGetVector(CAVE_WAND_FRONT,wandFront);
        CAVENavTranslate(wandFront[0]*jy*SPEED*dt,
                         wandFront[1]*jy*SPEED*dt,
                         wandFront[2]*jy*SPEED*dt);
    }
    if (fabs(jx)>0.2)
        CAVENavRot(-jx*90.0f*dt,'y');
}

/************\
> check_add  <
\************/
void check_add(struct _balldata *ball) {
   float wandPos[3], wandFront[3];
   if (CAVEButtonChange(1) == -1) {         /* left button is released */
        CAVEGetPosition(CAVE_WAND_NAV, wandPos);          /* wand position */
        CAVEGetVector(CAVE_WAND_FRONT_NAV, wandFront);   /* wand direction */
        ball->x = wandPos[0] + wandFront[0]*0.5;
        ball->y = wandPos[1] + wandFront[1]*0.5;
        ball->z = wandPos[2] + wandFront[2]*0.5;
        ball->radius = 1;
        ball->exist = 1;
   }
   if (CAVEButtonChange(3) == 1)
        ball->exist = 0;
}

/*************\
> init_shmem  <
\*************/
struct _balldata * init_shmem(void) {
    struct _balldata *ball;
    ball = (struct _balldata *)CAVEMalloc(sizeof(struct _balldata));
    bzero(ball, sizeof(struct _balldata));
    return ball;
}


static GLUquadricObj *sphereObj;
```

```
/**********\
> init_gl  <
\**********/
void init_gl(void) {
    GLfloat diffuse0[]  = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat ambient0[]  = { 0.0, 0.3, 0.3, 1.0 };
    GLfloat specular0[] = { 0.4, 0.4, 0.4, 1.0 };

    GLfloat light_diffuse[]  = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat light_ambient[]  = { 0.3, 0.3, 0.3, 0.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glEnable(GL_LIGHT0);
    glClearColor(0., 0., 0., 0.);
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    sphereObj = gluNewQuadric();

    mat0 = glGenLists(1);
    glNewList(mat0, GL_COMPILE);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  diffuse0);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambient0);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular0);
        glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 100.0);
    glEndList();

    boundary = glGenLists(1);           /* boundary of the CAVE room */
    glNewList(boundary, GL_COMPILE);
        glColor3f(0,1,0);
        glBegin(GL_LINE_LOOP);
            glVertex3f( 5, 0,-5); glVertex3f( 5,10,-5);
            glVertex3f(-5,10,-5); glVertex3f(-5, 0,-5);
        glEnd();
        glColor3f(0,0,1);
        glBegin(GL_LINE_LOOP);
            glVertex3f( 5, 0, 5); glVertex3f( 5,10, 5);
            glVertex3f(-5,10, 5); glVertex3f(-5, 0, 5);
        glEnd();
        glColor3f(1,0,0);
        glBegin(GL_LINES);
            glVertex3f( 5, 0,-5); glVertex3f( 5, 0, 5);
            glVertex3f(-5, 0,-5); glVertex3f(-5, 0, 5);
            glVertex3f( 5,10,-5); glVertex3f( 5,10, 5);
            glVertex3f(-5,10,-5); glVertex3f(-5,10, 5);
        glEnd();
    glEndList();
}
```

```
/************\
> draw_ball  <
\************/
void draw_ball(struct _balldata *ball) {
     glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
     glEnable(GL_LIGHTING);

/* Apply the navigation transformation */
     CAVENavTransform();

