計算機序論2
演習(CGを中心に)

2010年度

2010/11/15e改訂版

筑波大学

亀田能成

諸注意

- Eclipse 3.2の使い方については授業中に説明します
- https://moodle.tsukuba.ac.jpへの課題提出方法について不明な点があればＴＡに聞いて下さい

準備：Ubuntu上での設定

サンプルプログラムを正常に表示させるために必要な設定

- スクリーンの何もないところで右クリック
  - メニュー→「背景の変更」
    - 「外観の設定」
      - 視覚効果（パネル）で「効果なし」（N）を選択
  - これをしないとglutVisibilityFunc()の機能が有効になりません

目次

1. 関数によるプログラム構造化/スクリプト読み込みプログラム(関数によるプログラム構造化)
2. 構造体とポインタ/スクリプト読込プログラムにおける具体例(構造体とポインタ)
3. Callback型のプログラミングスタイル(Callback)
4. レンダリング基礎(レンダリング基礎)
5. 図形の表現(図形の表現)
6. 射影幾何(射影幾何)
7. 物体の運動(物体の運動)
8. 座標変換(座標変換)
9. 行列変換の実際(行列変換の実際)
10. OpenGLにおける描画フロー(描画フロー)
11. プログラム構成(描画プログラム構成)
12. アニメーション(アニメーション)
13. 物体の変形：モーフィング(モーフィング)

関数によるプログラム構造化

-スクリプト読み込みプログラム-

- 膨大な仕事を一気にこなすのは大変
- 仕事を分割して（＝複数の関数）で処理

規模の大きいプログラム

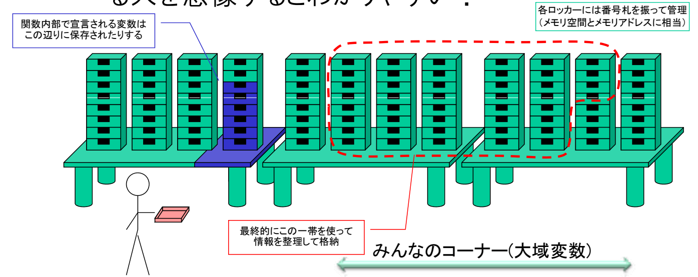
- 仕事だって大きくなれば一気にはできない
  - 作業を分割する
  - 分割した作業を人に任せる
  - 作業内容はとやかく詮索せずに結果だけ受領
- Ｃ言語でも基本は同じ
  - アルゴリズムを分割する
  - 分割したところを別の関数に任せる
  - 関数から結果だけ受領

スクリプト読込プログラム 概要

- アルゴリズム概観
  - 最大3つのファイルからそれぞれ1行ずつ書かれたデータを1つずつ読み込み、メモリ中のデータ構造に保管
  - できあがったデータ構造を標準出力に書き出し
- 詳細な仕様はWWWのほうを参照のこと
  - 物体ファイル
  - アニメーションファイル
  - 光源ファイル

スクリプト読込プログラムが働く様子（想像）

- データ構造はメモリ上に構成される
  - ロッカー（メモリに相当）みたいなもので作業している人を想像するとわかりやすい？



スクリプト読込プログラム チームプレイ（社長・部長）

- <u>社長</u>
  - 「物体ファイルを読込ませ結果を標準出力に出させる」作業を部下の『物体』部長にさせる
  - <u>『物体』部長</u>
    - ファイルを1行ずつ読み、読んだ行の中身に合わせて各「1行分のデータ構造を構築してくれる課長」を呼び出す
    - 『物体一行』課長
    - 『線分一行』課長
    - 『パッチ一行』課長
  - 『アニメーション』部長
    - 『アニメーション一行』課長
  - 『光源』部長
    - 『光源一行』課長

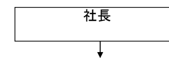
スクリプト読込プログラム チームプレイ（課長クラス）

- <u>『物体一行』課長</u>
  - 新しい物体構造を1つ用意する（だけ）
- <u>『線分一行』課長</u>
  - 現在の物体構造中の線分データ集合に線分を1個加える（だけ）
- <u>『パッチ一行』課長</u>
  - 現在の物体構造中の三角形パッチ集合に三角形パッチを1個加える（だけ）
- <u>『アニメーション一行』課長</u>
  - 新しいアニメーション構造を1つ用意し、その中身を設定
- <u>『光源一行』課長</u>
  - 新しい光源構造を1つ用意し、その中身を設定

スクリプト読込プログラム チームプレイ（下っ端）

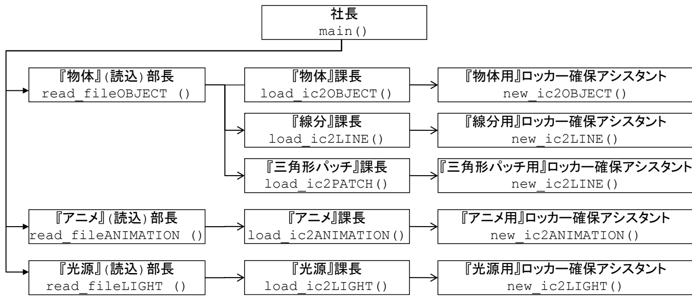
- <u>『物体サイズ』ロッカー確保アシスタント</u>
  - 新しい物体構造を1つ用意する（だけ）
  - ロッカーに行って物体1つ分のロッカーを確保して課長にどこを確保したか伝える
- <u>『線分サイズ』ロッカー確保アシスタント</u>
  - 現在の物体構造中の線分データ集合に線分1個加える（だけ）
  - ロッカーに行って線分1つ分のロッカーを確保して課長にどこを確保したか伝える
- （以下同様）

スクリプト読込プログラム 組織図



- 社長しかいないような会社（社長しか働かない会社）は大きくなれない

スクリプト読込プログラム 関数呼出関係図



構造体とポインタ

-スクリプト読込プログラムにおける具体例-

目的
構造体の中にポインタ変数を含める
⇒Linked-Listを構築

構造体

- データをまとめて扱えるようにする
  - 変数を幾つでもまとめられる
  - 違う型の変数でもまとめて扱える
- 構造体はintやfloatやcharと同じく、変数を宣言するのに使われる

構造体/定義

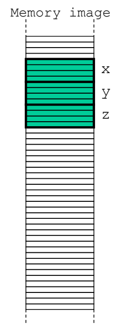
- 例：3つの浮動小数点数をまとめて空間中の1点を規定

今後は“struct ic2POINT”という型となる

定義

```
struct ic2POINT {
  float x;
  float y;
  float z;
};
```

構造体中の各変数をメンバと呼ぶ。ここではメンバは3つ。



構造体/利用[単純な例]

- 例：3つの浮動小数点数をまとめて空間中の1点を規定

“struct ic2POINT”という型の変数を使います、ということ。

利用する前にはプログラムの前方で宣言が必要。

宣言

```
struct ic2POINT chouten;
float d;
```

利用

```
d = chouten.x;
chouten.y = d * 2;
```

「ドット」で構造体変数とそのメンバを繋ぐと、メンバの値を参照できる。

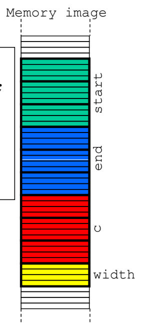
構造体の複雑な例

- 構造体自体も別の構造体に組み込める

```
struct ic2POINT {
  float x;
  float y;
  float z;
};
```

```
struct ic2COLOR {
  float r;
  float g;
  float b;
};
```

```
struct ic2LINEX {
  struct ic2POINT start;
  struct ic2POINT end;
  struct ic2COLOR c;
  float width;
};
```



構造体の複雑な例

- 構造体自体も別の構造体に組み込める

```
struct ic2LINEX {
  struct ic2POINT start;
  struct ic2POINT end;
  struct ic2COLOR c;
  float width;
};
```

```
struct ic2LINEX hasi;
float d;

d = hasi.w / 3.0;
hasi.start.y = d + 1.5;
```

構造体が入れ子になっている場合は、ドット演算子も続けて使う

構造体への演算

- 構造体は変数とはいえ、数値でもなければ文字列でもない(かもしれない)ので、可能な演算の種類は限られる
  - 代入(数少ない実際にできる演算)

```
struct ic2LINEX hasi;
struct ic2POINT aa, bb;
float d;

aa = hasi.start;
bb = aa;
bb.z = aa.z * 2.0;
```

  - 四則演算はできない(そもそも想像できないでしょ？)

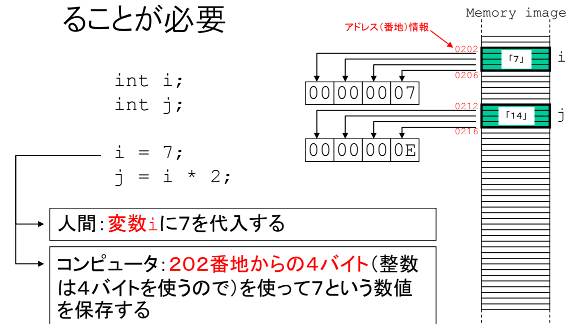
ポインタ・・の前に

- 変数とメモリ空間との関係を正確に把握することが必要。見慣れたプログラムは、実際、どのように動いているのか？

```
int i;
int j;

i = 7;
j = i * 2;
```

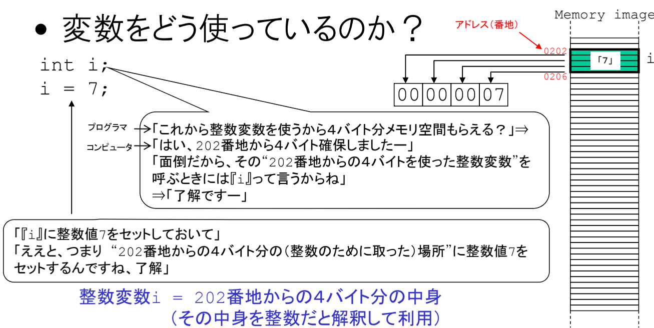

変数の実際

- 変数とメモリ空間との関係を正確に把握することが必要



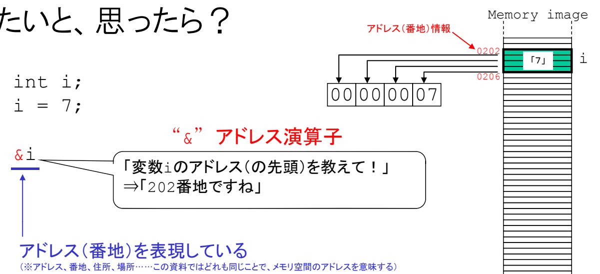
変数の宣言と利用

- 変数の宣言のときに何が起こったのか？
- 変数をどう使っているのか？



変数のアドレス(アドレス演算子&)

- もし万一、変数が格納されている場所を知りたいと、思ったら？



ポインタ変数

- アドレスを扱うという概念（が先ほどのスライドで発生）
- 今後、変数として使いたくなる（かも）
  - “&i”（例：「202番地」）というアドレス情報をどこかに保存したい

```
int i;
int *p;

i = 7;
p = &i;
```

ポインタ変数の宣言
（住所を書きとめるための変数）
⇒240番地からの4バイトを確保

ちなみに住所表記に4バイト使ってくれるのは、5階の計算機のOS上でMemory Imageの範囲が0番地～2^{32}-1番地[max4GB]だから。

「整数変数i（202番地からの4バイト）に整数値7を書き込んでおいて」
⇒「了解！」

「ところで整数変数iの住所教えてくれる？」
⇒「202番地（からの4バイト分）です」
「そっか、じゃあそれ書き留めておきたいから、（整数用）ポインタ変数p（240番地からの4バイト分）に『202番地』って書き込んでおいて」
⇒「ラジャー！」

Memory image
0202 「7」
0240 「0202番地」

特別なアドレス値 NULL

- 「アドレスが存在しない」状態を表現する

```
int i;
int *p = NULL;

i = 7;
p = &i;

if (p != NULL)
 printf(“num = %d¥n”, (*p));
```

ポインタ変数を確保して、その内容は「まだどのアドレスでもない」状態をNULLで表現

このようにポインタ変数の先の内容を参照する必要があるときは、ポインタ変数がNULLでないことを確認してから実行

このような確認は、例えばa/bの演算をする前にbがゼロでないことを確認するようなもので、健全なプログラム作成に必須
考えようによってはNULLは「住所未定・未記入」の意味。

間接演算子＊（ポインタ演算子ともいう）

- ポインタ変数を使うようになると、「アドレスしか知らない」状態（＝ポインタ変数に格納されたアドレス情報のみ）で、その中身を見たくなることがある（かも）
  - 「で結局202番地には何が入ってるのよ？」

```
int i;
int j;
int *p;

i = 7;
p = &i;
j = *p;
```

202番地から4バイト確保
212番地から4バイト確保
240番地から4バイト確保

「整数変数i（202番地からの4バイト）に整数値7を書き込んでおいて」
⇒「了解！」

「整数変数iの住所 “202番地” を（整数用）ポインタ変数p（240番地からの4バイト）に書き込んでおいて」
⇒「了解！」

「ところでポインタ変数pってもともと**整数用住所表記**だよね」
「そうです」
「じゃあそのpに書かれている住所（“202番地”）を訪ねて、そこに保存されているビット列を**整数値だと見なして**読んできて、整数変数j（212番地からの4バイト）に書き込んでおいてくれる？」
⇒「アイアイサー！」

Memory image
0202 「7」
0212 「7」最終的に
0240 「0202番地」

間接演算子＊と乗算演算子＊

- 間接演算子と乗算演算子は全く同じ記号
  - 使い分けは？
    - 演算子＊の前後両方に変数（数値）があれば乗算演算
    - 演算子＊の後方にしか変数（数値）がなければ間接演算
  - とにかく人間にとっては読みにくいので、できるだけ括弧（）を使って見やすく書くことを推奨

```
int i;
int j;
int *p;

i = 7;
p = &i;
j = (*p)*2;
```

