コンピュータグラフィックス特論 II

第2回　OpenGLプログラミングの基礎

九州工業大学　尾下 真樹

2021年度

今日の内容

- OpenGL＆GLUTの概要
- サンプルプログラムの概要
- 座標変換
- 変換行列の設定
- ポリゴンモデルの描画

今日の内容

- OpenGLプログラミングの基礎
  - C言語＋OpenGL＋GLUT によるプログラミング
- 座標変換の基礎
  - 同次座標変換（アフィン変換）にもとづく視野変換行列の設定

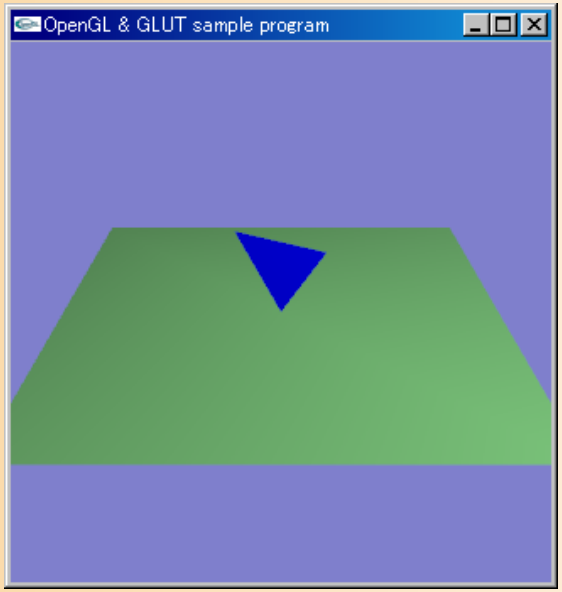
  - いずれも、学部の講義（レベルの内容）の復習

今日の内容

- OpenGL＆GLUTの概要
- サンプルプログラムの概要
- 座標変換
- 変換行列の設定
- ポリゴンモデルの描画

サンプルプログラム

- OpenGL＋GLUT のサンプルプログラム
  - 地面と1枚の青い三角形が表示される
  - OpenGL と GLUT の基本的な使い方を説明するためのプログラム



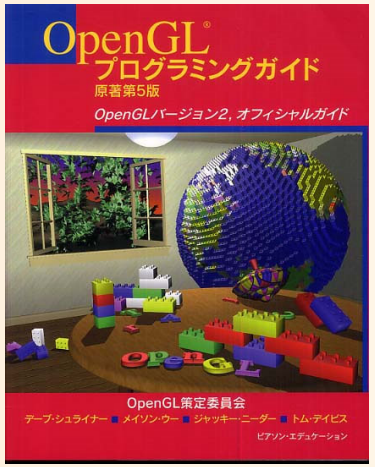
サンプルプログラム

- opengl_sample.c

opengl_sample.c

```
//
//  コンピュータグラフィックスＳ
//  OpenGLによる３次元グラフィックス演習 サンプルプログラム
//


// 基本的なヘッダファイルのインクルード
#ifdef _WIN32
    #include <windows.h>
#endif
#include <stdio.h>
#include <math.h>

// GLUTヘッダファイルのインクルード
#include <GL/glut.h>


// 視点操作のための変数
float  camera_pitch = -30.0;  // Ｘ軸を軸とするカメラの回転角度

// マウスのドラッグのための変数
int  drag_mouse_r = 0;  // 右ボタンをドラッグ中かどうかのフラグ (0:非ドラッグ中,1:ドラッグ中)
int  last_mouse_x;      // 最後に記録されたマウスカーソルのＸ座標
int  last_mouse_y;      // 最後に記録されたマウスカーソルのＹ座標


//
//  画面描画時に呼ばれるコールバック関数
//
void  display( void )
{
    // 画面をクリア (ピクセルデータとＺバッファの両方をクリア)
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // 変換行列を設定 (ワールド座標系→カメラ座標系)
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0,  0.0, -15.0 );
    glRotatef( -camera_pitch, 1.0, 0.0, 0.0 );

    // 光源位置を設定 (モデルビュー行列の変更にあわせて再設定)
    float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
    glLightfv( GL_LIGHT0, GL_POSITION, light0_position );

```

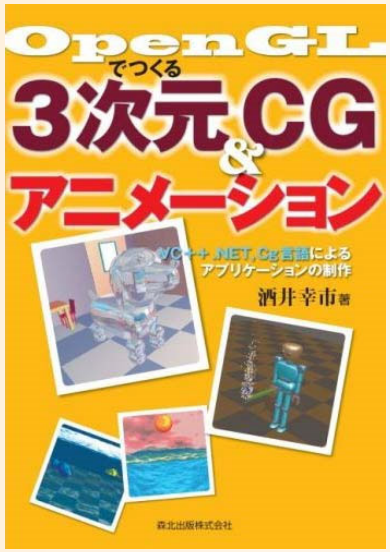
参考書



- 最低限の関数の使い方は資料を用意
- OpenGLの定番の本（高い）
  - OpenGLプログラミングガイド（赤本）, 12,000円
  - OpenGLリファレンスマニュアル（青本）, 8,300円
    - ピアソン・エデュケーション出版
- グラフィックスS（システム創成3年前期）演習資料
  - http://www.cg.ces.kyutech.ac.jp/lecture/cg/
  - OpenGLの使い方を段階的に学べるチュートリアル
  - OpenGLに不慣れな人は一通り試しておくことを推奨
- 適当な入門書
  - 他にもOpenGLの入門書は多数ある

参考書(続き)

- 他の参考書
  - 他にもOpenGLの入門書は多数ある
  - OpenGLでつくる 3次元CG & アニメーション (3600円)
    - 酒井 幸市 著
    - OpenGL･GLUTの使い方 ＋ 最新技術
    - 興味がある人は、買ってみると良い
  - OpenGL入門 (3,000円)
    - エドワード･エンジェル 著、滝沢 徹･牧野 祐子 訳
    - ピアソン･エデュケーション出版
    - OpenGL･GLUTの使い方



教科書・参考書

- 「コンピュータグラフィックス」
  CG-ARTS協会 編集・出版（3,600円）
- 「ビジュアル情報処理 ―CG・画像処理入門―」
  CG-ARTS協会 編集・出版（2,500円）
- 「3DCGアニメーション」
  栗原恒弥 安生健一 著、技術評論社
  出版（2,980円）

OpenGL & GLUT

演習環境

- 講義や資料で想定する標準環境
  - C言語 + OpenGL + GLUT
  - Windows + Visual Studio
    - 自力で対応できるなら、他の OS や開発環境を使用しても構わない

OpenGL & GLUT

- OpenGL
  - 現在、最も広く使われている3次元API
    - C言語を始め、いろんな言語から使える
  - ポリゴンの描画、Zバッファなどの3次元描画に必要な機能を提供
  - ウィンドウ生成やマウス・キーボード入力などの処理の機能は持たない
    - これらは、OSやウィンドウシステム固有の機能なので、各環境に応じたAPIを使って記述する必要がある
    - 実装が大変、環境ごとに実装する必要がある

GLUT

- OpenGL Utility Toolkit（GLUT）
  - ウィンドウ生成やイベント処理などの環境依存の部分を共通化したライブラリ
    - OpenGL標準ではないがかなり広く普及している
  - 内部に各OS用のコードを含んでいるため、一度プログラムを作ればいろんな環境で動く
  - 機能が限定されている代わりに非常にシンプル
  - とりあえずOpenGLを使いたい場合に適している

DirectXとの比較

- DirectX
  - Windowsのみでしか動かない
  - Windowsと密接に関連している
    - WindowsやCOMなどの仕組みを理解する必要がある
    - プログラミングが必要以上に面倒
  - 最新のハードウェアの機能を使えるという利点もある
  - 他のマルチメディア機能も持っている
    - DirectSound, DirectPlay, DirectInput
  - 基本な考え方はOpenGLと同じ