     if (ball->exist) {
         glPushMatrix();
           glTranslatef(ball->x, ball->y, ball->z);
           glCallList(mat0);
           gluSphere(sphereObj, ball->radius, 32, 32);
         glPopMatrix();
     }
     glDisable(GL_LIGHTING);
     glCallList(boundary);
}
```

\ end of navigate.c /

3.24 CompleXcope プログラミングのまとめ

以上で CompleXcope プログラミングの解説は終わりです。 CAVE ライブラリは素晴らしく良く出来たライブラリです。この CAVE ライブラリのおかげで、ユーザは OpenGL の基本機能さえ知っていれば、簡単にバーチャルリアリティが作成できます。もちろん OpenGL や CAVE ライブラリには、このガイドでは説明しなかった様々な機能が用意されていますが、我々の目的であるシミュレーションデータの可視化・解析のためには、ここで説明した機能だけで十分です。実際、 CompleXcope の最初の大規模アプリケーションである Virtual LHD プログラムも、基本的な構造はこれまでに解説したサンプルプログラムと全く変わりません。従ってこれまでのサンプルプログラムから出発し、それぞれのプログラムで重点的に解説している個々の機能を組合せながら、少しずつ変更していけば、自分の研究に活かせる CompleXcope アプリケーションが簡単に開発できるでしょう。

このガイドでは、 OpenGL を使って３次元データを構成し、表示する方法について解説しました。実は、 CAVE では OpenGL よりもより高レベルのグラフィクスライブラリ Iris Performer を利用して仮想現実空間を構成することも可能です。 Performer は、もともとフライトシミュレータ用に開発された大変高度なグラフィクスツールキットで、これを使うと複雑な３次元物体のデータベースを比較的簡単に構成することが可能です。さらに、遠くにある物体のポリゴン数を自動的に減らしてレンダリングを高速化させたりする等、優れた機能も用意されています。例えば、部屋のたくさんある仮想的な家を作り、それぞれの部屋にもまた様々な家具、物があるような大規模で複雑な仮想世界を本格的に構成する場合には Performer の様なソフトウエアツールは必須となります。 (このような世界を OpenGL だけでコーディングすることは、考えただけでも恐ろしくなります。) しかし、我々の研究の上では、このような複雑な３次元物体のデータベース構成が要求されることはまずないでしょう。むしろ、単純な点や曲線、面を細かく、正確に構成することが可能な OpenGL による方法が我々の目的には最適だと思われます。

最後に、 CAVE ライブラリのマニュアルとして、このライブラリの開発者である Dave Pape によって書かれた *CAVE User's Guide* があります。この中には CAVE ライブラリの全ての関数のリストとその解説があります。このマニュアルは EVL の WEB ページ (http://www.evl.uic.edu/EVL/index.html) から最新のバージョンを得ることが出来ます。

付録 A

CAVE システム診断コマンド

時々 CompleXcope の映像がおかしく見える場合があります。よくある症状としては

1. ワンドや頭の位置をトラッカーが検出してくれない。
2. (どれか一つの) スクリーンの映像が二重に見える。あるいは立体感がない。
3. スクリーンの境目で映像が滑らかにつながらない。

等があります。このような場合、トラッカーやプロジェクター、またはワークステーションのグラフィックスシステムに異常が生じている可能性があります。このような場合のために、グラフィックス及びトラッカーシステムを診断するためのプログラムが用意されています。特に便利なのは次の二つです。

```
/usr/local/CAVE/bin/cavevars
```

このコマンドを打って CompleXcope の部屋の中に立ち、ワンドを動かしてみて下さい、ワンドの先に仮想 3 次元マーカーがくっついて動いているならばトラッカーは異常無しです。また各スクリーンにはその他の細かい情報も表示されているのでそれをヒントに異常をチェックして下さい。

```
/usr/local/CAVE/bin/nifs.universal
```

このコマンドは各スクリーンにテストパターンを表示させます。パターンが歪んでいないかチェックして下さい。また、 right と left と書かれた二つの青いベルトがあります。左目を閉じて、右目だけで見ている時には right のベルトが、左目では left のベルトが見えていることを確認して下さい。映像が二重に見える場合の多くはこれが逆になっています。これらのコマンドで異常が確認出来たら CompleXcope 管理者に連絡して下さい。どちらの診断コマンドもキーボードでエスケープキーを入力すれば終了します。

付録 B

OpenGL の画像をファイルに保存する方法

OpenGL で、 X Window 上に表示させた画像をファイルに落すには、 SGI のワークステーションの場合、 snapshot というプログラムでモニター画面のダンプを取るのが一番簡単です。

しかし、このような間接的な方法ではなく、作った映像を任意の画像フォーマット (例えば pict) のファイルに直接落したい場合があります。残念ながら OpenGL には pict であれ他のフォーマットであれ、映像を画像ファイルとして直接保存する機能はありません。しかし、フレームバッファの内容をメモリに取り出すことは可能です。それには次の関数を使います。

