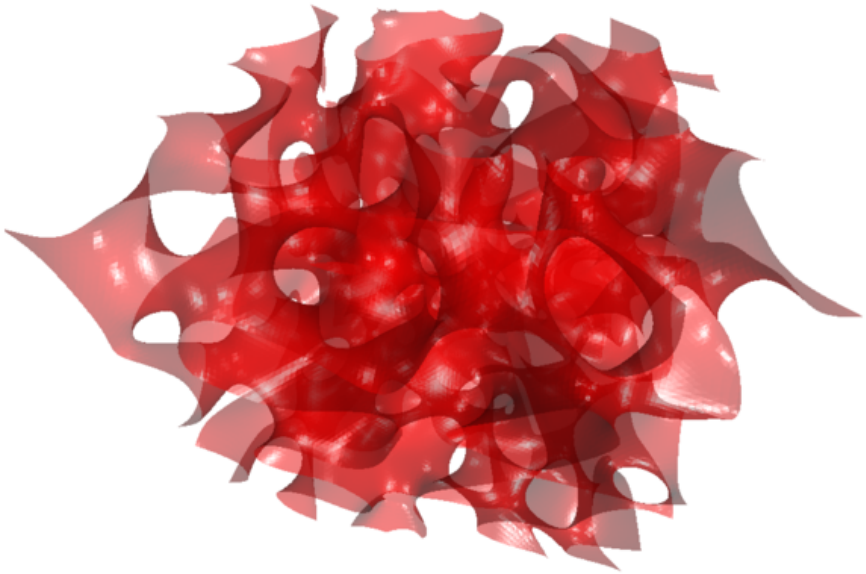
GLSC3D (Ver. 3.0.0) Manual

構想・制作・監督：秋山 正和
北海道大学 電子科学研究所



制作 (Ver. 2.2.1 以降)：舘入 数磨 *，須志田 隆道 †，小林　亮 ▽
* 北海道大学大学院 理学院 数学専攻, † 北海道大学 電子科学研究所,
▽ 広島大学　数理分子生命理学専攻

制作 (Ver. 2.2.1 まで)：平芳 悠人，岡本 守
北海道大学 理学部 数学科

2017 年 7 月 24 日

目次

1 はじめに

本章では GLSC の歴史や GLSC3D の開発に至る経緯などを詳しく紹介します．手っ取り早く使いたい方は本章を読み飛ばしください．

1.1 GLSC とその歴史

GLSC とは，Graphics Library for Scientific Computing の略で，科学計算の結果をディスプレイ上に表示するための簡単なグラフィックライブラリです[*1]．GLSC は小林 亮氏，高橋 大輔氏，中野 浩氏，松木平 淳太氏によって開発されました．当時（1980 年頃）はコンピュータ環境こそ整っていましたが，その計算結果を可視化する汎用のソフトウェアが殆どありませんでした．そのような時代に，「ユーザーに出来るだけ負担をかけずに簡単に数値計算結果を可視化したい」という目的で GLSC は開発されました．今日では，計算結果を可視化するソフトウェアは gnuplot [*2]を始め，Mathematica, MatLab[*3]など様々なものがあります．したがって「GLSC を使わなければ，可視化はできないか？」と問われれば答えは No となります．しかしながら，今でも GLSC にはコアなファン[*4]が沢山います．では，どのような点が GLSC は優れているのでしょうか？

[*1] “GLSC 小林 亮” と google 検索するか，`http://www-mmc.es.hokudai.ac.jp/~masakazu/` を見てください

[*2] `http://www.gnuplot.info/`

[*3] Mathematica, Matlab は数式処理や簡単な数値計算もできるので，可視化ソフトというよりは「可視化もできる」ソフトというべきですね．

[*4] もっとも授業などで最初に習った言語はその後を引きずるので，その影響で仕方なくという方もいるでしょうが...

1.2 GLSC の長所・短所

我々，数理科学者は現象と向き合い，数理モデルを作成しながら現象を理解することを生業としています．そして構成された数理モデルは，解析的に解くことは困難です．したがって，解を表示したりするためには，数値計算を行いのその結果を可視化するしかありません．この時多くの場合，数値計算のコードが先に出来上がります．次にそれをグラフィカルに表示するためにグラフィック用のコードを作成します．

ランダムウォークを例に取りましょう．次のページにランダムウォークを計算する C コードを貼りつけてあります．`r[i]` は `i` 番目の粒子の位置 (整数値) を示します．19 行目まで計算することによって，変数 `r[i]` の値が更新されます．計算の結果がうまく行われているかを確認するために，21 行目以降ではそれらを端末に表示するコードが示されています．試しに計算をしてみましょう．結果がグラフィカルに表示されていないので，わかりにくいと感じるはずです．そこで多くの場合，この結果を何らかの可視化ソフトを用い可視化します．先にも示したように，Free で手に入り，マニュアルも豊富に存在するということで，gnuplot は最近人気です．先の実行ファイルに対し，リダイレクション “>” を用いることで，計算結果を他のファイルに記録することができます．gnuplot にはこのデータを与えて可視化を行います．gnuplot は「与えられた座標に点を打つ」,「点どうしを先で結ぶ」,「鳥瞰図を作成する」など基本的な可視化を行うことができます．先の例では，端末での表示を見やすくするために計算結果に

```
---------------- Step = 0 ----------------
```

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #define     N       (10)
 4 #define     STEP    (100)
 5 int PlusMinus(void)
 6 {
 7     if(rand() < RAND_MAX / 2) return  1;
 8     else                      return -1;
 9 }
10 int main(void)
11 {
12     int         i, r[N],i_time;
13     //Initialize
14     for(i = 0;i < N;i ++) r[i] = 0;
15     //Time Loop
16     for(i_time = 0;i_time < STEP;i_time ++)
17     {
18         //Calc
19         for(i = 0;i < N;i ++) r[i] += PlusMinus();
20         //Print
21         printf("---------------- Step = %d ----------------\n",i_time);
22         for(i = 0;i < N;i ++) printf("%3d ",r[i]);
23         printf("\n");
24     }
25     return 0;
26 }
```

計算結果

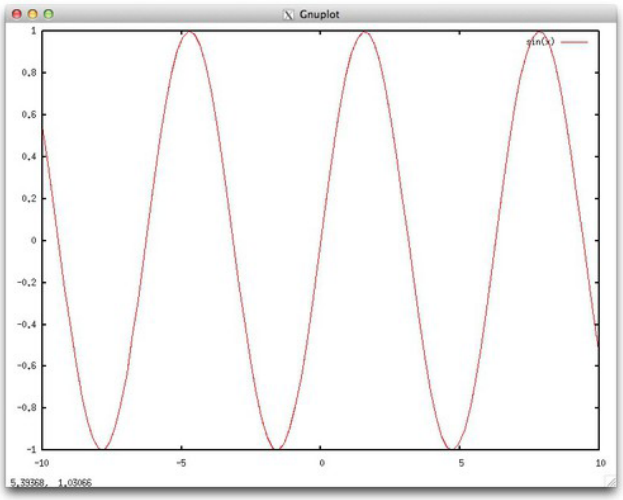
```
---------------- Step = 0 ----------------
  1   1  -1   1  -1   1   1  -1  -1  -1
---------------- Step = 1 ----------------
  2   0  -2   2   0   0   0   0   0   0
---------------- Step = 2 ----------------
                    :
                    :
---------------- Step = 98 ----------------
 -5 -15  13   1  -7  15  -3   1   7  -3
---------------- Step = 99 ----------------
 -4 -16  12   2  -8  14  -4   0   6  -2
```

などと表示をしていましたが，gnuplot ではこのような文字列は受け付けません．よって，多くの場合，gnuplot に適するように計算結果をフォーマットし直す必要があります．この例でもわかるように，数値計算のコードを書いた後に可視化のコードを書くということが，多くの現場で行われていることでしょう．

この方法には大きな欠点があります．それは「リアルタイムに可視化できていない」という問題です．先の例では，数値計算はほんの一瞬で終わりますが，中には長時間の計算を必要とする場合もあります．そのような場合，「計算と同時に可視化もできたらいいのに」と多くの人は思うことでしょう．もちろん，gnuplot では popen 関数を用いて，C 言語から直接 gnuplot を呼ぶこともでき，この目的を達成出来ます．しかしながら，数値計算のコードよりも可視化のコードの部分が肥大化するという問題がよく起こります．この問題も，できるだけコードが肥大化しないように，プログラマが適切に関数を作ればよいのですが，C 言語から gnuplot を簡単かつスマート呼べるようなツールはないようです[*5]．GLSC は C 言語から非常に簡単に呼ぶことができるように設計されており，数値計算をしながらリアルタイムに可視化を行うことが可能です．また，GLSC は内部で X Windowsy System[*6]を使用しており，描画も非常に高速である長所があります．

その一方で，GLSC には短所もあります．例えば，gnuplot などの汎用ツールは次のように端末に打つとグラフを出します．

```
gnuplot> plot sin(x)
```



グラフが出るだけでなく，軸，目盛り，関数の名前など様々な情報を付加してくれます．これは gnuplot が内部的に適切に処理をしてくれているお陰です．しかしながら，GLSC にはそのようなものはありません．そのような付加情報を出したければユーザー自身が直接コードを書かねばなりません．多くの方はこのような点を挙げ「GLSC はそのような機能がついていないのがやだ」，「自分で書くのが面倒だ」と批判的になります．確かにそのような側面はありますが，逆に言えば「なんでもできる」ということになります．やや脱線しますが，開発者の小林亮氏から次のような話を聞いたことがあります．

「GLSC は確かになんでもはやってくれない，でも自由度が高い分，「ああしたい」，「こうしたい」という痒い所に手が届くんだ」

なるほど，含蓄のあるお言葉です．

どんなツールにも短所と長所があります．ですから “GLSC でないとできない事” というのは，今日ないでしょう．しかしながら，GLSC は “数値計算のプロ” たちが，自身の経験から使いやすい形にまとめた関数の集大成となっています．つまりメインの数値計算のコードに “程よい程度の困難さで，ユーザーの思い描く自由なグラフィック発想を追加できる” という点が GLSC の長所なのです．

[*5] 2014.6.4 原稿作成時時点

[*6] http://www.x.org/

1.3 GLSC3D の開発まで

長所と短所の話をした後ですが，GLSC には致命的な問題がありました．それは 3 次元空間の描画を扱えない[*7]ということです．GLSC 開発当時，3 次元計算を行うことは当時のスパコンでも大変でしたし，ましてや数値計算と同時に 3 次元の可視化をする必要もありませんでした．しかしながら，今日ではそのような話は普通に聞きます．作者は博士論文作成時[*8]，非常に多くの 3 次元の数値計算を行っていましたが，GLSC では可視化ができず[*9]，泣く泣く他の描画ツールを使った覚えがあります．その描画ツールはまったく痒い所に手が届かず，苦い経験をしたものです．その後も様々な文献を読み漁ったり，人に聞いたりして，使いやすそうな 3 次元可視化ツールを模索しました．数年模索をしましたが，結論から言えばそのようなツールはない，もしくはあっても非常に高い[*10]ということでした．しかもそのようなソフトは，予想されたように，痒いところに手が届かなかったり，描画が非常に遅いという欠点もありました．“3 次元の良い描画ツールがないために，研究が滞る” ということを避けるためにも，何らかの手を打たなければならない．そこで，筆者は GLSC3D の開発に着手しました．

1.3.1 X Window System の限界

GLSC3D とはその名の通り，GLSC の 3 次元への拡張バージョンのことです．GLSC3D の基本的な思想や設計方針は GLSC と同じになるように設計されるべきです．このため，開発当初は新たな 3D の新関数を GLSC に追加する形で進められました[*11]．しかしながら，GLSC は内部的に X Window System を Call するため，新たに追加できる 3D 機能は X Window System の許す制限までということになります．X Window System は優れたライブラリですが，3 次元の描画関数が乏しく，この路線での拡張は原理的に不可能と判断しました．そこで，他の描画ライブラリの使用を検討しました．

1.3.2 OpenGL と GLUT の使用

3 次元の描画ライブラリとして有名なのは，やはり OpenGL(Open Graphics Library)[*12]でしょう．DirectX[*13]というライブラリも有名ですが，Windows 上でしか動かないので問題です．OpenGL は様々な環境で動作すること，描画が高速であること，そしてその名の通りオープンな API(Application Program Interface) であることから，GLSC3D の構築には最適です．

OpenGL はグラフィック専門の API なので，ウインドウを画面に出したり，マウスやキーボー

[*7] もちろん鳥瞰図や，等値面を描く関数はありますが...

[*8] 2010 年の冬頃

[*9] もちろん，3 次元データから 2 次元の断片データをいくつか作り，可視化をすることは出来ましたが...

[*10] AVS という有名な可視化ソフトがあります．教育向けの一番安いのでも 15 万円以上はします．Mathematica も 20 万円はします...

[*11] その名残が GLSC3.8 世代にて追加された `g_contln_3d` などです

[*12] `http://www.opengl.org/`

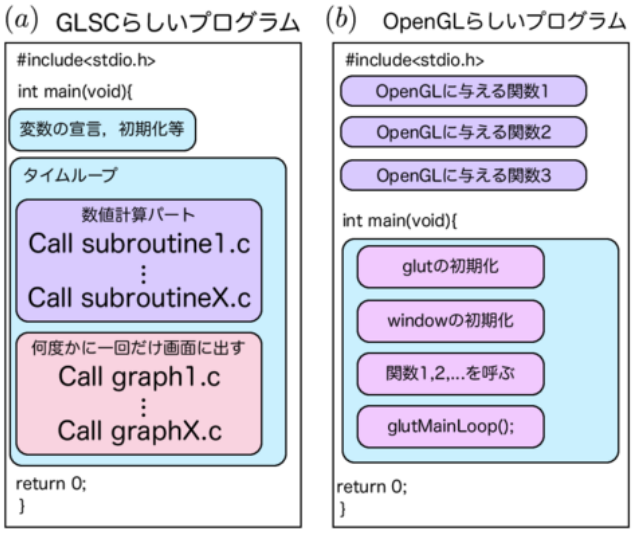
[*13] `http://ja.wikipedia.org/wiki/Microsoft_DirectX`

ドのようなデバイスから操作を受け付けるためには他のライブラリを使用する[*14]必要があります．それが GLUT(OpenGL Utility Toolkit)[*15]です．GLUT はこのような OS 間でのデバイスの違いの差も吸収してくれます．そこで OpengGL でインタラクティブなグラフィックプログラムを制作するといえば，大抵は GLUT も使用することになります．

実は GLUT は 1998 年に発表された Version 3.7 以降整備が行われなくなってしまいました．それでは困るということで，いくつかの団体 (OpenGLUT[*16]，FreeGLUT[*17]) が GLUT と互換性のある新たなライブラリを開発してきました．どちらの団体でも良かったのですが，freeGLUT のほうが作者の開発環境である MacOS と相性がよいので，GLSC3D の開発では freeGLUT を採用しました[*18][*19]．これにて，GLSC3D の開発に必要な土台は準備出来ました．あとは，どのように実装するかという問題が残されました．

1.3.3 GLSC らしさと OpenGL らしさ

先にも説明したように，GLSC は数値計算屋さんの作った描画ライブラリです．筆者は何かプロジェクトをスタートさせる時，数値計算のプログラムパートに精力を注ぎ，次にグラフィックパートを作成します[*20]．この時，数値計算を行いながら，可視化を行うわけですから，必然的にプログラムは (a) の様な構成になります．ここではこのようなプログラミング構成を GLSC らしいプログラミング構成と定義します[*21]．



さて，OpenGL のサンプルプログラムが載っている教科書等を一度読んでみてください．きっとプログラム構成は (b) のようになっていると思います．すなわち OpenGL や GLUT の関数の準備を行い，それらを main 関数内で呼ぶような構成です．そして一番のポイントは `glutMainLoop` 関数が呼ばれているということです．`glutMainLoop` はコールバック関数と呼ばれる特殊な関数です．この関数は「プログラムの実行中，何かが起こった時に」登録された関数を呼び出す（呼び戻す）ことができるような関数です．具体的にいえば，マウスなどで 3D グラフィックを回転させたい時がありますが，そのようなイベントが発生した時に (つまりマウスの動きを察した時)，OpenGL 関数に対して「再描画せよ」

[*14] このように書くと GLSC3D ではマウスやキーボードを使ってインタラクティブにグラフィックを行うことができそうですが，現時点 (2014.6.5) ではそのような関数は未実装です．その後 Ver2.2.1 で実装されました．

[*15] `http://www.opengl.org/resources/libraries/glut/`

[*16] `http://openglut.sourceforge.net/`

[*17] `http://freeglut.sourceforge.net/`

[*18] 何年か後に OpenGLUT 陣営が勝っていれば，それに合わせてプログラムを更新します...

[*19] FreeGLUT はいくつかの問題があることがわかり，Ver3.00 からは SDL に移行されました．

[*20] こう言うとグラフィックを疎かにしているようですが，グラフィックも綺麗にしないとインパクトのある講演はできませんからね... どっちも大事です．

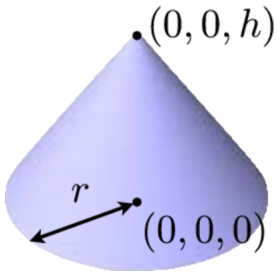
[*21] 「らしさ」を強要しているわけではありません．自由な発想に基づきプログラムをしてください．

と命令するのです．OpenGL はグラフィックをプログラミングするために適した API ですので，このようなプログラミング構成になるのは至極当然のことです．私は今まで仕事柄，多くの人のプログラムを読んできましたが，OpenGL らしいプログラミング構成の中に，何とか数値計算のコードを潜り込ませ，実行させているのをよく見かけます．しかしながら，上記のように整理すればわかるように，このような OpenGL らしいプログラミング構成と GLSC らしいプログラミング構成とは相容れない関係なのは明白です．OpenGL の参考文献に載ってるいるような「サンプルを数カ所変更すれば GLSC3D が完成！」というわけにはいかなかったからです．GLSC3D の開発は困難を極めました．

1.3.4 GLSC3D の赤ちゃんの誕生

GLSC らしいプログラミング構成となるように，どのように OpenGL の関数を呼び，GLSC3D の設計をすればよいのかは長らく未解決でした．ある時，研究室の樋口亮君ととある研究を行う中で，彼の OpenGL のプログラムをぼーっと見ていました．その時，このコードの一部に，これを解決するような構成方法を発見しました[*22]．筆者はこのコードと発見こそが GLSC3D 開発に大きく貢献したと思います．この場を借りて，樋口君には感謝の意を捧げます．

1.4 GLSC3D の開発



GLSC3D の根幹部分は先が見えたので，あとは実際の描画関数の設計を行うことを考えます．例えば，円錐を描画する関数 `DrawCone` を考えます．円錐の底の中心が原点 $(0,0,0)$，尖った先端を $(0,0,h)$，円錐の半径を r として円錐の描画関数を設計することを考えます．`DrawCone` の引数[*23]は h, r ということになります．当たり前ですが，この状態で描画される円錐はいつも底の中心が原点となっていますし，傾いた円錐というのも描画できません．このような時，3D オブジェクトに対して平行移動や回転などを施してやれば目的を達成できます．しかしながら `DrawCone` は呼ばれた時点で，底の中心が原点となるようなものしか描画できません．どうすればよいでしょうか．一つの戦略は `DrawCone` 関数の引数を増やし，円錐の底の座標や傾きなどを追加し，`DrawCone` の内部設計を変える方法です．でも，これは開発者側はちょっと面倒なプログラミングをせねばなりません．

二つ目の戦略は OpenGL に用意されている

`glPushMatrix`, `glPopMatrix`, `glTranslatef`, `glRotatef`

などの関数を使うことです．詳細は他書[*24]に委ねますが，簡単にいえば標準的な位置・方向の 3D オブジェクトに対して平行移動や回転などを施して，**描画時**に傾かせたり，移動させて見せたりする方法です．この方法はグラフィックスをメインとするようなプログラミング作成の際にはよく使われる手法です．しかしながら，GLSC3D はあくまでも数値計算をメインとするような人向

[*22] 樋口君，会社員になっても元気か？ついに開発したぞ！

[*23] 引数（ひきすう）と読みます．例えば関数 $f(x,y)$ なら x, y が引数です．とある学生がこの漢字を読めなかったもので...

[*24] OpenGL プログラミングガイド 原著第 5 版など

けのツールです．したがって，この手のやり方をユーザに強制するのはいかがなものか？ということになりました．そこで，GLSC3D では戦略 1 をとりました．戦略 1 は関数の引数が多くなり，開発に手間がかかるという欠点はありますが，3D オブジェクトの描画を任意の位置や方向に制御可能です．引数が多いなと感じる方もいるとは思いますが，試行錯誤の末，削りに削ったものです．もし，気に入らなければ，引数の少ない描画関数を自身で設計することも可能です．

このような経緯がありましたが，当面の問題は解決され GLSC3D の開発はスタートしました．

1.5 GLSC3D の設計哲学

GLSC3D は GLSC と同じ設計哲学を有します．それはおよそ以下の様な設計哲学です．

数値計算のコードを邪魔しない

先の述べたように，多くの場合数値シミュレーションのプログラムは，根幹をなす数値計算コードの部分が先に完成され，その後描画のパートが作成されます．したがって GLSC3D はもとの数値計算コードの部分に大きな変更を求めたり，後からの追加が困難であるように設計されてはなりません．したがって簡単な手続きで，GLSC3D のコードをもとのコードに埋め込めることができるように設計されるべきです．

関数の引数は必要最低限にせよ

関数の引数は多ければ多いほど豊かな動作が可能です．しかしながら，関数の引数が多いと「この関数の引数はなんだっけ？順番はどうだったっけ？」という問題が発生し，マニュアルとにらめっこしなければなりません．したがって関数の引数は，「その関数の動作に支障きたさない範囲内で最低限のレベルまで減らす」ことが求められます．例えば球を描画する場合はどのように工夫をしても，中心座標と半径は引数として必要でしょう．したがって `g_sphere_3D` 関数はこのような引数を必須としています．一方で `g_sphere` で球を描画していると，「ゴツゴツしているのでもう少しなめらかな球を描きたい，ワイヤーフレームで描きたい，etc...」といった欲望が出てくる場合もあります．そのような場合に備えて GLSC3D では，ほとんどすべての関数に添字 “`_core`” のついた上位版の関数を用意しています．例えば `g_sphere_3D_core` は球面の分割レベル，面の三角形分割のレベル，ワイヤフレームにするかどうかなどを指定することができます (詳細は関数の説明を御覧ください)．GLSC3D では一般に関数 `g_A`, `g_A_core` が存在する場合 `g_A` は内部的に `g_A_core` を Call しています．

だからと言って構造体などで包まない

構造体を使えば，一つの構造体を関数に渡せばあたかも多数の引数を渡したことと同等のことが実現できます．しかしながら，この方法も結局「この関数の構造体のメンバーはなんだっけ？型の名前はなんだっけ？」という問題が発生し，マニュアルとにらめっこしなければなりません．そのようなことから GLSC3D では構造体を通して引数を渡すことをしません．もし「引数が多くて嫌だな...」と思われる場合は，ユーザー自身で構造体を作成し GLSC3D の関数をラッピングしてご使用ください．[*25]

[*25] とは言っても 3 次元の計算ツールなので，座標ぐらいは構造体にしたい気分です．そのうちリリースするかもしれません

ユーザーが自由に関数を設計できる

先に g_sphere_3D を例にしましたが，例えば「球面の北極部分を赤く，南極部分を青く，さらに赤道部分は緑にしたい」という欲望が出た場合，g_sphere_3D_core ではもはや手におえません．GLSC3D は「ユーザーが自由に関数を設計できるよう」に設計されるべきです．したがって一番細かいレベルの目的を達成できるようなプリミティブな関数を提供すべきです．GLSC3D では点を打つ，線を描く，面を描く，文字を書くといったような基本的な関数群を揃えてありますので，それを組み合わせて使用することで，どんな関数でも設計することが可能です．これらの使用方法を熟読し自由な発想で描画を楽しんでください．

2 GLSC および GLSC3D Ver2.x からの変更点及び注意点

GLSC3D は GLSC の設計哲学を引き継ぎ構築されています．理想的には，関数の命名規則，引数などがシームレスに繋がり，ユーザには「どこが変わったかわからない」と思わせるように設計すべきです．しかしながら，それぞれの基盤ライブラリである OpenGL と X Window System は全く異なるものです．したがって，GLSC3D の開発は 0 から作り上げていく方式を取らざるを得ませんでした．そして開発中，どうしても変更せざるを得ないくつかの点が生じました．また，3 次元化されたこと及び新関数の導入によって，GLSC とは別の新たな注意点が発生しました．マニュアルの順番としては GLSC3D の関数を紹介した後に本章を置くべきですが，非常に重要な変更のため本章にてお知らせします．

2.1 GLSC からの変更点

2.1.1 Fortran 言語をサポートしていない

GLSC では Fortran をサポートしていましたが，GLSC3D ではサポートしていません．未来永劫に渡り，Fortran をサポートしないわけではありません．GLSC 同様に内部的にはすべてをポインタ渡しに Wrap するとで，Fortran から C を Call できますので，Fortran に強い方で Wrapper プログラムを書いてもいいよという方は筆者までお知らせください．

2.1.2 最終の描画スタイルの変更

3 次元オブジェクトを描画する際，OpenGL では描画命令をいくつかのまとまった単位になるまで保持し，一気に描画するというスタイルが取られます．また描画の最初と最後を明示的に指定しなければなりません (下記参照)．[*26]

OpenGL の描画スタイル (3 次元空間内で線分を描画する場合)