Java3Dとの比較

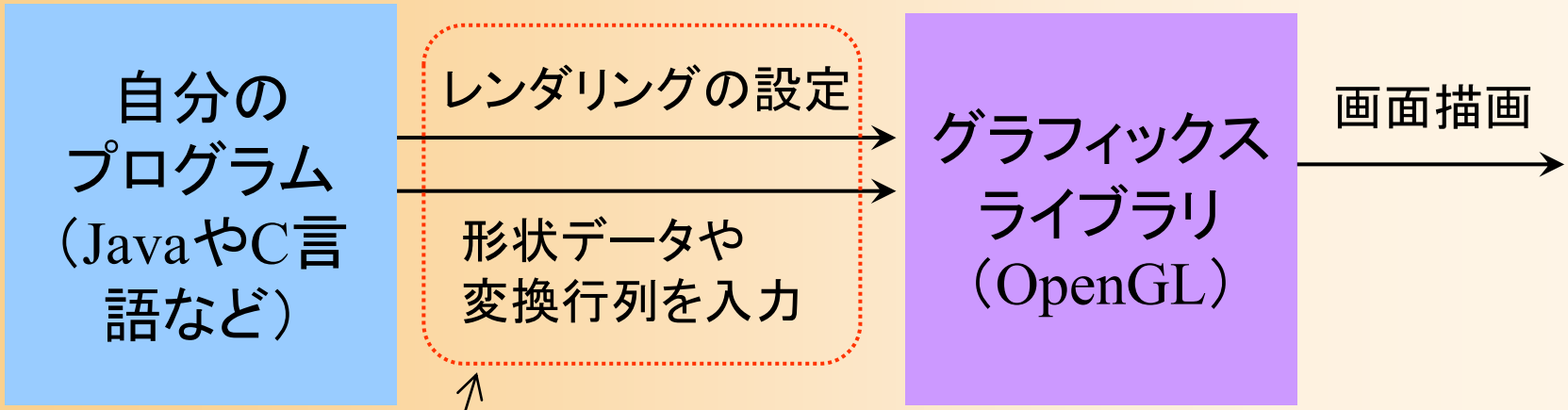
- Java3D
  - シーングラフ API（高レベルAPI）
    - カメラや物体などのシーンの階層構造を設定してやると、細かい描画は自動的に行ってくれる
    - 高機能で便利、CGの原理をよく知らなくても使える
  - デメリット
    - 独自のライブラリなので、Java3Dの使い方だけ覚えても使い回しが利かない
    - クラスライブラリになっているので、本当にきちんと理解しようとすると全体像を把握する必要があり、大変

他のライブラリとの比較

- 携帯端末（iOS/Androidなど）用アプリ
  - OpenGL ES が採用されている
    - OpenGL のサブセット（機能限定版）
    - 描画処理の基本は OpenGL と基本的に同様
- ゲームエンジン
  - Unity, Unreal など
  - Java 3D 同様、シーン情報やアニメーションを設定すれば、細かい描画処理は自動的に行ってくれる

OpenGLの利用

- 自分のプログラム と OpenGL の関係



最低限、これらの方法だけ学べば、
プログラムを作れる

これらの処理は、自分でプログラ
ムを作る必要はないが、しくみは
理解しておく必要がある

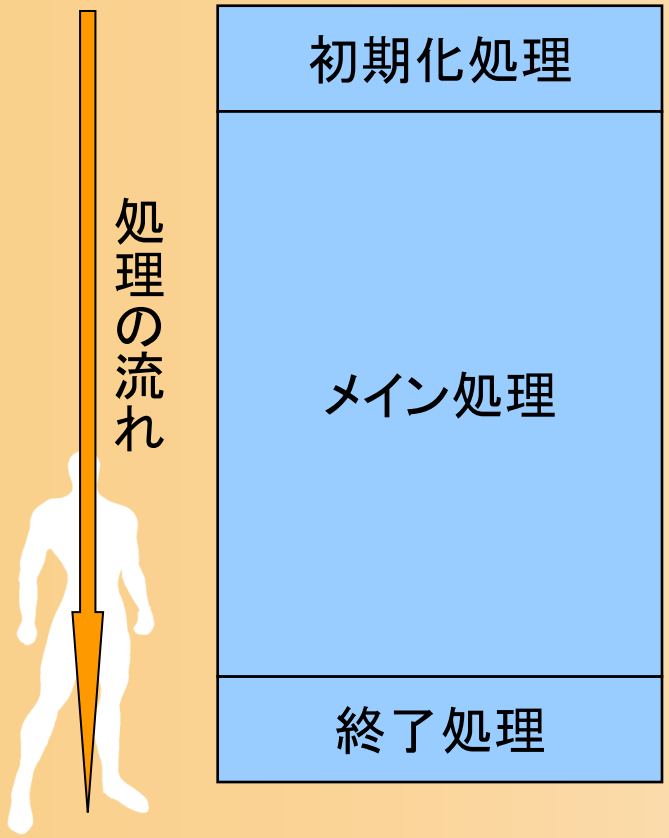
レンダリング（＋座標変換、
シェーディング、マッピング）
などの処理を行ってくれる

GLUTのイベントモデル

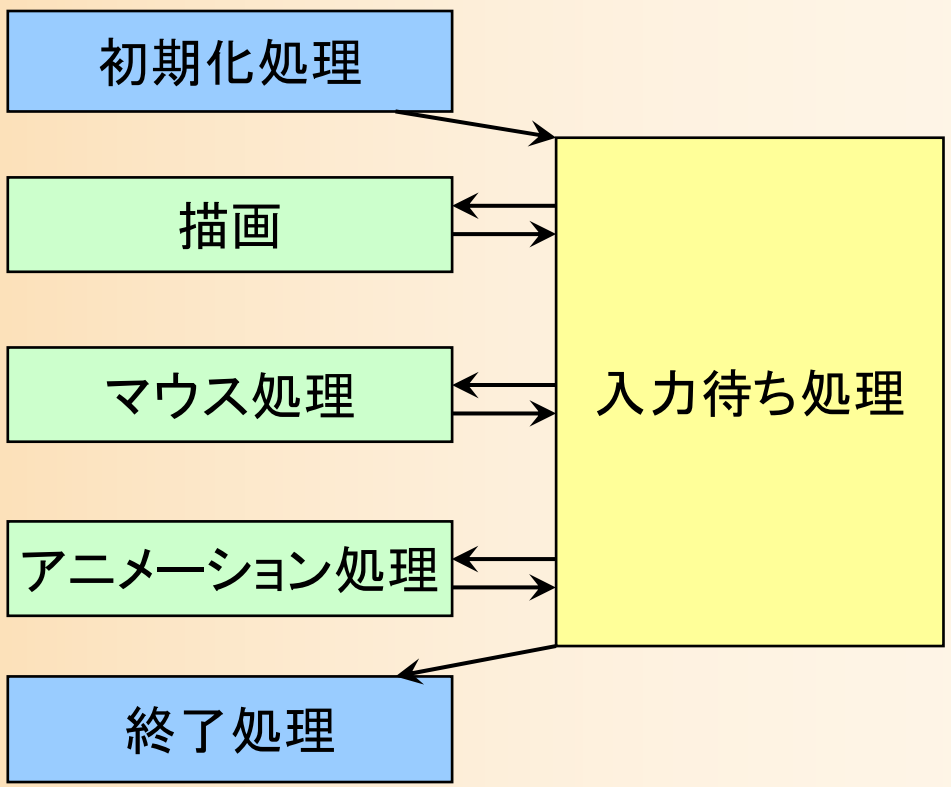
- ウィンドウシステムでのプログラミング
  - Windows や X Window などの一般的なウィンドウシステム
  - ウィンドウ管理やマウス操作などはシステムがまとめて処理するため、ユーザプログラムは扱わない
  - ユーザプログラムは初期化処理を行った後は処理をウィンドウシステムに移す
  - ウィンドウシステムは、画面の再描画やマウスの操作などのイベントが起こるたびにユーザプログラムに処理を一時的に戻す