```
glReadPixels()
```

従ってこの機能を利用して各瞬間のフレームバッファの内容 (それが通常はウインドウに表示されています) を読みだし、そのデータを各画像フォーマットの規格に従って変換し、必要なヘッダをつけて保存すればいいのです。しかし、 pict 等はかなり複雑なイメージデータ形式なので直接 pict に書き出すことは大変です。

そこで、まず AVS のイメージデータ (*.x) にしてファイルに書き出し、これを何らかのファイルフォーマット変換コマンドで pict 等、望みのファイルフォーマットに変えるというのが最も簡単です。 AVS のイメージデータは始めの 2 バイトに画像の width と height のピクセル数の整数が書かれ、そのあとに各ピクセルの alpha, red, green, blue が次々に並べられた (おそらく) 最も単純なイメージデータフォーマットなのです。他にも同様に簡単なフォーマットがありますが、それらは普通ヘッダにマジックナンバー (フォーマットの種類をアプリケーションに知らせるための記号) が必要なため少々厄介です。

この様な方針で OpenGL プログラムから AVS のイメージデータを書き出すサンプルプログラム以下に示します。(基本は p.42で紹介した torus.c です。) 一番大事な部分は glReadPixels() を使っている toAVSx() の部分です。

```
/*
 *   saveimage.c
\*
     An OpenGL sample program.
        - Read the color buffer and make an AVS image file (*.x) from it
          by "glReadPixels" command.
        - An AVS image file named "picture.x" is automatically generated.
        - AVS image file has a simple data structure;
                    int  width
                    int  height
                    char alpha <-+  1st pixel
                    char red     |
                    char green   |
                    char blue  <-+
                    char alpha <-+  2nd pixel
```

```
                      char red     |
                      char green   |
                      char blue  <-+
                      char alpha <-+  3rd pixel
                      char red     |
                      char green   |
                      char blue  <-+
                         .
                         .
                         .
                         .
          - Here, it is supposed that the window size is 600x500 pixels
            and the position of its upper-left corner is (0,0).
          - A torus is drawn by the aux library.
          - With a blue lighting.
          - Perspective view by the glu library.
          - No animation.
*/

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdio.h>
#include "aux.h"

void initial(void) {
    GLfloat light_diffuse[]  = { 0.5, 0.5, 1.0, 1.0 };
    GLfloat light_ambient[]  = { 0.5, 0.5, 1.0, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);

    glClearColor(0.2, 0.2, 0.2, 1.0);
    }


void toAVSx(void) {

    FILE    *fp;
    char    *file1 = "./picture.x";
    /*  Here we supposed that
         (1) the window size is 600 x 500 pixels,
         (2) The position of the window's upper left corner is (0,0).
     */
```

```
    char    pic_red  [600*500];
    char    pic_green[600*500];
    char    pic_blue [600*500];
    char    pic_alpha[600*500];
    char    pic_rgba [600*500*4];  /* R,G,B and Alpha  */
    int     pic_size [2];          /* width and height */
    int     i;

    /*
        We cannot use the this (simpler) command;
           glReadPixels(0, 0, 600, 500, GL_RGBA, GL_UNSIGNED_BYTE, pic),
        since the each element of RGBA in the color buffer isn't
        necessarily the same as that in the AVS-X image file (which is
        "A-R-G-B").
    */

    glReadPixels(0, 0, 600, 500, GL_RED,   GL_UNSIGNED_BYTE, pic_red);
    glReadPixels(0, 0, 600, 500, GL_GREEN, GL_UNSIGNED_BYTE, pic_green);
    glReadPixels(0, 0, 600, 500, GL_BLUE,  GL_UNSIGNED_BYTE, pic_blue);
    glReadPixels(0, 0, 600, 500, GL_ALPHA, GL_UNSIGNED_BYTE, pic_alpha);