202番地から4バイト確保
212番地から4バイト確保
240番地から4バイト確保

「整数変数i（202番地からの4バイト）に整数値7を書き込んでおいて」
⇒「了解！」

「整数変数iの住所 “202番地” を（整数用）ポインタ変数p（240番地からの4バイト）に書き込んでおいて」
⇒「了解！」

「ところでポインタ変数pってもともと**整数用住所**だよね」
「そうです」
「じゃあそのpに書かれている住所（“202番地”を訪ねて、そこに保存されているビット列を**整数値だと見なして**読んできて**その値を2倍して**、整数変数j（212番地からの4バイト）に書き込んでおいてくれる？」
⇒「アイアイサー！」

Memory image
0202 「7」
2倍
0212 「14」最終的に
0240 「0202番地」

構造体変数へのポインタ操作

- 構造体変数であっても、要領は概ね同じ

```
struct ic2LINEX ab;
struct ic2LINEX ac;
struct ic2LINEX *pl;
float *pf;
float f;
```

代入の場合、左辺の型と右辺の型が同じになるように注意すること

```
ab.start.x = 2.2;
pl = &ab;
pf = &(ab.end.z);

f = (*pl).start.y;
ac = (*pl);
ac.end.x = (*pf) * 1.5;
```

- ab.start.x = 2.2; → float型
- pl = &ab; → “struct ic2LINEX”型のためのアドレス
- pf = &(ab.end.z); → float型のためのアドレス
- f = (*pl).start.y; → float型
- ac = (*pl); → “struct ic2LINEX”型
- ac.end.x = (*pf) * 1.5; → float型（1.5倍する等の演算をしてもfloat型であることは変わらない）

構造体ポインタからのメンバアクセス

- C言語において、構造体ポインタからメンバへのアクセスには特別にアロー演算子“ー＞”が用意されている
  - (*p).x より p->x のほうが読みやすい（この2つは等価）と言われている
  - ちなみに *p.x はアウト
    [“*”よりも“. ”のほうが優先順位が高いという約束があるので
    p.xは(p.x)を意味することになり意味不明]

```
struct ic2LINEX ab;
struct ic2LINEX *pl;
float f;

pl = &ab;

f = (*pl).start.y;
f = pl->start.y;
```

plは構造体struct ic2LINEX型のポインタ

どちらの書き方でも演算結果は同じ（だが、下の書き方のほうが読みやすいので推奨しておきます）

ポインタ変数に可能な演算

- 代入
- 間接演算
- アロー演算

```
struct ic2LINEX ab;
struct ic2LINEX *pl;
float f;
pl = &ab;
f = (*pl).start.y;
f = pl->start.y;
```

※アロー演算子はポインタ変数が構造体のために用意されていた場合のみ有効

- 加算・減算（特殊事例）
  - 配列の再勉強をしてから。

ちょっと変わった演算子 sizeof

- ある型があったとき、その型の宣言に必要なバイト数を教えてくれる
- ある変数があったとき、その変数が使用しているバイト数を教えてくれる

```
struct ic2LINEX x;
float f;
printf("size of float is %d¥n", sizeof(float));
printf("size of ic2LINEX is %d¥n", sizeof(struct ic2LINEX));
printf("size of x is %d¥n", sizeof(x));
```

表示結果は、上から順に,4,40,40
sizeof演算子へは、伝統的に引数を()で括って渡す

Linked List
（スクリプト読込プログラムを例として）

目的
事前に予想できない量のデータをメモリ上に
動的に確保できるようにする
ポインタと構造体を駆使
Linked Listの利用

さて準備は整った。

物体に関してデータ構造を使って表現したいもの
- 線分
- パッチ
- 物体

問題はスクリプトファイルを開いてみるまで、
- 1つの物体に
  - 何本線分があるかわからない
  - 何枚パッチがあるかわからない
- 物体が何個入っているかわからない
- どれだけアニメーションが続くのかわからない

個数不明なデータに対する処理

Cプログラミング上の問題
- 配列では無理
  - C言語ではプログラム記述時に配列の要素数を指定
    （配列の数はコンパイル時に固定しなくてはいけない）

```
float ff[1000];
struct ic2LINE ll[100];
```

固定値←今回のような問題設定では固定値の指定不能

「十分に余裕をとった数値を設定」というのも1つの考え方だが、
今回はよろしくない（100万個データが来るかもしれない）

動的メモリ確保

- C言語における可変データ数に対する解決策
- データが増えるたびに必要なメモリをOSから貰ってくる
  - 貰ってくるための関数(malloc, calloc)を利用すること

```
#include <stdlib.h>
void *malloc(size_t NBYTES);
void *calloc(size_t N, size_t S);
```

calloc関数

calloc = cleared memory allocationという噂です

- Sバイトのメモリブロックを連続N個分確保
- 先頭のアドレスを返してくれる
  - 確保に失敗した場合はNULLを返してくる
- 確保されたメモリ空間は全て0で初期化済
  - malloc()では初期化してくれない

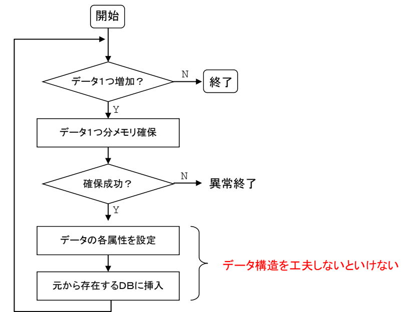
```
#include <stdlib.h>

struct ic2LINEX *ptr = NULL;
ptr = (struct ic2LINEX *)calloc(1, sizeof(struct ic2LINEX));
if (ptr == NULL) return;
ptr->start.x = 2.0;
ptr->end.x   = 4.0;
```

[1] struct ic2LINEを1つ分(40バイトが1つで合計40バイト)確保してもらい、その先頭番地をcalloc()関数が返す。

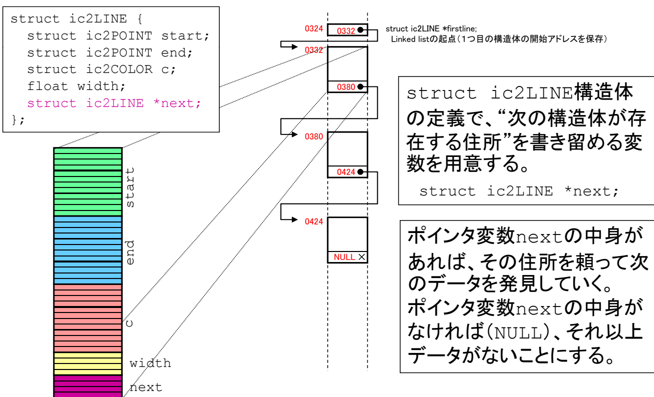
[2] calloc()が返す番地がどういう変数(struct ic2LINEX)のための番地なのかを明示するためのcast

データの増加に対応できるプログラム

- 必要な分だけメモリを確保



Linked List



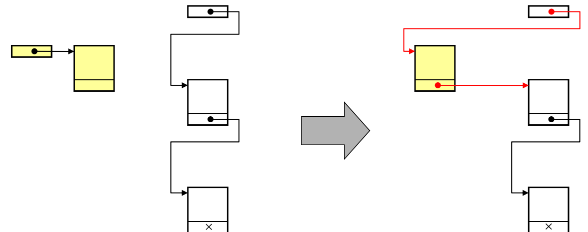
struct ic2LINE構造体の定義で、“次の構造体が存在する住所”を書き留める変数を用意する。

struct ic2LINE *next;

ポインタ変数nextの中身があれば、その住所を頼って次のデータを発見していく。ポインタ変数nextの中身がなければ(NULL)、それ以上データがないことにする。

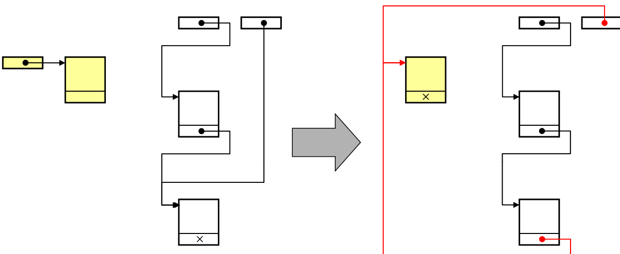
Linked Listの生成

- リストの先頭に挿入する場合
  - 課題で求められた場合は、本図よりも詳しく具体的な図と説明を用意すること
  - 特にプログラムソースとの具体的な対応を説明すること



Linked Listの生成

- リストの末端に追加する場合
  - 課題で求められた場合は本図よりも詳しく具体的な図と説明を用意すること
  - 特にプログラムソースとの具体的な対応を説明すること



Linked List中のデータへのアクセス

- ポインタ変数の中身に書いてある住所を辿ればよい
- 起点は最初から用意されているポインタ変数

Linked Listの利点と欠点

- 利点
  - データの個数に合わせて動的にメモリを確保するので、実行時までデータ数が不明でも対応できる
  - ・
- 欠点
  - ・
  - ・

スクリプト読込プログラム

- スクリプトファイル中の全ての記述をメモリ上に保存
- データ構造の設計
  - Linked Listを多用（可変データ数に対応）
    - 物体
      - 線分
      - 三角形パッチ
    - アニメーション
    - 光源

ある指定ファイルのメモリイメージを見てみよう

ファイルの例

```
# Introduction Computing II
# 2007/09/25 kameda

# 1: 0 lines, 3 patches
O
P   0.1  0.0  0.0    0.3 -0.2  0.0    0.5  0.0  0.0    0.2 0.2 1.0
P  -0.5  0.0  0.0   -0.3 -0.2  0.0   -0.1  0.0  0.0    0.2 0.2 1.0
P  -0.1  0.1  0.0    0.1  0.1  0.0    0.0  0.3  0.0    0.5 0.5 1.0

# 2: 2 lines, 2 patches
O
P   0.1  0.0  0.0    0.5  0.0  0.0    0.3  0.2  -0.1   0.3 0.5 1.0
P  -0.5  0.0  0.0   -0.1  0.0  0.0   -0.3  0.2  -0.1   0.3 0.5 1.0
L   0.1  0.0  0.0    0.0  0.2  0.0                     1.0 0.3 0.3  4
L  -0.1  0.0  0.0    0.0  0.2  0.0                     1.0 0.3 0.3  4

# 3: 3 lines, 1 patch
O
P   0.1  0.0  0.0   -0.1  0.0  0.0    0.0 -0.2   0.1   0.3 1.0 0.3
L   0.1  0.0  0.0    0.0  0.1  0.0                     0.5 1.0 0.6  4
L  -0.1  0.0  0.0    0.0  0.1  0.0                     0.5 1.0 0.6  4
L   0.0  0.1  0.0    0.0  0.3  0.1                     0.8 1.0 0.9  4

# END
```

物体ファイル

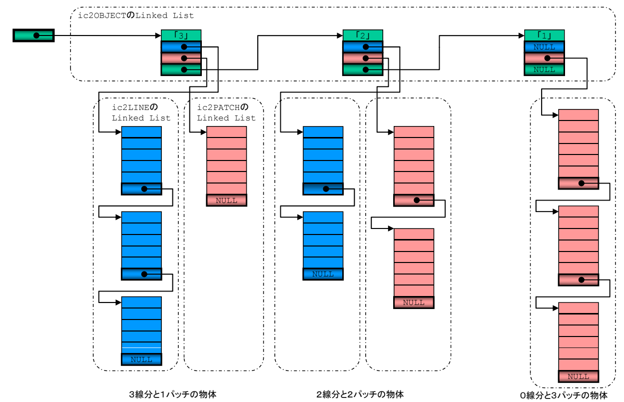
```
# ANIME
A   1   400   1    0.0   2.0   0.0  360     0    0   1.0 1.0 1.0
C   2   400   1    0.0   0.0   0.0    0     0    0   1.0 1.0 1.0
C   2   400   1    0.0   0.0   0.0    0   360    0   1.0 1.0 1.0

A   3     1   0    0.0   2.0   0.0    0     0    0   1.0 1.0 1.0
C   3   800   1    0.0  -2.0   0.0    0  -720    0   1.0 1.0 1.0
```

アニメーションファイル

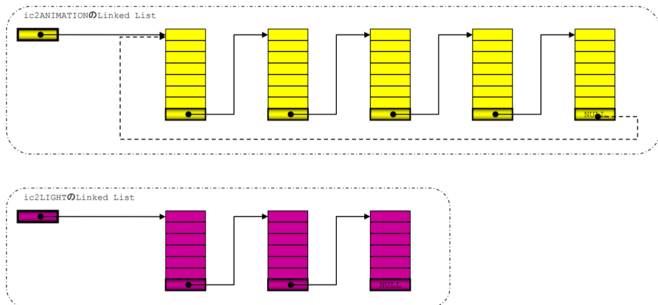
```
# LIGHT
S  1.0   0.2   1.0   1.0 1.0 1.0   1.0 1.0 1.0
S  0.0   1.5   0.5   1.0 1.0 1.0   1.0 1.0 1.0
S  0.0   1.5  -0.5   1.0 1.0 1.0   1.0 1.0 1.0
```