```
glBegin(GL_LINES);          /*線を描き始めることを指定*/
glVertex3f(0,0,0);          /*線分の端点を指定*/
glVertex3f(0,1,0);          /*線分の端点を指定*/
glEnd();                    /*線を描き終えたことを指定*/
glFlush();                  /*線を描く命令を実行*/
```

一方，GLSC で描画命令はその都度実行されます．したがって，描画の最初と最後を明示的に指定する必要はありません (下記参照)．

GLSC の描画スタイル (2 次元空間内で線分を描画する場合)

```
g_move(0,0);                /*線分の端点を指定*/
g_plot(0,1);                /*線分の端点を指定*/
```

[*26] この方法は OpenGL 3.1 以降では廃止されており Vertex Buffer と Vertex Array を使用する必要があります．

GLSC のほうが記述が楽で良さそうに思えますが，描画命令がその都度実行されるので，OpenGL に比べ（若干ですが）描画が遅くなったり，画面がちらついたりする可能性があります．[*27]．

GLSC3D では GLSC の哲学と OpenGL の良さを引き継いでいますので，次のように書くことができます．

GLSC3D の描画スタイル (3 次元空間内で線分を描画する)

```
g_move_3D(0,0,0);                    /*線分の端点を指定*/
g_plot_3D(0,1,0);                    /*線分の端点を指定*/
g_finish();                        /*バッファの描画命令を実行*/
```

最後の g_finish ですが，GLSC ではなかった記述です．g_finish はタイムループ内で描画関数が終わった時**最後に一度だけ**呼ぶ必要があります．この関数を呼ぶことで，今までバッファに溜まっていた描画命令が一気に実行されます．多くの場合，最終的な描画の後で画面を短い時間停止させます．GLSC ではそのような場合，g_sleep 関数を用いていました．GLSC3D では最終的な描画の後で g_finish を用いる必要がありますので，次のようにプログラミングすることが多くなるでしょう．これが大きな変更点です．

GLSC3D の描画スタイル

```
g_xxxx;                                  /*描画 1*/
g_xxxx;                                  /*描画 2*/
g_xxxx;                                  /*描画 3*/
.
.
.
g_finish();                            /*バッファの描画命令を実行*/
g_sleep(0.05);                         /*0.05 秒間停止する*/
```

[*27] OpenGL ではちらつきを抑えるためにダブルバッファリングという手法を用いることができます．GLSC3D はこの手法を用いています．

2.2 GLSC3D Ver2.x からの変更点

2.2.1 OpenGL の最新規格への追随

OpenGL は 1992 年に誕生しました．1990 年代，GPU は決められた描画アルゴリズムしか実行できませんでした．その後 GPU は劇的に進化し，現在はシェーダーと呼ばれる GPU 上で実行されるプログラムを用いてグラフィックスの表示を行うことが一般的となっております．

2008 年に OpenGL 3.0 が発表され，従来の固定機能に由来する大量の API が非推奨と宣言され，翌年の OpenGL 3.1 で削除されました．現在も古いバージョンの OpenGL は残されているため固定機能を使い続けることはできますが，将来にわたって存在が保証されているわけではありません．GLSC は Ver 3.0 で OpenGL の最新バージョンと互換性を保つこととしました．

2.2.2 ウインドウライブラリを FreeGLUT から SDL への変更

以前の GLSC3D では FreeGLUT を用いてウインドウを表示していました．

Mac OS X 上では FreeGLUT はネイティブに動作せず, 代わりに XQuartz を用いて X Window System 上で動作させる必要がありました．しかし XQuartz は

- XQuartz 上で利用できる OpenGL がバージョン 2.1 に限定されるため,
  非推奨な (OpenGL 3.1 以降廃止された) 描画方法しかできない
- ウインドウのサイズ変更・最大化ができない
- Retina Display に対応していないため，MacBook Pro などでの表示が美しくない
- 起動が遅い
- XQuartz 自体が将来にわたって存在するのかどうか怪しい

といった問題点があったため，使用するライブラリを FreeGLUT から SDL に変更しました．

2.2.3 フォント埋め込みの廃止, フォント指定関数の変更, 日本語などの Unicode 文字への対応

以前の GLSC3D では 4 種類の日本語フォントをライブラリに埋め込んでいましたが，ファイルサイズが肥大化するためこれを廃止しました．Ver 3.0 以降ではインストーラーが，GLSC3D に同梱されたフォント `Noto Sans CJK JP Regular` [*28] を

Mac OS X	`~/Library/Fonts/`
Linux	`/usr/share/fonts/opentype/noto/`
Windows	`C:/Windows/Fonts/`

に配置し，GLSC3D はここからデフォルトフォントを読み込みます．違うフォントを使いたい場合，`g_text_font_core` にフォントファイル名を指定してください．

[*28] Google Noto Fonts (https://www.google.com/get/noto/) にて配布されています．

埋め込みフォントを指定する関数 g_text_font を廃止しました．また，テキストのフォント指定 g_text_font_core とサイズ指定 g_text_size を分離しました．また，g_def_text 関数の font 引数を削除しました．

日本語や数学記号などの Unicode 文字に対応しました．これを用いる場合文字列のエンコードは必ず UTF-8 にしてください．gcc では文字列リテラルのエンコードはデフォルトで UTF-8 になります．お使いのコンパイラーのデフォルトが UTF-8 でない場合，C++11 の u8 プレフィックスを使用してみてください．(C++11 としてコンパイルする必要があります)

```
g_text_virtual_3D(0, 0, 0, u8"あいうえお");
```

2.2.4 ウィンドウのサイズ変更・最大化と Apple Retina Display の対応

Ver 3.0 以降ではウィンドウのサイズ変更・最大化と Apple Retina Display に対応しています．このため，GLSC3D は内部的にスケールファクターを保持しており，g_init(_core) に渡されたウインドウサイズに対する現在のサイズの比率 (の水平方向と垂直方向の値の小さいほう) を使用します．スケールの対象になるのは次の関数です．

- g_sel_scale
- g_marker_size, g_line_width, g_text_size
- g_text_standard
- g_input_state (マウス座標の逆変換)

Ver 3.0 以降では g_sel_scale を (最初に 1 回だけ呼ぶのではなく) ループ内で使用することをお勧めします．これによりウインドウサイズの変更に適切に対応できるようになります．

g_capture を用いる場合，ウインドウサイズの変更は行わないでください．

Apple Retina Display の対応はデフォルトでは無効になっています．
有効化する場合，g_enable_highdpi() を g_init(_core) の前に呼び出してください．
この場合，Retina Display で実行したときにはスケールファクターの初期値は 2 になります．
それ以外の場合，スケールファクターの初期値は 1 です．

2.2.5 ワイヤーフレームとサーフェイス塗りつぶしの引数の変更

以前の GLSC3D は各関数に WireFill パラメータが存在しており，G_WIRE=0(ワイヤーフレーム) または G_FILL=1(サーフェイスを塗りつぶす) を指定できました．

GLSC 3.0 では旧 GLSC に近づけるため，Wire パラメータと Fill パラメータに分離し，それぞれに G_YES=1 または G_NO=0 を指定するようにしました．

注意：この変更は主に旧 GLSC ユーザーのためのものです．Wire パラメータと Fill パラメータの両方に G_YES を指定した場合，線分と三角形が混合しパフォーマンスが低下します．できるだけワイヤーフレームの描画とサーフェイスの描画を分離してください．

2.2.6 2D の描画順の変更

以前の GLSC3D では 2D で先に描画したものが後に描画したものより優先して表示されていました．Ver 3.0 以降では後に描画したものが優先して表示されるように変更しました．3D モードの場合も奥行きが完全に一致したときには，同様の挙動になるように変更されています．

2.2.7 g_box_2D，g_box_3D(_core) 関数の仕様変更

以前の GLSC3D では長方形や直方体を描画する g_box_2D，g_box_3D(_core) 関数には，中心と大きさを指定していました．

Ver 3.0 では旧 GLSC と同様に両端の x,y (,z) 座標を指定するようにしました．以前のように中心と大きさを指定する場合，g_box_center_2D，g_box_center_3D(_core) を使用してください．

2.2.8 g_area_color 関数の仕様変更

以前の GLSC3D では g_area_color_2D 関数，g_area_color_3D 関数がありましたが，2D でも 3D でも面の色の指定法は同じ指定方法の方が簡便であると考え，この関数を廃止しました．新しい関数は g_area_color となりますので，ご注意ください．

2.2.9 マーカーの種類の追加

g_marker_type として正方形，円のほかに球を指定できるようにしました．引数には定数 G_MARKER_SQUARE=0，G_MARKER_CIRCLE=1，G_MARKER_SPHERE=2 を指定できます．

球マーカーは 3 次元で小さな球を大量に描画する目的に最適化されており高速に動作します．そのような目的では g_sphere_3D(_core) を使用しないでください．

どの種類のマーカーでも，サイズは g_marker_size によりピクセル単位の直径を指定できるほか，g_marker_radius により自由座標系での半径を指定することができます．

2.2.10 スムーズシェーディング用の関数，構造体を引数とする関数の追加

3D の三角形を描画するときに，陰影をスムーズにするために頂点法線を指定できる関数 g_triangle_3D_smooth(_core) を追加しました．

GLSC3D Ver 3.0 では色 (r,g,b,a) を保持する G_COLOR 構造体と座標 (x,y,z) を保持する G_VECTOR 構造体を公開しています．各成分は float 型です．

G_COLOR 構造体は以下の関数で使用することができます．

```
g_area_color_s
g_marker_color_s, g_line_color_s, g_text_color_s
```

G_VECTOR 構造体は以下の関数で使用することができます．

g_marker_s, g_move_s, g_plot_s

g_triangle_2D_s, g_triangle_3D_s, g_triangle_3D_core_s

g_triangle_3D_smooth_s, g_triangle_3D_smooth_core_s

2.2.11 その他

Ver 3.0 以降ではライトの最大数を 3 つに変更しました．これは 4 つ以上のライトは不要と判断してのことですが，要望がある方は本マニュアルの “おわりに” をご覧ください．

アンチエイリアシングを使用して描画することを指定できるようにしました．
アンチエイリアシングは線や面の境界で発生するジャギーを低減して滑らかに表示します．
詳細は g_set_antialiasing を参照してください．

プログラムを Esc キーで終了できるようにしました．
そのため，プログラムがキーボード入力として Esc を使用することはできません．

g_init_core に指定するウインドウの位置として，画面の中央に表示することを意味する G_WINDOW_CENTERED を指定できるようにしました．

多次元配列を受け取る関数

g_bird_view_3D, g_contln_2D, g_data_plot_3D, g_isosurface_3D

はプリプロセッサマクロにより対応する 1 次元配列版

g_bird_view_f_3D, g_contln_f_2D, g_data_plot_f_3D, g_isosurface_f_3D

に変換されるようにしました．今後は C++ でも多次元配列版を使うことができます．

2.3 GLSC3D の注意点

2.3.1 物体の透明化における注意点 (Ver1.x をお使いの方へ)

2 枚の板が重なっている状況を考えます．3 次元では奥行き情報があるので，それを元にどちらが手前にあるかを判定し (z バッファ法と呼ばれます) 描画を行います．GLSC では物体の透明化処理はできませんでしたが，GLSC3D では α ブレンディング法を用いて透明化処理を行うことができます．α ブレンディング法はオブジェクトとオブジェクトが重なるとき，視点に対して手前にあるオブジェクトの色を，奥にあるオブジェクトの色と混ぜ合わせ描画することによって，透明化処理を行います．この事からわかるように，α ブレンディング法は z バッファ法と同時に使うことは原理的に不可能です．そこで，α ブレンディング法を使用する際は，z バッファ法は OFF にし，視点に対して奥の方から先に 3D オブジェクトの描画を行う必要があります．GLSC3D では α ブレンディング法を使用する際に自動的に z バッファ法は OFF になり[*29]ますが，**視点に対して奥の方から先に 3D オブジェクトの描画行う**ことはプログラマの責任となります．

このように書くと，それを自動化できないのか？と思う方もいると思います．回答として「原理的には可能なはず，しかしながら，非常に困難」であるといえます．α ブレンディング法の肝は「視点に対して奥の方から先に 3D オブジェクトの描画行う」ことです．つまり視点に対してどのオブジェクトが奥にあるか？ということを自動的に判定せねばなりません．`g_finish` が呼ばれた瞬間にこの自動判定を行うことができれば，原理的には可能です．しかしながら，物体が何かの物体の手前あるもしくは奥にあるとはどういうこととして定義されるでしょうか？

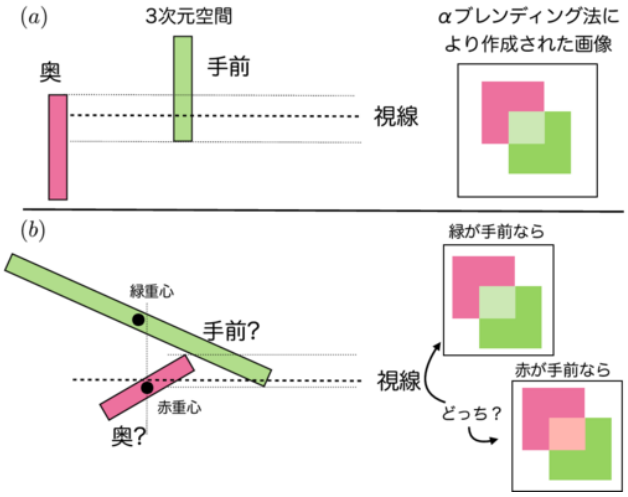


図 (a) では視線に対して，緑のオブジェクトが手前，赤が奥となっています．緑のオブジェクトを構成するどの頂点も，赤のオブジェクトを構成する頂点よりも手前にあることが計算できるので，自動的に奥行き判定をすることができ，α ブレンディング法を実行できます．

一方 (b) も同じ状況ですが，緑のオブジェクトを構成する頂点は，赤のオブジェクトを構成する頂点よりも手前，もしくは奥にあるため一概には判定できません．情報を減らして，それぞれの重心位置で比較することも考えられますが，この場合は「緑が奥にある」と判定されてしまい，意図した結果となりません．

α ブレンディング法を実行するために 3 次元オブジェクトの手前と奥を特徴づけるよい方法があれば良いのですが，筆者の調べた限りでは，そのような方法は無いようです．バージョン 1.x の

[*29] GLSC3D では α ブレンディング法の使用が終わると z バッファ法は自動的に ON になります．つまり α ブレンディング法が特別扱いにされています．

GLSC3D では，α ブレンディング法における本問題はプログラマの責任で対処するしかなさそうです．

2.3.2 物体の透明化における注意点 (Ver1.x 以降 をお使いの方へ)

しかしながら，これでは不便であるということで Ver2.0 からはある手続きを行うことで透明化処理を自動的に行わせることが可能となりました．(この処理を自動的に実行させるためには，GLSC3D の初期化関数 g_init_core にて g_enable_transparent_out = 1 としてください．) ですので，以降の説明は必ずしも読む必要はありませんが，高度なプログラミングを行いたい方は知識として知っておいてください．

関数を基本的なアイデアは簡単で以下のような手続きを踏んでいます．

- ステップ 1：任意の 3 次元オブジェクトを 3 角形分割する
- ステップ 2：3 角形分割された描画命令を溜め込む
- ステップ 3：3 角形の重心を求め，視点に対して遠い順番に並べ替え溜め込む
- ステップ 4：g_finish が呼ばれた段階で，視点に対して遠い 3 角形から描画を行う

ステップ 1：任意の 3 次元オブジェクトを 3 角形分割する

任意の 3 次元オブジェクトは 3 角形分割によって描画することが可能です．分割数は多いほど滑らかな表示になります．GLSC3D では面を描く場合，内部の関数では最終的にはすべて g_triangle_3D_core が Call されています．逆にユーザーが自身で関数を設計する場合も g_triangle_3D_core を用いてプログラミングを行ってください．用いない場合は透明化処理が適切に自動化されません．

ステップ 2：3 角形分割された描画命令を溜め込む

g_enable_transparent = 1 の場合，GLSC3D は描画命令を溜め込むバッファ (以降 TRIANGLE_BUFFER と呼びます) を自動的に確保します．デフォルトの TRIANGLE_BUFFER のサイズは 2^{20} 個の三角形を登録できるようなサイズ（大体 500MB に相当）です．ほとんどこのサイズで問題は起きないはずですが，使用中 too many triangles for triangle buffer などと表示される場合は関数 g_init_core にてバッファサイズ (TRIANGLE_BUFFER_SIZE) をさらに大きくしてください．

ステップ 3：3 角形の重心を求め，視点に対して遠い順番に並べ替え溜め込む

3 角形の重心を求めることは容易ですが，視点に対して遠い順番に並べ替える方法をどのように実装するかは難しい問題でした．GLSC3D では 3 角形の描画命令を TEMPORARY_TRIANGLE_BUFFER_SIZE の数だけクイックソートアルゴリズムを用いてソートし，テンポラリなバッファ (以降 TEMPORARY_TRIANGLE_BUFFER と呼びます) にスタックします．その後，マージソートアルゴリズムを用いて TRIANGLE_BUFFER にマージ＆ソートし情報をアップデートします．このように TRIANGLE_BUFFER は現時点の TRIANGLE_BUFFER と TEMPORARY_TRIANGLE_BUFFER から作られます．どのようなアルゴリズムを用いてソートを行うことが最も高速となるかは自明ではないため，様々なアルゴリズムを組み合わせて用いました．その結果，少なくとも筆者の環境ではこの方式が最も高速でした．デフォルトの設定では

TEMPORARY_TRIANGLE_BUFFER のサイズ TEMPORARY_TRIANGLE_BUFFER_SIZE は 2^{10} 個の三角形を登録できるようなサイズです．このサイズが極端に大きかったり，小さかったりすると描画が非常に遅くなることが確認されています．ただし，ここでいうサイズの大きさの問題は，描画するべき対象に依った問題となりますので，最適値はわかりません[*30]．多くの場合，デフォルト値で問題は生じないと思われますが，描画遅いなどの問題が生じた場合は関数 g_init_core にて TEMPORARY_TRIANGLE_BUFFER_SIZE を変更し最適値を見つけてください．

ステップ 4：g_finish が呼ばれた段階で，視点に対して遠い 3 角形から描画を行う

ステップ 3 までの処理で，TRIANGLE_BUFFER には視点から遠い順番で三角形の描画命令が登録されています．これらの三角形は g_finish 関数が呼ばれた時点で描画されます．

GLSC3D では GLSC と同様に標準座標系の中に複数の自由座標系を構築することができますが，このステップ 1~4 は複数の自由座標系において自動的に適用されます．したがって，**ユーザーは g_triangle_3D_core を用いて描画さえすれば，特に何も意識しなくても，正しく透明化処理を行うことができます**．ただし，透明化処理を行う際には別の注意も必要です．それは「3 次元オブジェクトの三角形分割は十分に細かくなければならない」という点です．ステップ 3 において，3 次元オブジェクトを大きな三角形で分割すると，分割が荒いことが影響し視点に対する奥行き判定の際に，ミスが生じます．このことを防ぐためには，3 次元オブジェクトの三角形分割は十分に細かくする必要があります．例えば，g_sphere という関数は与えられ点を中心に，与えられた半径の球を描画する関数です．この関数にはその上位関数として g_sphere_core が存在しますが，その引数として int DivideLevel があります．この DivideLevel を 0 として指定する場合は 1 つの 3 角形を描きますが，1 を指定すると 4 個の 3 角形，2 を指定すると 16 個の 3 角形という具合に親 3 角形をさらに小さな子三角形へと分解し描画します．このような機能はほぼすべての描画関数に対して存在しますので，有効に活用してください．

透明化処理を行う際は，この点に注意いただくとともに，もし，さらに良いアイデアをお持ちの方がいましたら教えてください．

[*30] 将来的には描画の時間を内部的に取得し，最適値を漸近的に求めるようにするかもしれません

3　動作環境とその構築

GLSC3D Ver. 3.0.0 以降では，依存ライブラリとインストール方法が変更されています．必ずお読みください．

3.1　動作環境

GLSC3D Ver. 3.0.0 以降では，OpenGL，SDL，libPNG，FreeType，Math ライブラリに依存します．OpenGL，SDL は GLSC3D の根幹部分です．PNG ライブラリは画面をキャプチャするために必要です[*31]．C 言語の Math ライブラリは描画関数内で数学関数を用いているために必要です．

頂点データを保持用に，9MB のメモリー領域を使用します．これは 2^{16} 個の三角形を一度に GPU に転送できるサイズです．

3.2　動作環境の構築

GLSC3D Ver. 3.0.0 以降では，GLSC3D をインストールためのシェルスクリプトを OS 毎に用意しています．`http://www-mmc.es.hokudai.ac.jp/~masakazu/` の公開ソフトウェアから対応するスクリプトをダウンロードし，実行することによってインストールが行われます．

3.2.1　Mac OS X の場合

(1) Xcode のインストール：

Mac OS X は Xcode を導入しなければ，様々な開発環境がインストールされません．まず Xcode をインストールしてください．App Store で Xcode を検索すればトップに表示されます．

(2) Mac OS X 用シェルスクリプト：

Script_on_mac.zip を解凍すると以下の実行ファイルが入手できます：

1_Install_macports_mac
2_Install_dependency_library_mac
3_Test_GLSC3D_on_mac
4_Install_GLSC3D_on_your_mac

(3) 1_Install_macports_mac の実行：

1 つ目のスクリプトでは，MacPorts がインストールされているかを確認します．以下のように実行すると，右図のようにブラウザで `https://www.macports.org/`のページが開くので，インストールしていない場合はダウンロー

[*31] `http://www.libpng.org/pub/png/libpng.html`

ドおよびインストールを行って下さい．

スクリプトの実行