    pic_size[0] = 600;   /* width  in pixel */
    pic_size[1] = 500;   /* height in pixel */

    for (i=0; i<600*500; i++) {
        pic_rgba[4*i+0] = pic_alpha[i];
        pic_rgba[4*i+1] = pic_red  [i];
        pic_rgba[4*i+2] = pic_green[i];
        pic_rgba[4*i+3] = pic_blue [i];
        }

    fp = fopen(file1, "w");
    if (fp == NULL) {
        printf(" open error : %s\n", file1);
        exit(1);
        }
    fwrite(pic_size, sizeof(pic_size), 1, fp);
    fwrite(pic_rgba, sizeof(pic_rgba), 1, fp);
    fclose(fp);

}


void display (void) {
    GLfloat mat_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT,  mat_ambient);
```

```
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);

    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        auxSolidTorus(0.2,1.0);
    glPopMatrix();

    glFlush();

    toAVSx();           /****** This is it. ******/

}

void reshape(int width, int height) {
    GLdouble  ang = 60.0;
    GLdouble near = 3.0;
    GLdouble  far = 10.0;

    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (ang, (GLdouble)width/(GLdouble)height, near, far);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char **argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 600, 500);
    auxInitWindow (argv[0]);
    initial();
    auxReshapeFunc (reshape);
    auxMainLoop(display);
}
```

付録 C

C 言語入門 (diff f c)

CompleXcope でアプリケーションを作るには、 C 言語でプログラムを書く必要があります。それは CAVE ライブラリと OpenGL が C 言語の使用を前提としているからです。そこでこの Appendix では、 Fortran には詳しいけれども C 言語になじみが少ないという研究者のために、 C 言語の簡単な解説を行います。 Fortran と C では計算機言語としての基本的な体系にそれほど差がないので、ここではその違いに重点をおき、必要最小限と思われる部分だけを説明します。もちろん、この短いページで C 言語の全てを網羅することは不可能なので、必要に応じて市販の C のテキストを参照して下さい。

C.1 C 言語の基本的構造

C は “main” 関数と、そこから呼び出されるその他の関数の集まりで書かれます。そのファイルは必ず “.c” という拡張子を持ち、コンパイルには普通 “cc” コマンドを使います。コンパイルすると Fortran コンパイラと同様、 “a.out” が生成されます。以下に示す sample01.c は C の簡単なプログラムです。

```
/* sample01.c */

#include <stdio.h>
int  func1(int);

main()  {
     int i,j,k;
     i=3;
     j=i*i;
     printf("calling func1\n");
     k = func1(j);                  /* function call */
     printf("    k = %d\n",k);
}

int func1(int k)  {
     int m;
     printf("I am in func1\n");
     m = 3*k;
     return(m);
}
```

参考のためこれに対応する Fortran プログラムを以下に記します。

```
      i = 3
      j = i**2
      write(6,*)' calling nfunc1'
      k = nfunc1(j)
      write(6,*)'       k = ',k
      stop
      end
c
      function nfunc1(k)
c
      write(6,*)'I am in nfunc1'
      nfunc = 3*k
      return
      end
```

さて、 C に関して注意すべきポイントをいくつか記しましょう。

- 変数、関数名では、小文字と大文字を区別する
- 関数の中身は “{” と “}” で囲まれる
- 文法的一文はセミコロン “;” で終わる。コードの一行中に二文 (例えば `i=1; j=2;`) と書いても良い
- インデントは自由
- コメントは、 “/*” で始まり “*/” で終わる
- 入出力には標準入出力ライブラリ (stdio) を使う。このときプログラムの始めにstdio.h というファイルを include する必要がある

C.2 コメント

上に書いたように、コメントは “/*” と “*/” で囲みます。コメントの典型的な書き方を以下に記します。

```
    i = 3;     /*    comment 1    */

    /*                     *
     *                     *
     *    comment 2        *
     *                     *
     *                     */