イベントドリブン型プログラム

コンソール・プログラム



ウィンドウ・プログラム（イベントドリブン）

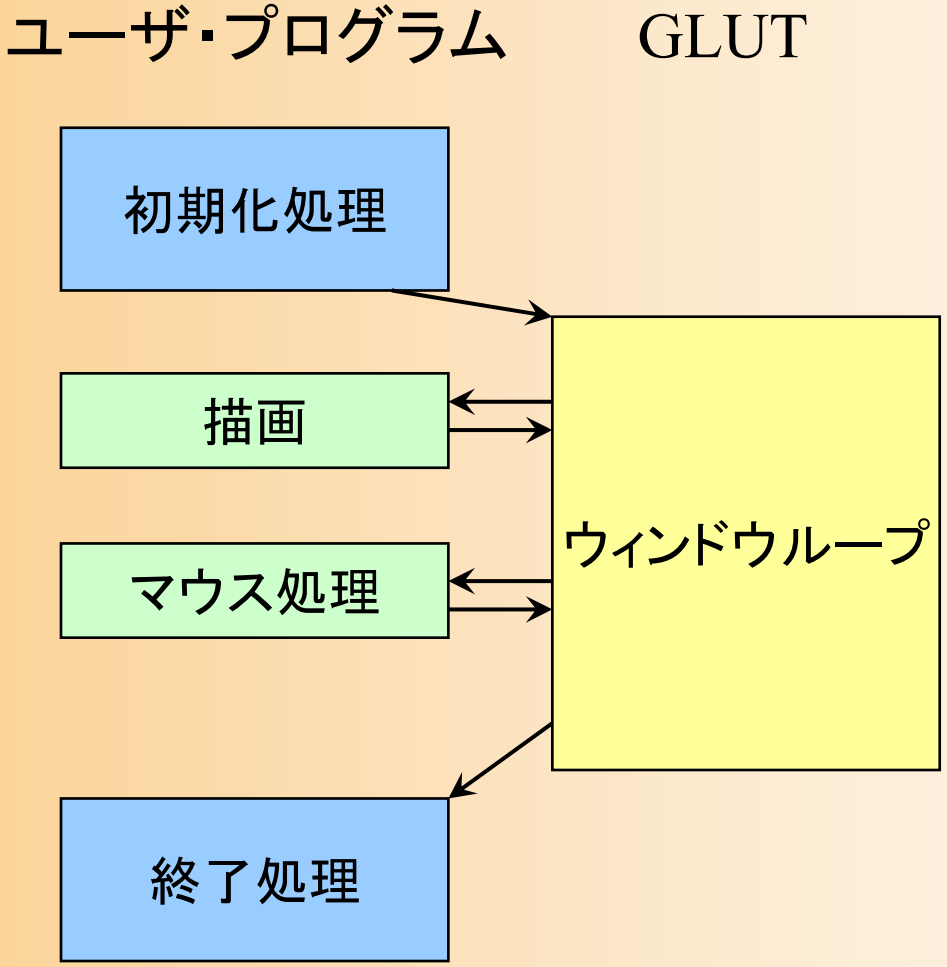


GLUTのイベントモデル

- イベントループとコールバック
  - イベントが起こった時にそのイベントを処理する関数をあらかじめ登録しておく
  - プログラムは初期化が終わったら、GLUTに処理を移す
  - マウス操作などのイベントが起こったらあらかじめ登録した関数が呼ばれる(コールバック)

- Javaの AWT や Swing などでは、同様の機能をリスナクラスを使って実現している

GLUTのイベントモデル



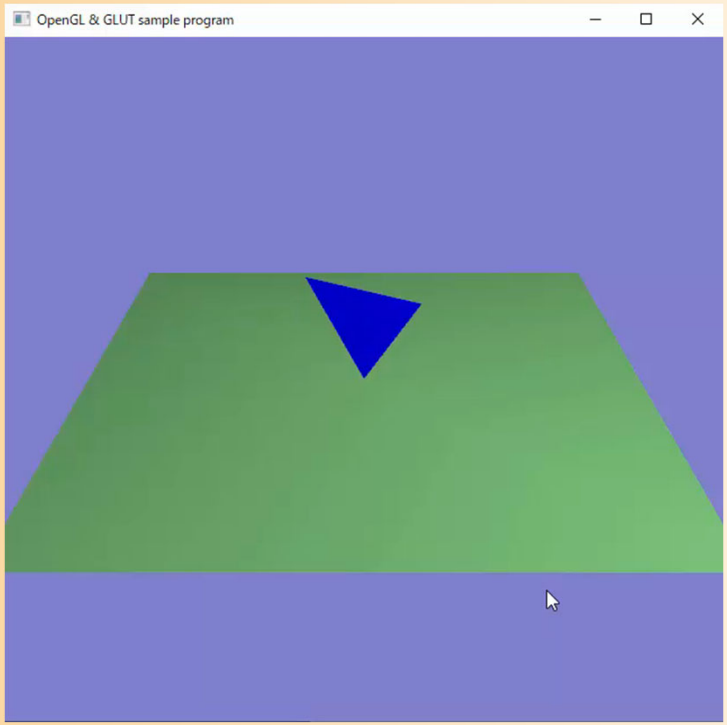
GLUTのコールバック関数の種類

- 描画コールバック関数
  - 描画が必要な時に呼ばれる
- サイズ変更コールバック関数
  - ウィンドウサイズ変更時に呼ばれる
- マウスクリック・コールバック関数
  - マウスのボタンが押されたとき、離されたときに呼ばれる
- マウスドラッグ・コールバック関数
  - マウスがウィンドウ上でドラッグされたときに呼ばれる
- キーボード・コールバック関数
  - キーボードのキーが押されたときに呼ばれる
- アイドル・コールバック関数
  - 処理が空いた時に定期的に呼ばれる

サンプルプログラムの解説

サンプルプログラム

- opengl_sample.c
  - 地面と1枚の青い三角形が表示される
  - マウスの右ボタンドラッグで、視点を上下に回転



サンプルプログラム

- opengl_sample.c

opengl_sample.c

```
//
//  コンピュータグラフィックスＳ
//  OpenGLによる３次元グラフィックス演習 サンプルプログラム
//


// 基本的なヘッダファイルのインクルード
#ifdef _WIN32
    #include <windows.h>
#endif
#include <stdio.h>
#include <math.h>

// GLUTヘッダファイルのインクルード
#include <GL/glut.h>


// 視点操作のための変数
float  camera_pitch = -30.0;  // Ｘ軸を軸とするカメラの回転角度

// マウスのドラッグのための変数
int  drag_mouse_r = 0;  // 右ボタンをドラッグ中かどうかのフラグ (0:非ドラッグ中,1:ドラッグ中)
int  last_mouse_x;      // 最後に記録されたマウスカーソルのＸ座標
int  last_mouse_y;      // 最後に記録されたマウスカーソルのＹ座標


//
//  画面描画時に呼ばれるコールバック関数
//
void  display( void )
{
    // 画面をクリア (ピクセルデータとＺバッファの両方をクリア)
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // 変換行列を設定 (ワールド座標系→カメラ座標系)
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0,  0.0, -15.0 );
    glRotatef( -camera_pitch, 1.0, 0.0, 0.0 );

    // 光源位置を設定 (モデルビュー行列の変更にあわせて再設定)
    float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
    glLightfv( GL_LIGHT0, GL_POSITION, light0_position );