```
$: ./1_Install_macports_mac
```

無事にインストールされたら，次のステップに進んで下さい．

(4) 2_Install_dependency_library_mac の実行：

2 つ目のスクリプトでは，GLSC3D が依存しているライブラリで Mac OS X に必要なもの (cmake, libPNG, libsdl2) を導入します．

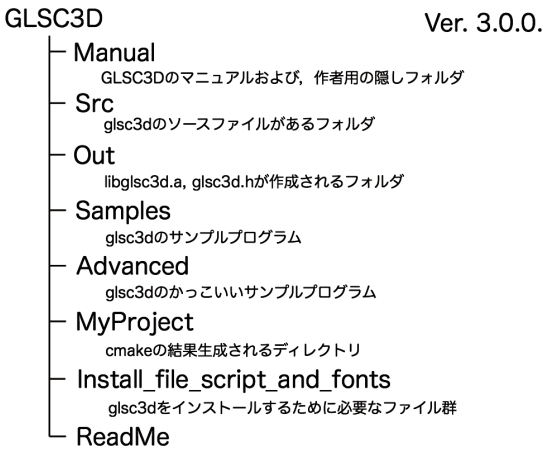
スクリプトの実行

```
$: ./2_Install_dependency_library_mac
```

無事にインストールされたら，次のステップに進んで下さい．

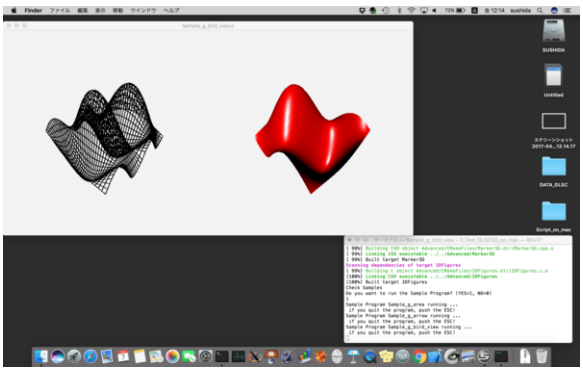
(5) 3_Test_GLSC3D_on_mac の実行：

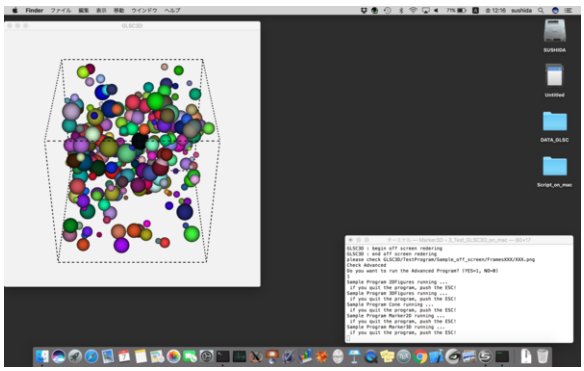
3 つ目のスクリプトでは，はじめに$HOME に GLSC3D_Working_Directory というディレクトリが作成され，GLSC3D_Working_Directory に https://github.com/GLSC3DProject/GLSC3D から最新版の GLSC3D がダウンロードされます．GLSC3D_Working_Directory 内のファイルは下図のようになります．また，フォントファイルがシステムにない場合は$HOME/Library/Fonts にインストールされます．次に，全てのサンプルプログラムとアドバンスドプログラムが実行できるかを確認します．サンプルプログラムとアドバンスドプログラムは実行時間がすこしかかるので，No を選択してスキップすることもできます．各プログラムは esc キーを押すと終了します．



スクリプトの実行

```
$: ./3_Test_GLSC3D_on_mac
```





無事にインストールされたら，次のステップに進んで下さい．

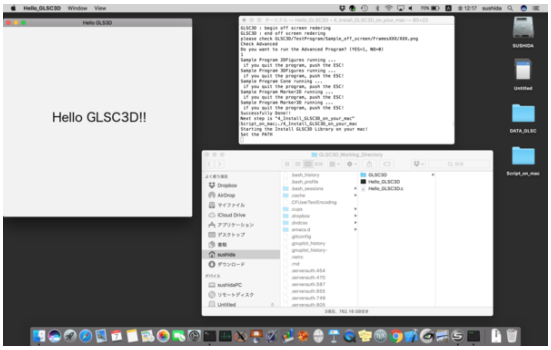
(6) 4_Install_GLSC3D_on_your_mac の実行：

4 つ目のスクリプトでは，はじめに$HOME 直下に 3 つのフォルダ bin，include，lib を作成し，ccg を bin に，glsc3d_3.h，glsc3d_3_math.h を include に，libglsc3d_3.a を lib に，Hello_GLSC3D.c を GLSC3D_Working_Directory の直下に保存します．次に，bin，include にパスが通っているかの確認をします．

スクリプトの実行

```
$: ./4_Install_GLSC3D_on_your_mac
```

パスが通っていない場合はメッセージに従って，下記のように使用しているシェル (.bashrc or .profile など) を編集して下さい．



パスの通し方

```
export PATH=$PATH:$HOME/bin
export PATH=$PATH:$HOME/include
```

無事にパスが通っていれば，bin 内の ccg コマンドによって自動的にサンプルプログラム Hello_GLSC3D.c がコンパイルされ，実行されます．(右図のように実行画面が表示されれば正しいです．) 実行がうまく行った場合は，システムに正しく GLSC3D がインストールされたことになります．

3.2.2 Ubuntu の場合

(1) Ubuntu 用シェルスクリプト：

Script_on_ubuntu.zip を解凍すると以下の実行ファイルが入手できます：

1_Install_dependency_library_ubuntu
2_Test_GLSC3D_on_ubuntu
3_Install_GLSC3D_on_your_ubuntu

(2) 1_Install_dependency_library_ubuntu の実行：

1 つ目のスクリプトでは，GLSC3D が依存しているライブラリで Ubuntu に必要なもの (git, cmake, libPNG, libsdl2, freetype) を導入します．

スクリプトの実行

```
$: ./2_Install_dependency_library_ubuntu
```

無事にインストールされたら，次のステップに進んで下さい．

(3) 2_Test_GLSC3D_on_ubuntu の実行：

2 つ目のスクリプトでは，はじめに$HOME に GLSC3D_Working_Directory というディレクトリが作成され，GLSC3D_Working_Directory に https://github.com/GLSC3DProject/GLSC3D

から最新版の GLSC3D がダウンロードされます．また，フォントファイルがシステムにない場合は/usr/share/fonts/opentype/noto にインストールされます．次に，全てのサンプルプログラムとアドバンスドプログラムが実行できるかを確認します．サンプルプログラムとアドバンスドプログラムは実行時間がすこしかかるので，No を選択してスキップすることもできます．実行結果は Mac OS X と同様です．

スクリプトの実行

```
$: ./2_Test_GLSC3D_on_ubuntu
```

無事にインストールされたら，次のステップに進んで下さい．

(4) 3_Install_GLSC3D_on_your_ubuntu の実行：

4 つ目のスクリプトでは，はじめに$HOME 直下に 3 つのフォルダ bin，include，lib を作成し，ccg を bin に，glsc3d_3.h，glsc3d_3_math.h を include に，libglsc3d_3.a を lib に，Hello_GLSC3D.c を GLSC3D_Working_Directory の直下に保存します．次に，bin，include にパスが通っているかの確認をします．

スクリプトの実行

```
$: ./3_Install_GLSC3D_on_your_ubuntu
```

パスが通っていない場合はメッセージに従って，下記のようにシェルを編集して下さい．

パスの通し方: 使用しているシェル (.bashrc or .profile など) において書きを追加します．

```
export PATH=$PATH:$HOME/bin
export PATH=$PATH:$HOME/include
```

無事にパスが通っていれば，bin 内の ccg コマンドによって自動的にサンプルプログラム Hello_GLSC3D.c がコンパイルされ，実行されます．実行結果は Mac OS X と同様です．実行がうまく行った場合は，システムに正しく GLSC3D がインストールされたことになります．

3.2.3 CentOS の場合

CentOS でのインストールでは，glxinfo にて OpenGL のバージョンが 4.1 以降であることと libpng および font に対するパスの指定が正しくできていれば動作する可能性があります．ライブラリのバージョンが Ubuntu に比べて古いので動作の保証ができません．GLSC3D Ver. 2.1.1 を使用されることをお勧めします．

3.2.4 Windows の場合

32 ビット版 Windows には対応しておりません．64 ビット版 Windows をご使用ください．

(1) Visual Studio のインストール：

GLSC3D を使用するには Visual Studio 2017 または 2015 が必要です．Community Edition は無料で使用することができます．インストールされていない方は

`https://www.visualstudio.com`

からダウンロードしてください．
インストールの際は C++ によるデスクトップ開発にチェックを入れてください．

(2) GLSC3D のダウンロード：

Github から GLSC3D のバイナリをダウンロードしてください．

`https://github.com/GLSC3DProject/GLSC3D/releases`

から `GLSC3D_3.zip` をダウンロードして展開します．

生成された `GLSC3D_3` フォルダを好きな場所に移動します (インストール先になります)．

(3) パスの設定：

`GLSC3D_3` フォルダ内にある `Bin` フォルダの場所を環境変数 `Path` に追加します．

1. `Bin` フォルダのパスをコピーします．
2. エクスプローラーの左側にあるツリーで “PC” を右クリックして “プロパティ (R)” を選択します．“システム” ウインドウが表示されます．
3. 左側にある ‘システムの詳細設定’ を選択します．
4. “環境変数 (N)” をクリックします．
5. ユーザー環境変数の `Path` をダブルクリックします．
6. “新規 (N)” をクリックします．
7. 入力欄に `Bin` フォルダのパスを貼り付けします．

(4) フォントのインストール：

フォント “Noto Sans CJK JP” をインストールします．

同梱された `NotoSansCJKjp-Regular.otf` を右クリックして “インストール (I)” を選択します．インストールが終わったらこのファイルは削除してかまいません．

以上で GLSC3D のインストールは終わりです．次のページでコンパイル方法を説明します．

(5) プログラムのコンパイル：

Bin フォルダにある ccg.bat を使用してコンパイルすることができます．例えば，Hello_GLSC3D.c をコンパイルするには，それを含むフォルダをコマンドプロンプトで開き

```
ccg Hello_GLSC3D.c
```

とすると Hello_GLSC3D.c をコンパイルすることができます．

Visual Studio のプロジェクトから使用するときには，プロジェクトのプロパティからインクルードディレクトリとリンクライブラリを指定してください．Lib フォルダにある全てのファイルのほか，opengl32.lib，user32.lib をリンクする必要があります．

(6) サンプルプログラムの実行：

Samples にある RunAllSamples.bat を実行すれば，サンプルプログラムやアドバンスドプログラムをコンパイルして実行することができます (セキュリティーの警告が表示されることがあります)．実行するとサンプルプログラムとアドバンスドプログラムで各 1 回ずつ確認メッセージが表示されるので，Y/N(小文字でも可) で答えてください．

また，Clean.bat を実行すれば，RunAllSamples.bat の生成物である*.obj，*.exe ファイルと Frames.*フォルダーを一括で削除することができます．

(7) Windows 特有の注意点：

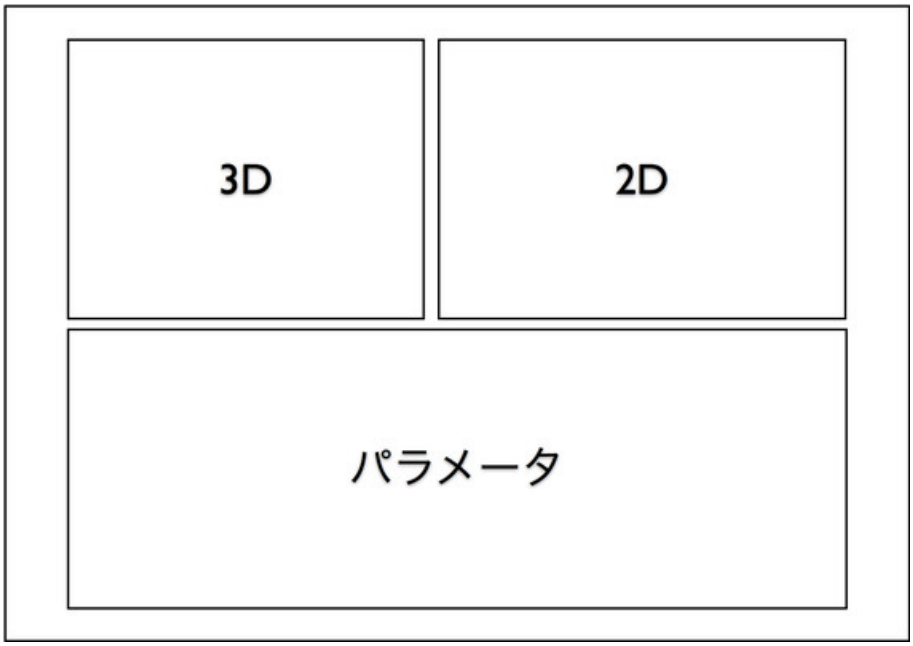
Visual Studio のコンパイラにおいて，文字列リテラルのエンコードはデフォルトでは Shift-JIS になります．GLSC3D は UTF-8 以外には対応していないため，ASCII 以外の文字 (日本語など) を表示するときは必ずプログラムの先頭に

```
#pragma execution_character_set("UTF-8")
```

と記述してください．(Sample_g_text.c に例があります．)

4　GLSC3D の関数

GLSC3D は GLSC と同様に一枚のキャンバスに対して，いくつかの絵を描画するようなソフトであると考えてください．適切な関数を用い，キャンバスをいくつかのエリアに区切り，それぞれのエリアに描きたいもの（3 次元，2 次元，テキスト等）を描画することができます．



GLSC3D を利用する際，多くの場合基本的な枠組みは以下のような形態となるでしょう．ここでは我々が用意した関数それぞれについての説明とその実行例も記載しておりますので，個々の関数の使用方法はそちらをご覧ください．なお，具体的な使用例はサンプルプログラムがありますので，そちらをご覧ください．

基本的な枠組み

```
#include <...>
...
#include <glsc3d_3.h>

int main()
{
    g_init(...); //用紙の設定
    g_def_scale_3D(0,...); //0 番の自由座標系の定義
    g_def_scale_3D(1,...); //1 番の自由座標系の定義
    ...

/****************************************/
                数値計算
/****************************************/

    g_cls(); //用紙を背景色で塗りつぶす

    //自由座標系を選択→属性を指定→描画関数で描画.
    g_sel_scale(0); //0 番の自由座標系を選択（以下，0 番の自由座標系に描かれる．）
    g_area_color(...); //塗りつぶしの色を指定
    g_box_3D(...); //3 次元空間に box を描画
    ...

    //自由座標系を選択→属性を指定→描画関数で描画.
    g_sel_scale(1); //1 番の自由座標系を選択（以下，1 番の自由座標系に描かれる．）
    g_area_color(...); //塗りつぶしの色を指定
    g_sphere_3D(...); //3 次元空間に sphere を描画
    ...

    g_finish(); //描画する
    return 0;
}
```

ここで GLSC3D を使いこなすために必要な幾つかの概念について解説します.

描画ウィンドウと標準座標系

すでに述べたように，GLSC3D においては，ディスプレイ上に描画ウィンドウを開いて，その上にいくつかの描画エリアを定め，それぞれの描画エリアにグラフィクスを表示します．標準座標系は描画ウィンドウ上に定義される座標系で，描画エリアを定義するのに使われます[*32]．

標準座標系は，描画ウィンドウ左上のコーナーを $(0.0, 0.0)$ とするピクセル単位[*33]の座標系です．標準座標系では GLSC2D と同様，標準 x 座標はウィンドウ左から右への方向，標準 y 座標はウィンドウ上から下への方向をそれぞれ正の方向としています．

描画オブジェクト空間と固定座標系

3 次元グラフィクスにおいて，描画されるオブジェクトが置かれる 3 次元ユークリッド空間を描画オブジェクト空間と呼びます．そして，この空間における通常の直交座標系を固定座標系と呼びます[*34]．ここでいう「通常」の意味は，この座標系における 3 つの基本ベクトル $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ の長さが全て 1 であることと，これらが右手系をなすことを意味しています．

この固定座標系を用いて描画オブジェクトを記述することももちろんできますが，GLSC3D では以下に述べる自由座標系を用いて描画オブジェクトを記述します．

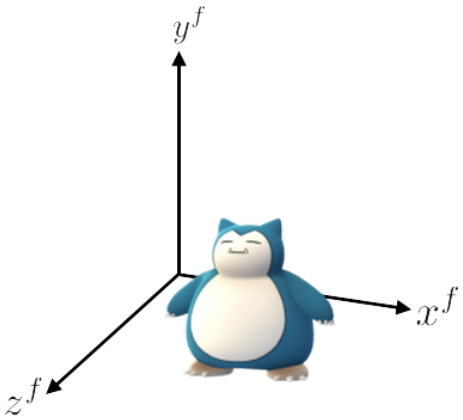


図 1　描画オブジェクト空間と固定座標系 (x^f, y^f, z^f)．この座標系は等縮尺で右手系です．なお，この図では y^f 軸が上を向いているように見えますが，描画オブジェクト空間自体には上下という概念はありません．オブジェクトをどう見るかを決める段階で上下が生じます．(p.** 参照)

自由座標系

例えば $z = f(x, y)$ のグラフの鳥瞰図を描く場合を考えてみますと，少なくとも変数 z の値はユークリッド空間における z 座標には対応していない場合がほとんどです（$f(x, y)$ が位置 (x, y)

[*32] 2 次元グラフィクスにおいては，後述の固定座標の役割を兼務します．

[*33] MacBookPro などのような Retina Display を搭載する機種は，必ずしも表示される映像は指定したピクセルとはなりません．従って，標準座標の数値は，厳密にはピクセルを指しません．ただし，多くの環境では標準座標の数値=ピクセルと考えてもらって良いです．

[*34] GLSC2D では描画オブジェクト空間と描画ウィンドウを同一視することができるため，わざわざ固定座標系という概念を持ち出さなくても，標準座標系だけで話が完結していましたが，GLSC3D 3.0.0 以降では，本概念が必須となりました．

における温度や気圧を表している場合など）．もちろん変数 x, y においても同様です（$f(x, y)$ が身長 x 体重 y の頻度分布である場合など）． このような場合，固定座標を用いてそのままの変数値で描画すると，とんでもなく背が高いグラフや平べったいグラフになってしまいかねません．そこで，描画オブジェクト空間に対し，固定座標とは別の自由なスケールを持った自由座標系を導入して，その座標系を用いて描画オブジェクトを生成する必要があります．

この自由座標系は次のようにして定義します．まず，描画オブジェクト空間に固定座標 (x^f, y^f, z^f) を用いて直方体 $[x_0^f, x_1^f] \times [y_0^f, y_1^f] \times [z_0^f, z_1^f]$ を定めます．これは基本的に描画オブジェクトを収納したい領域です．そして，その直方体が自由座標 (x, y, z) では $[x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$ となるように両座標系の対応を定めるのです．式で書けばその対応は以下のようになります．

$$\frac{x - x_0}{x_1 - x_0} = \frac{x^f - x_0^f}{x_1^f - x_0^f}, \qquad \frac{y - y_0}{y_1 - y_0} = \frac{y^f - y_0^f}{y_1^f - y_0^f}, \qquad \frac{z - z_0}{z_1 - z_0} = \frac{z^f - z_0^f}{z_1^f - z_0^f}$$

この自由座標系の定義は `g_def_scale_3D` という重要な関数を用いて行います（p.** ）．

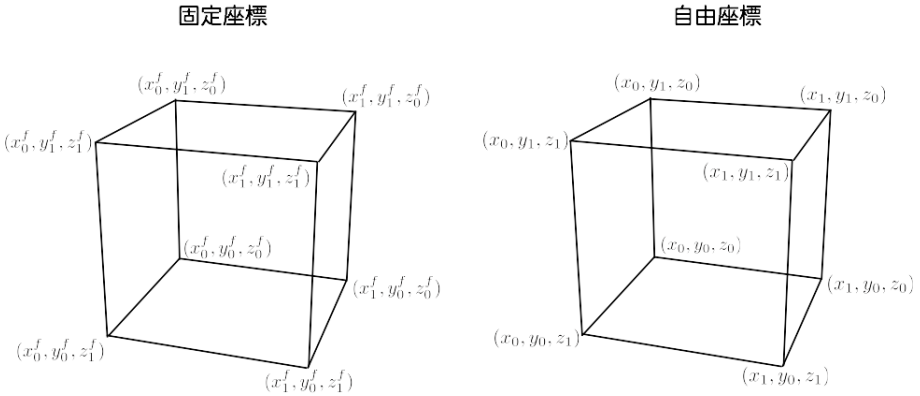


図 2　左図：固定座標で定めた直方体　　右図：同じ直方体の自由座標での表現

なお，この座標系は旧 GLSC では仮想座標系と呼ばれていたもので，おなじみの概念ではありますが，GLSC3D における 2 次元グラフィクスにおいても，対応する座標系を自由座標系と呼ぶことにします． 2 次元グラフィクスにおける自由座標系の定義に関しては `g_def_scale_2D` を参照してください（p.***）

自由座標系による描画

GLSC3D では全ての描画は自由座標系を用いて行われます[*35]．固定座標系を使用することはありません．とは言っても，3 次元グラフィクスにおいては，固定座標系を用いて（すなわち xyz 方向等縮尺で）描画オブジェクトを生成したいこともあるでしょう．そのような場合は，固定座標系に一致する自由座標系を定義した上で，その自由座標系を使用してください．$(x_0 = x_0^f,\ x_1 = x_1^f,\ y_0 = y_0^f,\ y_1 = y_1^f,\ z_0 = z_0^f,\ z_1 = z_1^f)$ [*36] ．

[*35] 例外は `g_text_standard` です．

[*36] 左手系にしたければ，例えば $x_1 = x_0^f,\ x_0 = x_1^f$ のように一つの自由座標をフリップしてください．

描画における属性

例えば線を描く場合，色や線の太さなどを指定したいこともあります．色や線の太さなどを属性と呼び，属性を適切に設定することで，描きたい描画対象の属性を変更し描くことが出来ます．この属性は属性コントロール関数を用いて変化させることができます．

関数名について

・“_2D”や “_3D”がついていないもの・・・2 次元描画と 3 次元描画で共に使用可能．

・“_2D”がついてるもの・・・2 次元描画のみで使用可能，もしくは，3 次元描画で使用すると描画が不自然となるもの．

・“_3D”がついてるもの・・・3 次元描画のみで使用可能，もしくは，2 次元描画で使用すると描画が不自然となるもの．

4.1 制御関数

4.1.1 g_init

g_init 関数