     /*---------------------- 


           comment 3

       ---------------------*/
```

コメントを入れ子にすることはできないので注意して下さい。つまり下のように書くとエラーになります。

```
/*
 *
 *    This
```

C.5 関係演算子

二つの変数の大小関係等を調べる関係演算子は次の様に書きます。コメントは対応する Fortran 表記です。

```
>=          /*  .GE.  */
>           /*  .GT.  */
<=          /*  .LE.  */
<           /*  .LT.  */
==          /*  .EQ.  */
!=          /*  .NE.  */
```

C.6 if 文

if 文は表記方法が Fortran と少し違うだけです。

```
 if (y>=0) 文;

 if (y==1) {       /*   if (y==1)              */
    文 1;          /*   { 文 1;                 */
    文 2;          /*     文 2;      でもよい。 */
    文 3;          /*     文 3;                 */
 }                 /*   }                      */

 if(y!=0) {
     文 1;
     文 2;
 }
 else {
     文 1;
     文 2;
     文 3;
}
```

C.7 関数

Fortran のサブルーチンに相当するものは C にはありません。あるのは関数だけです。関数なので、原則として常に何かの値を返します。

```
/* sample03.c */

float sum(float,float);      /* プロトタイプ宣言 */

main() {
       float a;
       a=sum(2.0,3.0);
}

float sum(float a,float b) {
      return(a+b);
}
```

返り値が必要ない場合、つまり Fortran のサブルーチンと同じ事がしたい時には、形式的に “void” という値を返す関数を作ります。 “void” 値には特に意味がありません。

自分で定義した関数を使う時には関数の “プロトタイプ宣言” が必要です。プロトタイプ宣言とは、関数の定義や使用の前に、その関数の返り値の型と引数の型を明示的に宣言することです。これにより、関数呼び出し時の、型の不一致に起因する見つけにくいエラーをコンパイル時に検出できるようになります。

Fortran と違い、関数の呼び出し時に引数で渡されるのは “値そのもの” です。つまり、次の Fortran プログラム

```
          .
          .
        i=2
        call sub1(i)
          .
          .
        stop
        end
c
        subroutine  sub1(j)
          .
          .
        j=0
        return
        end
```

では、サブルーチン sub1 の呼び出し後、変数 `i` の値は 0 になっていますが、対応する C のプログラム

```
main() {
    .
    .
   i=2;
   sub1(i);
    .
    .
}

void sub1(int j) {
    .
    .
   j=0;
}
```

では、関数 sub1() の呼び出し後も i は 2 のままです。

Fortran とのもう一つの違いに、 C の関数は再帰呼び出しが可能だという事があります。つまり、 C の関数は自分自身を呼び出す事ができるわけです。(あまり使うことはありませんが。)

C.8 for 文

Fortran で最も重要な制御文である、いわゆる do 文は C では for 文で実現されます。下の例では関数 func() は 10 回 call されます。

```
for (i=0; i<10; i++) {      /*  do i = 1 , 10  */
    func();                 /*    in Fortran   */
}
```

for 文中 ( ) 内のセミコロンで区切られた 3 つの部分はそれぞれ

(初期化 ; 判定条件 ; 繰り返し処理)

となっています。

C.9 while 文

C では標準の FORTRAN77 にはない制御文 while() が使えます。

```
/* sample04.c */

#include <stdio.h>

void func1(int);
void func2(int);

main() {
    int n = 10;
    puts("----------------"); /* "puts" is a standard function */
    func1(n);
    puts("----------------");
    func2(n);                 /* the value n is not changed */
}

void func1(int i) {
    while (i>0) {
        printf(" int = %d\n",i);
        i--;                            /* i = i-1 */
    }
}

void func2(int i) {
    while (i>0)
      printf(" int = %d\n",i--);
}
```

上の func1() と func2() は次の形にも書けます。

```
void func3(int i) {
    while (i)        /* ( ) 内の値が 0 以外なら実行 */
        printf(" inn = %d\n",i--);
}
```

C.10 配列

C の配列の表記法は Fortran と少し異なります。