```

サンプルプログラムの解説

- ここでは、プログラム全体を眺めて、大まかに、各部分でどのような処理を行っているかを確認する
- 各自、実際にコンパイルをしてみて、動作を確認する

OpenGLの関数

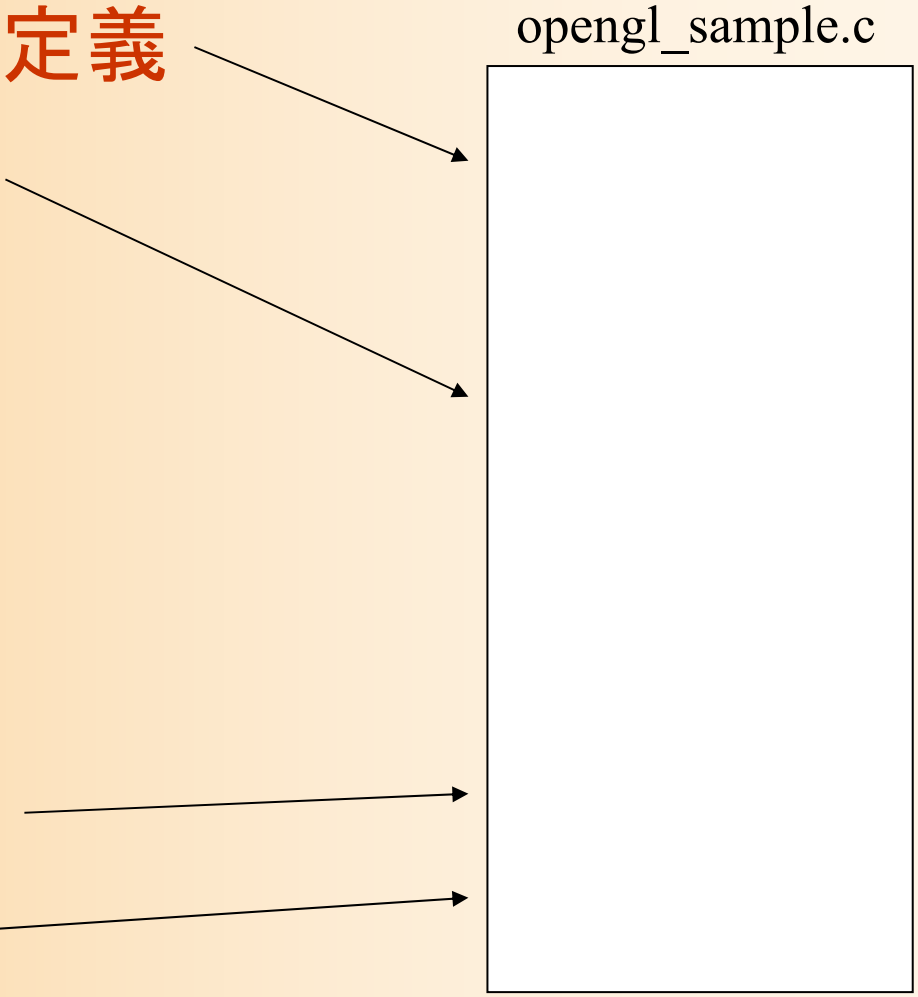
- gl～ で始まる関数
  - OpenGLの標準関数
- glu～ で始まる関数
  - OpenGL Utility Library の関数
  - OpenGLの関数を内部で呼んだり、引数を変換したりすることで、使いやすくした補助関数
- glut～ で始まる関数
  - GLUT（OpenGL Utility Toolkit）の関数
  - 正式にはOpenGL標準ではない

OpenGLの関数名

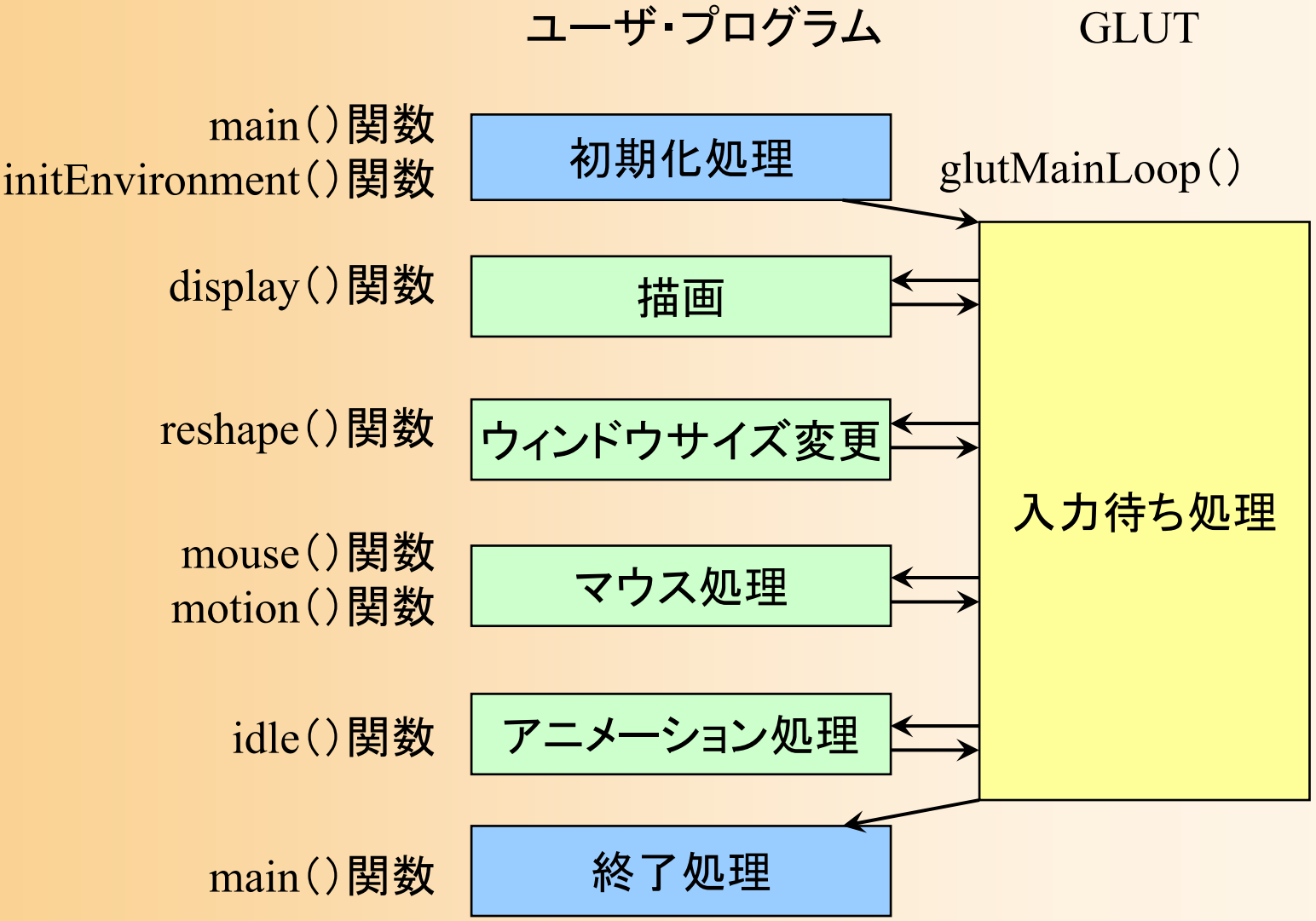
- 同じ機能で、微妙に違う名前の関数がある
  - 例：glVertex3f(x, y, z), glVertex3d(x, y, z)
    - f は引数が float 型であることを表す
    - d は引数が double 型であることを表す
      - C言語なので、関数のオーバーロード（同じ名前で引数が異なる関数）はサポートしていない
    - 必要に応じて使い分ける

サンプルプログラムの構成

- グローバル変数の定義
- コールバック関数
  - display()
  - reshape()
  - mouse()
  - motion()
  - idle()
- initEnvironment()
- main()