```
void g_init(
        const char *WindowName,
        int width, int height);
```

WindowName ; ウィンドウの名前，又は G_OFF_SCREEN を設定します．G_OFF_SCREEN では画面は表示されません
width ; ウィンドウの幅（ディスプレイ上の座標（ピクセル単位））
height ; ウィンドウの高さ（ ディスプレイ上の座標（ピクセル単位））

g_init 関数の説明

描画するウィンドウを設定するための関数です．ウィンドウはディスプレイの中央に表示されます．G_OFF_SCREEN を設定した場合，画面は表示されませんが，キャプチャーは問題なく行えます．※当然ながら，マウス・キーボードを用いたインタラクティブな操作は出来なくなります．

4.1.2 g_init_core

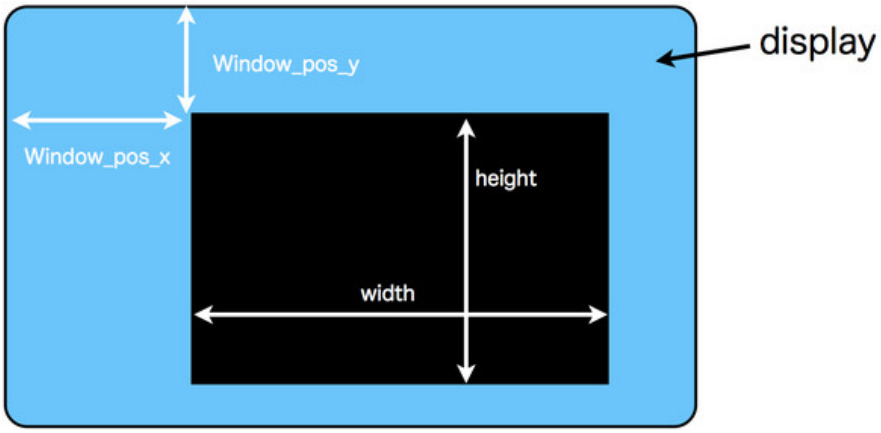
g_init_core 関数

```
void g_init_core(
        const char *WindowName,
        int width, int height,
        int pos_x, int pos_y,
        float r, float g, float b,
        int enable_transparent,
        int temporary_triangle_buffer_size,
        int triangle_buffer_size);
```

WindowName ; ウィンドウの名前，又は G_OFF_SCREEN を設定します．G_OFF_SCREEN では画面は表示されません
width ; ウィンドウの幅（ピクセル単位）
height ; ウィンドウの高さ（ピクセル単位）
pos_x ; ウィンドウの左上 x 座標（ピクセル単位）または G_WINDOW_CENTERED
pos_y ; ウィンドウの左上 y 座標（ピクセル単位）または G_WINDOW_CENTERED
r，g，b ; 背景色の初期値を設定（不透明度は 1 に固定）
enable_transparent ; 透明化処理にて並び替え機能を使用するか (1) しないか (0).
temporary_triangle_buffer_size ;
透明化処理用の TEMPORARY_TRIANGLE_BUFFER_SIZE を指定する．デフォルトは 2^{10}
triangle_buffer_size ; 透明化処理用の TRIANGLE_BUFFER_SIZE を指定する．デフォルトは 2^{20}

g_init_core 関数の説明

描画するウィンドウを設定するための関数です．g_init に比べて細かい制御が可能です．
Ver 3.0 以降: pos_x，pos_y に G_WINDOW_CENTERED を指定すると，ウインドウをディスプレイの中央に表示します．


Window_pos_y
Window_pos_x
height
width
display

4.1.3 g_init_light

g_init_light 関数

```
void g_init_light(
        int lightnum,
        float x, float y, float z);
```

lightnum ; ライトの番号（0〜2）
x, y, z ; ライトの方向座標

g_init_light 関数の説明

ライトの方向を設定するための関数です．ライトは常に平行光源です．ライト 0 から 2 の 3 個までの光源を設定できます．この関数を呼ばずに描画した場合，デフォルト値として g_init_light(0, 0, 0, 1) の光源が設定されます．

4.1.4 g_init_light_core

g_init_light_core 関数

```
void g_init_light_core(
        int lightnum,
        float x, float y, float z, float power);
```

lightnum ; ライトの番号（0〜2）
x, y, z ; ライトの方向座標
power ; ライトの強さ

g_init_light_core 関数の説明

g_init_light とほぼ同じですが，光源の強さを指定できます．

4.1.5 g_disable_light

g_disable_light 関数

```
void g_disable_light(
        int lightnum);
```

lightnum ; ライトの番号（0〜2）

g_disable_light 関数の説明

g_init_light, g_init_light_core によって有効化されたライトを無効化します．

4.1.6 g_scr_color

g_scr_color 関数

```
void g_scr_color(
        float r, float g, float b, float a);
```

r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）

g_scr_color 関数の説明

背景色を設定するための関数です.

4.1.7 g_cls

g_cls 関数

```
void g_cls();
```

g_ cls 関数の説明

新しいフレームを背景色で塗りつぶす.

4.1.8 g_finish

g_finish 関数

```
void g_finish();
```

g_ finish 関数の説明

この関数を呼ぶことで描画することができます.

4.1.9 g_sleep

g_sleep 関数

```
void g_sleep(
        double wait_time);
```

wait_time ; 静止時間（秒）

g_ sleep 関数の説明

指定した秒数だけ静止します.

4.1.10 g_capture_set, g_capture

g_capture_set, g_capture 関数

```
void g_capture_set(
        const char *name);
void g_capture();
```

name ; 保存先のフォルダ名（“” または NULL の場合，フォルダ名は「Frames 年. 月. 日. 時. 分. 秒」となる.）

g_capture_set, g_capture 関数の説明

画面を取り込むための準備を行う関数.（g_capture_set）
画面を取り込む関数.（g_capture）

4.1.11 g_enable_highdpi

g_enable_highdpi 関数

```
void g_enable_highdpi();
```

g_enable_highdpi 関数の説明

Apple Retina Display のネイティブ解像度で描画するように指定します.
この関数を使用するときは，必ず g_init(_core) より前に使用してください.

4.1.12 g_set_antialiasing

g_set_antialiasing 関数

```
void g_set_antialiasing(
        unsigned int level);
```

level ; アンチエイリアシングのレベル

g_set_antialiasing 関数の説明

アンチエイリアシングのレベルを指定します.
アンチエイリアシングは線や面の境界で発生するジャギーを低減して滑らかに表示します.

有効な値は 0 (アンチエイリアシングは無効), 1, 2, 3, 4 (最大のクオリティ) ですが，環境によっては有効な最大値が 3 以下かもしれません．デフォルトの値は 0 です.

この関数を使用するときは，必ず g_init(_core) より前に使用してください.

4.2 補助関数

4.2.1 g_key_state, g_input_state

g_key_state 関数

```
G_INPUT_STATE g_key_state(
        G_KEY_CODE code);
```

code ; 取得する入力のコード

g_input_state 関数

```
G_INPUT_STATE g_input_state(
        G_KEY_CODE code, int *x, int *y);
```

code ; 取得する入力のコード
x,y ; 最後にクリックされた位置（標準座標系）を格納する変数へのポインタ（ヌルポインタを許容します.）

g_key_state, g_input_state 関数の説明

code で指定された入力を取得します．code には char 型のリテラル（'a', '@', '/' 等）か，G_KEY_F1, G_KEY_LEFT, G_MOUSE_LEFT 等の定数を指定します．各入力の状態は，g_finish が呼ばれた時点で更新されます．G_KEY**定数，および G_INPUT_STATE の詳細は事項以降を見てください.

G_KEY**定数

G_MOUSE_LEFT,	G_KEY_INVALID,	G_KEY_INSERT,	G_KEY_F1,	G_KEY_F7,
G_MOUSE_MIDDLE,	G_KEY_PAGE_UP,	G_KEY_LEFT,	G_KEY_F2,	G_KEY_F8,
G_MOUSE_LEFT,	G_KEY_PAGE_DOWN,	G_KEY_UP,	G_KEY_F3,	G_KEY_F9,
	G_KEY_HOME,	G_KEY_RIGHT,	G_KEY_F4,	G_KEY_F10,
	G_KEY_END,	G_KEY_DOWN,	G_KEY_F5,	G_KEY_F11,
			G_KEY_F6,	G_KEY_F12

G_input_state 詳細

G_NONE ; キーは押されていない
G_DOWN ; キーが押された
G_UP ; キーが解放された
G_REPEAT ; キーが押しっぱなし

```
if(g_key_state('a') == G_DOWN)
{
    //a が入力された時の動作
}
int x, y;
if(g_input_state(G_MOUSE_LEFT, &x, &y) == G_DOWN)
{
    //左クリックされた時の動作
    printf( “x : %d\n” ” y : %d\n” , x, y);
}
if(g_input_state(G_MOUSE_LEFT, NULL, NULL) == G_DOWN)
{
    //座標が必要無ければこれでも呼び出せます.
    //x 座標と y 座標両方に NULL を指定するのは g_key_state を使うのと等価です.
}
```

4.3 スケール関数

4.3.1 g_def_scale_2D

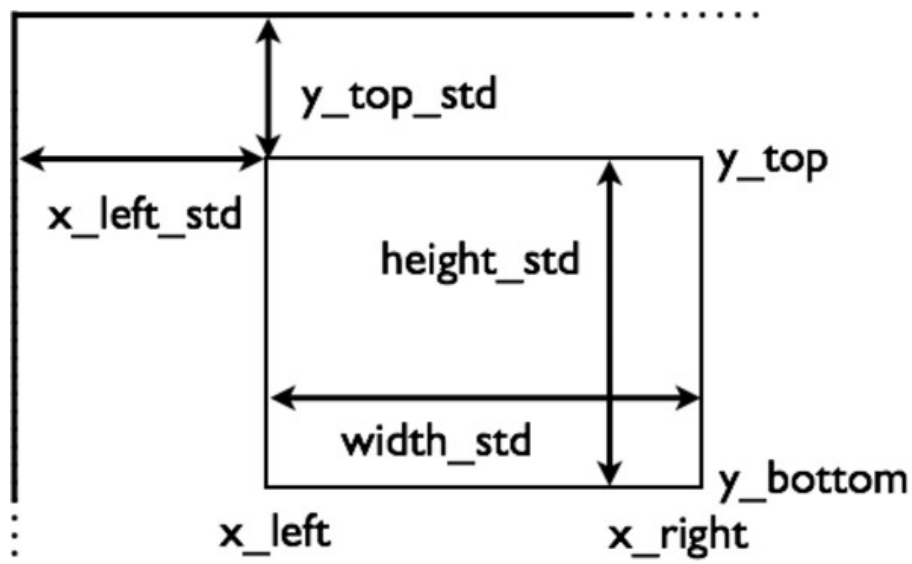
g_def_scale_2D 関数

```
void g_def_scale_2D(
        int id,
        double x_0, double x_1,
        double y_0, double y_1,
        double x_left_std, double y_top_std,
        double width_std, double height_std);
```

id ; スケール番号
x_0 ; 自由座標系での x 座標左端
x_1 ; 自由座標系での x 座標右端
y_0 ; 自由座標系での y 座標下端
y_1 ; 自由座標系での y 座標上端
x_left_std ; 標準 (固定) 座標系に定義する描画枠の左端の x 座標
y_top_std ; 標準 (固定) 座標系に定義する描画枠の左端の y 座標
width_std ; 標準 (固定) 座標系に定義する描画枠の幅
height_std ; 標準 (固定) 座標系に定義する描画枠の高さ

g_def_scale_2D 関数の説明

標準座標系上に二次元のオブジェクトを描画するための描画枠およびオブジェクトが定義される自由座標系を設定する．g_def_scale_2D では標準座標系が固定座標系の役割を担う．



4.3.2 g_def_scale_3D

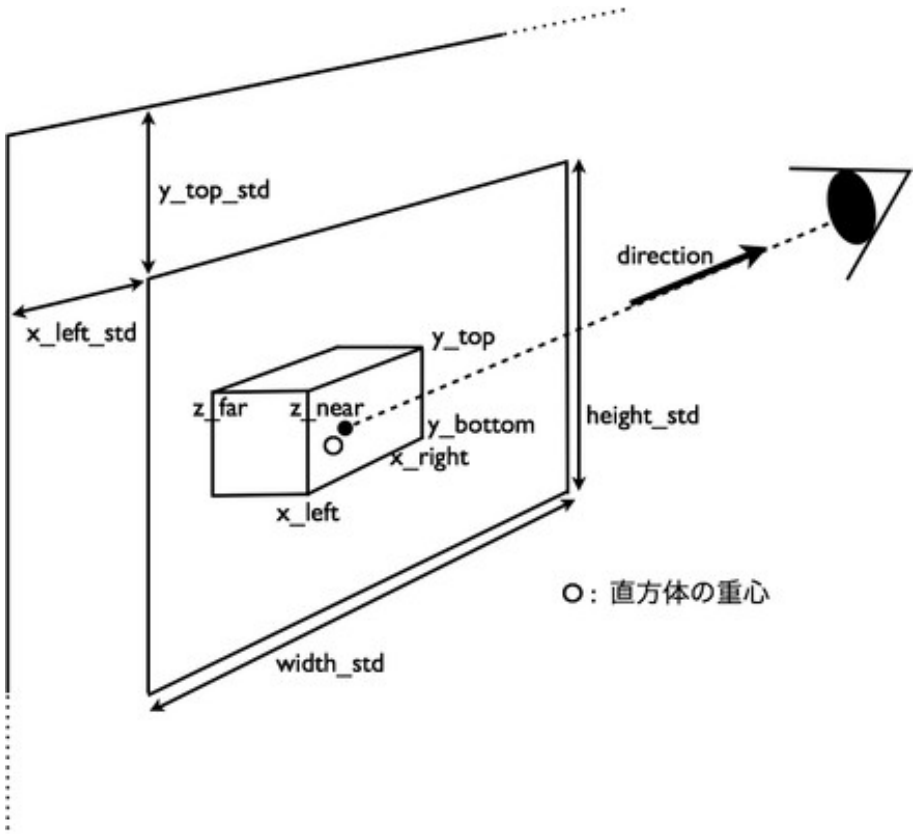
g_def_scale_3D 関数

```
void g_def_scale_3D(
        int id,
        double x_0, double x_1,
        double y_0, double y_1,
        double z_0, double z_1,
        double x_f_0, double x_f_1,
        double y_f_0, double y_f_1,
        double z_f_0, double z_f_1,
        double eye_x, double eye_y, double eye_z,
        double up_x, double up_y, double up_z,
        double zoom,
        double x_left_std, double y_top_std,
        double width_std, double height_std);
```

id ; スケール番号
x_0, x_1, y_0, y_1, z_0, z_1 ; 自由座標系における x, y, z の範囲の端点
x_f_0, x_f_1, y_f_0, y_f_1, z_f_0, z_f_1 ; 固定座標系における x, y, z の範囲の端点
eye_x, eye_y, eye_z ; 描画直方体 * の重心から視点位置までのベクトルの座標
up_x, up_y, up_z ; 画面上方向を指定するベクトルの座標
zoom ; 拡大率
x_left_std ; 標準座標系に定義する描画枠の左端の x 座標
y_top_std ; 標準座標系に定義する描画枠の左端の y 座標
width_std ; 標準座標系に定義する描画枠の幅
height_std ; 標準座標系に定義する描画枠の高さ

g_def_scale_3D 関数の説明

標準座標系上に三次元のオブジェクトを描画するための描画枠・描画するための固定座標系・オブジェクトが定義される自由座標系を設定する．* 描画するための固定座標系で指定される直方体を描画直方体と呼ぶ．
eye を指定することで，オブジェクトを見る視点位置を指定できる．
up を指定することで，描画直方体の上方向を指定できる．
GLSC3D では，固定座標系ないし自由座標系の範囲を全て *_0 < *_1 のように定めると “右手系” として描画される．ただし，奇数個の範囲を *_0 > *_1 のように定めると “左手系” として描画される．偶数個であれば “右手系” として描画される．


y_top_std
x_left_std
direction
y_top
z_far
z_near
y_bottom
x_right
x_left
height_std
width_std
○：直方体の重心

4.3.3 g_def_scale_3D_fix

g_def_scale_3D_fix 関数

```
void g_def_scale_3D_fix(
        int id,
        double x_0, double x_1,
        double y_0, double y_1,
        double z_0, double z_1,
        double eye_x, double eye_y, double eye_z,
        double up_x, double up_y, double up_z,
        double zoom,
        double x_left_std, double y_top_std,
        double width_std, double height_std);
```

id ; スケール番号
x_0, x_1, y_0, y_1, z_0, z_1 ; 固定 (自由) 座標系における x, y, z の範囲の端点
eye_x, eye_y, eye_z ; 描画直方体の重心から視点位置までのベクトルの座標
up_x, up_y, up_z ; 画面上方向を指定するベクトルの座標
zoom ; 拡大率
x_left_std ; 標準座標系に定義する描画枠の左端の x 座標
y_top_std ; 標準座標系に定義する描画枠の左端の y 座標
width_std ; 標準座標系に定義する描画枠の幅
height_std ; 標準座標系に定義する描画枠の高さ

g_def_scale_3D_core 関数の説明

標準座標系上に三次元のオブジェクトを描画するための描画枠・描画するための固定 (自由) 座標系を設定する．g_def_scale_3D_fix では，オブジェクトを定義する自由座標系が固定座標系と同じである．

4.3.4 g_sel_scale

g_sel_scale 関数

```
void g_sel_scale(
        int id);
```

id ; スケール番号

g_sel_scale 関数の説明

定義された id 番号の座標系 (g_def_scale_*) を選択する.

4.3.5 g_clipping

g_clipping 関数

```
void g_clipping(
        G_BOOL value);
```

value ; G_YES or G_NO

g_clipping 関数の説明

デフォルトでは g_clipping(G_YES) が設定されており，この場合は各スケール番号ごとに用意された標準座標系内においてオブジェクトが描画される．g_clipping(G_NO) を g_sel_scale_*(id) の後に呼ぶと，選ばれたスケール番号の描画が設定されたスクリーン全体に渡って描画される．g_sel_scale が呼ばれると g_clipping(G_YES) が選択されていることに注意.

4.4 属性コントロール関数

4.4.1 g_marker_color

g_marker_color 関数

```
void g_marker_color(
        float r, float g, float b, float a);
```

r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）

g_marker_color 関数の説明

マーカーの色を変更する．

4.4.2 g_marker_size

g_marker_size 関数

```
void g_marker_size(
        float size);
```

size ; マーカーの大きさ（ピクセル単位）

g_marker_size 関数の説明

マーカーの大きさを変更する．ピクセル単位の直径を指定する．

4.4.3 g_marker_radius

g_marker_radius 関数

```
void g_marker_radius(
        float size);
```

size ; マーカーの大きさ（自由座標系での半径）

g_marker_radius 関数の説明

マーカーの大きさを変更する．自由座標系での半径を指定する．

4.4.4 g_marker_type

g_marker_type 関数

```
void g_marker_type(
        unsigned int type);
```

type ; マーカーの種類

g_marker_type 関数の説明

マーカーの種類を変更する．使用できるマーカーの種類は以下の通り．

定数名	値	形状
G_MARKER_SQUARE	0	正方形
G_MARKER_CIRCLE	1	円
G_MARKER_SPHERE	2	球

マーカーに関するパフォーマンス上の注意

マーカータイプの変更はシェーダーの切り替えを伴うため，できるだけ同じ種類のマーカーをまとめて描画するようにすること．
また，マーカーサイズの変更は問題ないが，マーカーサイズの座標系 (g_marker_size または g_marker_radius) の変更もシェーダーの切り替えを伴うため，できるだけ同じ座標系のマーカーサイズをまとめて使用すること．

4.4.5 g_def_marker

g_def_marker 関数