```
t[N]     /*  t(N) in Fortarn   */
```

Fortran では、t はt(1),t(2),t(3),...,t(N) という値をもちますが、 C では、

```
t[0],t[1],t[2],...,t[N-1]
```

のように、0 からN-1 までの値をとります。 2 次元配列の表記法は次の通りです。

```
        t[M][N]       /* 2次元配列  */
```

それぞれの添字は、やはり 0 からM-1 と 0 からN-1 まで動きます。配列t は全部でN*M 個のデータがあり、これがメモリ中に並んでいるわけですが、以下に述べるようにその順番が Fortran とは異なっているため注意が必要です。

例えば Fortran では、3×4 次元の配列 t(3,4) は

```
t(1,1)
t(2,1)
t(3,1)
t(1,2)
t(2,2)
   .
   .
   .
t(3,4)
```

の順番でメモリに格納されていますが、 C では配列 t[3][4] は

```
t[0][0]
t[0][1]
t[0][2]
t[0][3]
t[1][0]
   .
   .
   .
t[2][3]
```

というふうに置かれています。 Fortran で生成された多次元配列データを C 言語で読み込む時に (あるいはその逆の時にも) これは特に忘れてはならない点です。

3 次元以上の配列も使えて、例えば 3 次元の場合その表記はt[L][M][N] となります。

C.11 ポインタ

C 言語の特徴の一つは、メモリ上に格納されている変数のアドレス値を直接参照したり[1]、アドレス値を値としてもつ変数 (ポインタとよばれる) を扱うことができることです。ポインタをうまく使えばプログラムが簡潔になり、また実行効率もよくなります。

ある実数変数のアドレスを格納するポインタ ap は次のように宣言します。

```
        float *ap;
```

配列を多用する科学技術計算やグラフィクスプログラムでは、ポインタを使うと記述が非常に簡潔になる場合があります。それは配列の名前と、その配列が割り当てられているメモリ中の位置 (アドレス) に密接な関係があるからです。いまプログラム中で実数配列 temp[100] を使っているとしましょう。このとき計算機のメモリのどこかにこのデータは

```
temp[0], temp[1], temp[2], ..., temp[99]
```

[1] 変数x のアドレスは&x で得られます。

の順番で格納されています。(以下の説明では説明のためにアドレス値を具体的に書きますが、これは単に説明のためだけで、その値はマシンや実行毎に変わり得ます。) ひとつひとつの要素はそれぞれ (例えば)4 バイト分のメモリ空間を占めています。今、仮に `temp[0]` というデータがメモリ中の 4100 番の位置に格納されているとしましょう。すると

`temp[1], temp[2], ...`

は、それぞれ

`4104, 4108, ...`

のアドレスにあります。

ここで重要な事実は

> 配列の名前はポインタであり、その値には自動的に配列の先頭要素のアドレスが入っている。

ということです。上の例で言えば `temp` はポインタで、その値は 4100 です。プログラム中で `temp[100]` という配列は宣言しますが、`temp` という変数は特に明示的には宣言していません。しかし実は `temp` という変数は使えて、しかもその値は自動的に 4100 にセットされているのです。

このことを利用すれば、プログラム中で、ある関数 `func()` に配列 `temp` のデータを渡す必要がある時、`func()` に (100 個)×(4 バイト) のデータをまるごと渡すような大変なことをする必要はありません[2]。(配列の大きさが 100 であることを `func()` が知っているということを前提として、) `func()` には単に「必要なデータはメモリ中の4100 番地以降に置かれているよ。」と教えるだけで十分です。言い替えれば、`func()` にはポインタ`temp` を引数として渡せばよいわけです。つまり以下の形にします。