光源ファイル

データ構造の概観I（物体）



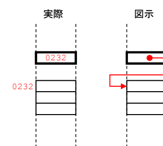
データ構造の概観II（他）



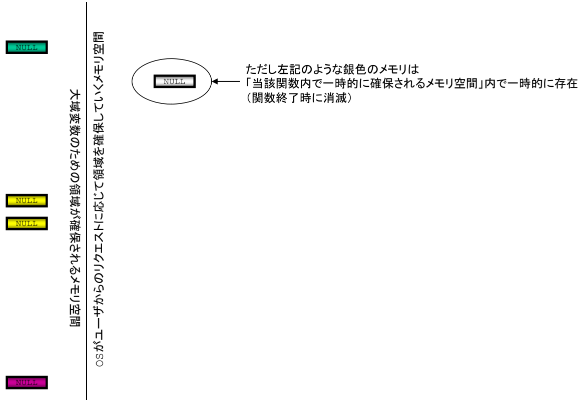
データ構造の構築の様子

- プログラムの実行の様子をメモリイメージで考えてみる

NULL ポインタ変数（アドレスが格納される）

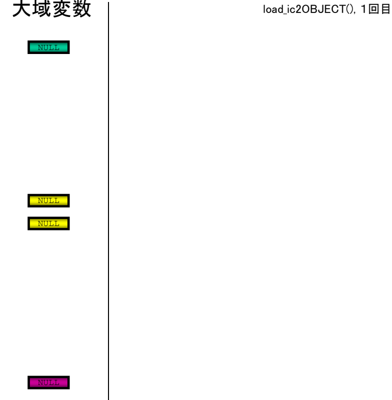


三種類のメモリ空間



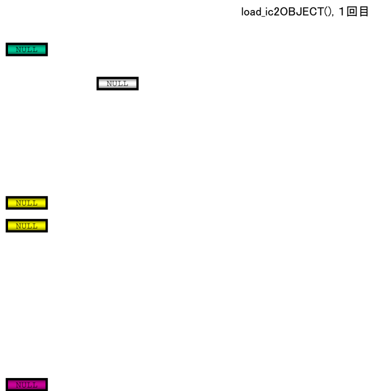
データ構造の構築[01]

load_ic2OBJECT(), 1回目



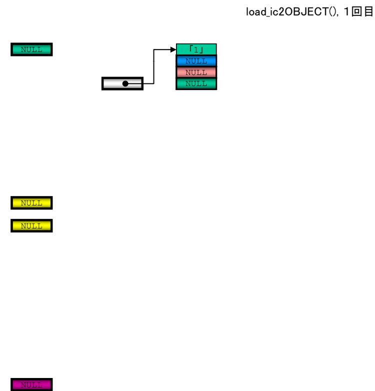
データ構造の構築[02]

load_ic2OBJECT(), 1回目



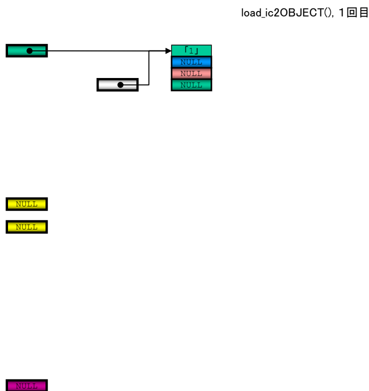
データ構造の構築[03]

load_ic2OBJECT(), 1回目



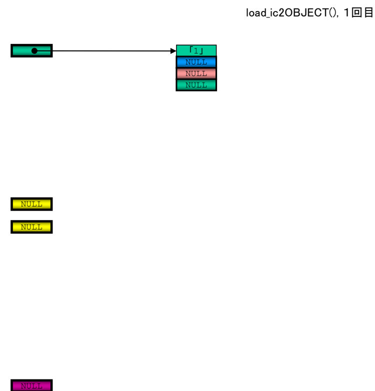
データ構造の構築[04]

load_ic2OBJECT(), 1回目



データ構造の構築[05]

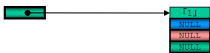
load_ic2OBJECT(), 1回目



Linked List

データ構造の構築[06]

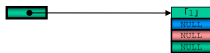
load_ic2PATCH(),1回目

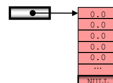


Linked List

データ構造の構築[07]

load_ic2PATCH(),1回目

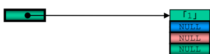


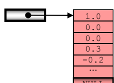


Linked List

データ構造の構築[08]

load_ic2PATCH(),1回目

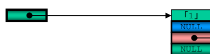


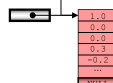


Linked List

データ構造の構築[09]

load_ic2PATCH(),1回目

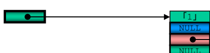


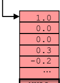


Linked List

データ構造の構築[10]

load_ic2PATCH(),1回目

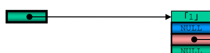


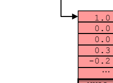


Linked List

データ構造の構築[11]

load_ic2PATCH(),2回目

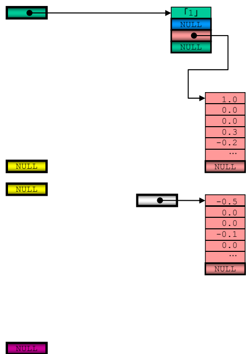




Linked List

データ構造の構築[12]

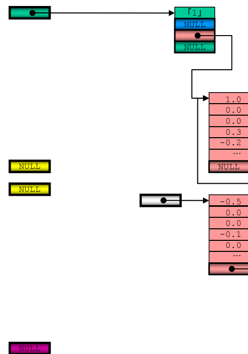
load_ic2PATCH(),2回目



Linked List

データ構造の構築[13]

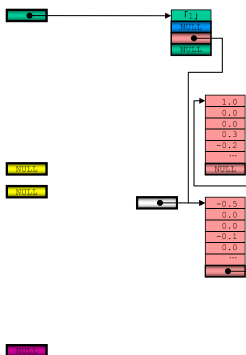
load_ic2PATCH(),2回目



Linked List

データ構造の構築[14]

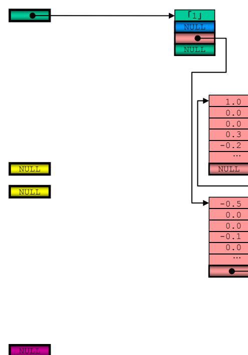
load_ic2PATCH(),2回目



Linked List

データ構造の構築[15]

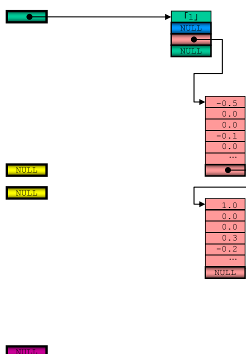
load_ic2PATCH(),2回目



Linked List

データ構造の構築[15′]

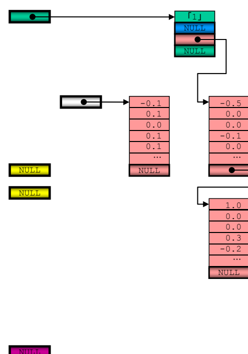
load_ic2PATCH(),2回目



Linked List

データ構造の構築[16]

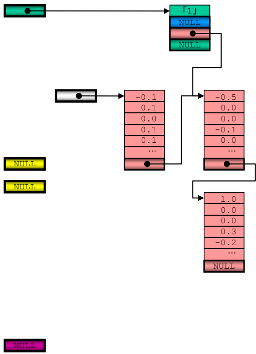
load_ic2PATCH(),3回目



Linked List

データ構造の構築[17]

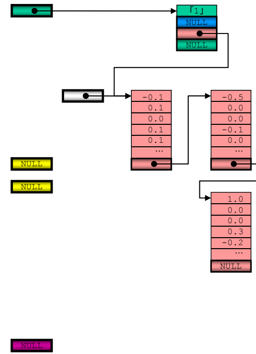
load_ic2PATCH(),3回目



Linked List

データ構造の構築[18]

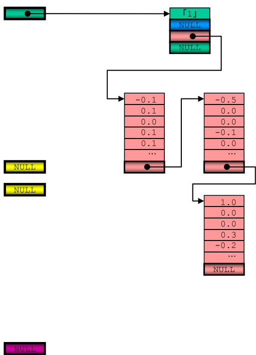
load_ic2PATCH(),3回目



Linked List

データ構造の構築[19]

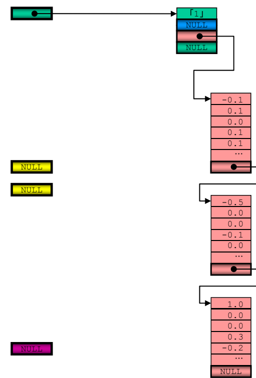
load_ic2PATCH(),3回目



Linked List

データ構造の構築[19']

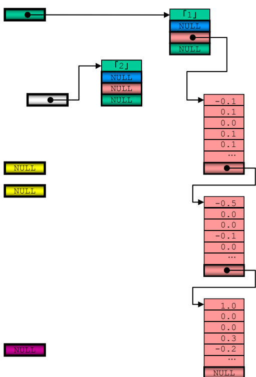
load_ic2PATCH(),3回目



Linked List

データ構造の構築[20]

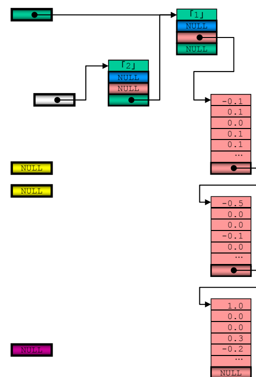
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[21]

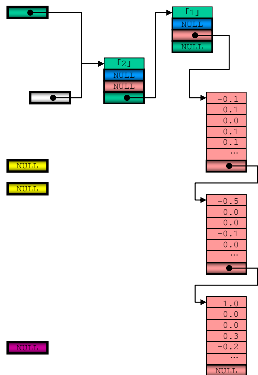
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[22]

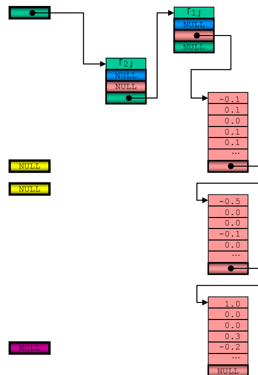
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[22]

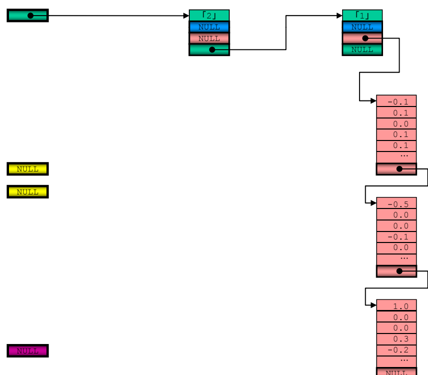
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[22′]

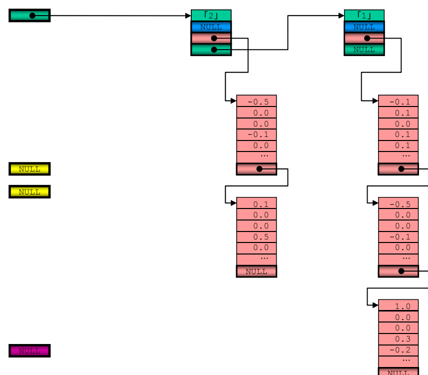
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[23]

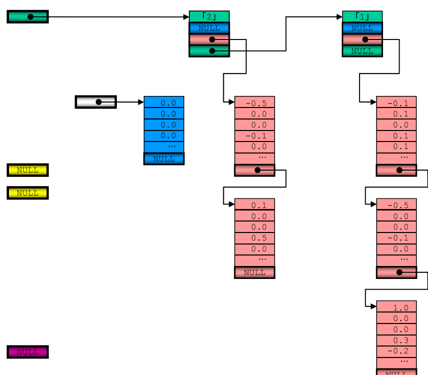
load_ic2LINE(),4-5回目



Linked List

データ構造の構築[24]

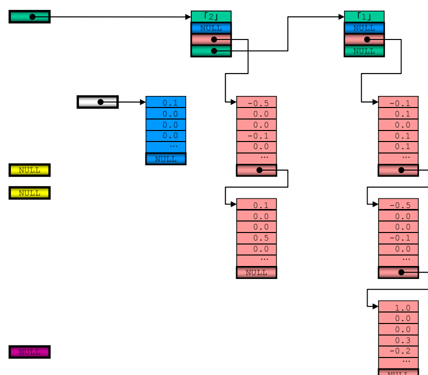
load_ic2LINE(),1回目



Linked List

データ構造の構築[25]

load_ic2LINE(),1回目



Linked List

データ構造の構築[26]

load_ic2LINE(),1回目

Linked List

データ構造の構築[27]

load_ic2LINE(),1回目

Linked List

データ構造の構築[28]

load_ic2LINE(),2回目

Linked List

データ構造の構築[29]

load_ic2LINE(),2回目

Linked List

データ構造の構築[30]

load_ic2LINE(),2回目

Linked List

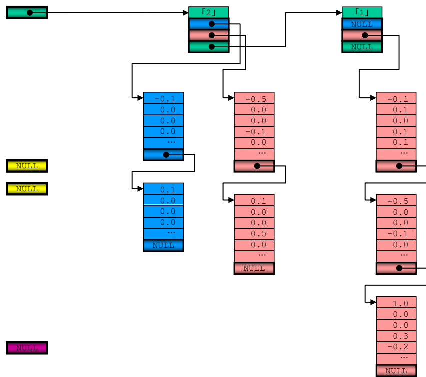
データ構造の構築[31]

load_ic2LINE(),2回目

Linked List

データ構造の構築[31′]

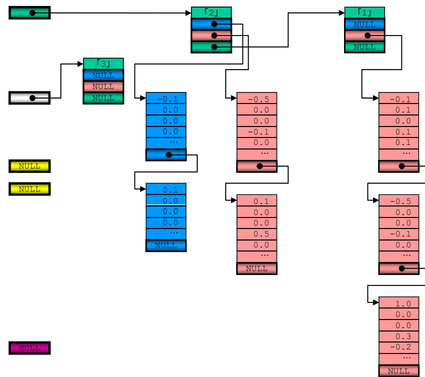
load_ic2LINE(),2回目



Linked List

データ構造の構築[32]