```
void g_def_marker(
        int id,
        float r, float b, float g, float a,
        int type, float size);
```

id ; マーカーの属性番号
r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）
type ; マーカーの種類（g_marker_type の説明を参照）
size ; マーカーの大きさ

g_def_marker 関数の説明

マーカーの属性セットを定義する.

4.4.6 g_sel_marker

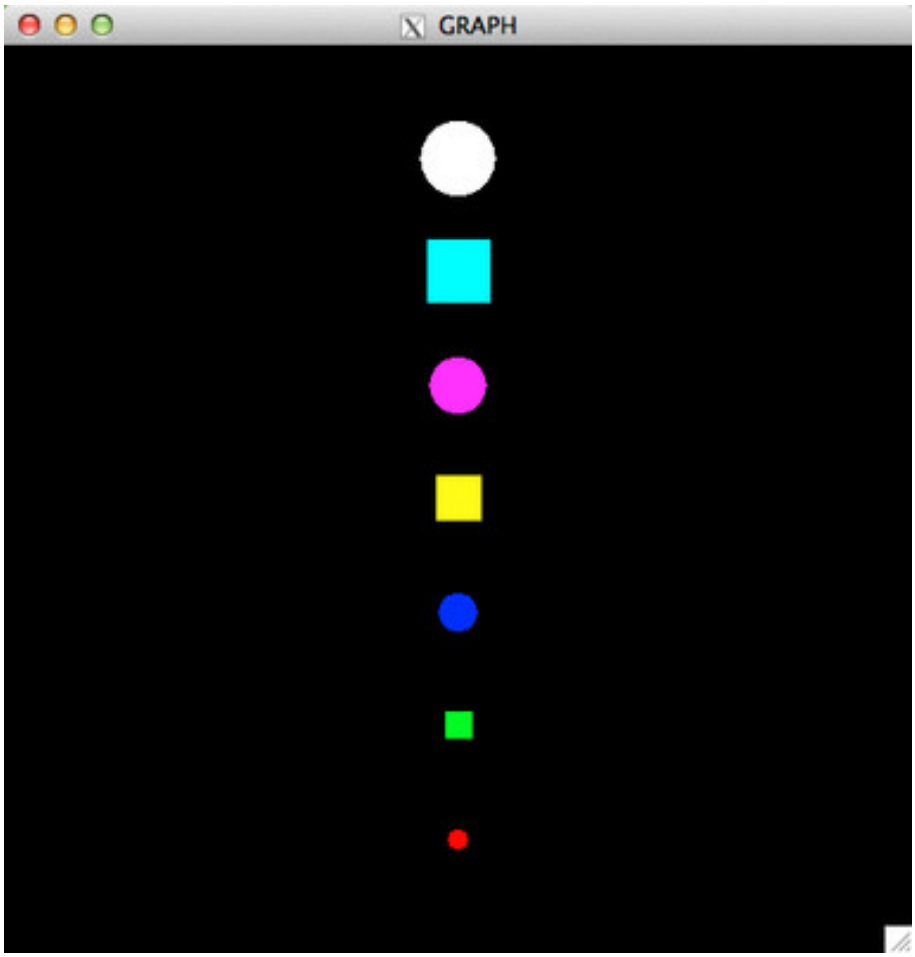
g_sel_marker 関数

```
void g_sel_marker(
        int id);
```

id ; マーカーの属性番号

g_sel_marker 関数の説明

マーカーの属性セットを選択する.


GRAPH

4.4.7 g_line_color

g_line_color 関数

```
void g_line_color(
        float r,float g,float b,float a);
```

r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）

g_line_color 関数の説明

線の色を変更する.

4.4.8 g_line_width

g_line_width 関数

```
void g_line_width(
        float size);
```

size ; 線の太さ

g_line_width 関数の説明

線の太さを変更する.

4.4.9 g_line_type

g_line_type 関数

```
void g_line_type(
        int type);
```

type ; 線の種類（0〜8）

g_line_type 関数の説明

線の種類を変更する.

4.4.10 g_def_line

g_def_line 関数

```
void g_def_line(
        int id,
        float r, float b, float g, float a,
        float width, int type);
```

id ; 線の属性番号
r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）
width ; 線の太さ
type ; 線の種類（0〜8）

g_def_line 関数の説明

線の属性セットを定義する.

4.4.11 g_sel_line

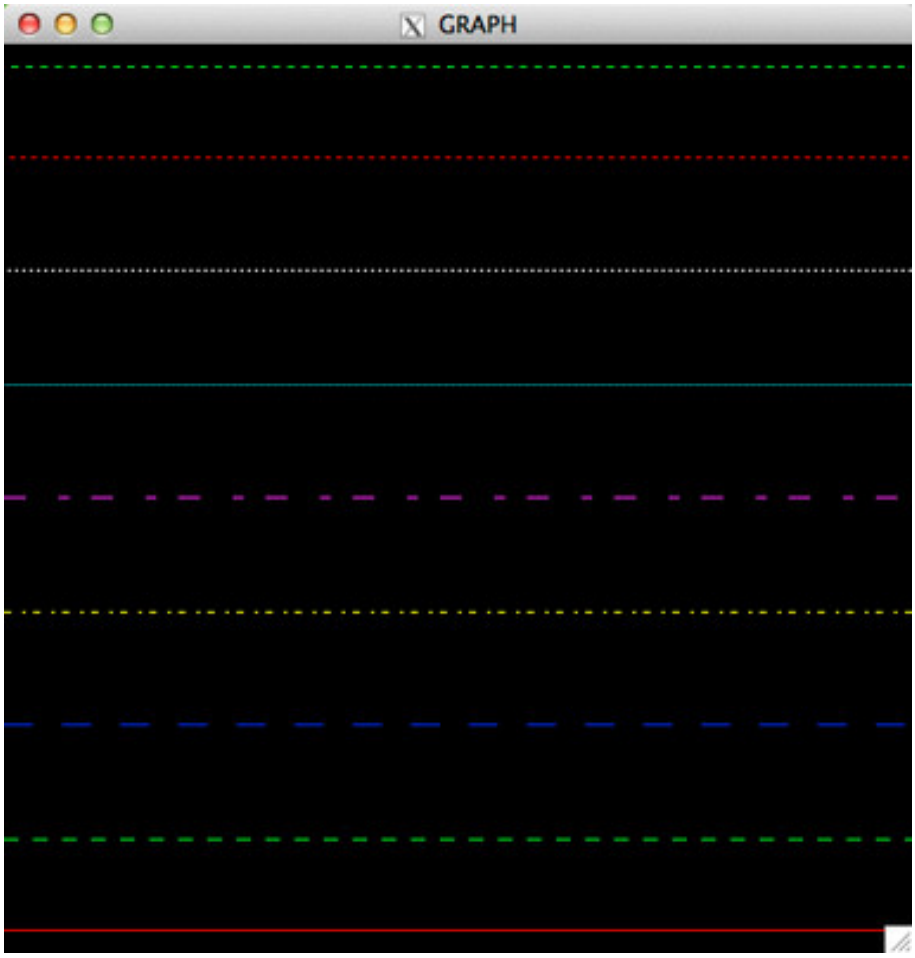
g_sel_line 関数

```
void g_sel_line(
        int id);
```

id ; 線の属性番号

g_sel_line 関数の説明

線の属性セットを選択する.



4.4.12 g_area_color

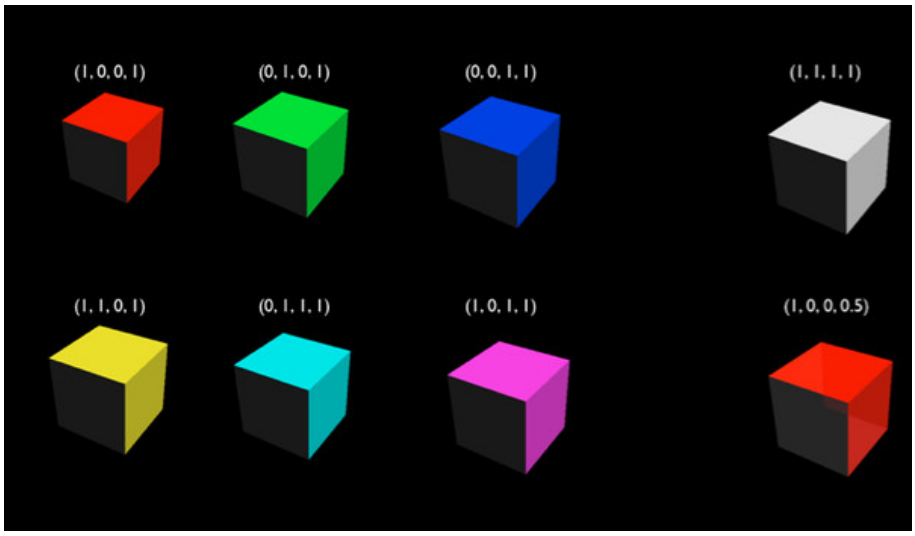
g_area_color 関数

```
void g_area_color(
        float r, float g, float b, float a);
```

r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）

g_area_color_2D, g_area_color_3D 関数の説明

面を塗りつぶす色を変更する.



4.4.13 g_def_area_2D, g_def_area_3D

g_def_area_2D, g_def_area_3D 関数

```
void g_def_area_2D(
        int id,
        float r, float b, float g, float a);
void g_def_area_3D(
        int id,
        float r, float b, float g, float a);
```

id ; 塗りつぶしの属性番号
r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）

g_def_area_2D, g_def_area_3D 関数の説明

塗りつぶしの属性セットを定義する.

4.4.14 g_sel_area_2D, g_sel_area_3D

g_sel_area_2D, g_sel_area_3D 関数

```
void g_sel_area_2D(
        int id);
void g_sel_area_3D(
        int id);
```

id ; 塗りつぶしの属性番号

g_sel_area_2D, g_sel_area_3D 関数の説明

塗りつぶしの属性セットを選択する.

4.4.15 g_text_color

g_text_color 関数

```
void g_text_color(
        float r, float g, float b, float a);
```

r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）

g_text_color 関数の説明

テキストの色を変更する.

4.4.16 g_text_font_core

g_text_font_core 関数

```
void g_text_font_core(
        const char *font_file);
```

const char *font_file ; テキストのフォントを相対パス名で指定する. テキストのフォントは ttf(True Type Font), otf(Open Type Font) などの主要な形式を指定できる.

g_text_font_core 関数の説明

テキストのフォントを変更する.

4.4.17 g_text_size

g_text_size 関数

```
void g_text_size(
        float size);
```

size ; テキストのサイズを指定する.

g_text_font_core 関数の説明

テキストのサイズを変更する.

4.4.18 g_def_text, g_def_text_core, g_sel_text

g_def_text 関数

```
void g_def_text(
        int id,
        float r, float g, float b, float a,
        unsigned int font_size);
```

id ; テキストの属性番号
r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）
font_size ; テキストのサイズ（自然数）

g_def_text 関数の説明

テキストの属性セットを定義する．

g_def_text_core 関数

```
void g_def_text_core(
        int id,
        float r, float g, float b, float a,
        const char *font_type, unsigned int font_size);
```

id ; テキストの属性番号
r,g,b,a ; 光の三原色＋不透明度（$0 \leq$ r,g,b,a ≤ 1）
font_type ; フォントファイル名，または NULL
font_size ; テキストのサイズ（自然数）

g_def_text 関数の説明

テキストの属性セットを定義する．font_type==NULL のとき，色とサイズのみ使用される．

g_sel_text 関数

```
void g_sel_text(
        int id);
```

id ; 線の属性番号

g_sel_text 関数の説明

テキストの属性セットを選択する．

4.5 描画関数

4.5.1 g_marker_2D, g_marker_3D

g_marker_2D, g_marker_3D 関数

```
void g_marker_2D(
        double x, double y);
void g_marker_3D(
        double x, double y, double z);
```

x,y,z ; マーカーの中心座標

g_marker_2D, g_marker_3D 関数の説明

マーカーを描画する.

4.5.2 g_text_standard

g_text_standard 関数

```
void g_text_standard(
        double x, double y,
        const char *str);
```

x,y ; テキスト左下の座標（標準座標系）
str ; 文字列

g_text_standard 関数の説明

文字列を描画する.（座標指定が標準座標系であることに注意.）

4.5.3 g_text_virtual_2D, g_text_virtual_3D

g_text_virtual_2D, g_text_virtual_3D 関数

```
void g_text_2D_virtual(
        double x, double y,
        const char *str);
void g_text_3D_virtual(
        double x, double y, double z,
        const char *str);
```

x,y,z ; テキスト左下の座標（自由座標系）
str ; 文字列

g_text_virtual_2D, g_text_virtual_3D 関数の説明

文字列を描画する.（座標指定が自由座標系であることに注意.）

4.5.4 g_move_2D, g_move_3D

g_move_2D, g_move_3D 関数

```
void g_move_2D(
        double x, double y);
void g_move_3D(
        double x, double y, double z);
```

x,y,z ; 線の始点の座標

g_move_2D, g_move_3D 関数の説明

線の始点を指定する.

4.5.5 g_plot_2D, g_plot_3D

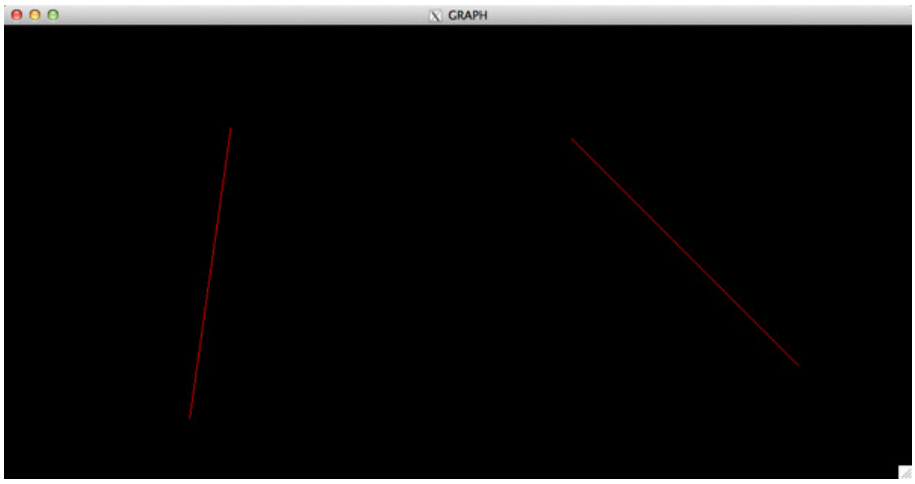
g_plot_2D, g_plot_3D 関数

```
void g_plot_2D(
        double x, double y);
void g_plot_3D(
        double x, double y, double z);
```

x,y,z ; 線の始点の座標

g_plot_2D, g_plot_3D 関数の説明

線の終点を指定する.



4.5.6 g_box_2D

g_box_2D 関数

```
void g_box_2D(
        double x_left, double x_right,
        double y_bottom, double y_top,
        G_BOOL Wire, G_BOOL Fill);
```

x_left, x_right ; 両端の x 座標
y_bottom, y_top ; 両端の y 座標
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_box_2D 関数の説明

長方形を描画する.
Ver 3.0 以降: GLSC3D Ver.2.x とは引数が変更されているので注意.
旧仕様は g_box_center_2D

4.5.7 g_box_3D

g_box_3D 関数

```
void g_box_3D(
        double x0, double x1,
        double x0, double x1,
        double z0, double z1,
        G_BOOL Wire, G_BOOL Fill);
```

x0, x1 ; 両端の x 座標
y0, y1 ; 両端の y 座標
z0, z1 ; 両端の z 座標
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_box_3D 関数の説明

直方体を描画する.
Ver 3.0 以降: GLSC3D Ver.2.x とは引数が変更されているので注意.
旧仕様は g_box_center_3D

4.5.8 g_box_3D_core

g_box_3D_core 関数

```
void g_box_3D_core(
        double x0, double x1,
        double x0, double x1,
        double z0, double z1,
        int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

x0, x1 ; 両端の x 座標
y0, y1 ; 両端の y 座標
z0, z1 ; 両端の z 座標
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_box_3D_core 関数の説明

直方体を描画する．（DivideLevel の設定が可能）
Ver 3.0 以降: GLSC3D Ver.2.x とは引数が変更されているので注意.
旧仕様は g_box_center_3D_core

4.5.9 g_box_center_2D

g_box_center_2D 関数

```
void g_box_center_2D(
        double x, double y,
        double width, double height,
        G_BOOL Wire, G_BOOL Fill);
```

x,y ; 重心の座標
width,height ; 幅と高さ
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_box_center_2D 関数の説明

長方形を描画する.

4.5.10 g_box_center_3D

g_box_center_3D 関数

```
void g_box_center_3D(
        double x, double y, double z,
        double width, double height, double depth
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 重心の座標
width,height,depth ; 幅と高さと奥行き
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_box_center_3D 関数の説明

直方体を描画する.

4.5.11 g_box_center_3D_core

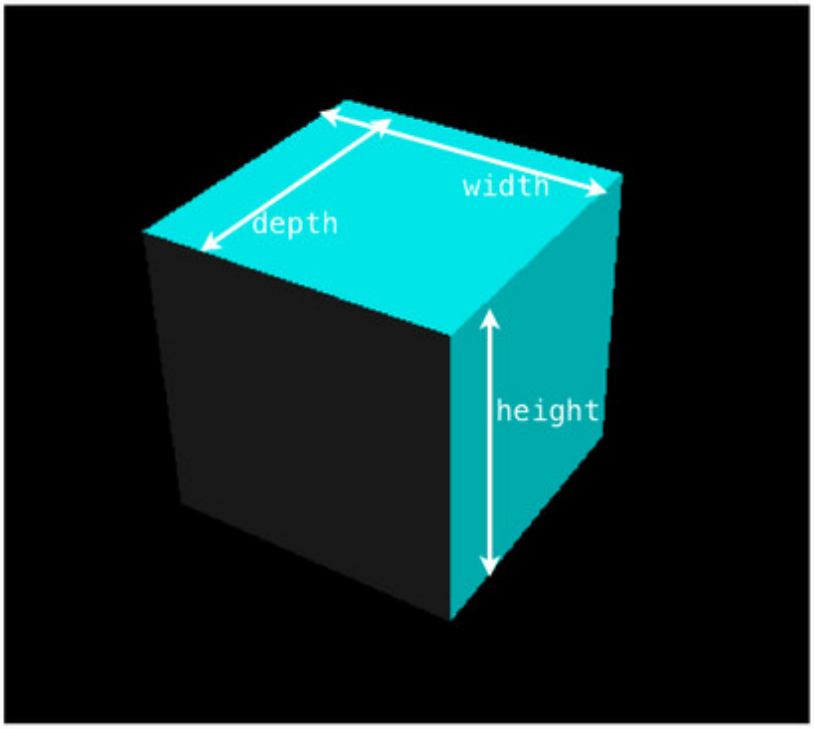
g_box_center_3D_core 関数

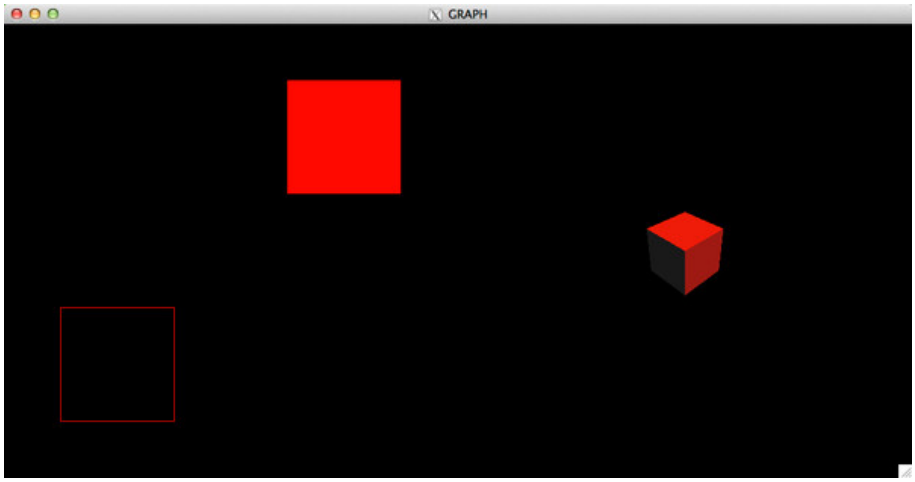
```
void g_box_center_3D_core(
        double x, double y, double z,
        double width, double height, double depth,
        int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 重心の座標
width,height,depth ; 幅と高さと奥行き
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_box_center_3D_core 関数の説明

直方体を描画する．（DivideLevel の設定が可能）




GRAPH

4.5.12 g_sphere_3D

g_sphere_3D 関数

```
void g_sphere_3D(
        double x, double y, double z,
        double radius,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 重心の座標
radius ; 半径
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_sphere_3D 関数の説明

球を描画する.
Ver 3.0 以降: この関数を小さな球を大量に表示する目的には使用しないでください. 代わりに球マーカーを使用してください.

4.5.13 g_sphere_3D_core

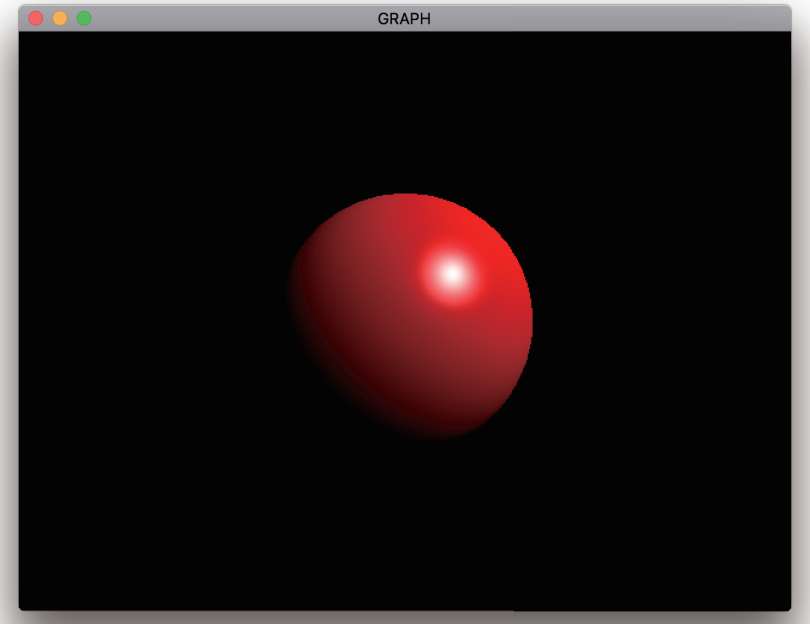
g_sphere_3D_core 関数

```
void g_sphere_3D_core(
        double x, double y, double z,
        double radius,
        int FaceNumberLevel, int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 重心の座標
radius ; 半径
FaceNumberLevel ; 球面の分割レベル
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_sphere_3D_core 関数の説明

球を描画する．（より細かい設定が可能）
Ver 3.0 以降: この関数を小さな球を大量に表示する目的には使用しないでください．代わりに球マーカーを使用してください．