```
void func(float *);    /* プロトタイプ宣言。これを            */
                       /* void func(float *t)                  */
                       /* void func(float t[])                 */
                       /* void func(float t[100]) としてもよい。 */
main() {
   float temp[100];
      ...
   func(temp);
}

void func(float *t) {  /* これを void func(float t[]) としてもよい */
                       /*      void func(float t[100]) でもよい */
   ...
   a2 = t[2];
   a5 = t[5];
   ...
}
```

このとき `func()` には先頭のアドレス4100 という整数値だけを渡しています。受ける方の`func()` は先頭のアドレスを知っているので例えば`t[2]` というのが4108 番のアドレスに格納されているデータだということが分かるわけです。もちろんこのプログラムは`temp[0]` のアドレスが、実行するごとに毎回変わったとしても (4100 に限らず何であろうが) ちゃんと動きます。ポインタのこうした使い方は数十メガバイト以上にも及ぶ大きな配列を扱うサイエンティフィックビジュアリゼーションでは特に必須です。

[2]実は C ではこのように配列をまるごと渡すことは、最初から出来ないようになっています。

C.12 構造体

最後に C 言語の最も便利な機能の一つであり、 CompleXcope プログラミングでも多用する構造体について解説します。複数の変数を一まとめのグループとして扱いたい場合がよくあります。例えば磁力線のデータについて考えてみましょう。３次元空間の磁力線を例えば１０本、 VR 空間中に描きたいとします。磁力線の追跡は既になされており、必要なデータ

```
/*  j 本目の磁力線を構成する線素の数                */
/*  j 本目の磁力線の i 番目の線素の x 座標          */
/*  j 本目の磁力線の i 番目の線素の y 座標          */
/*  j 本目の磁力線の i 番目の線素の z 座標          */
/*  各線素の位置でのある物理量、例えば電流          */
/*   (これは例えば磁力線に色をつけるのに使う)       */
```

が全て揃っているとします。一般にプログラムでは、複数の磁力線を描く関数 `draw()` を用意し、それを呼ぶことにより 10 本の磁力線を描く様にするのが普通でしょう。問題はこのときの上記の多数のデータの扱い方です。 Fortran では、 (線素の最大数を 1000 として)

```
integer nten(10)        /* j 本目の磁力線を構成する線素の数        */
real    xpos(10,1000)   /* j 本目の磁力線の i 番目の線素の x 座標 */
real    ypos(10,1000)   /* j 本目の磁力線の i 番目の線素の y 座標 */
real    zpos(10,1000)   /* j 本目の磁力線の i 番目の線素の z 座標 */
real    curr(10,1000)   /* 各線素の各位置での電流                  */
```

という配列を宣言し、用意したサブルーチン draw の引数に上の全ての配列 (５個も！) を渡すという方法が普通だと思われます。

C ではデータを一まとめにした “構造体” `magline` を以下の様に定義することが可能です。

```
struct magline {
   int   nten;
   float xpos[1000];
   float ypos[1000];
   float zpos[1000];
   float curr[1000];
};
```

この構造体 `magline` は一本の磁力線に必要な情報を収める鋳型のようなもので、要するに int とか float と同じレベルの、変数の新しい “型” を定義したことになります。ただし、このままでは未だメモリ中に変数としての実体がありません。この構造体 `magline` を使い、次のようにして変数 (構造体変数) を`line_1` 宣言します。

```
struct magline line_1;
```

こうすれば変数 `line_1` には一本の磁力線の記述に必要な全てのデータを収めることができ、プログラム中で `line_1` は、磁力線そのものと同一視できる便利な変数です。構造体変数は配列にも出来るので１０本の磁力線を扱うこの例では