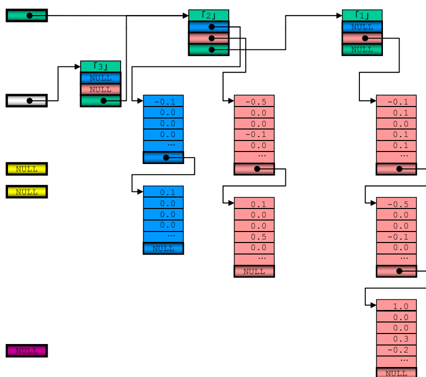
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[33]

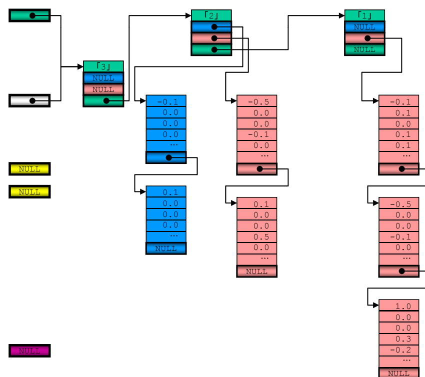
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[34]

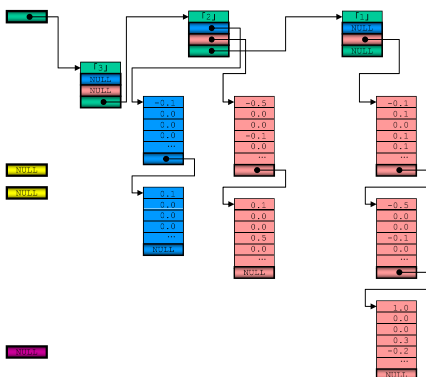
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[35]

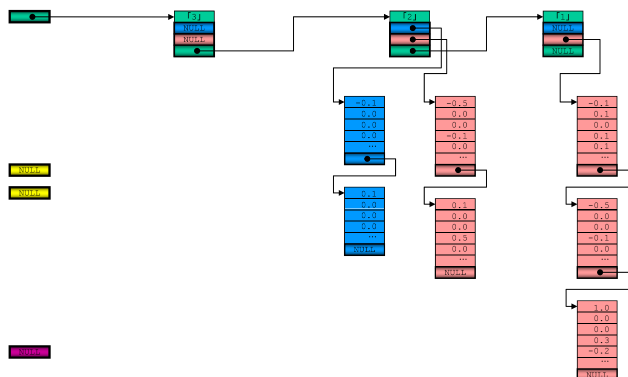
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[35′]

load_ic2OBJECT(),3回目



Linked List

データ構造の構築[36]

load_ic2PATCH(),6回目
Load_ic2LINE(),3-5回目

Linked List

データ構造の構築[37]

load_ic2ANIMATION(),1回目

Linked List

データ構造の構築[38]

load_ic2ANIMATION(),1回目

Linked List

データ構造の構築[39]

load_ic2ANIMATION (),1回目

Linked List

データ構造の構築[40]

load_ic2ANIMATION (),2回目

Linked List

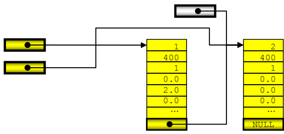
データ構造の構築[41]

load_ic2ANIMATION (),2回目

Linked List

データ構造の構築[42]

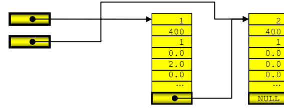
load_ic2ANIMATION (),2回目



Linked List

データ構造の構築[43]

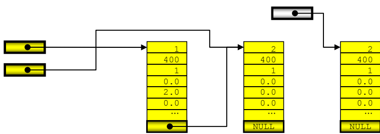
load_ic2ANIMATION (),2回目



Linked List

データ構造の構築[44]

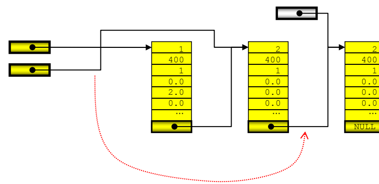
load_ic2ANIMATION (),3回目



Linked List

データ構造の構築[44]

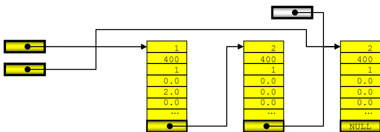
load_ic2ANIMATION (),3回目



Linked List

データ構造の構築[45]

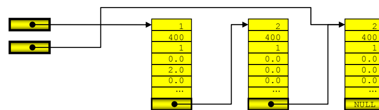
load_ic2ANIMATION (),3回目



Linked List

データ構造の構築[46]

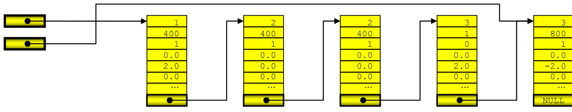
load_ic2ANIMATION (),3回目



Linked List

データ構造の構築[47]

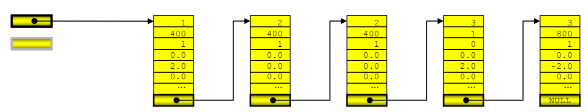
load_ic2ANIMATION (),4-5回目



Linked List

データ構造の構築[48]

load_ic2ANIMATION (),4-5回目

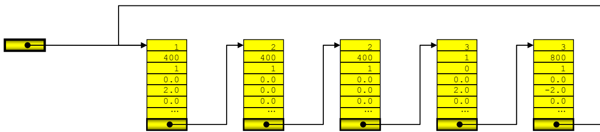


光源については自分で書いてみよう

Linked List

データ構造の構築[49]

read_scriptfile()の最後

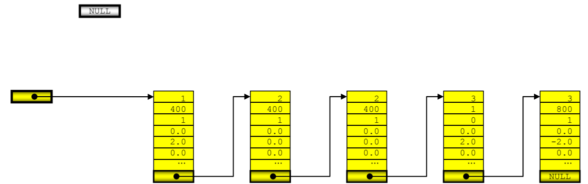


光源については自分で書いてみよう

Linked List

Linked Listの全探索[1]

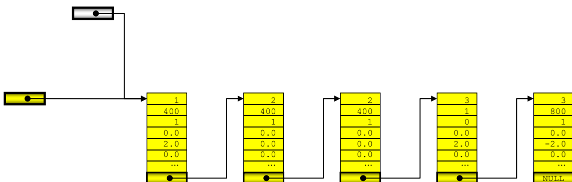
print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



Linked List

Linked Listの全探索[2]

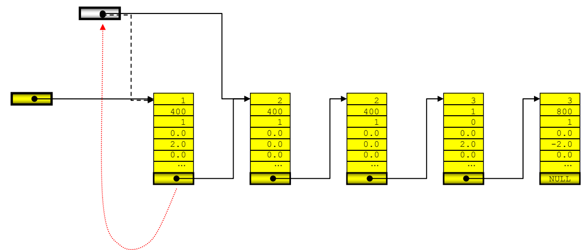
print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



Linked List

Linked Listの全探索[3]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



Linked List

Linked Listの全探索[4]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等

Linked List

Linked Listの全探索[5]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等

Linked List

Linked Listの全探索[6]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等

Linked List

Linked Listの全探索[7]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等

Linked List

Linked Listの全探索[8]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等

Linked List

Linked Listの全探索[9]

print_all_ic2OBJECT(), print_all_ic2ANIMATION ()等

Callback型の プログラミングスタイル

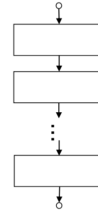
- •CGの描画は大変
- •人間(動きが非常に遅い)からのデータを待っていられない

↓

- •データの受け取りは他所に任せてデータが来たときだけ対応

通常のプログラミングスタイル

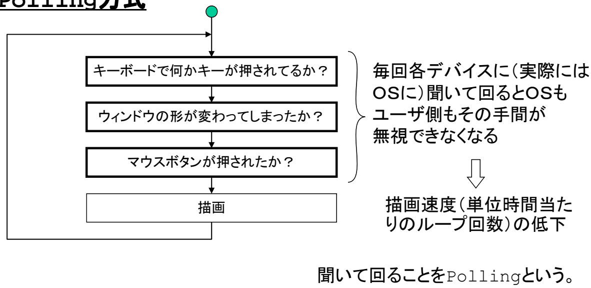
- 順序構造(Sequence)
- 先頭から始まって最後の行で終わる



順序処理の問題

- 滅多に起きない、いつ来るかわからないデータを いちいち確認するのは面倒かも

Polling方式

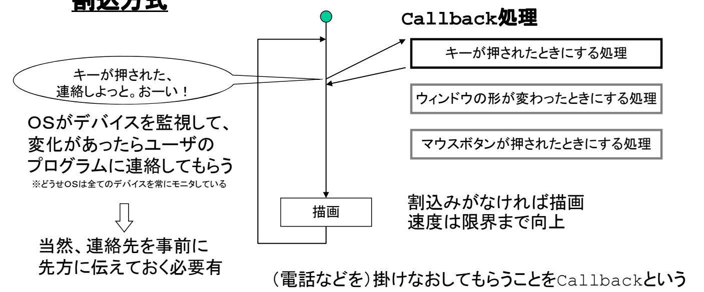


聞いて回ることをPollingという。

発想の逆転：割込方式

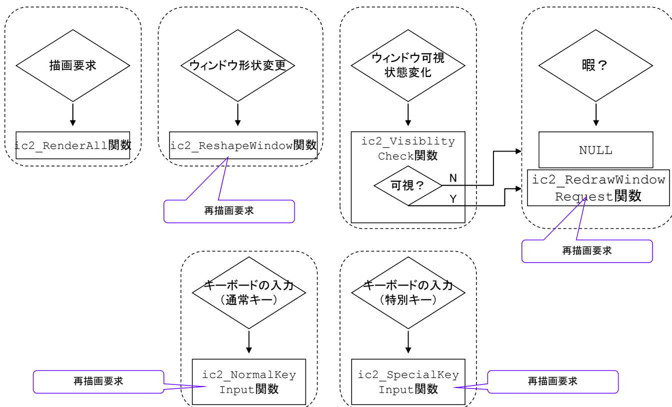
- 待てよ、どうせOSにデータの到着の有無をしつこく聞きまくって顰蹙買ってるのなら、いっそ向こうから連絡してもらおう

割込方式

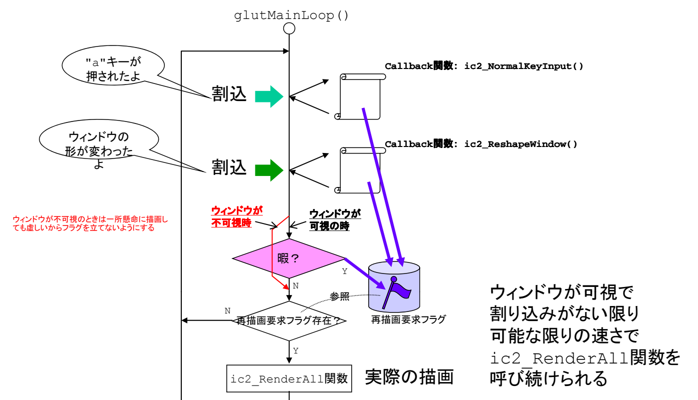


(電話などを)掛けなおしてもらうことをCallbackという

Callbackの設定



Callbackを使った処理の流れ



Callback関数の設定例

- 関数名を引数に
  - 「関数名も変数」参照
  - 呼び出される関数の型・引数は正確に仕様通りに

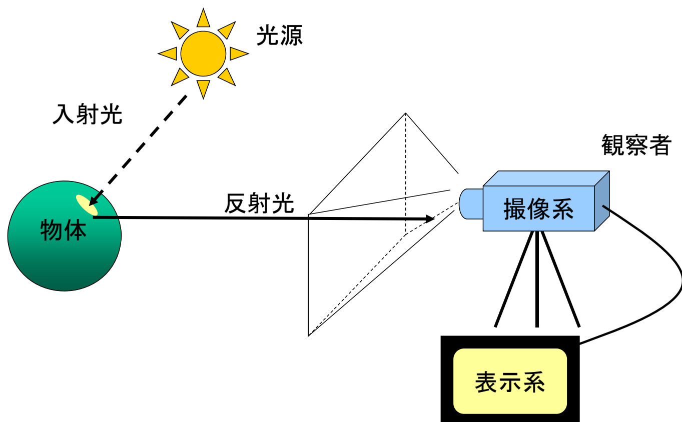
```
#include <stdio.h>
#include <gl/glut.h>
void myreshape(int x, int y){
  printf("(%d, %d)¥n",x,y);
}

int main(){
  glutReshapeFunc(myreshape);
  glutMainLoop();
  return 0;
}
```