4.5.14 g_ellipse_3D

g_ellipse_3D 関数

```
void g_ellipse_3D(
        double x, double y, double z,
        double Sx, double Sy, double Sz,
        double direction_x, double direction_y, double direction_z,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 重心の座標
Sx,Sy,Sz ; x,y,z 方向へのスケーリング率
direction ; 向き
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_ellipse_3D 関数の説明

楕円球を描画する.

4.5.15 g_ellipse_3D_core

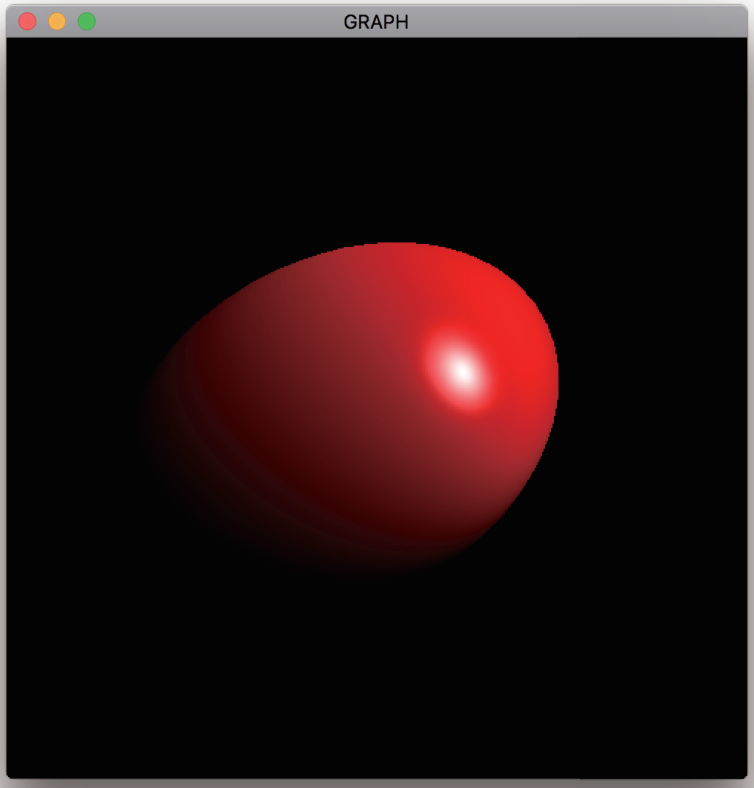
g_ellipse_3D_core 関数

```
void g_ellipse_3D_core(
        double x, double y, double z,
        double Sx, double Sy, double Sz,
        double direction_x, double direction_y, double direction_z,
        int FaceNumberLevel, int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 重心の座標
Sx,Sy,Sz ; x,y,z 方向へのスケーリング率
direction ; 向き
FaceNumberLevel ; 球面の分割レベル
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_ellipse_3D_core 関数の説明

楕円球を描画する.（より細かい設定が可能）


GRAPH

4.5.16 g_prism_3D

g_prism_3D 関数

```
void g_prism_3D(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double height, double psi, int N,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
height ; 高さ
psi ; direction に対する回転角
N ; 側面の数
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_prism_3D 関数の説明

角柱を描画する.

4.5.17 g_prism_3D_core

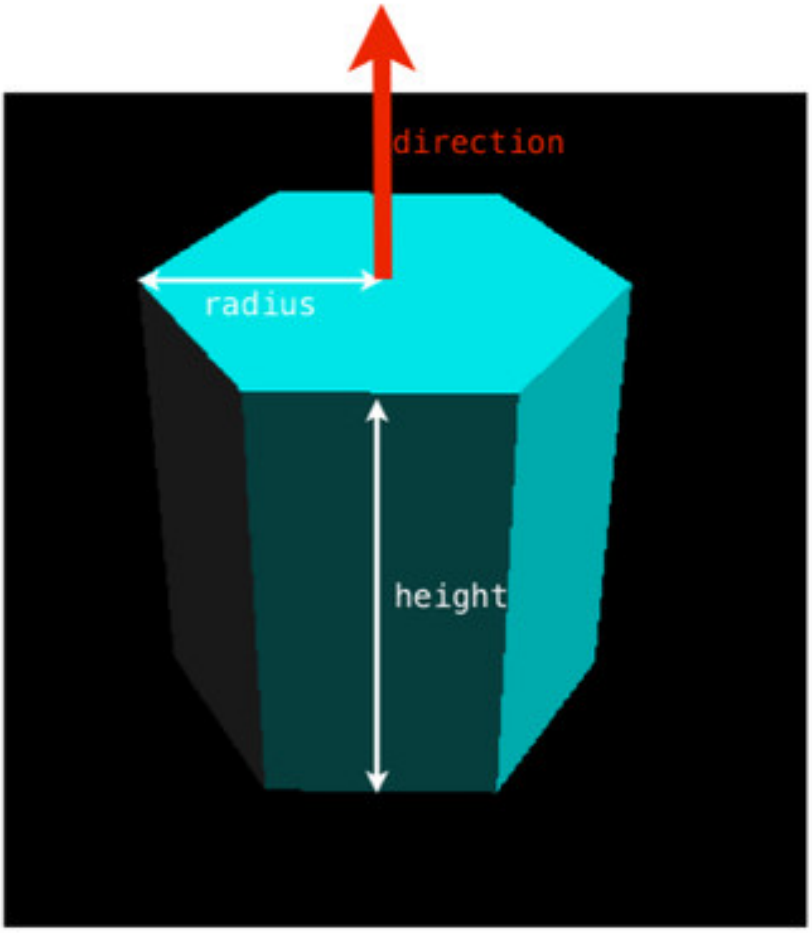
g_prism_3D_core 関数

```
void g_prism_3D_core(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double height, double psi, int N,
        int DivideLevel, G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
height ; 高さ
psi ; direction に対する回転角
N ; 側面の数
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_prism_3D_core 関数の説明

角柱を描画する．（より細かい設定が可能）


direction
radius
height

4.5.18 g_cylinder_3D

g_cylinder_3D 関数

```
void g_cylinder_3D(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double height, double psi,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
height ; 高さ
psi ; direction に対する回転角
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_cylinder_3D 関数の説明

円柱を描画する.

4.5.19 g_cylinder_3D_core

g_cylinder_3D_core 関数

```
void g_cylinder_3D_core(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double height, double psi,
        int N, int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
height ; 高さ
psi ; direction に対する回転角
N ; 側面の数
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_cylinder_3D_core 関数の説明

円柱を描画する．（より細かい設定が可能）


GRAPH

4.5.20 g_cone_3D

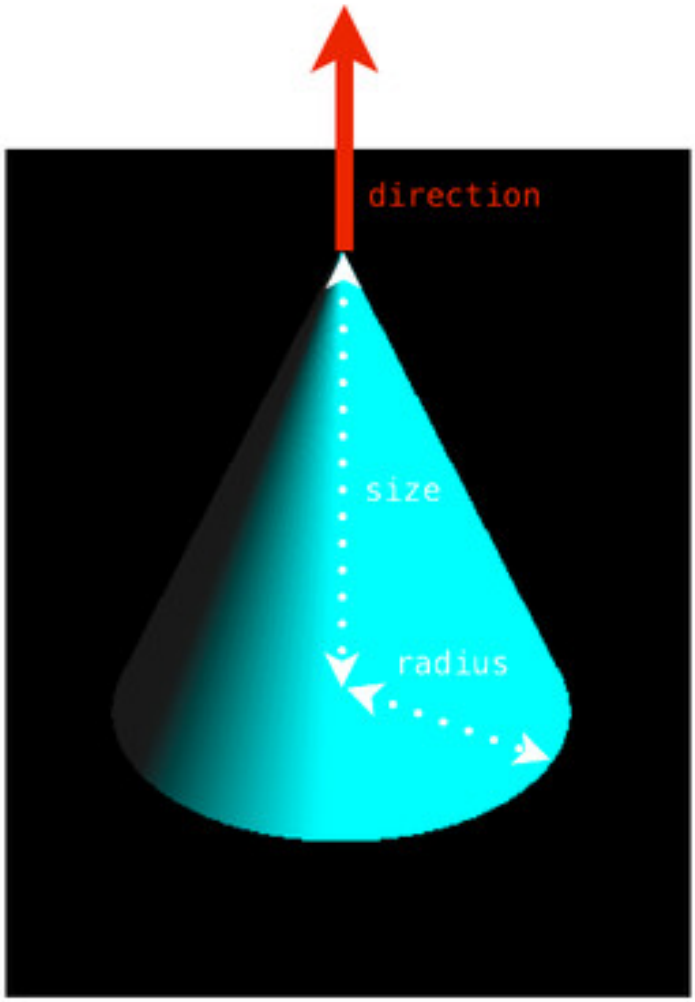
g_cone_3D 関数

```
void g_cone_3D(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double head_size
        G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
head_size ; 高さ
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_cone_3D 関数の説明

円錐を描画する.


direction
size
radius

4.5.21 g_cone_3D_core

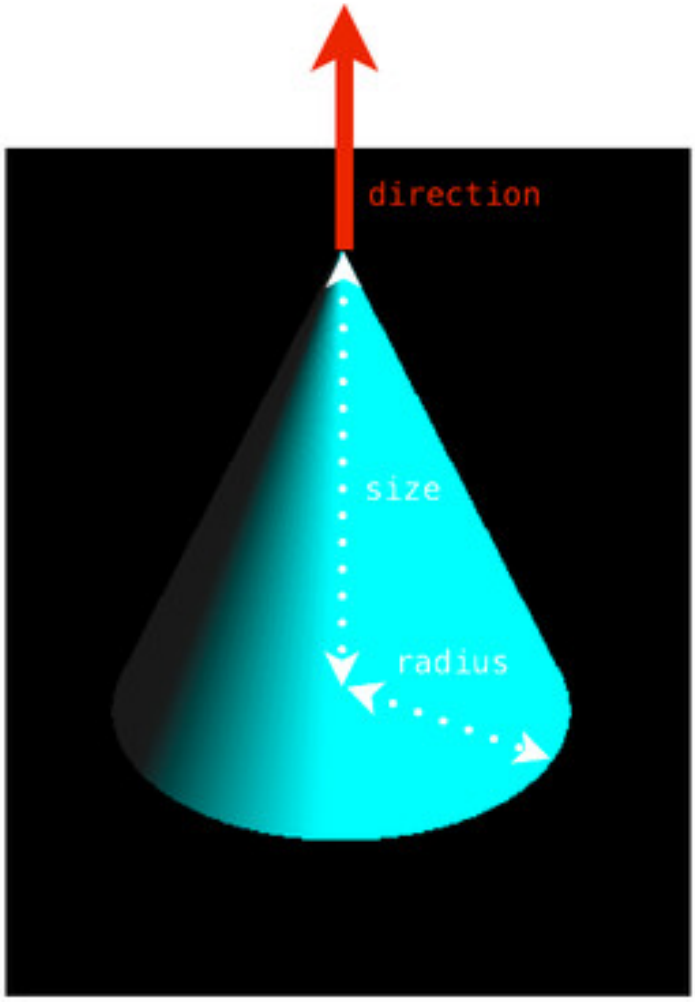
g_cone_3D_core 関数

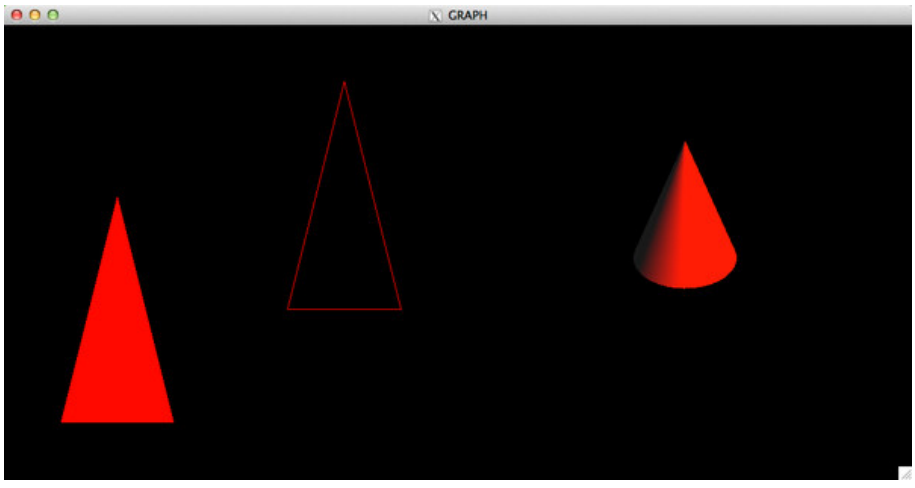
```
void g_cone_3D_core(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double head_size,
        int N, int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
head_size ; 高さ
N ; 円周の分割数
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_cone_3D_core 関数の説明

円錐を描画する．（より細かい設定が可能）


direction
size
radius


GRAPH

4.5.22 g_pyramid_3D

g_pyramid_3D 関数

```
void g_pyramid_3D(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double head_size, double psi, int N,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
head_size ; 高さ
psi ; direction に対する回転角
N ; 側面の数
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_pyramid_3D 関数の説明

角錐を描画する.

4.5.23 g_pyramid_3D_core

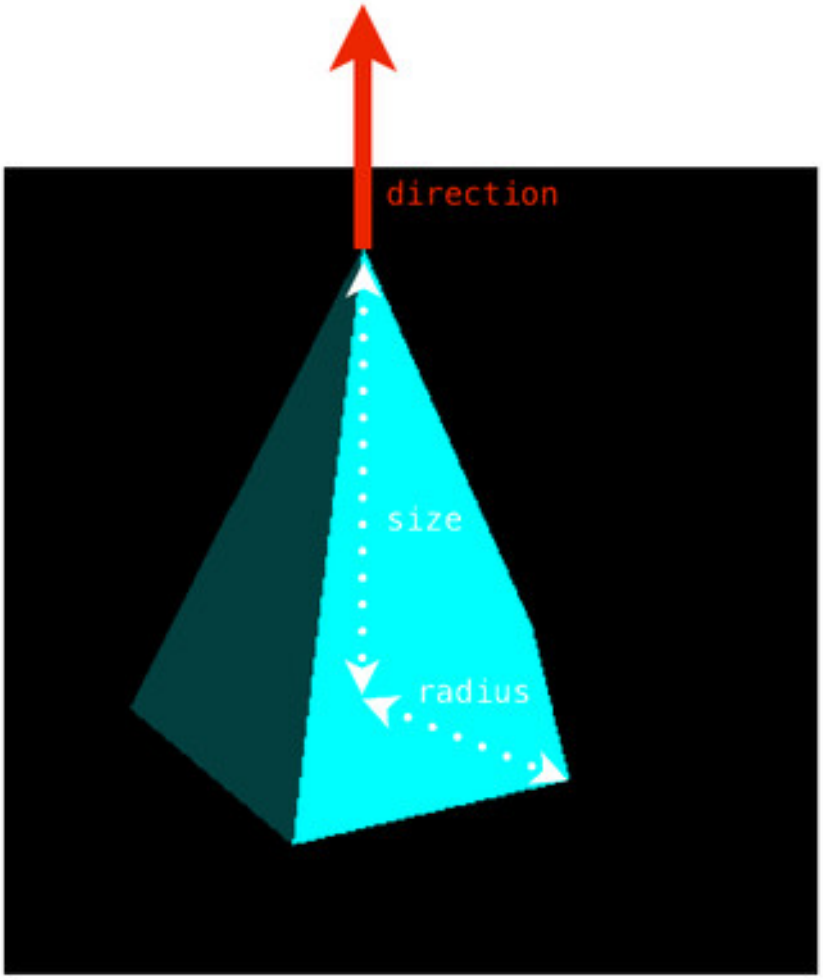
g_pyramid_3D_core 関数

```
void g_pyramid_3D_core(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double head_size, double psi, int N,
        int DivideLevel, G_BOOL Wire, G_BOOL Fill);
```

center ; 重心の座標
direction ; 向き
radius ; 半径
head_size ; 高さ
psi ; direction に対する回転角
N ; 側面の数
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_pyramid_3D_core 関数の説明

角錐を描画する．（より細かい設定が可能）


direction
size
radius

4.5.24 g_arrow_2D

g_arrow_2D 関数

```
void g_arrow_2D(
        double base_x, double base_y
        double direction_x, double direction_y
        double arrow_size, double head_size, int type);
```

base ; 根元の座標
direction ; 向き
arrow_size ; 全体の長さ
head_size ; 傘の部分の長さ
type ; 矢印の種類 (0〜2)

g_arrow_2D 関数の説明

矢印を描画する.

4.5.25 g_arrow_3D

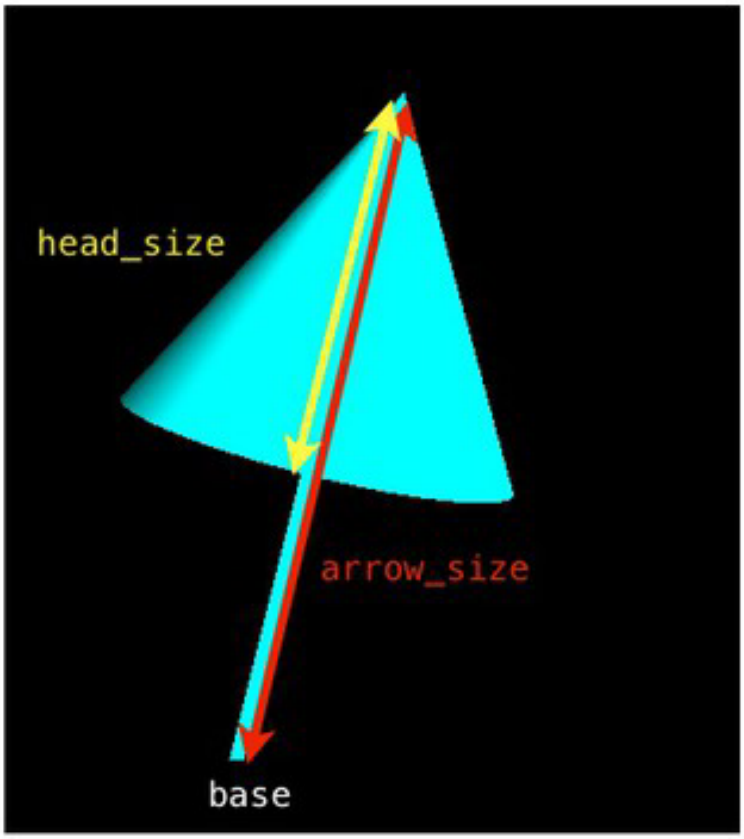
g_arrow_3D 関数

```
void g_arrow_3D(
        double base_x, double base_y, double base_z,
        double direction_x, double direction_y, double direction_z,
        double arrow_size, double head_size,
        G_BOOL Wire, G_BOOL Fill);
```

base ; 根元の座標
direction ; 向き
arrow_size ; 全体の長さ
head_size ; 傘の部分の長さ
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_arrow_3D 関数の説明

矢印を描画する.


head_size
arrow_size
base

4.5.26 g_arrow_3D_core

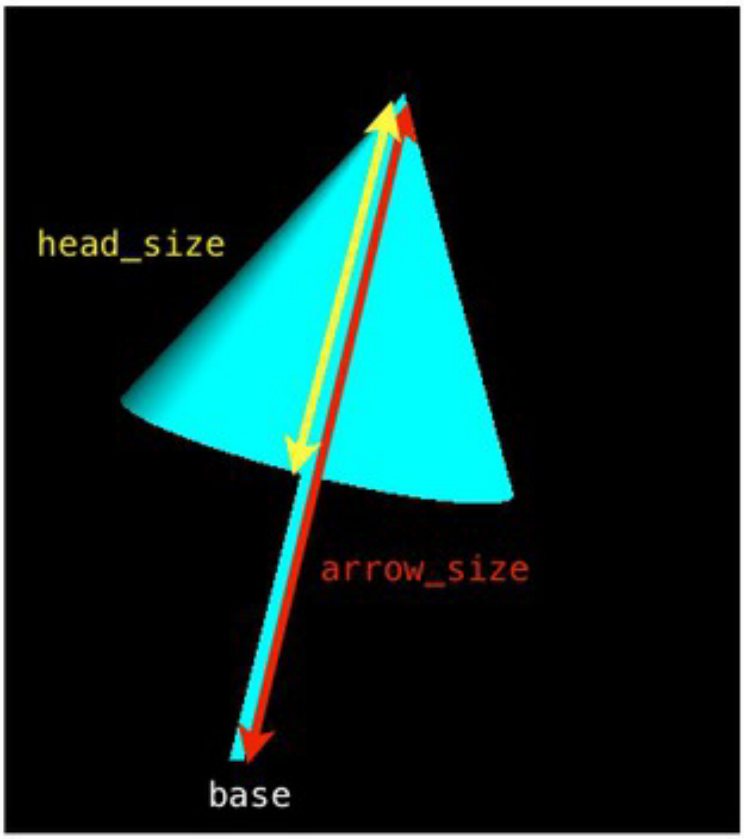
g_arrow_3D_core 関数

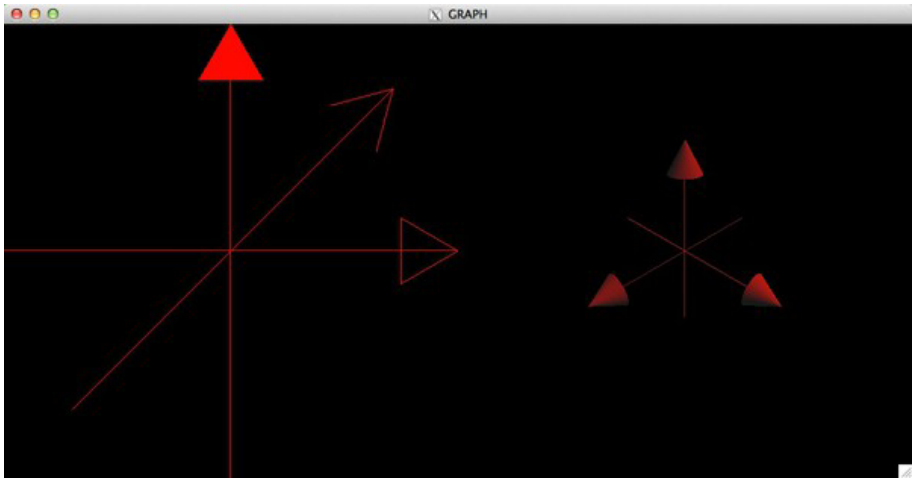
```
void g_arrow_3D_core(
        double base_x, double base_y, double base_z,
        double direction_x, double direction_y, double direction_z,
        double arrow_size, double head_size,
        int N, int DivideLevel,
        G_BOOL Wire, G_BOOL Fill);
```

base ; 根元の座標
direction ; 向き
arrow_size ; 全体の長さ
head_size ; 傘の部分の長さ
N ; 傘の部分の分割数
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_arrow_3D_core 関数の説明

矢印を描画する．（より細かい設定が可能）


head_size
arrow_size
base


GRAPH

4.5.27 g_triangle_2D

g_triangle_2D 関数