サンプルプログラムの構成

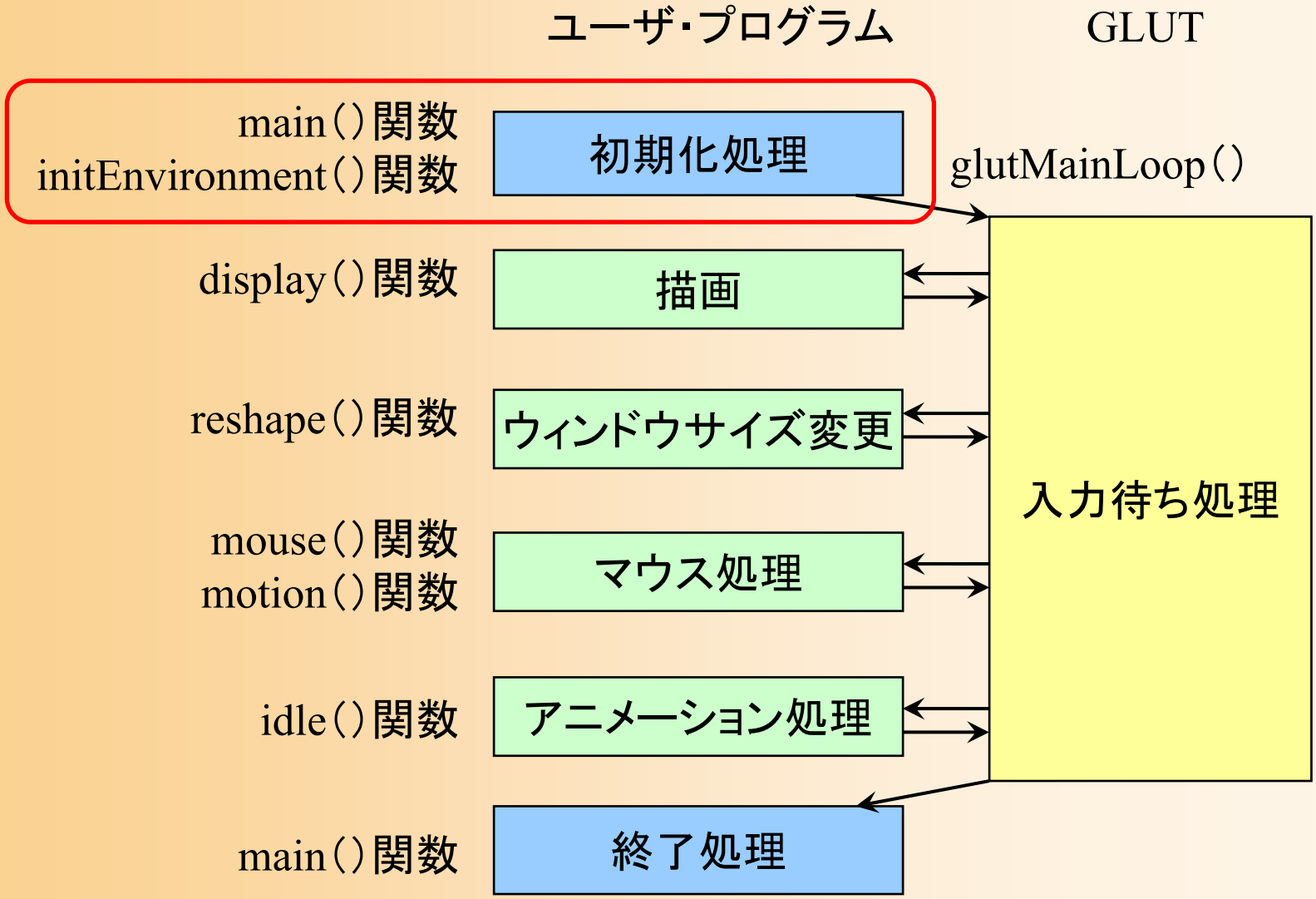


グローバル変数の定義

- グローバル変数
  - 全ての関数からアクセスできる変数
  - ここでは視点操作に関する変数を定義
    - 詳細は後で説明

```
// 視点操作のための変数
float  camera_pitch = -30.0;  // X軸を軸とするカメラの回転角度

// マウスのドラッグのための変数
int  drag_mouse_r = 0;  // 右ボタンをドラッグ中かどうかのフラグ
                                    (0:非ドラッグ中,1:ドラッグ中)
int  last_mouse_x;      // 最後に記録されたマウスカーソルのX座標
int  last_mouse_y;      // 最後に記録されたマウスカーソルのY座標
```

サンプルプログラムの構成



開始・初期化処理

- main関数
  - GLUTの初期化（メイン関数）
  - コールバック関数の設定
  - initEnvironment関数の呼び出し
  - GLUTのメインループの開始
- initEnvironment関数
  - レンダリングの設定
  - 光源の設定

1. GLUTの初期化

```
int  main( int argc, char ** argv )
{
        // GLUTの初期化
        glutInit( &argc, argv );
        glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
        glutInitWindowSize( 320, 320 );
        glutInitWindowPosition( 0, 0 );
        glutCreateWindow(“OpenGL & GLUT sample program");

        ......
}
```

2. コールバック関数の設定

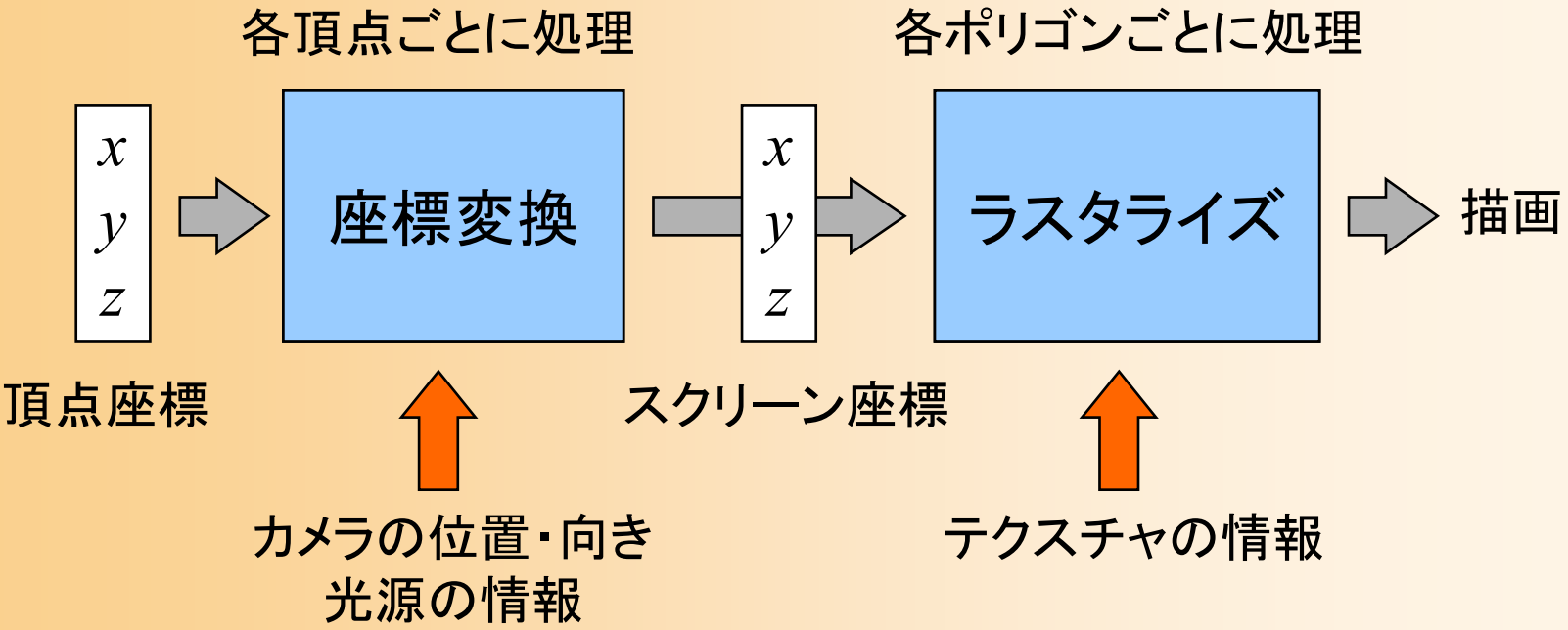
```
int  main( int argc, char ** argv )
{
        ......
        // コールバック関数の登録
        glutDisplayFunc( display );
        glutReshapeFunc( reshape );
        glutMouseFunc( mouse );
        glutMotionFunc( motion );
        glutIdleFunc( idle );

        // 環境初期化
        initEnvironment();

        // GLUTのメインループに処理を移す
        glutMainLoop();
        return 0;
}
```

3. レンダリングの設定

- Zバッファ法によるレンダリングの各種設定
  - 標準的な描画機能を設定（詳しい内容は後日説明）

```
void  initEnvironment( void )
{
        ......
        // 光源計算を有効にする
        glEnable( GL_LIGHTING );

        // 物体の色情報を有効にする
        glEnable( GL_COLOR_MATERIAL );

        // Zテストを有効にする
        glEnable( GL_DEPTH_TEST );

        // 背面除去を有効にする
        glCullFace( GL_BACK );
        glEnable( GL_CULL_FACE );

        // 背景色を設定
        glClearColor( 0.5, 0.5, 0.8, 0.0 );
}
```