void型関数でint型引数を2つの関数を用意すること（glutReshapeFunc関数の利用方法の説明にそう書いてある）

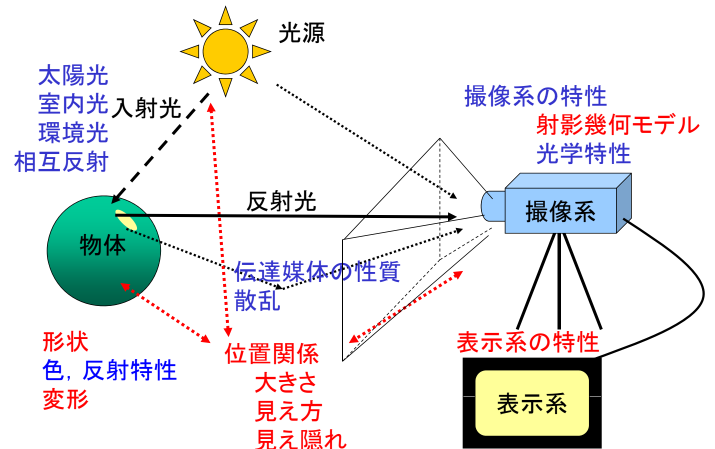
レンダリング基礎

光源・物体・観測者・・三位一体

画像撮影

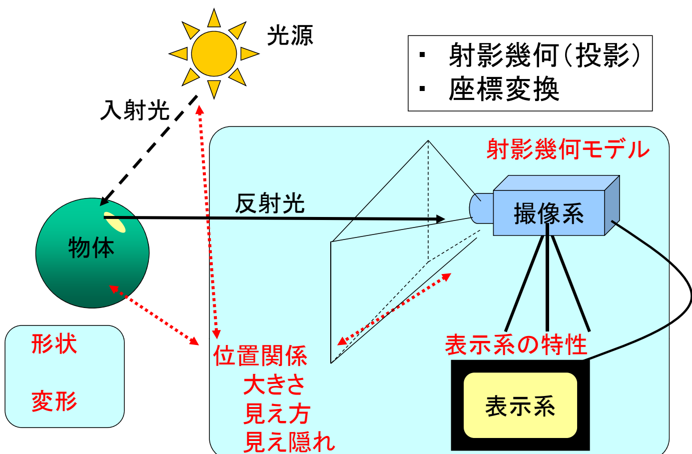


CGで考慮すべき要因



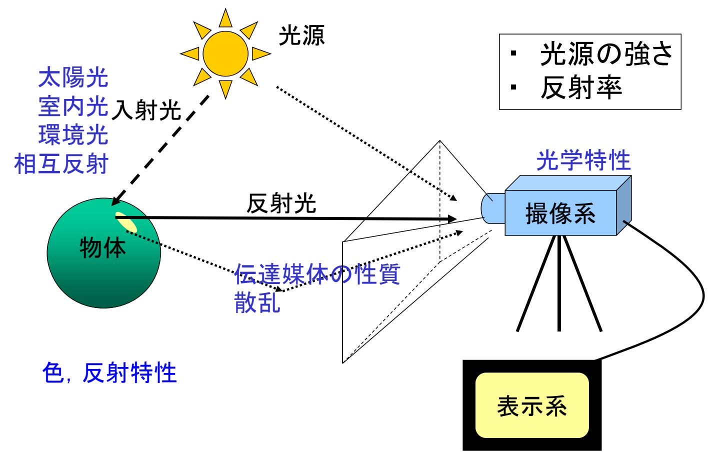
幾何的要因

- 射影幾何（投影）
- 座標変換



光学的要因（課題Eに向けて）

- 光源の強さ
- 反射率

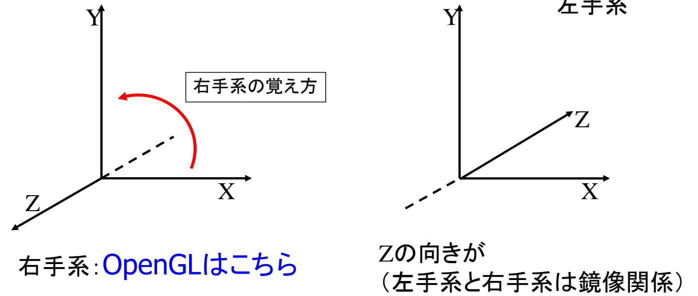


図形の表現

物体座標系・世界座標系・カメラ座標系
同次(斉次)座標表現

3次元空間の座標系

- 右手系と左手系の違い



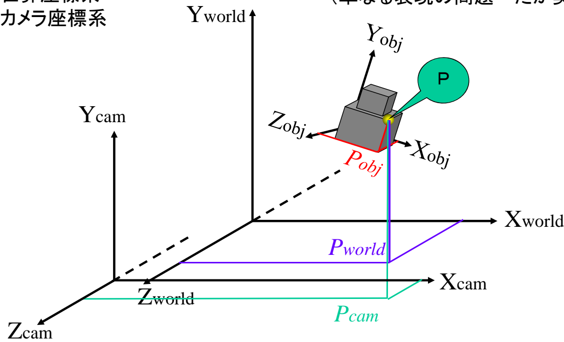
右手系：OpenGLはこちら

Zの向きが
（左手系と右手系は鏡像関係）

座標系

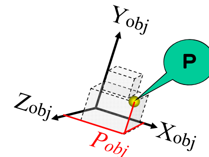
- 物体座標系
- 世界座標系
- カメラ座標系

ある点Pがどこに存在するか？
（単なる表現の問題‥だが奥は深い）



物体座標系

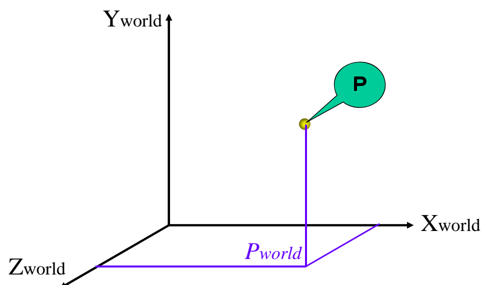
物体を記述するための座標系



$P_{obj} = (X_{obj}, Y_{obj}, Z_{obj})$

世界座標系

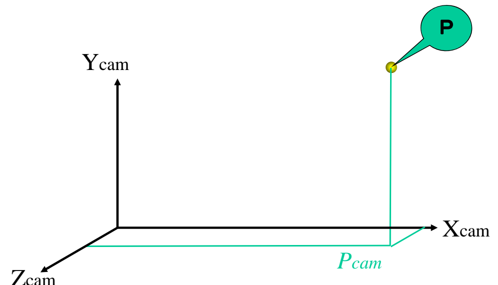
世界全体を記述するための座標系



$P_{world} = (X_{world}, Y_{world}, Z_{world})$

カメラ座標系

撮影カメラを中心とした座標系



$P_{cam} = (X_{cam}, Y_{cam}, Z_{cam})$

同次(斉次)座標表現

- CG高速レンダリング(幾何学計算)を可能にするアイデア

通常の3次元位置の表現
３要素によるベクトル

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

同次座標表現
４要素目を追加した4次元ベクトル

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

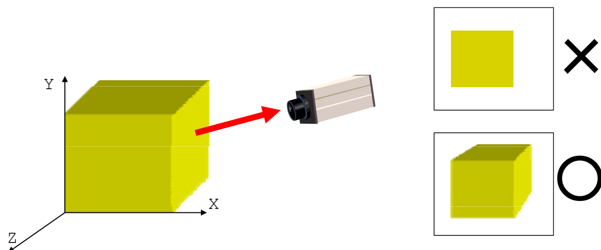
当然4つ目に意味はない‥
が計算のトリックに多大な貢献
⇒あとで出てきます

射影幾何

3次元空間中の物体を
2次元で表現する方法

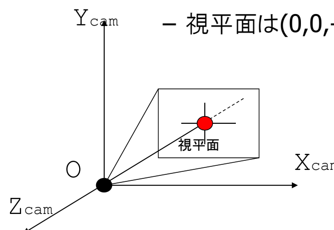
平面上で3次元物体をどうやって描く？

- 描画する（見る）のは平面上※
  - 3次元空間から2次元平面への投影（幾何変換）
  - 次元が一つ減るだけではなく、見え方の計算が必要（見え方にはいろいろ考えられる！



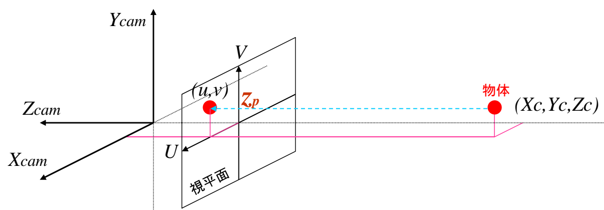
OpenGLでの撮影カメラの設定

- デフォルト設定
  - カメラ視点（焦点）は原点に設置
  - カメラの向き（光軸）は(0,0,-1)
  - 画面上では、X軸正方向が右、Y軸正方向が上
- 本演習のプログラムでの追加設定
  - 視平面は(0,0,-1)を通りZ軸に直交



平行投影（直行変換）

- ある点を光軸と平行に視平面に投影する.



- 物体からの光は視平面にそれぞれ平行に到着.
- 近くの物体も遠くの物体も同じ大きさに見える.
- 視点の概念は必要ない（視線は必要）.
- 視点と物体の距離が十分大きい(物体の奥行きが，物体への距離に比べて十分小さい)場合に適する(ただしスケーリング必須).
- 製品の分解図等には重宝．日本の古来の画法.

平行投影の行列演算

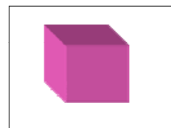
視平面の奥行きがZpの場合

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & Z_p \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} X_c \\ Y_c \\ Z_p \\ 1 \end{bmatrix}$$

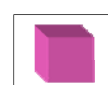
X項、Y項だけを残す

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & 0 & Z_p \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} sX \\ sY \\ Z_p \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ Z_p \\ 1 \end{bmatrix}$$

大抵の場合、画像平面と大きさが合わないのでXc,Ycについて
S倍のスケーリング



s倍

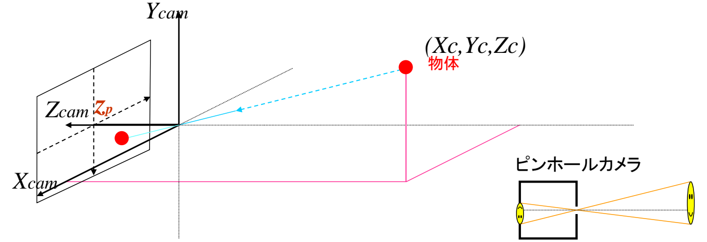


平行投影(直行変換)・余談

- 日本絵画の例(洛中洛外図)
  - 平行な直線の組は画像上で常に平行
  - 消失点がない
  - スケール感を雲とかで誤魔化す

透視投影(射影変換)

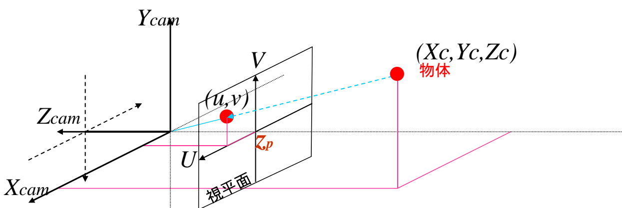
- 一点(焦点)を通過した光線のみを視平面に投影する.



- 視点位置によって見え方が異なる
- 物体との距離が任意の場合に適する.
- 像の上下が反転する.

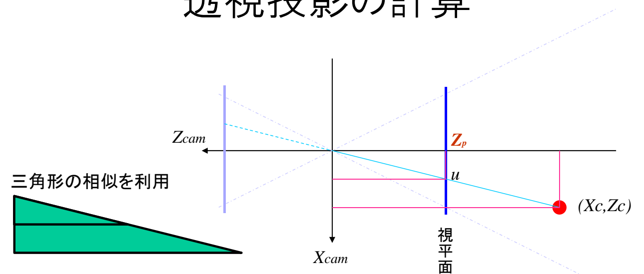
透視投影(射影変換)

- 一点(焦点)を通過した光線のみを視平面に投影する.



- 視平面を物体側(マイナス側)に設定することにより上下反転の問題を解消

透視投影の計算



求めたいのは u
与えられているのは(Xc,Zc)と Z_p(焦点距離)

$$\frac{X_c}{Z_c} = \frac{u}{Z_p} \Rightarrow u = \frac{Z_p}{Z_c} X_c = Z_p \frac{X_c}{Z_c}$$

透視投影の行列演算

$$\begin{bmatrix} \frac{Z_p}{Z_c} & 0 & 0 & 0 \\ 0 & \frac{Z_p}{Z_c} & 0 & 0 \\ 0 & 0 & \frac{Z_p}{Z_c} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{Z_p}{Z_c} X_c \\ \frac{Z_p}{Z_c} Y_c \\ Z_p \\ 1 \end{bmatrix}$$

行列内に Z_c が混じる
⇒点ごとに異なる行列が必要

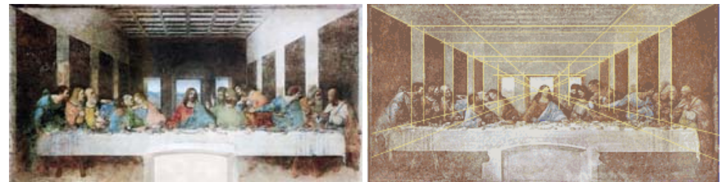
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/Z_p & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ Z_c / Z_p \end{bmatrix} \Rightarrow \frac{Z_p}{Z_c} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ Z_c / Z_p \end{bmatrix} = \begin{bmatrix} \frac{Z_p}{Z_c} X_c \\ \frac{Z_p}{Z_c} Y_c \\ Z_p \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ Z_p \\ 1 \end{bmatrix}$$

同次項を利用してZcを外す
(右辺の大きさは変わる)

得られた同次表現の第4項が
1になるように除算!

透視投影(射影変換)・余談

- ルネッサンス期の西洋絵画(最後の晩餐)
  - 平行な直線群はどこか一点(消失点)で交わる
    (画面に平行な直線群だけは例外として画面上でも並行)
  - 消失点を前提に描かれた絵画として有名



物体の運動

座標系の変換で考える

斉次(同次)座標系と蓄積行列

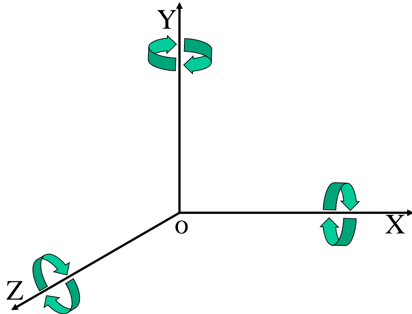
平行移動

- 同次座標表現を用いれば、行列の積和計算だけで可能

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} X+tx \\ Y+ty \\ Z+tz \\ 1 \end{bmatrix}$$

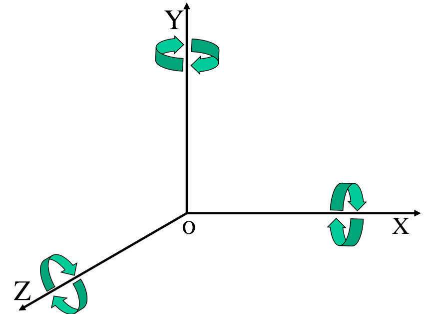
回転の表現

- 本講義では直交軸まわりの回転のみを考える.