```
void g_triangle_2D(
        double x0, double y0,
        double x1, double y1,
        double x2, double y2,
        G_BOOL Wire, G_BOOL Fill);
```

x,y ; 各頂点の座標
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_triangle_2D 関数の説明

三角形を描画する.

4.5.28 g_triangle_3D

g_triangle_3D 関数

```
void g_triangle_3D(
        double x0, double y0, double z0,
        double x1, double y1, double z1,
        double x2, double y2, double z2,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 各頂点の座標
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_triangle_3D 関数の説明

三角形を描画する.（頂点の指定順によって表裏が変わることに注意.）

4.5.29 g_triangle_3D_core

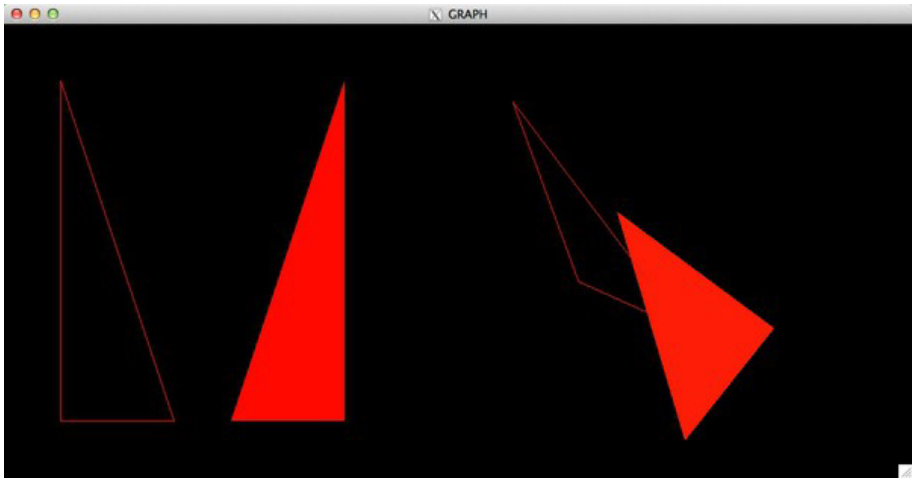
g_triangle_3D_core 関数

```
void g_triangle_3D_core(
        double x0, double y0, double z0,
        double x1, double y1, double z1,
        double x2, double y2, double z2,
        int DivideLevel, G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 各頂点の座標
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_triangle_3D_core 関数の説明

三角形を描画する．（頂点の指定順によって表裏が変わることに注意．）



4.5.30 g_triangle_3D_smooth

g_triangle_3D_smooth 関数

```
void g_triangle_3D_smooth(
        double x0, double y0, double z0,
        double x1, double y1, double z1,
        double x2, double y2, double z2,
        double nx0, double ny0, double nz0,
        double nx1, double ny1, double nz1,
        double nx2, double ny2, double nz2,
        G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 各頂点の座標
nx,ny,nz ; 各頂点での法線ベクトルの x, y, z 成分
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_triangle_3D_smooth 関数の説明

三角形を描画する.（頂点の指定順によって表裏が変わることに注意.）

4.5.31 g_triangle_3D_smooth_core

g_triangle_3D_smooth_core 関数

```
void g_triangle_3D_smooth_smooth(
        double x0, double y0, double z0,
        double x1, double y1, double z1,
        double x2, double y2, double z2,
        double nx0, double ny0, double nz0,
        double nx1, double ny1, double nz1,
        double nx2, double ny2, double nz2,
        int DivideLevel, G_BOOL Wire, G_BOOL Fill);
```

x,y,z ; 各頂点の座標
nx,ny,nz ; 各頂点での法線ベクトルの x, y, z 成分
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_triangle_3D_smooth_core 関数の説明

三角形を描画する.（頂点の指定順によって表裏が変わることに注意.）

4.5.32 g_fan_2D

g_fan_2D 関数

```
void g_fan_2D(
        double center_x, double center_y,
        double direction_x, double direction_y,
        double radius, double angle,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 中心の座標
direction ; 向き
radius ; 半径
angle ; 扇の中心角
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_fan_2D 関数の説明

扇形を描画する.

4.5.33 g_fan_3D

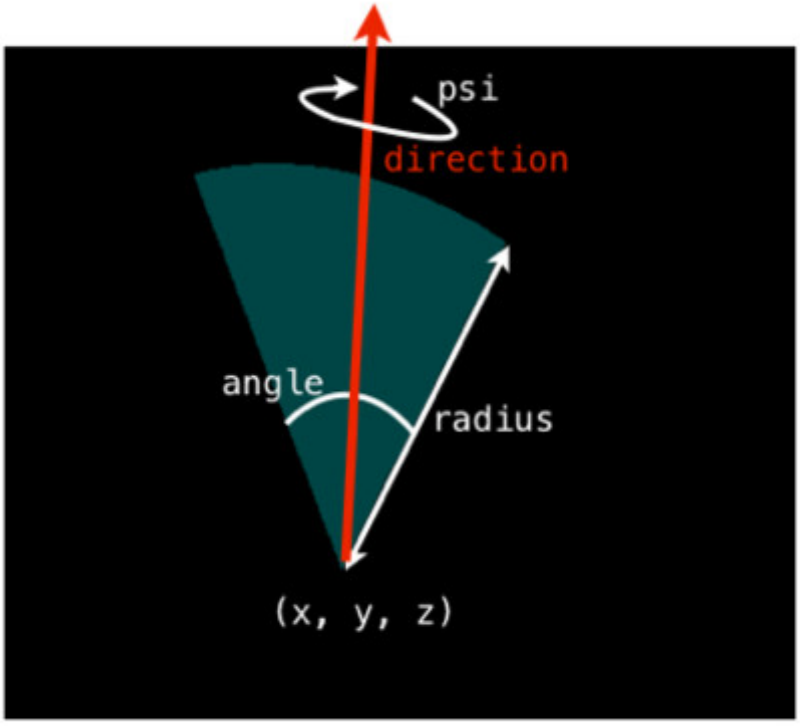
g_fan_3D 関数

```
void g_fan_3D(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double angle, double psi,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 中心の座標
direction ; 向き
radius ; 半径
angle ; 扇の中心角
psi ; direction に対する回転角
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_fan_3D 関数の説明

扇形を描画する.



4.5.34 g_fan_3D_core

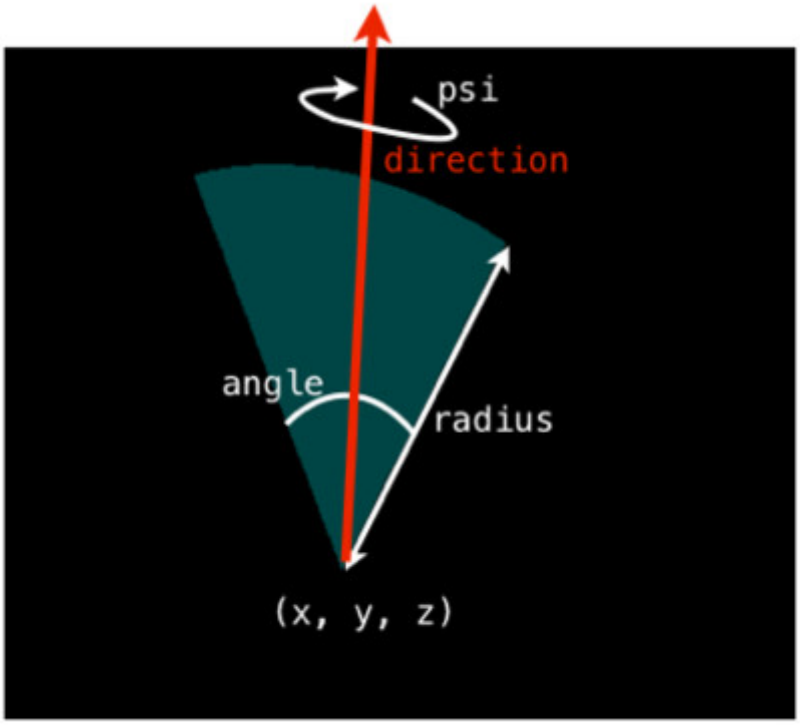
g_fan_3D_core 関数

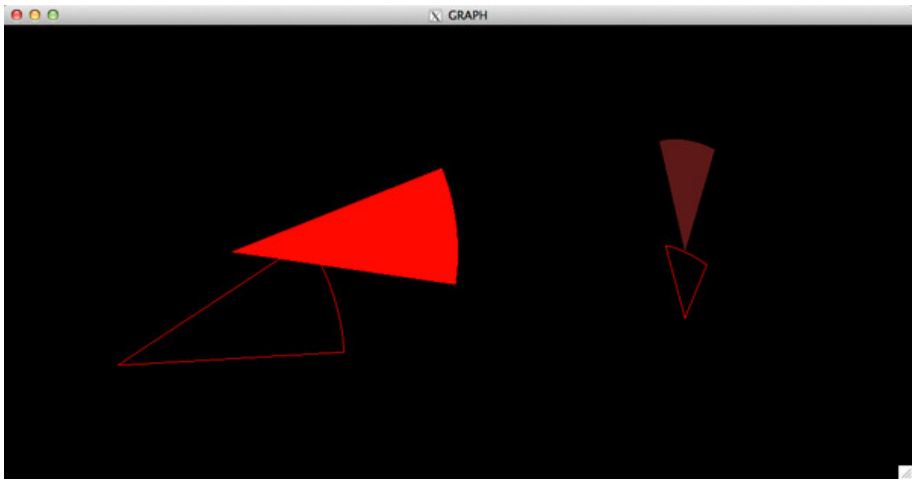
```
void g_fan_3D_core(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double radius, double angle, double psi,
        int FaceNumberLevel, int DivideLevel, G_BOOL Wire, G_BOOL Fill);
```

center ; 中心の座標
direction ; 向き
radius ; 半径
angle ; 扇の中心角
psi ; direction に対する回転角
FaceNumberLevel ; 扇形の分割レベル
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_fan_3D_core 関数の説明

扇形を描画する．（より細かい設定が可能）


psi
direction
angle
radius
(x, y, z)


GRAPH

4.5.35 g_circle_2D

g_circle_2D 関数

```
void g_circle_2D(
        double center_x, double center_y,
        double radius, G_BOOL Wire, G_BOOL Fill);
```

center ; 中心の座標
radius ; 半径
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_circle_2D 関数の説明

円を描画する.

4.5.36 g_circle_3D

g_circle_3D 関数

```
void g_circle_3D(
        double center_x, double center_y,
        double radius, double theta, double phi,
        G_BOOL Wire, G_BOOL Fill);
```

center ; 中心の座標
radius ; 半径
psi ; 半径
theta ; 半径
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_circle_3D 関数の説明

円板を描画する.

4.5.37 g_circle_3D_core

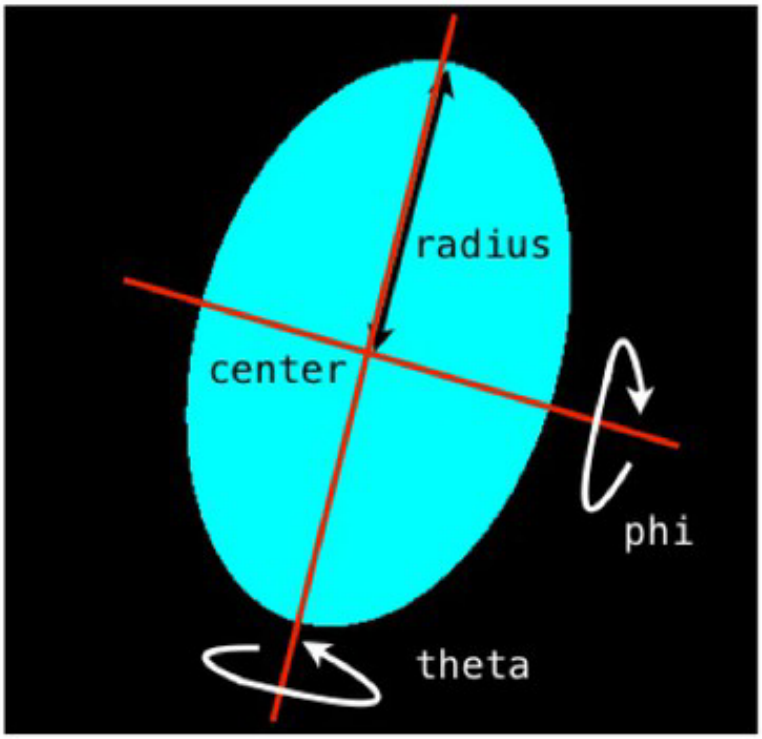
g_circle_3D_core 関数

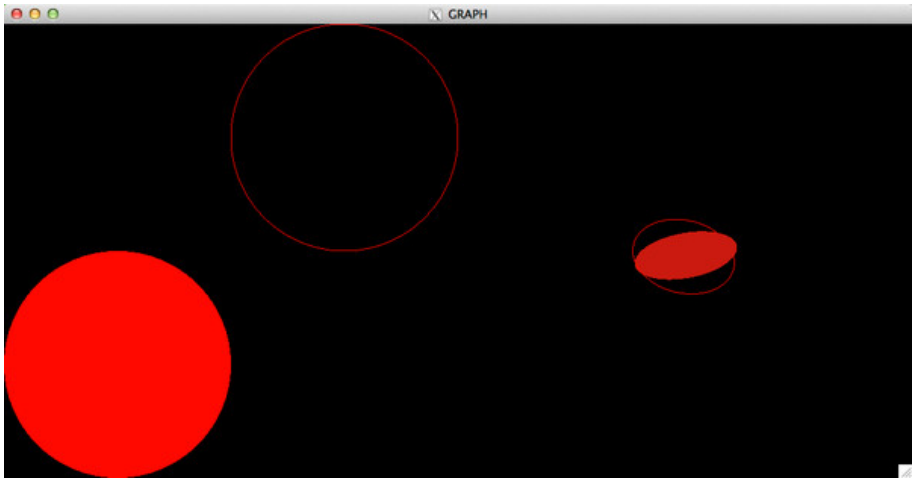
```
void g_circle_3D_core(
        double center_x, double center_y,
        double radius, double theta, double phi,
        int N, int DivideLevel, G_BOOL Wire, G_BOOL Fill);
```

center ; 中心の座標
radius ; 半径
psi ; 半径
theta ; 半径
N ; 円周の分割数
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_circle_3D_core 関数の説明

円板を描画する．（より細かい設定が可能）




GRAPH

4.5.38 g_polygon_2D

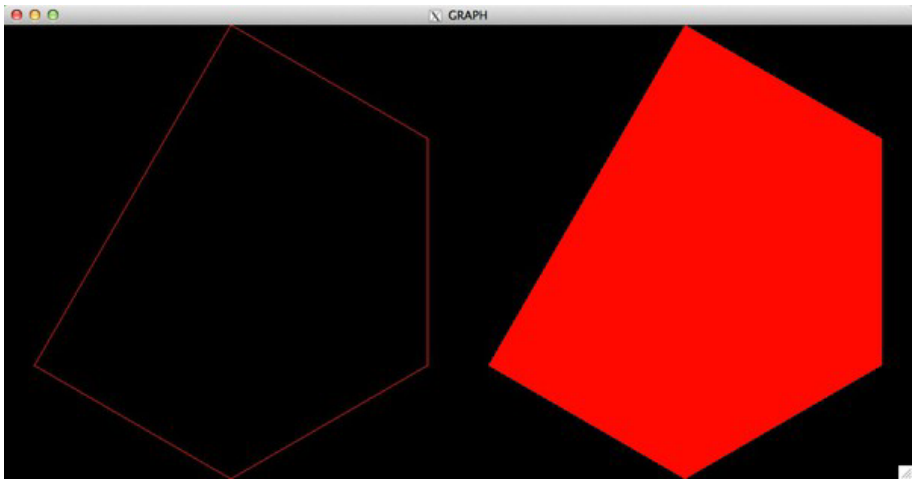
g_polygon_2D 関数

```
void g_polygon_2D(
        double *xx, double *yy,
        int n, G_BOOL WIRE, G_BOOL FILL);
```

xx,yy ; 頂点データを格納した一次元配列
n ; 配列の数
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_polygon_2D 関数の説明

多角形を描画する.



4.5.39 g_polyline_2D

g_polyline_2D 関数

```
void g_polyline_2D(
        double *xx, double *yy,
        int n);
```

xx,yy ; 頂点データを格納した一次元配列
n ; 配列の数

g_polyline_2D 関数の説明

与えられた点列を線分で連続的に結んだものを描画する.

4.5.40 g_polyline_3D

g_polyline_3D 関数

```
void g_polyline_3D(
        double *xx, double *yy, double *zz,
        int n);
```

xx,yy,zz ; 頂点データを格納した一次元配列
n ; 配列の数

g_polyline_3D 関数の説明

与えられた点列を線分で連続的に結んだものを描画する.

4.5.41 g_rectangle_3D

g_rectangle_3D 関数

```
void g_rectangle_3D(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double width, double depth, double psi,
        G_BOOL WIRE, G_BOOL FILL);
```

center ; 重心の座標
direction ; 向き
width,height,depth ; 幅，高さ，奥行き
psi ; direction に対する回転角
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_rectangle_3D 関数の説明

長方形を描画する.

4.5.42 g_rectangle_3D_core

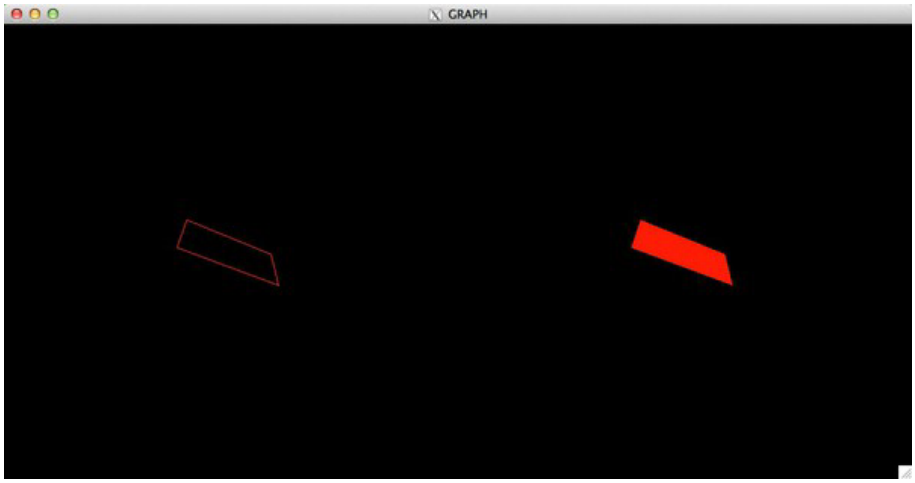
g_rectangle_3D_core 関数

```
void g_rectangle_3D_core(
        double center_x, double center_y, double center_z,
        double direction_x, double direction_y, double direction_z,
        double width, double depth, double psi,
        int DivideLevel, G_BOOL WIRE, G_BOOL FILL);
```

center ; 重心の座標
direction ; 向き
width,height,depth ; 幅，高さ，奥行き
psi ; direction に対する回転角
DivideLevel ; 面の三角形分割レベル（$4^{\texttt{DivideLevel}}$ 倍ずつ三角形の分割数が増える）
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_rectangle_3D_core 関数の説明

長方形を描画する．（より細かい設定が可能．）



4.5.43 g_data_plot_2D

g_data_plot_2D 関数

```
void g_data_plot_2D(
        double x0, double x1,
        double y0, double y1,
        int number_x,
        double u[number_x]);
```

x,y ; 描画範囲の指定
number ; 配列 u のサイズ
u ; データの格納された 1 次元配列

g_data_plot_2D 関数の説明

1 次元配列 u のデータをプロットする.

4.5.44 g_data_plot_3D

g_data_plot_3D 関数

```
void g_data_plot_3D(
        double x0, double x1,
        double y0, double y1,
        double z0, double z1,
        int number_x, int number_y,
        double u[number_x][number_y]);
```

x,y,z ; 描画範囲の指定
number ; 配列 u の各方向のサイズ
u ; 2 次元配列もしくは 1 次元配列を 2 次元配列化*した配列

g_data_plot_3D 関数の説明

2 次元配列 u のデータをプロットする．(2 次元配列化*に関しては前章をお読みください．)

4.5.45 g_data_plot_f_3D

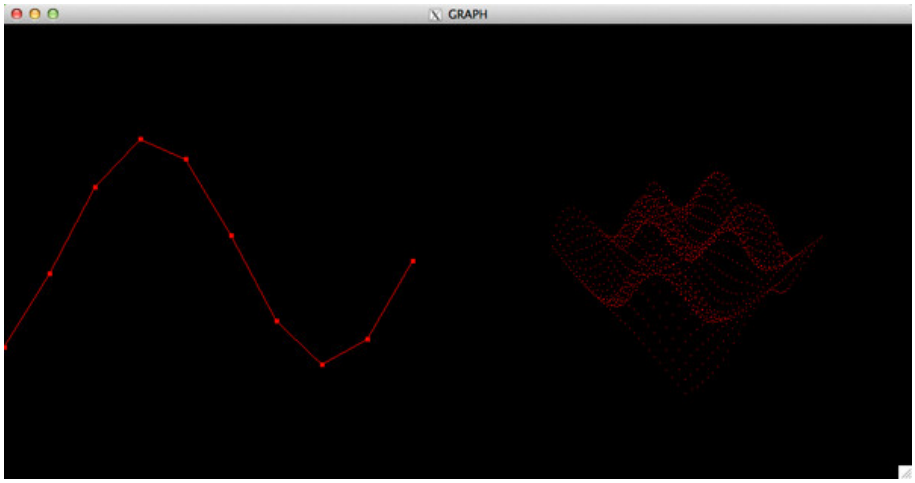
g_data_plot_f_3D 関数

```
void g_data_plot_f_3D(
        double x0, double x1,
        double y0, double y1,
        double z0, double z1,
        int number_x, int number_y,
        double *u);
```

x,y,z ; 描画範囲の指定
number ; 配列 u の各方向のサイズ
u ; 1 次元配列を 2 次元配列化*した配列

g_data_plot_f_3D 関数の説明

1 次元配列 u のデータをプロットする．(2 次元配列化*に関しては前章をお読みください．)



4.5.46 g_boundary

g_boundary 関数

```
void g_boundary();
```

g_boundary 関数の説明

g_sel_scale 関数で選択されているスケール番号の描画枠の枠線を描画する.