レンダリング・パイプラインの設定（復習）



- 描画の前に、さまざまな設定を行うことができる
- 各機能を使うかどうか（Zバッファ、背面除去等）
- カメラの位置・向き（変換行列）の設定
- 光源の情報（位置・向き・色など）を設定

描画機能の設定

- さまざまな描画機能のオン･オフを設定
  - 不必要な処理はオフにすることで、高速できる
  - 初期状態ではオフになっている機能が多いので、必要な機能はオンに設定する必要がある
- glEnable（機能の種類）, glDisable（･･･）
  - 各機能のオン･オフを変更する
    - GL_LIGHTING, GL_COLOR_MATERIAL, GL_DEPTH_TEST, CL_CULL_FACE, etc
  - 各機能の動作はそれぞれ別の関数で設定

サンプルプログラムの描画機能の設定

- 標準的な描画の設定（最初に一度だけ設定）

```
void  initEnvironment( void )
{
        ......
        // 光源計算を有効にする
        glEnable( GL_LIGHTING );

        // 物体の色情報を有効にする
        glEnable( GL_COLOR_MATERIAL );

        // Zテストを有効にする
        glEnable( GL_DEPTH_TEST );

        // 背面除去を有効にする
        glCullFace( GL_BACK );
        glEnable( GL_CULL_FACE );

        // 背景色を設定
        glClearColor( 0.5, 0.5, 0.8, 0.0 );
}
```

描画機能の設定（その他）

- 背面除去の設定
  - glCullFace(GL_BACK)
  - 表面・背面のどちらを描画しないかを設定
- 背景色の設定
  - glClearColor(r, g, b, a)
  - 画面をクリアしたときの色を設定

4. 光源の設定

- シェーディングのための光源情報の設定
  - 1つの点光源を設定（詳しい内容は後日説明）

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );

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

OpenGLの光源処理の概要

- 光源と物体の素材（頂点の色）・法線によって、描画される頂点（ポリゴン）の色が決まる
- OpenGLの光源処理
  - OpenGLの関数を使って、光源や物体の素材・法線の情報を指定
  - OpenGLは、各頂点ごとに、自動的に光源処理を行い、各頂点の色を決定
    グローシェーディングにより、各頂点の色をもとに、ポリゴンが描画される

光のモデル（復習）

- 輝度の計算式
  - 全ての光による影響を足し合わせることで、物体上の点の輝度が求まる

$$I = I_a k_a + \sum_{i=1}^{n_L} I_i \left[k_d \left(N \cdot L \right) + k_s \left(R \cdot V \right)^n \right] + k_r I_r + k_t I_t$$

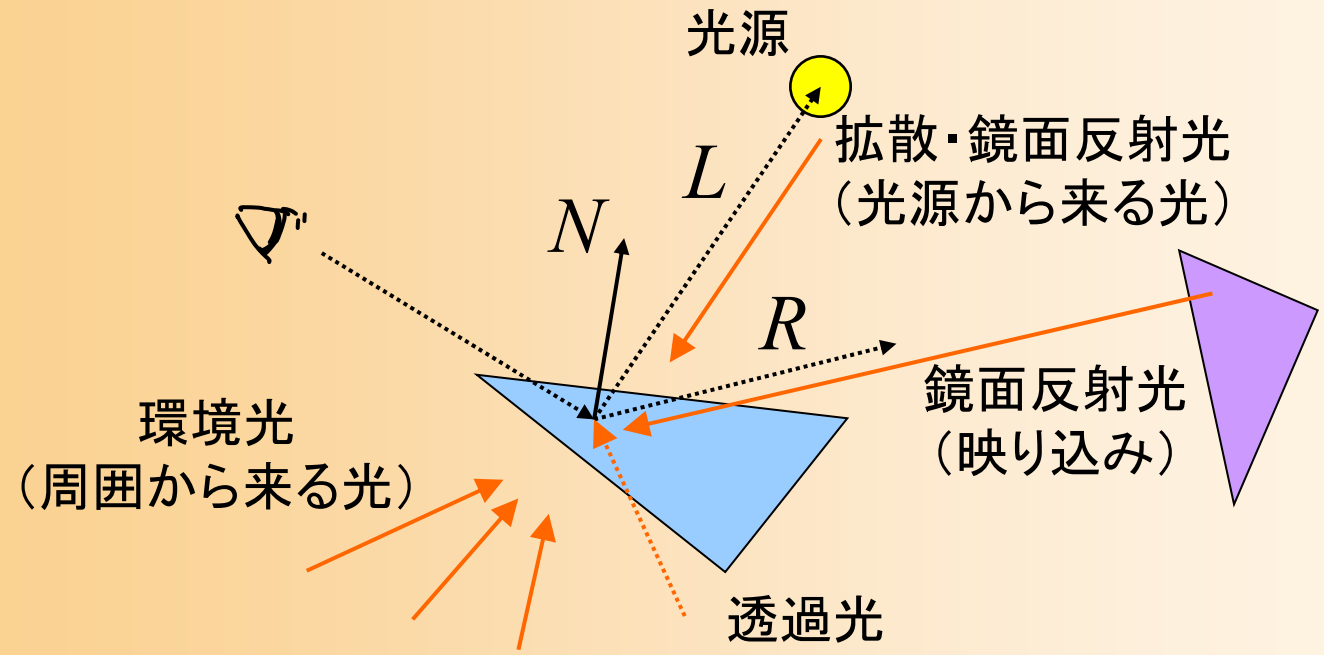
環境光　拡散反射光　鏡面反射光（局所照明）　鏡面反射光（大域照明）　透過光

それぞれの光源からの光（局所照明）　大域照明

$$k_a + n_L \left(k_d + k_s \right) + k_r + k_t = 1$$

各係数の和は1

光のモデル（復習）



$$I = I_a k_a + \sum_{i=1}^{n_L} I_i \left[k_d \left(N \cdot L \right) + k_s \left(R \cdot V \right)^n \right] + k_r I_r + k_t I_t$$

環境光　拡散反射光　鏡面反射光（局所照明）　鏡面反射光（大域照明）　透過光

それぞれの光源からの光（局所照明）　大域照明

OpenGLの光源処理

- 光のモデルにもとづき、各光源による輝度を、RGBごとに次式で計算して加算

$$Color = L_{\text{ambient}} \cdot M_{\text{ambient}} + \max\{\ \mathbf{l}\cdot\mathbf{n},\ 0\ \}\ L_{\text{diffuse}} \cdot M_{\text{diffuse}}$$
$$+ \max\{\ \mathbf{s}\cdot\mathbf{n},\ 0\ \}^{M_{\text{specular\_factor}}}\ L_{\text{specular}} \cdot M_{\text{specular}}$$

  - max{A, B}は、A, B のうち大きい値を使用
    内積が負の場合は、その項は0になる
  - 全ての値を足し合わせた結果は、0.0～1.0の範囲に丸められる

- $L_{\text{ambient}}, L_{\text{diffuse}}, L_{\text{specular}}$ は光の輝度
- $M_{\text{ambient}}, M_{\text{diffuse}}, M_{\text{specular}}, M_{\text{specular\_factor}}$ は素材の特性

光源情報の設定

- 光源情報の設定
  - glLight(), glLightv() 関数 を使用
    - 光源番号、設定パラメタの種類、設定する値、を指定
    - glLight() 関数はスカラ値を設定
    - glLightv() 関数はベクトル値を設定
- 光源処理を有効にする
  - 光源処理を有効にする glEnable(GL_LIGHTING)
  - 各光源の影響を有効にする glEnable(GL_LIGHT0)

光源情報の設定の例(1)

- 初期化処理での設定

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );

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

詳細は、後ほど説明

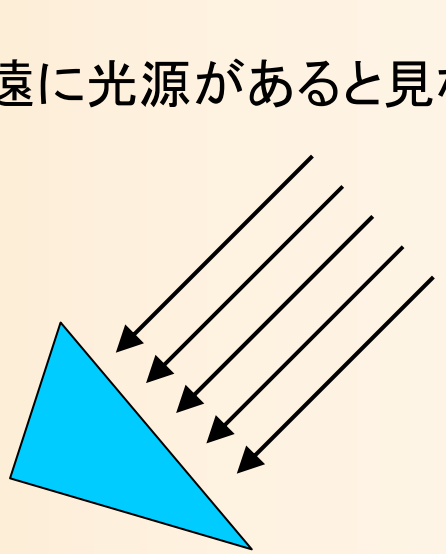
光源情報の設定の例(2)