直交軸回転

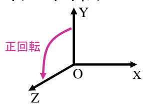
- 直交軸回転には3種類ある. (x軸回り, y軸回り, z軸回り)
- 順番が違うと結果が違う(行列演算では交換則は成立しない).



直交軸回転の行列表現

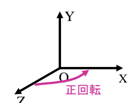
正回転は軸に対して右ネジの法則

- X軸回りの回転.



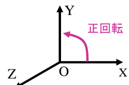
$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Y軸回りの回転.



$$R_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Z軸回りの回転.



$$R_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

拡大・縮小

- 対角要素で表現

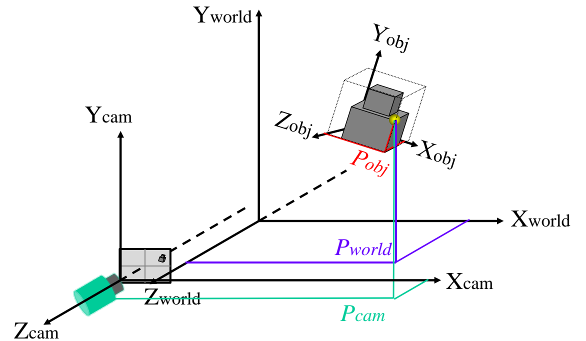
$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} sx \times X \\ sy \times Y \\ sz \times Z \\ 1 \end{bmatrix}$$

座標変換

OpenGLにおける画像提示の流れ
物体座標系→世界座標系→カメラ座標系

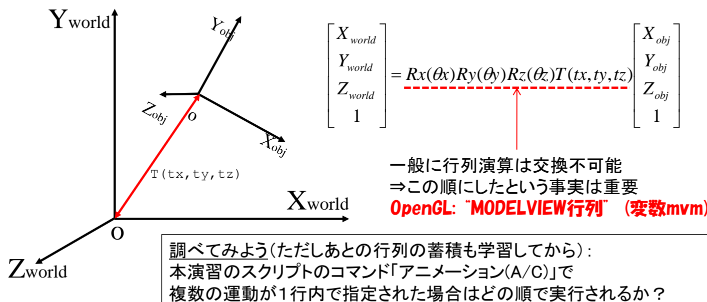
座標系

- 物体座標系(obj)　　　物体表現を簡単に
- 世界座標系(world)　　物体とカメラが載る空間
- カメラ座標系(cam)　　写像計算を簡単に



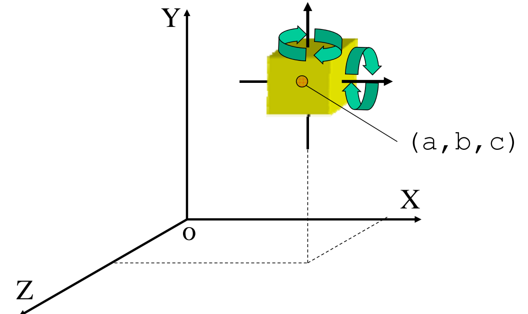
座標系の間の射影変換

- 座標系の間の射影変換も行列計算
  - 原点を動かす　:平行運動
  - 回す　　　　　:回転運動



物体の複合運動に対応する行列計算

例:(a,b,c)を中心にX軸回りα、Y軸回りβ回転
・・実はこれでは表現があいまい(不十分)



物体運動と座標系の計算

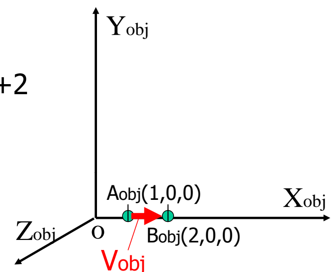
行列に掛けるベクトル「の座標系の上で」
- 回転行列
- 移動行列

の演算は行われる。以後:
　解釈①　座標系をどんどん乗り換えて
　解釈②　すべてを世界座標系の値で
演算する例を示す(結果は同じ！)

物体運動と座標変換の行列計算

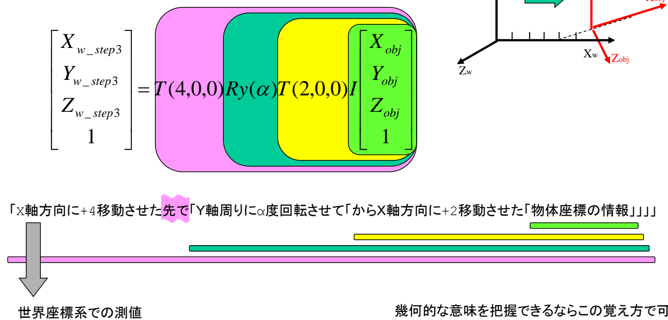
物体の表現で用意したもの:ここではベクトル1本だけ
物体座標系:点Aobj(1,0,0)から点Bobj(2,0,0)に向かうベクトルVobj

スタート:物体座標系と世界座標系が一致
1. X軸に方向に+4
2. その場で物体座標系の
   Y軸回りにα回転
3. 物体座標系のX軸方向に+2



物体運動の行列計算①

ある意味、日本語の語順と同じ
(座標系を運んでいく感覚)



「X軸方向に+4移動させた先で「Y軸周りにα度回転させて「からX軸方向に+2移動させた「物体座標の情報」」」」

世界座標系での測値

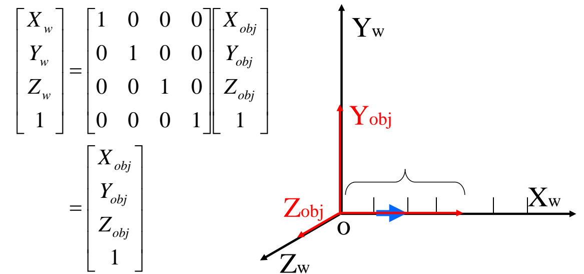
幾何的な意味を把握できるならこの覚え方で可

物体運動の行列計算②

あくまで世界座標系で考える方式

初期状態の考え方

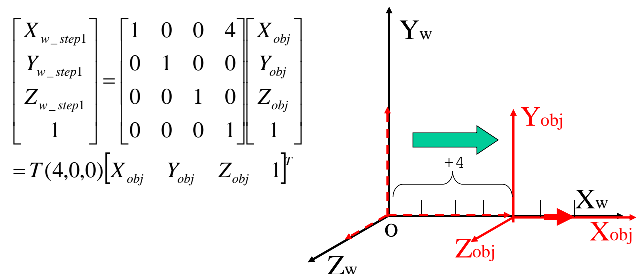
- 物体座標系と世界座標系が一致



物体運動の行列計算②

「世界座標系の中で」

1. X軸に方向に+4

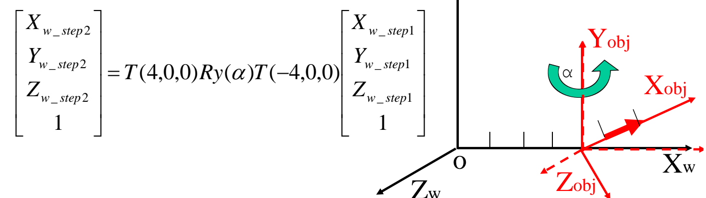


物体運動の行列計算②

「世界座標系での回転行列を使って」

2. その場で物体座標系の
 Y軸回りに **α** 回転

原点へ戻さないと軸回転は表現できない



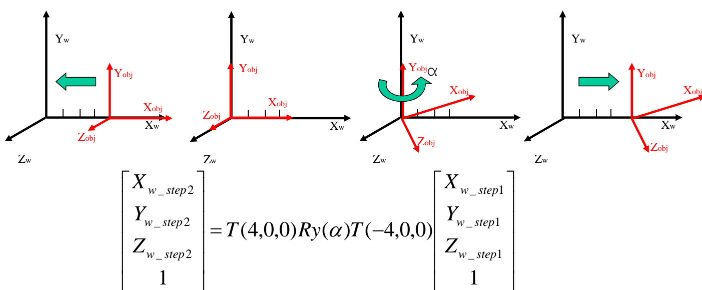
Yw Yobj α Xobj O Xw Zobj Zw

物体運動の行列計算②

「世界座標系での回転行列を使って」

2. その場で物体座標系のY軸回りに **α** 回転

原点へ戻さないと軸回転は表現できない

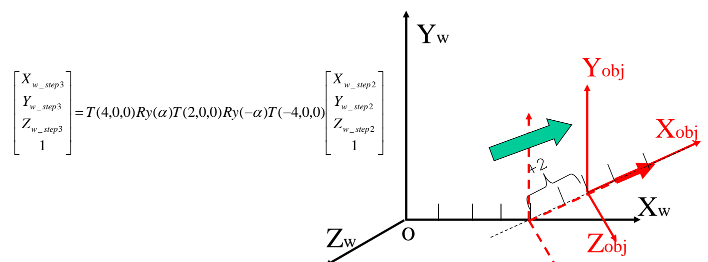


物体運動の行列計算②

「世界座標系の回転行列を使って」

3. 物体座標系のX軸方向に+2

原点へ戻して変換を施す

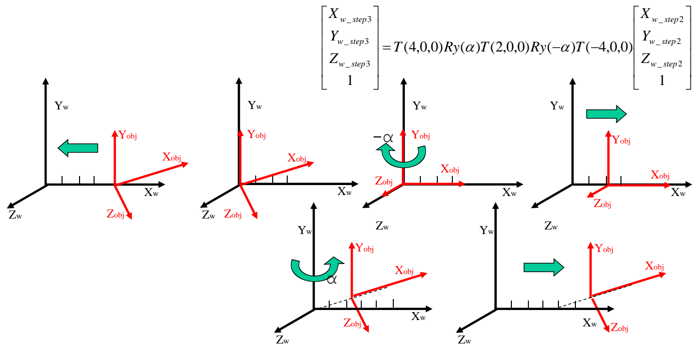


物体運動の行列計算②

3. 物体座標系のX軸方向に+2

原点へ戻して変換を施す

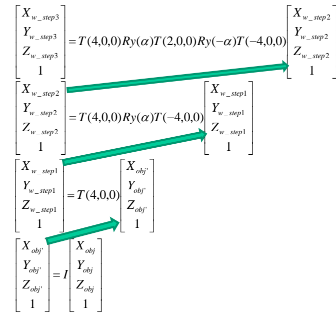
$$\begin{bmatrix} X_{w_step3} \\ Y_{w_step3} \\ Z_{w_step3} \\ 1 \end{bmatrix} = T(4,0,0)Ry(\alpha)T(2,0,0)Ry(-\alpha)T(-4,0,0)\begin{bmatrix} X_{w_step2} \\ Y_{w_step2} \\ Z_{w_step2} \\ 1 \end{bmatrix}$$



物体運動の行列計算②

全体としては：

これまでの演算結果を合成して変換を作成



$$\begin{bmatrix} X_{w_step3} \\ Y_{w_step3} \\ Z_{w_step3} \\ 1 \end{bmatrix} = T(4,0,0)Ry(\alpha)T(2,0,0)Ry(-\alpha)T(-4,0,0)\begin{bmatrix} X_{w_step2} \\ Y_{w_step2} \\ Z_{w_step2} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_{w_step2} \\ Y_{w_step2} \\ Z_{w_step2} \\ 1 \end{bmatrix} = T(4,0,0)Ry(\alpha)T(-4,0,0)\begin{bmatrix} X_{w_step1} \\ Y_{w_step1} \\ Z_{w_step1} \\ 1 \end{bmatrix}$$

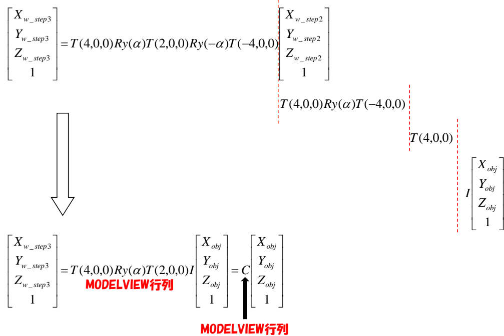
$$\begin{bmatrix} X_{w_step1} \\ Y_{w_step1} \\ Z_{w_step1} \\ 1 \end{bmatrix} = T(4,0,0)\begin{bmatrix} X_{obj'} \\ Y_{obj'} \\ Z_{obj'} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_{obj'} \\ Y_{obj'} \\ Z_{obj'} \\ 1 \end{bmatrix} = I\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

物体運動の行列計算②

全体としては：

これまでの演算結果を合成して変換を作成



$$\begin{bmatrix} X_{w_step3} \\ Y_{w_step3} \\ Z_{w_step3} \\ 1 \end{bmatrix} = T(4,0,0)Ry(\alpha)T(2,0,0)Ry(-\alpha)T(-4,0,0)\begin{bmatrix} X_{w_step2} \\ Y_{w_step2} \\ Z_{w_step2} \\ 1 \end{bmatrix}$$

$$T(4,0,0)Ry(\alpha)T(-4,0,0) \quad T(4,0,0) \quad I\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_{w_step3} \\ Y_{w_step3} \\ Z_{w_step3} \\ 1 \end{bmatrix} = T(4,0,0)Ry(\alpha)T(2,0,0)I\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix} = C\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

行列変換の実際

OpenGLでの行列の実装

- 幾何変換などの変換行列
  - 1次元の配列を利用して4x4の行列を表現
  - 行列に対して右から縦ベクトルを掛ける（という約束）

```
float m[16];
```

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & -a \\ 0 & 1 & 0 & -b \\ 0 & 0 & 1 & -c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