4.6 上位関数

4.6.1 g_contln_2D

g_contln_2D 関数

```
void g_contln_2D(
        double x_left, double x_right,
        double y_bottom, double y_top,
        int N_x, int N_y,
        double data2D[N_x][N_y],
        double level);
```

x,y ; 描画範囲の指定
N ; 配列 u の各方向のサイズ
data2D ; データの格納された 2 次元配列もしくは 1 次元配列を 2 次元配列化*した配列
level ; 等高線を引きたい値.

g_contln_2D 関数の説明

2 次元配列 u に対して，値 level に等高線を描画する．(2 次元配列化*に関しては前章をお読みください.)

4.6.2 g_contln_f_2D

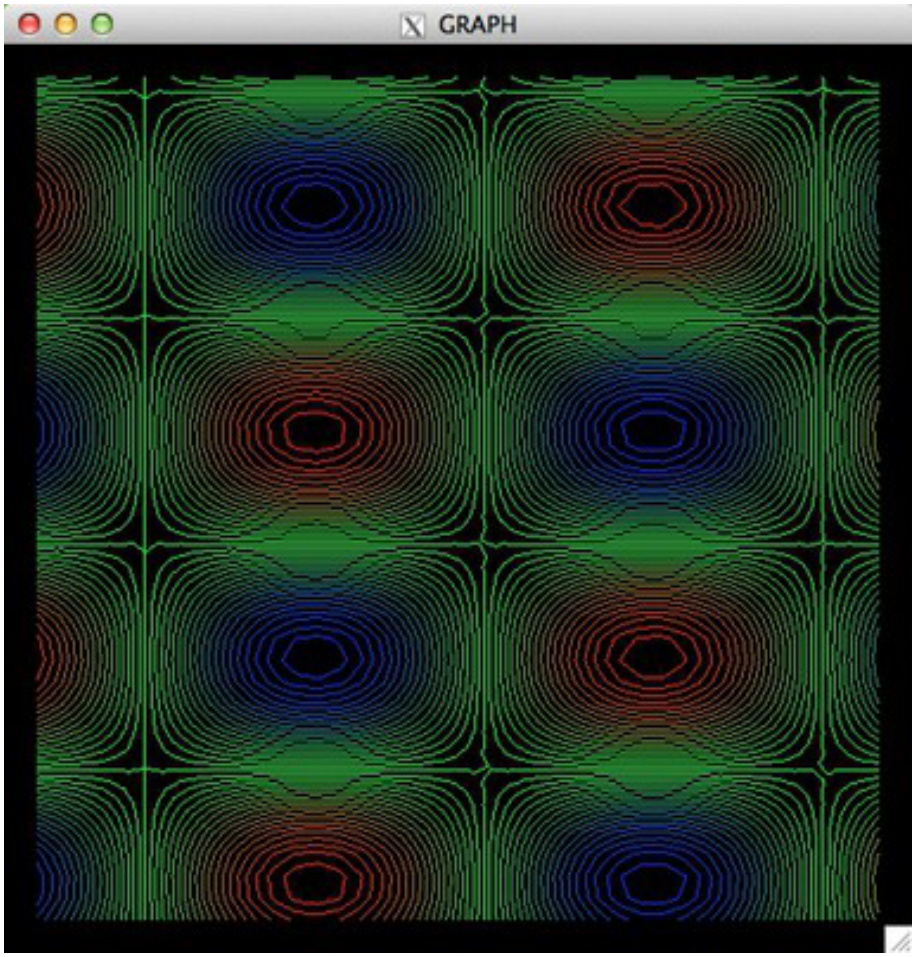
g_contln_f_2D 関数

```
void g_contln_f_2D(
        double x_left, double x_right,
        double y_bottom, double y_top,
        int N_x, int N_y,
        double *data2D,
        double level);
```

x,y ; 描画範囲の指定
N ; 配列 u の各方向のサイズ
data2D ; データの格納された，1 次元配列を 2 次元配列化*した配列
level ; 等高線を引きたい値.

g_contln_f_2D 関数の説明

1 次元配列 u に対して，値 level に等高線を描画する．(2 次元配列化*に関しては前章をお読みください.)



4.6.3 g_bird_view_3D

g_bird_view_3D 関数

```
void g_bird_view_3D(
        double x_left, double x_right,
        double y_bottom, double y_top,
        int number_x, int number_y,
        double u[number_x][number_y],
        G_BOOL WIRE, G_BOOL FILL);
```

x,y,z ; 描画範囲の指定
number ; 配列 u の各方向のサイズ
u ; 2 次元配列もしくは 1 次元配列を 2 次元配列化*した配列
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_bird_view_3D 関数の説明

2 次元配列 u に対して鳥瞰図を描画する．y_top=y_bottom とすると y 方向のスケーリングをせず，値をそのまま反映する．(2 次元配列化*に関しては前章をお読みください.)

4.6.4 g_bird_view_f_3D

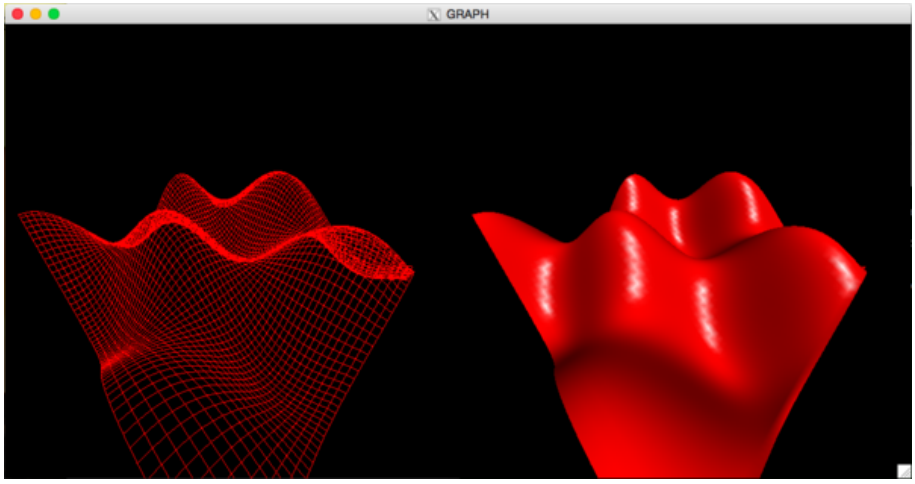
g_bird_view_f_3D 関数

```
void g_bird_view_f_3D(
        double x_left, double x_right,
        double y_bottom, double y_top,
        int number_x, int number_y,
        double *u, G_BOOL WIRE, G_BOOL FILL);
```

x,y,z ; 描画範囲の指定
number ; 配列 u の各方向のサイズ
u ;1 次元配列を 2 次元配列化*した配列
Wire ; G_YES:枠線を描く, G_NO:枠線を描かない
Fill ; G_YES:塗りつぶす, G_NO:塗りつぶさない

g_bird_view_f_3D 関数の説明

1 次元配列 u に対して鳥瞰図を描画する．y_top=y_bottom とすると y 方向のスケーリングをせず，値をそのまま反映する．(2 次元配列化*に関しては前章をお読みください.)



4.6.5 g_isosurface_3D

g_isosurface_3D 関数

```
void g_isosurface_3D(
        double x0, double x1,
        double y0, double y1,
        double z0, double z1,
        int number_x, int number_y, int number_z,
        double u[number_x][number_y][number_z],
        double height);
```

x,y,z ; 描画範囲の指定
number ; 配列 u の各方向のサイズ
u ; 3 次元配列もしくは 1 次元配列を 3 次元配列化*した配列
height ; 等値面を描きたい値

g_isosurface_3D 関数の説明

3 次元配列 u に対して与えられた高さ height の位置でマーチングテトラヘドン法を用いて，等値面を描画する．（フラットシェーディングのみサポートしている．フラットシェーディングはメモリ使用量が増大するため機能しない．3 次元配列化*に関しては前章をお読みください．）

4.6.6 g_isosurface_f_3D

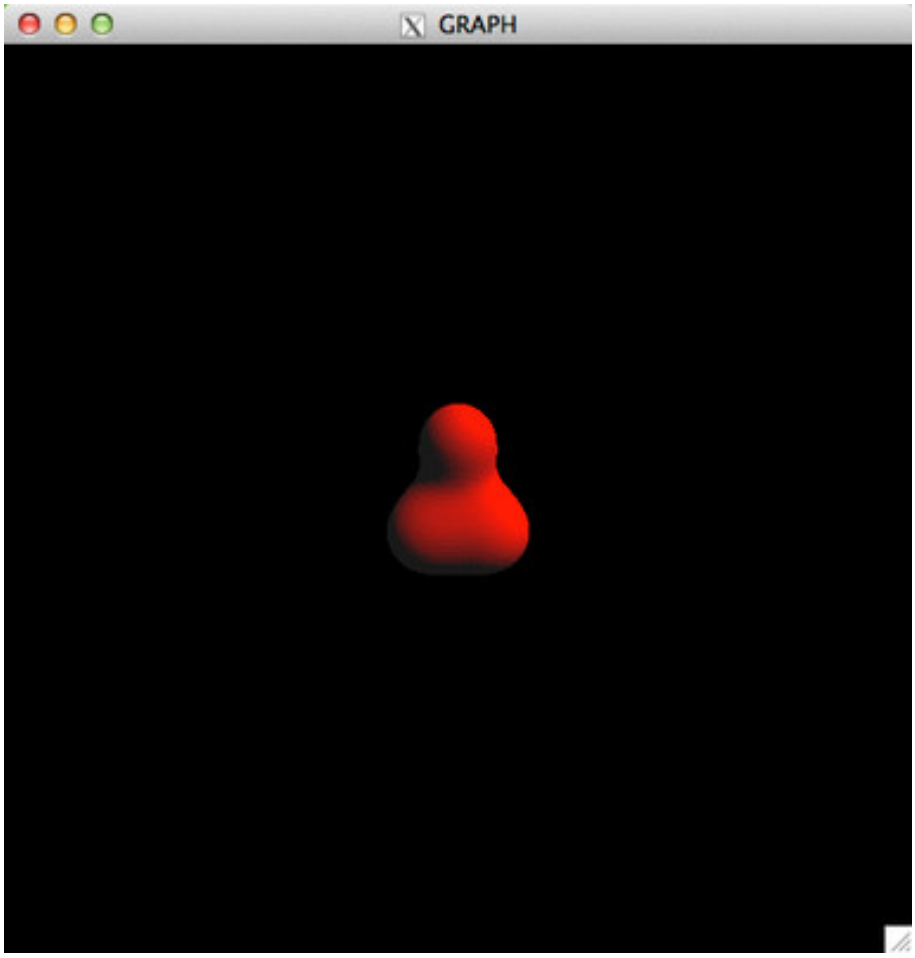
g_isosurface_f_3D 関数

```
void g_isosurface_f_3D(
        double x0, double x1,
        double y0, double y1,
        double z0, double z1,
        int number_x, int number_y, int number_z,
        double *u,
        double height);
```

x,y,z ; 描画範囲の指定
number ; 配列 u の各方向のサイズ
u ; 1 次元配列を 3 次元配列化*した配列
height ; 等値面を描きたい値

g_isosurface_f_3D 関数の説明

1 次元配列 u に対して与えられた高さ height の位置でマーチングテトラヘドン法を用いて，等値面を描画する．（フラットシェーディングのみサポートしている．フラットシェーディングはメモリ使用量が増大するため機能しない．3 次元配列化*に関しては前章をお読みください．）



5 Version の履歴

2016 5.12 Ver 2.2.1

・G_Font_id,G_WIRE,G_FILL,G_BOOL を define において直接数値を与えるように変更した.
(岡本, 秋山)

2016 4.26 Ver 2.2

・g_text 系関数を刷新.

・static ライブラリ (*.a) と shared ライブラリ (mac なら*.dylib, Linux なら*.so) を同時に生成するように変更

2016 4.14 Ver 2.1.1.1

・g_fan_2D のバグを修正

2015 4.15 Ver 2.1.1

・g_data_plot, g_isosurface, g_bird_view のバグを修正

2015 4.13 Ver 2.1

・オフスクリーンレンダリングを実装

・Manual を更新（岡本）

2014 12.4 Ver 2.0　（全員）

・ほとんどすべての関数の実装を見直した.

・遠距離並び替え型透明化処理を正式にサポート

・Manual を更新

2014 10.16 Ver 1.13

・Manual を更新（秋山）

2014 10.16 Ver 1.13-experimental

・Manual を更新

・視点位置の内部的な計算方法を改善

・gluPerspectinve の引数（fovy, z_near, z_far）を変更（平芳）

2014 10.14 Ver 1.12

・Ver1.12-experimental の変更点を採用

・manual の付録を Tex に書き換え（未完成）

・後方互換性がないので注意してください（平芳）

2014 10.3 Ver 1.12-experimental

・3D 版と 2D 版で同じ機能を提供する関数を統一

・3D 版しか存在しない関数から 3D 表示を削除

破壊的な変更のため、experimental で評価を待ちます（岡本）

2014 10.3 Ver 1.11 C++ 対応、及び gcc 対策のための雑多な修正

・g_init_3D g_init_3D_core の char* 型引数を const char * 型に

・変数長配列が c++ 利用時には削除されるように

その他の修正

・G_INPUT 構造体を削除

・マニュアルの g_init_3D_core を修正（岡本）

2014 9.16 Ver 1.10 g_bird_view を改良（malloc を使わないプログラムに改変）

g_rectangle_3D の法線ベクトルの向きを修正（平芳）

2014 9.8 Ver 1.09 g_text_standard の追加
マニュアル（付録）の更新（平芳）

2014 9.4 Ver 1.08 g_def_scale_3D_core の追加（画面上方向の指定を可能に）
font のデフォルト値を設定
マニュアル（付録）の更新（平芳）

2014 8.13 Ver 1.07 g_init_3D, g_init_3D_core の引数の win_pos,width,height を double 型から int 型に変更
g_bird_view_f_3D, g_contln_f_3D, g_data_plot_f_3D を作成し，test_program に使用例を追記
マニュアル（付録）の更新（平芳）

2014 8.11 Ver 1.06
g_input.c を正式に追加．ASCII 文字に加えて，ファンクションキー，矢印キーなどの特殊キーを入力可能．
マウス入力に対応．最後にクリックされたポイントを入手可能．
G_INPUT_STATE g_input_state(G_KEY_CODE code, int *x, int *y) 入力データの取得
詳しくはマニュアルで（岡本）
test_program に g_input_state の使用方法を追記（平芳）

2014 7.28 Ver 1.05-experimental
g_input.c を仮追加．ASCII 文字を入力出来るように．
void g_input_init() 入力機構の初期化
G_INPUT g_get_input() 入力データの取得
G_INPUT_STATE g_input_state(G_INPUT in,unsigned char key) key に対応するキーの情報を取得。G_INPUT_STATE 列挙体の詳細は g_input.h で

Sample_g_input を追加．上記の分のサンプルコード．
g_text_3D/2D_virtual を変更．printf 形式でのフォーマットを使えるように．
glsc3d.h に M_PI の定義を追加．M_PI が定義されていない環境（CentOS とか）に対応するように．
尚，long double の精度が環境依存なので，四倍精度にも対応出きるように 36 桁定義．
experimental 取れたらマニュアル更新します．（岡本）

2014 6.27 Ver 1.04 g_init_light_3D_core の追加.
TestProgram を変更.
マニュアルを更新.（平芳）

2014 6.27 Ver 1.03 g_cls_3D 内の glutMainLoopEvent を g_finish_3D に移動.
g_scr_color_3D の引数から不透明度（a）を削除.
g_init_3D_core で背景色を変更可能にした.
上に伴い，TestProgram と SampleProgram を変更.
g_rectangle_3D の direction のバグを修正.
g_contln_2D の線が途切れる問題を修正.
マニュアルの更新.（平芳）

2014 6.27 Ver 1.02 g_isosurface_f_3D を作成した．TestProgram も作成した．マニュアルへは記載していない.（秋山）

2014 6.24 Ver 1.01 TesProgram に RunAllSample スクリプトを追加．glsc3d.h を c++ にも対応可能にした.（秋山）

2014 6.22 Ver 1.0 マニュアルを更新（平芳）

2014.6.x GLSC3D version 1.0 完成

6 おわりに

GLSC3D は gnuplot ような便利さも無ければ，OpenGL のように 3D オブジェクトに対して，詳細な属性設定をすることもできません．しかしながら，その分コーディングが簡素となり，あなたが思い描くことは何でも表現できます．本マニュアルを熟読し，サンプルプログラムを参考にしながら使用してください．

新関数やライトの上限を増やしてほしいなどの要望，またはバグを発見した場合は秋山正和[*37]までご連絡ください．

論文や学術研究会などでも本ライブラリを使用された際は，その旨を記載してください．開発の励みとなります．

本ライブラリを改変し，再配布することは禁止しません．しかしながら，その場合には私へメールでご報告ください．また使用者から私へ改善要求や不満などが来ることが無いように配慮していただければ幸いです．また科学技術のさらなる発展のために作成したライブラリですので，有料化などの行為はおやめください．

7 謝辞

GLSC3D 開発プロジェクトは科学研究補助金 (科研費) 新学術領域研究 (研究領域提案型) 「生物の 3 D形態を構築するロジック (15H05857)」および若手研究 (B)「平面内細胞極性に関する統一的数理モデルの構築 (15K20835)」の助成を受けている．

[*37] akiyama@es.hokudai.ac.jp ですが，未来永劫このメールアドレスが使用可能かは不明です...

8 作者の覚書

8.1 新関数の追加方法

- 新関数を作る際は，Src のプログラムをよく見て他に倣って作成すること．
- 作成したプログラムが正常に動作するかを確認するために，TestProgram フォルダにて関数のテストプログラムを作成すること．

8.2 Manual の作成方法

- Manual フォルダに隠しフォルダがあるので，そのフォルダ内を見ること．
- Canvas.key には絵が沢山あるのでそれをみること．
- Figures には eps ファイルがあるが，そのファイルは jpg2epsconverter をうまく使えば楽に作成できる．

8.3 設計上の基本原則

GLSC3D の関数は，大きく分けて二つの関数が存在します．常用関数とコア関数です．コア関数は大量の引数を要求する代わりに，ユーザーにより高度な選択肢を提供します．例えば描画関数多くは，ユーザーはコア関数を用いることで，描画の質と実行速度のトレードオフに関与することが出来ます．常用関数は逆に，一般的でないと思われる選択肢を隠蔽することで，より使いやすいよう，関数の引数を最小限に留めています．

現在，常用関数はコア関数のアダプタにすぎません．内部では適切なパラメータをしてコア関数を呼び出しています．つまり，同等な引数を設定する場合，性能的な差はほとんどありません．関数呼び出しが増える分，理論的には常用関数の方が僅かに劣った性能を持つと考えられます．

今後の更新においても，一つの例外を除いてこの構造は継承されるべきでしょう．一つの例外とは，実装言語が C 言語から，関数オーバーロードや引数のデフォルト値などの機能を備えたより先進的な言語に切り替えられた場合です．この場合，引数の多寡によって関数名を切り替える必要はありません．

しかしながら，その場合にも，C 言語から呼び出し可能な構造は維持すべきです．なぜなら，ユーザーは数値計算の専門教育を受けていることが想定されますが，プログラミング全般に対してはそうではありません．また，現在，数値計算の教育では C 言語の使用を前提としていることが多いと考えられます（あるいは FORTRAN!）．また，C 言語は内部的にも比較的単純な構造を持つため，他の言語からも比較的容易に呼び出すことが出来ます．即ち，C 言語から呼び出し可能な構造を維持することで，他の多くの言語からも呼び出し可能となります．

8.4 ファイル構成

git を用いて最新版を入手することが出来ます．ただし，サーバーは動的数理モデリング研究室内にしか公開されていません．

開発版は LatestVer です．これは開発のため不安定になり得ます．最新の安定版は GLSC3DVerXXXX で、XXXX が最も大きいものです．これは安定して使用することができます．

ソースコードは Src/Source にあります．インクルードファイルは Src/Include です．Src で `make` を行うと，Src/Object にオブジェクトファイル，Src/Depend に依存関係ファイルが生成され，最終的に Out に公開用のファイルが生成されます．通常，Src/Source にファイルを追加しても Makefile を書き換える必要はありません．

ユーザーは Src で `make` した後，Out 以下のファイルを適切な位置にコピーする必要があります．これを自動化するスクリプトは現在存在しません．

TestProgram 以下に各種サンプルコード，SampleProgram 以下にデモプログラムが存在します．開発者は新たな関数を公開する場合，対応したサンプルコードを追加して下さい．

Manual 以下にはマニュアルが存在します．あなたが読んでいるこのマニュアルは，おそらく Manual/GLSC3D_Manual.pdf でしょう．開発者は，Manual/.TexManual 以下を見て下さい．新たな関数を公開する場合，GLSC3D_Manual.tex にその関数の情報を追加してください．書式等はすでにある関数のものをテンプレートにして下さい．画像は Figure/eps 内に追加して下さい．

8.5 描画処理

GLSC3D はある程度の透明化処理をサポートしています．このため，全ての三角形描画は一旦保持され，最終的に視点から重心までの距離を用いてソートされます．このため，今後追加される描画関数は全て，このプロトコルに従うべきです．即ち，

- `g_triangle_3D(_smooth)(_core)(_s)` 関数
- `g_triangle_3D_smooth_worker` 関数
- `g_triangle_3D_flat_worker` 関数

のどれかを用いて描画するべきです．

8.6 Future Works

1. テキストのサイズとフォントを個別に指定．→Ver2.2 で実装!
2. 日本語表示．→Ver3.0 で実装!
3. 3 次元描画における線の問題を解消．
4. オフスクリーンレンダリング．→Ver2.1 で実装!

5. g_def_scale を自動化.
6. triangle buffer を動的に変化.
7. オフスクリーンレンダリング時，通常とキャプチャーしたファイルサイズが食い違います．g_off_screen.c/g_init_off_screen_rendering 内で用意しているバッファが原因かもしれません．書き出される画像は肉眼では違わないように見えます.
8. C の多次元配列を内部的に呼べるが，配列が転置されて表示されます．従って，将来的にはそのような関数は消します.
9. OpenGL の最新規格への対応．シェーダの実装．バーテックスシェーダとフラグメントシェーダを OpenGL の glShaderSource 関数を用いて呼ぶことができる，従って，三角形を描画する場合にはこの関数を通してラップすれば GLSL への対応が可能となるはず.
10. g_isosurface.c のコードを書き直す
11. 線分と三角形が混じるとき極端に遅くなる