- 変換行列の変更後に、光源位置を再設定
  - 光源計算は、カメラ座標系で適用されるため

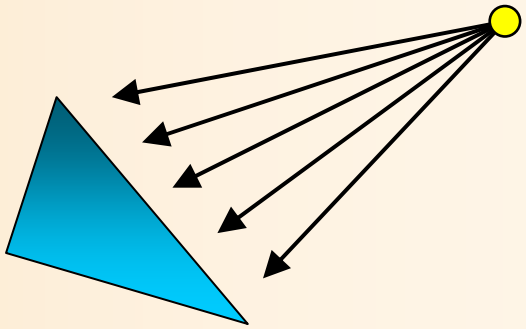
```
void  display( void )
{
        ・・・・・・
        // 変換行列を設定(ワールド座標系→カメラ座標系)
        glMatrixMode( GL_MODELVIEW );
        ・・・・・・

        // 光源位置を設定(変換行列の変更にあわせて再設定)
        float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
        glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
        ・・・・・・
```

光源の種類と設定方法(1)

- 平行光源
  - (x,y,z)の方向から平行に光が来る
  - 光源位置のw座標を0.0に設定



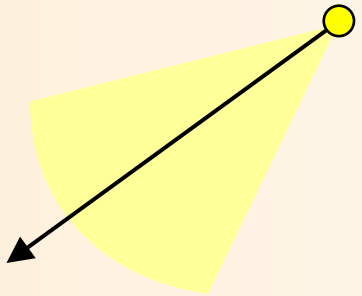
- 点光源
  - (x,y,z)の位置に光源がある
  - 光源位置のw座標を1.0に設定

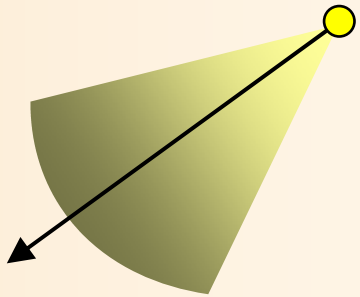


光源の種類と設定方法(2)

- スポットライト光源
  - 点光源にさらに、スポットライトの向き・角度範囲などの情報を設定したもの
- 光源の減衰も設定可能
  - 点光源・スポットライト光源から距離が離れるほど暗くなるような効果を加える
- 設定方法の説明は省略

指定した方向・角度にのみ有効な点光源





光源情報の設定の例

- サンプルプログラムの例

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );

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

光源位置のw座標が1.0なので、点光源となる

LIGHT0の
・光源の位置・種類
・拡散反射成分の色
・鏡面反射成分の色
を設定

LIGHT0の
・環境光成分の色
を設定

一般的な光源の設定方針

- LIGHT0を使って環境の主な光源を設定
  - その環境の明るさに応じて環境光を設定
  - 全体の明るさを決めるような、平行光源or点光源を設定
- LIGHT1以降を使って追加の光を設定
  - 電灯や車など、空間中にあるオブジェクトが周囲のオブジェクトを照らすような場合に、点光源やスポットライトを追加する
  - ２番目以降の光源では、環境光はあまり大きくしないことが多い

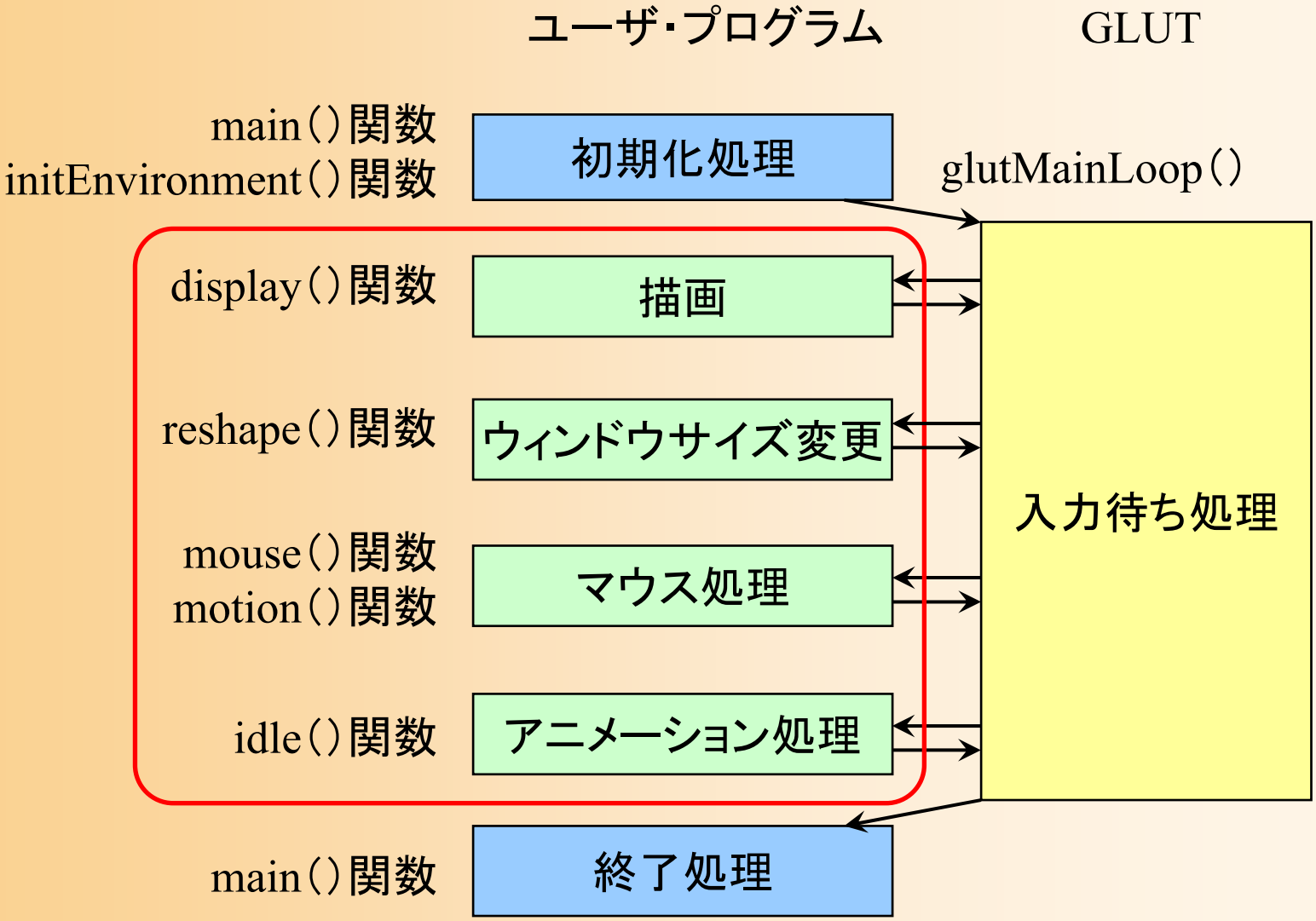
素材の設定

- 頂点の色の設定
  - glColor()関数
    - デフォルトでは、頂点の環境特性と拡散反射特性を同時に設定（個別に設定することも可能）
- その他の素材特性を個別に設定（詳細は省略）
  - glMaterial()関数
  - 環境特性、拡散反射特性、鏡面反射特性、鏡面反射係数など

$$Color = L_{\text{ambient}} \cdot M_{\text{ambient}} + \max\{\ \mathbf{l}\cdot\mathbf{n},\ 0\ \}\ L_{\text{diffuse}} \cdot M_{\text{diffuse}}$$

$$+\max\{\ \mathbf{s}\cdot\mathbf{n},\ 0\ \}^{M_{\text{specular\_factor}}}\ L_{\text{specular}} \cdot M_{\text{specular}}$$