```
struct magline lines[10];
```

とすれば便利です。そして複数本の磁力線を描く関数 draw() が用意されているとすれば

```
draw(lines);
```

と、簡潔なプログラムを書くことが可能です。

付録 D

テクスチャの作成法

第 2.32章で説明したように、 OpenGL には特定のフォーマットの画像データからテクスチャを自動的に生成する機能がありません。そこでここでは、任意の画像データフォーマットから 256×256 ピクセルのテクスチャデータを生成する C シェルスクリプトを紹介します。このスクリプトの内部では本文 (p.61) で説明した理由から一度 AVS のイメージデータ (*.x) を経由してそこからテクスチャデータに変換しています (x2texture コマンド)。この変換プログラムのソースも紹介します。

D.1 任意の画像ファイルからテクスチャを生成するコマンド

ここでは画像処理プログラム imtools の一つ imscale を利用して任意の画像形式から 256×256 ピクセルの AVS の x 形式のファイルに変換し、そこからテクスチャを生成するコマンドを示します。もちろん imscale 以外にもデータフォーマット変換コマンドはありますので、そのどれを使っても構いません。大事な点は、中間ファイルである AVS のイメージデータ (*.x) の width と height がどちらも (64 以上の) 2 の冪乗でなければならない点です。(width と height が同じ値である必要はありません。)

mkTexture

```
#!/bin/csh -f
#
# mkTexture
#     To make a texture data with 256x256 pixels from
#     any image file, via AVS image file format.
#

if ( $2 == "" ) then
  echo usage: $0 file1.rgb keyword
  echo output: keyword.256x256
  exit
endif

/usr/sdsc_imtools/imtools/bin/imscale -xsize 256 -ysize 256 \
      -infile $1 -outfile /tmp/$2.x

x2texture /tmp/$2.x $2          # cc -o x2texure x2texture.c

rm /tmp/$2.x
```

\ end of mkTexture /

D.2 x2texture コマンドのソースコード

上の mkTexture コマンドで使われている x2texture コマンドのソースコードを以下に示します。

x2texture.c

```
/*
 *  x2texure.c
 * To make a texture data from an AVS image file.
 */

#include <stdio.h>

main(int argc,char **argv) {
        FILE *fpin,*fpout;
        int  i;
        char red, green, blue, alpha;
        char sizeinfo[20];
        int  nx,ny;

        if (argc<3) {
          fprintf(stderr," usage: %s file_in file_out\n", argv[0]);
          exit(8);
        }

        fpin =  fopen(argv[1],"r");
        if(fpin==NULL) {
            fprintf(stderr,"  infile open error.\n");
            exit(9);
        }
        fread((char *)&nx, sizeof(int), 1, fpin);
        fread((char *)&ny, sizeof(int), 1, fpin);
        sprintf(sizeinfo,".%dx%d\0",nx,ny);
        printf(" size = %s\n", sizeinfo);

        fpout = fopen((char *)strcat(argv[2],sizeinfo),"w");
        if(fpout==NULL) {
            fprintf(stderr,"  outfile open error.\n");
            exit(9);
        }

        for (i=0; i<nx*ny; i++) {
            fread((char *)&alpha,  sizeof(char), 1, fpin);
            fread((char *)&red,    sizeof(char), 1, fpin);
            fread((char *)&green,  sizeof(char), 1, fpin);
            fread((char *)&blue,   sizeof(char), 1, fpin);
            fwrite((char *)&red,   sizeof(char), 1, fpout);
            fwrite((char *)&green, sizeof(char), 1, fpout);
            fwrite((char *)&blue,  sizeof(char), 1, fpout);
        }
}
```

\ end of x2texture.c /

参考文献

[1] .J. Kilgard, *OpenGL Programming for the X Window System*, Addison-Wesley, 1996

[2] OpenGL Architecture Review Board, J. Neider, T. Davis and . Woo, *OpenGL^{TM} Programming Guide*, Adison-Wesley, 1993

[3] システムソフトエンジニアリング訳, *OpenGL^{TM} Programming Guide (日本語版)*, アジソンウェスレイジャパン, 1993

[4] OpenGL Architecture Review Board, J. Neider, T. Davis and . Woo, *OpenGL^{TM} Programming Guide, Second Edition*, Adison-Wesley, 1997

[5] システムソフトエンジニアリング訳, *OpenGL^{TM} Programming Guide, Second Edition (日本語版)*, アジソンウェスレイジャパン, 1997

[6] OpenGL Architecture Review Board, *OpenGL^{TM} Reference Manual, Second Edition*, Adison-Wesley, 1996

[7] 三浦憲二朗, *OpenGL^{TM} 3D* グラッフィクス入門, 浅倉書店, 1995