配列に格納される順番に注意

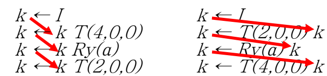
- ic2_Translate,ic2_RotateX,ic2_RotateY,ic2_RotateZの代入要素
- ic2_MultMatMat()における要素参照の順番に注目

行列の計算例

「物体運動の行列演算」と同じ出現順序(になるように手順を規定)

行列k に演算処理がドンドン蓄積させていく

(以下はどちらでも結果は同じ)



$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = T(4,0,0)Ry(\alpha)T(2,0,0)I\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

ic2_AnimationMatrix ()での演算

- 行列tを行列kに蓄積
  行列tと行列kの積算結果を行列kに代入する
  ic2_MultMatMat(k, k, t); ないし ic2_MultMatMat(k, t, k);
  (右から掛ける) v.s. (左から掛ける)

- 注意事項
  - 行列計算は演算用のコピーで実行
  - 最終的にプログラムに戻す行列(ポインタ渡し)は？

アニメーション描画の場合、さらに蓄積の回数が増えるが、
それについては次回以降

OpenGLにおける描画フロー

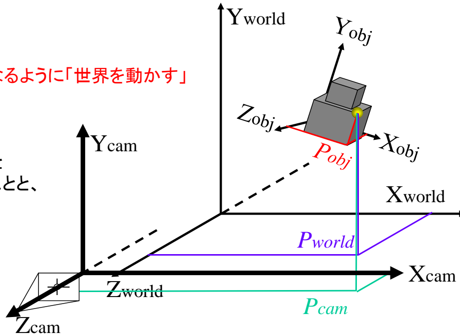
物体座標系→世界座標系→カメラ座標系

物体を撮影する手順

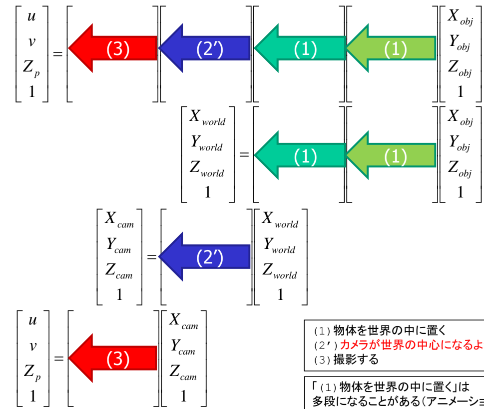
考えがちな手順
（1）物体を世界の中に置く
（2）カメラを世界の中に置く
（3）撮影する

CGの実際手順
（1）物体を世界の中に置く
（2’）カメラが世界の中心になるように「世界を動かす」
（3）撮影する

これは（3）の撮影の計算が
自由なカメラ位置に対してだと
投影の計算式が面倒になることと、
（2）と（2’）の計算コストが
変わらないことから。



物体を撮影する行列演算



（1）物体を世界の中に置く
（2’）カメラが世界の中心になるように「世界を動かす」
（3）撮影する

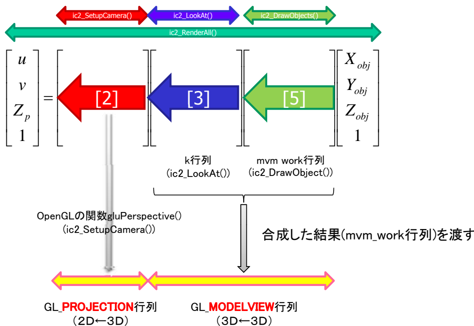
「（1）物体を世界の中に置く」は
多段になることがある（アニメーションの継続など）

物体を撮影する行列演算

コマンドA（アニメーション）のみが来た場合

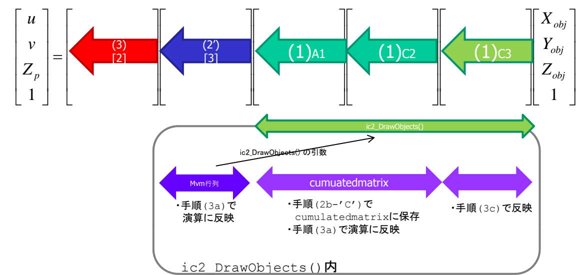
概念（前のスライド参照）
（1）物体を世界の中に置く
（2’）カメラが世界の中心になるように「世界を動かす」
（3）撮影する

プログラム上での出現順序(ic2_RenderAll())
[5] 物体を世界の中に置く
[3] カメラが世界の中心になるように「世界を動かす」
[2] 撮影する



物体を撮影する行列演算

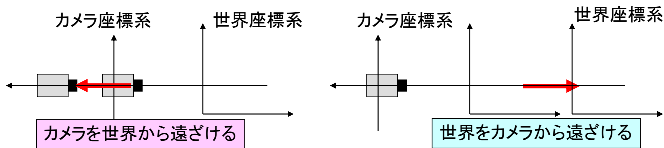
コマンドA1/C2/C3（アニメーション）が連続して来た場合
（1）のc3はさらにic2_AnimationMatrix ()で制御
（2’），（3）は前と同じ



描画フロー

(2)カメラを世界の中に置く
(2’)カメラが世界の中心になるように「世界を動かす」

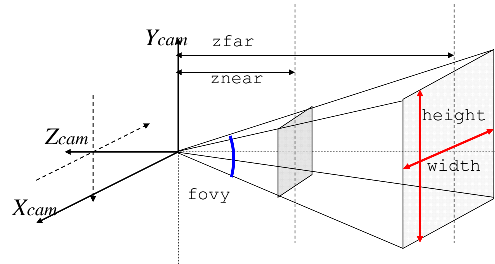
- カメラの移動は、物体を逆向きに移動させることで表現
  - 実は、カメラの動いた分、世界が逆に“動かされて”いる
  - 世界座標系からカメラ座標系への変換
  - デフォルトでは、撮影カメラはカメラ座標系の中で原点に配置され、(0,0,-1)を向いている
- ic2_LookAt(float *mvm)
  - mvm: 動かされた量を返す(蓄積)
  - 大域変数 camdist でカメラを世界中心から遠ざけている
  - 他にもカメラを原点の周りで回転させるための「緯度・経度」変数



描画フロー

(3)撮影する

- gluPerspective(double fovy, double aspect, double znear, double zfar)
  - fovy:垂直画角 *field of view in y direction*
  - aspect:縦横比(width/height)
  - znear:前面
  - zfar:後面

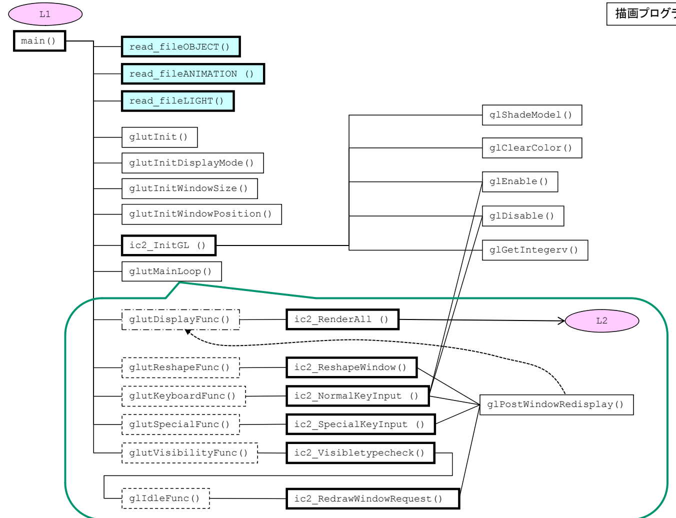


描画プログラム構成

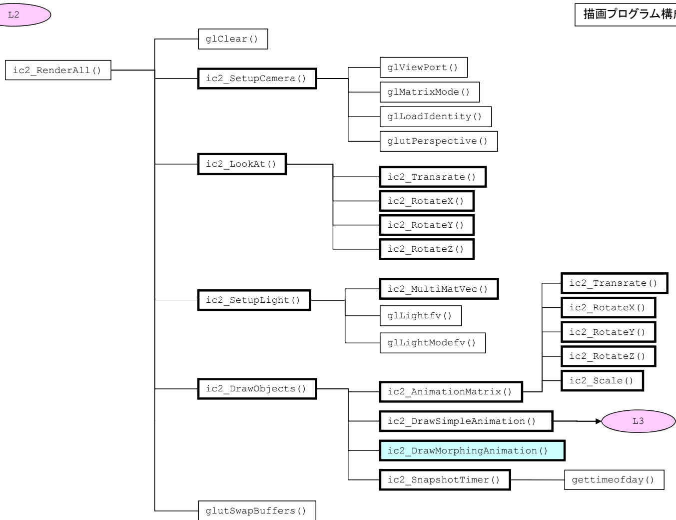
プログラム構成

特に描画部

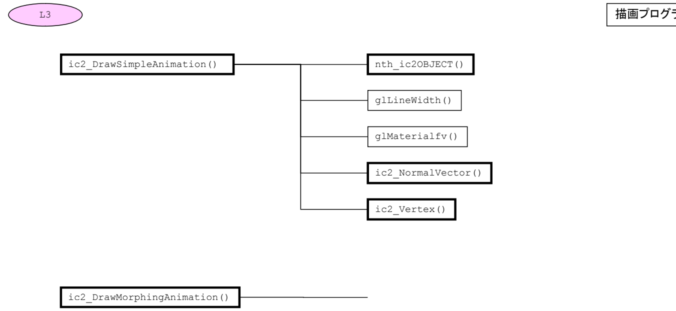
描画プログラム構成



描画プログラム構成



描画プログラム構成



アニメーション

コマ送りの実装

アニメーション

- 動作や形が少しずつ異なる多くの絵や人形を一齣(ひとこま)ずつ撮影し、映写した時に画像が連続して動いて見えるようにするもの。ビデオ-レコーダーによるものやコンピューターグラフィックスを応用するものもある。アニメ。動画。

三省堂提供「大辞林 第二版」

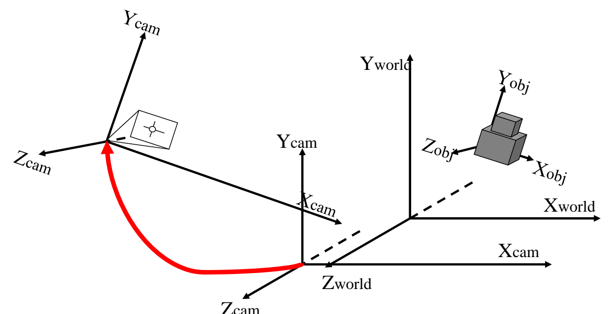
CGを用いたアニメーション作成

アニメーションの実現手段

- カメラの移動
  - Ic2_LookAt()
- CG物体の移動・回転
  - ic2_DrawSingleAnimation ()
- CG物体の変形（モーフィング【後述】）
  - ic2_DrawMorphingAnimation ()

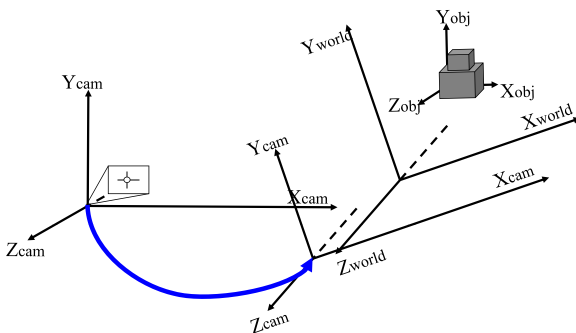
カメラの移動によるアニメーション

- 視点の移動による見え方の変化



カメラの移動によるアニメーション

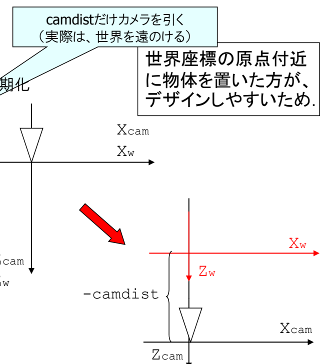
- 実際は、世界・物体座標系が移動している



カメラの位置・姿勢設定 ic2_LookAt(float *mvm)

```
void ic2_LookAt (float *r) {
    float t[16];     // 一時的な幾何変換行列
    float k[16];     // 蓄積行列

    ic2_LoadIdentity(k);  // k[]を単位行列で初期化

    // Z軸上での移動
    ic2_Translate(t, 0, 0, -camdist);
    ic2_MultMat(k, k, t);

    // 上下方向の回転
    ic2_RotateX(t, camtheta_v);
    ic2_MultMat(k, k, t);

    // 左右方向の回転
    ic2_RotateY(t, -(camtheta_h - 90.0));
    ic2_MultMat(k, k, t);

    // 最終的に得られた蓄積行列を返す
    ic2_CopyMatrix(r, k);
}
```

camdistだけカメラを引く
（実際は、世界を遠のける）

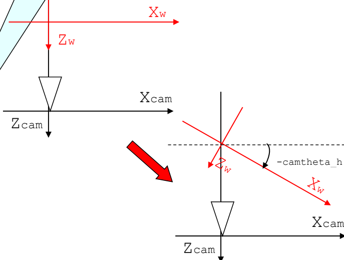


カメラの位置・姿勢設定 ic2_LookAt(float *mvm)

```
void ic2_LookAt (float *r) {
    float t[16];        // 一時的な幾何変換行列
    float k[16];        // 蓄積行列

    ic2_LoadIdentity(k);  // k[]を単位行列で初期化

    // Z軸上での移動
    ic2_Translate(t, 0, 0, -camdist);
    ic2_MultMatMat(k, k, t);

    // 上下方向の回転
    ic2_RotateX(t, camtheta_v);
    ic2_MultMatMat(k, k, t);

    // 左右方向の回転
    ic2_RotateY(t, -(camtheta_h - 90.0));
    ic2_MultMatMat(k, k, t);

    // 最終的に得られた蓄積行列を返す
    ic2_CopyMatrix(r, k);
}
```

Yw軸周りに-camtheta_h回転
（実際は、逆方向に世界が回転）

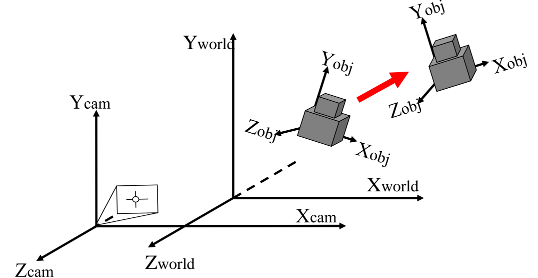
物体の周りを周回する
(r=camdist)カメラ移動



物体の移動によるアニメーション

ここで「移動」には平行移動・回転・スケーリングを含む

- 物体の移動による見え方の変化
  - カメラ～世界座標系の幾何関係は一定
  - 世界～物体座標系の幾何関係が変化
  - 課題プログラムのスクリプトで実行されるアニメーション



スクリプトによるアニメーションの実行

- 世界～物体座標系の移動変換
- 2通りの移動情報（平行・回転）の与え方
  1. コマ毎に、(tx,ty,tz)(rx,ry,rz)を関数に入力
  2. 全体の移動情報を最初に与え、それをコマ数で分割した移動情報（1コマの移動情報）を関数に入力
- 本演習では 2. を採用

スリプト中のアニメーション記述１

A: [animation]　一つ目のアニメーション

A ObjectID step interval tx ty tz rx ry rz sx sy sz

世界座標系に対する物体座標系の移動・回転・伸縮の操作を示す
- tx,ty,tz: 平行移動
- rx,ry,rz: 回転（軸回り） [degree]
- sx,sy,sz: 拡大縮小
- step : アニメーションにかける回数
- interval: アニメーション１回にかける時間[ミリ秒]

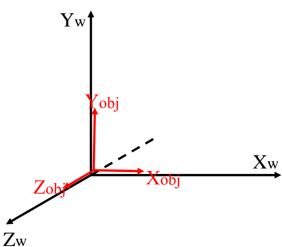
同時指定した場合の演算は、プログラム上の出現順序で 平行移動、回転（X,Y,Z軸周りの順）、拡大縮小の通り

（例）A 1 200 0 0.0 0.0 0.0 0 0 360 1.0 1.0 1.0
200コマかけて、360度回転

スリプト中のアニメーション記述１

A: [animation]　一つ目のアニメーション

A ObjectID step interval tx ty tz rx ry rz sx sy sz



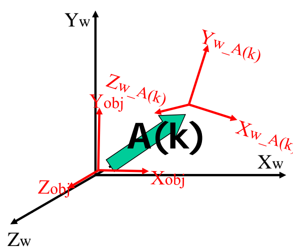
アニメーション開始時に、
蓄積変換行列C を単位行列に初期化する

$$C = I$$

スリプト中のアニメーション記述１

A: [animation]　一つ目のアニメーション

A ObjectID step interval tx ty tz rx ry rz sx sy sz



アニメーションの進行度k(0≦k≦1)の場合の座標変換は、

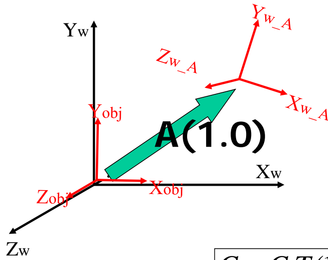
$$\begin{bmatrix} X_{w_A(k)} \\ Y_{w_A(k)} \\ Z_{w_A(k)} \\ 1 \end{bmatrix} = C\,T(k)Rx(k)Ry(k)Rz(k)S(k) \begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

蓄積変換行列　C(=I)
平行移動変換行列　T(k)
回転移動変換行列　Rx(k), Ry(k), Rz(k)
スケーリング　S(k)

スリプト中のアニメーション記述１

A: [animation]　一つ目のアニメーション

A ObjectID step interval tx ty tz rx ry rz sx sy sz



アニメーションの終了時の、平行移動変換
T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)
S(1.0)を蓄積変換行列C に蓄積.

$$\begin{bmatrix} X_{w_A} \\ Y_{w_A} \\ Z_{w_A} \\ 1 \end{bmatrix} = C\,T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)S(1.0)\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

$$C = C\,T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)S(1.0)$$

代入

スクリプト中のアニメーション記述2

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

世界座標系に対する物体座標系の移動・回転・伸縮の操作に加え物体の変形を表現する

(1) 移動・回転・伸縮 A:と同じ

(2) 変形 ObjectIDが前のAnimation（A行 or C行）と同じ場合は、アニメーション操作を表す

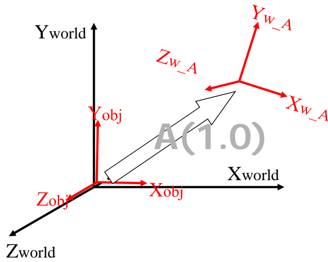
(※) 変形 ObjectIDが前のAnimation（A行 or C行）と異なる場合は、変形操作を表す→ モーフィングで再出

スクリプト中のアニメーション記述2

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

ObjectIDがAの行と同じ場合：アニメーションの継続



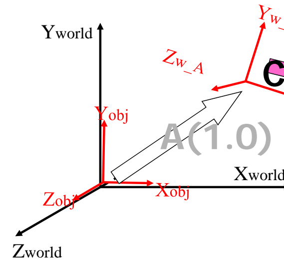
アニメーション開始時、現在の蓄積変換行列Cをそのまま利用（＝アニメーションの継続）

スリプト中のアニメーション記述2a

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

ObjectIDがAの行と同じ場合：アニメーションの継続



アニメーションの進行度
k(0.0≦k≦1.0)の場合の座標変換はA行の場合と同様.

一つ前のAを示す行列がここに格納されている

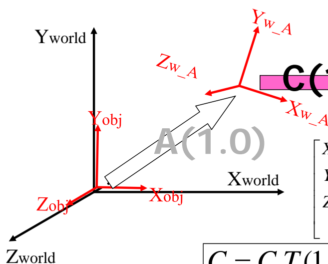
$$\begin{bmatrix} X_{w_C(k)} \\ Y_{w_C(k)} \\ Z_{w_C(k)} \\ 1 \end{bmatrix} = C\,T(k)Rx(k)Ry(k)Rz(k)S(k)\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

スリプト中のアニメーション記述2a

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

ObjectIDがAの行と同じ場合：アニメーションの継続



アニメーションの終了時の、移動変換T(1.0)Rx(1.0)
Ry(1.0)Rz(1.0)
S(1.0)を蓄積変換行列
Cに蓄積.

$$\begin{bmatrix} X_{w_C} \\ Y_{w_C} \\ Z_{w_C} \\ 1 \end{bmatrix} = C\,T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)S(1.0)\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

$$C = C\,T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)S(1.0)$$

さらに代入

アニメーションデータの構造体

１回分のアニメーションのための構造体

```
struct ic2ANIMATION {
  int type; // [A] new anime, [C] anime continued
  int id; // object ID
  int step; // intermediate snapshots to morph
  int interval; // interval between intermediate snapshots in milliseconds
  float tx; // translate along x axis
  float ty; // translate along y axis
  float tz; // translate along z axis
  float rx; // rotate around x axis
  float ry; // rotate around y axis
  float rz; // rotate around z axis
  float sx; // scale in x
  float sy; // scale in y
  float sz; // scale in z
  struct ic2ANIMATION *next;  //次のアニメーションへのポインタ
};
```

アニメーション処理

ic2_AnimationMatrix()関数
- 変換情報は構造体animnowに記録
  struct ic2ANIMATION *animnow
  連続アニメーションはanimnow->next で連結
- 連続するアニメーション操作では、(1つめのアニメーションからの)相対的な変換情報が必要
  float cumulatedmatrix[16] に蓄積
- 現在のアニメーションについては、今の表示回に応じてアニメーションの進行度合い(0.0~1.0)が決まる
  r = (float)snapshotnum / animnow->step;

書式
ic2_AnimationMatrix (mvm_work, animnow, r, mvm_work);

ic2_AnimationMatrix()

アニメーション中の1コマ(A(k)ないしC(k))の座標変換

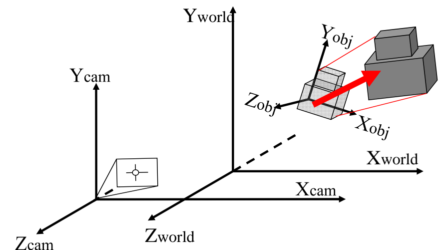
```
void ic2_AnimationMatrix
(
float *rmat,
// アニメーション終了のモデルビュー行列

struct ic2ANIME*animnow,
// 今まさに処理中のアニメーション構造体(平行･回転移動･スケールの
   情報が格納されている)

float r,
// アニメーションの進度(0.0-1.0) ※0.0はstep=0(アニメーションなし)
   となる特殊値

float *basematrix
// アニメーション開始時のMODELVIEW行列
) ;
```


物体の変形:モーフィング

今回は線形補間で単純に実装

物体の変形によるアニメーション

- 物体の変形による見え方の変化
  - カメラ～世界座標系の幾何関係は一定
  - 世界～物体座標系の幾何関係は一定(独立)
  - 課題プログラムのスクリプトで実行されるアニメーション



モーフィング

- ある形状から別の形状へ徐々に変化していく様子を動画で表現するために、その中間を補うための画像を作成すること。(中略)前後のコマの画像の要素から、中間の画像を作り出すことを何度も行なうと、滑らかに変化するモーフィングの動画(CG物体の変形によるアニメーション)が得られる。
  (e-words)

線形補間によるモーフィング

- 始点A(Xa,Ya,Za)と終点B(Xb,Yb,Zb)をr:1-rに内分(内挿)した点C(Xc,Yc,Zc)

$$\begin{pmatrix} Xc \\ Yc \\ Zc \end{pmatrix} = \begin{pmatrix} Xa \\ Ya \\ Za \end{pmatrix} \times r + \begin{pmatrix} Xb \\ Yb \\ Zb \end{pmatrix} \times (1-r)$$

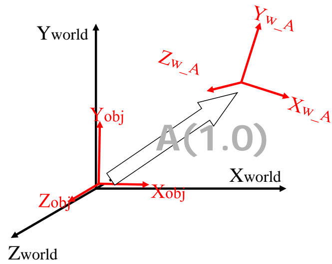
- 色情報の内挿も忘れずに

スリプト中のアニメーション記述2b

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

ObjectIDが前の行(例:A行)と異なる場合:モーフィングの実行



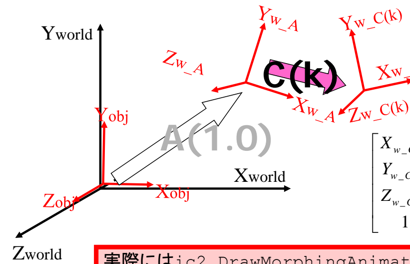
アニメーション開始時、現在の蓄積変換行列Cをそのまま利用
(=アニメーションの継続)

スリプト中のアニメーション記述2b

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

ObjectIDが前の行(例:A行)と異なる場合:モーフィングの実行



アニメーションの進行度k(0.0≦k≦1.0)の場合の座標変換は、A行の場合と同様.

$$\begin{bmatrix} X_{w_C(k)} \\ Y_{w_C(k)} \\ Z_{w_C(k)} \\ 1 \end{bmatrix} = C\,T(k)Rx(k)Ry(k)Rz(k)S(k)\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

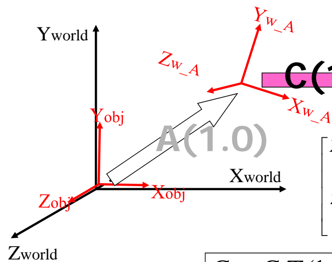
実際にはic2_DrawMorphingAnimation()内で、物体の色と形状(頂点の位置)の両方の内挿処理が行われる

スリプト中のアニメーション記述2b

C: [animation Continued]　二つ目以降のアニメーション

C ObjectID step interval tx ty tz rx ry rz sx sy sz

ObjectIDが前の行(例:A行)と異なる場合:モーフィングの実行



C(1.0)　Yw_C(k)　Xw_C(k)　Zw_C(k)

アニメーションの終了時の移動変換T(1.0) Rx(1.0) Ry(1.0) Rz(1.0) S(1.0)を蓄積変換行列Cに蓄積.

$$\begin{bmatrix} X_{w_C} \\ Y_{w_C} \\ Z_{w_C} \\ 1 \end{bmatrix} = C\,T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)S(1.0)\begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ 1 \end{bmatrix}$$

$$C = C\,T(1.0)Rx(1.0)Ry(1.0)Rz(1.0)S(1.0)$$

モーフィング処理用関数

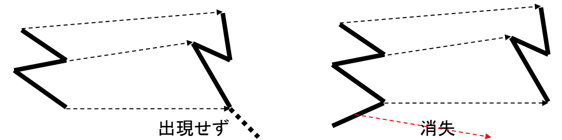
- float ic2_LinearInterpolation (float a, float b, float r)
  - aとbの内分値を戻り値とする関数
- void ic2_POINTLinearInpterpolation (struct ic2POINT *i, struct ic2POINT a, struct ic2POINT b, float r)
  - 3次元点aと点bの内分点iを計算する関数

モーフィング前後のCG要素数

- モーフィングの前後で物体を構成するCG要素(線や面)の数が異なる場合どうなる？
- 前後で対応が取れるだけ、モーフィングが行われる(数が少ない方の要素数まで)
  - もし新しく要素が出現するようなモーフィングをしたい場合は、前のCG物体中に冗長な要素を持たせる.

モーフィング前後のCG要素数

- 前後で対応が取れる分(数が少ない方の要素数)だけモーフィングが行われる



- 新しく要素が出現するようなモーフィングをしたい場合は、前のCG物体にこっそりと冗長な要素を持たせておく.