サンプルプログラムの構成



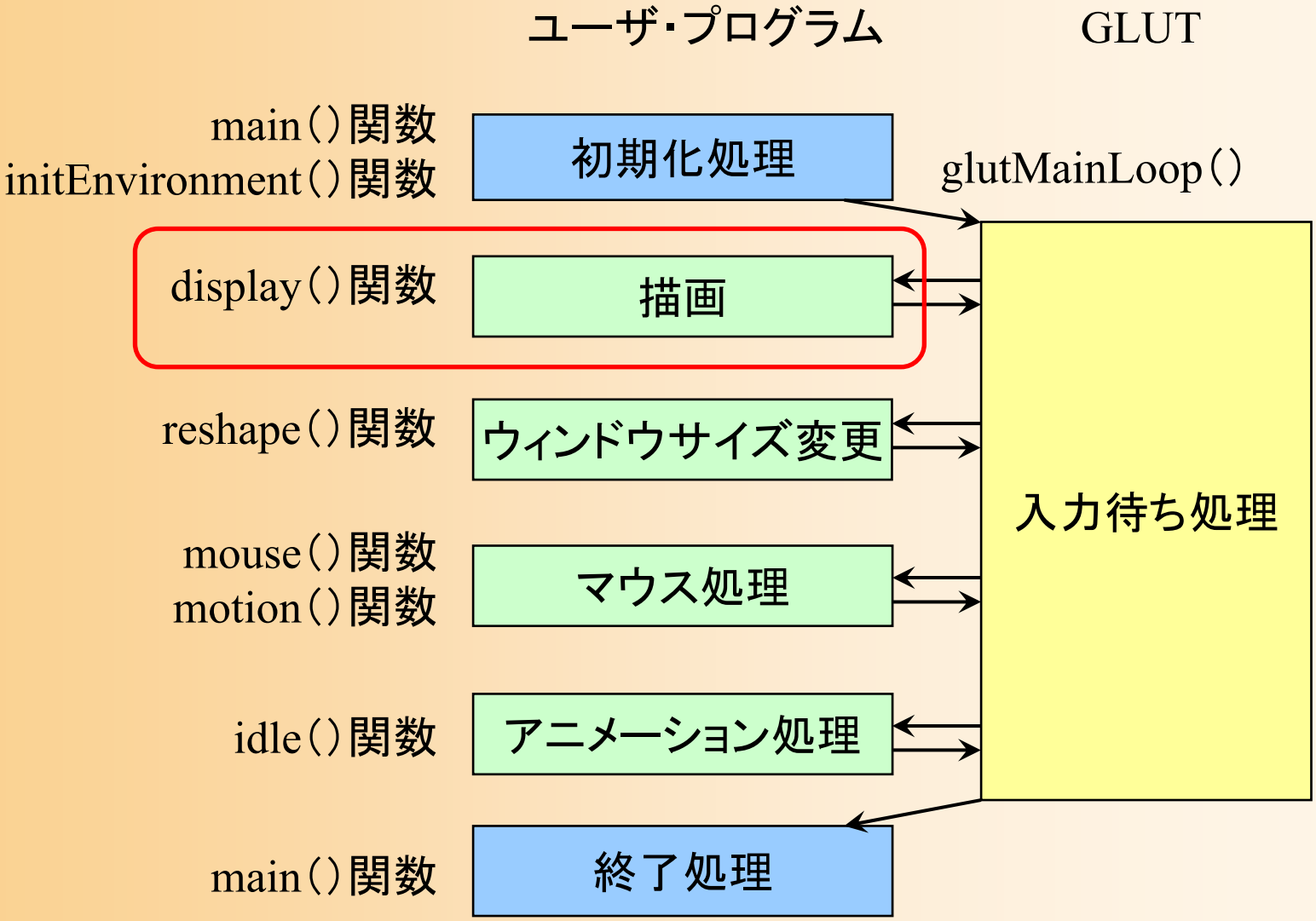
コールバック関数(1)

- 描画コールバック関数 display()
  - 再描画が必要な時に呼ばれる
  - 本プログラムでは、変換行列の設定、地面と1枚のポリゴンの描画、を行っている
- サイズ変更コールバック関数 reshape()
  - ウィンドウサイズ変更時に呼ばれる
  - 本プログラムでは、視界の設定、ビューポート変換の設定、を行っている

コールバック関数(2)

- マウスクリック・コールバック関数 mouse()
  - マウスのボタンが押されたとき、離されたときに呼ばれる
  - 本プログラムでは、右ボタンの押下状態を記録
- マウスドラッグ・コールバック関数 motion()
  - マウスがウィンドウ上でドラッグされたときに呼ばれる
  - 本プログラムでは、右ドラッグされたときに、視点の回転角度を変更
- アイドル・コールバック関数 idle()
  - 処理が空いた時に定期的に呼ばれる
  - 本プログラムでは、現在は何の処理も行っていない

サンプルプログラムの構成



描画関数の流れ

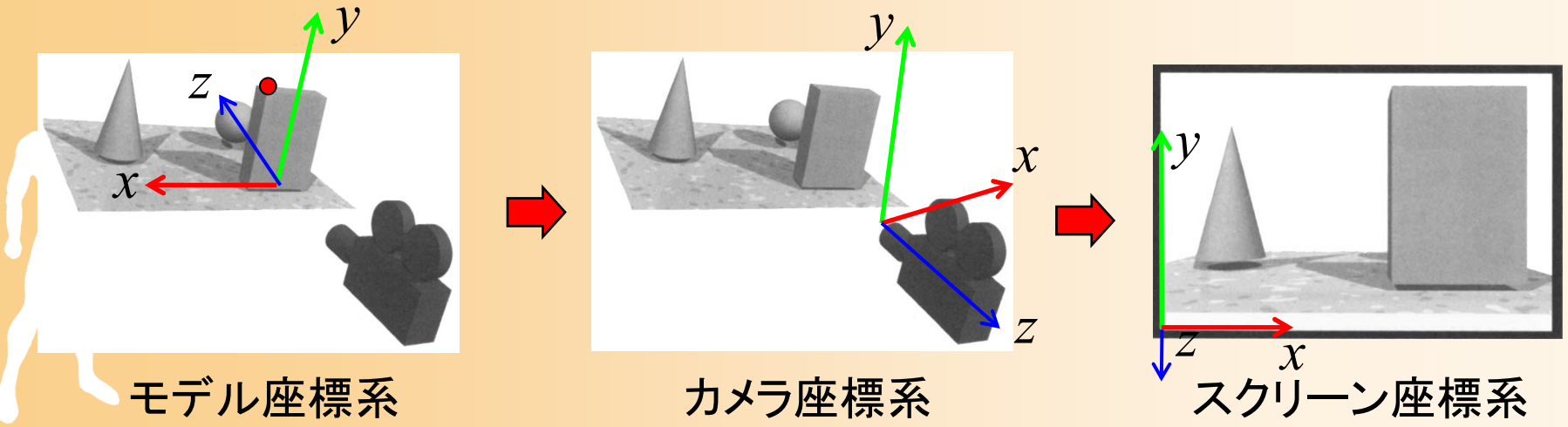
```
//
//  ウィンドウ再描画時に呼ばれるコールバック関数
//
void  display( void )
{
        // 画面をクリア（ピクセルデータとZバッファの両方をクリア）
        // 変換行列を設定（ワールド座標系→カメラ座標系）
        // 光源位置を設定（モデルビュー行列の変更にあわせて再設定）
        // 地面を描画
        // 変換行列を設定（物体のモデル座標系→カメラ座標系）
        // 物体（１枚のポリゴン）を描画
        // バックバッファに描画した画面をフロントバッファに表示
}
```

描画関数(1/4)

```
void  display( void )
{
        // 画面をクリア(ピクセルデータとZバッファの両方をクリア)
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        // 変換行列を設定(ワールド座標系→カメラ座標系)
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity();
        glTranslatef( 0.0,  0.0, - 15.0 );
        glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

        // 光源位置を設定(モデルビュー行列の変更にあわせて再設定)
        float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
        glLightfv( GL_LIGHT0, GL_POSITION, light0_position );

        ......
}
```

座標変換(復習)

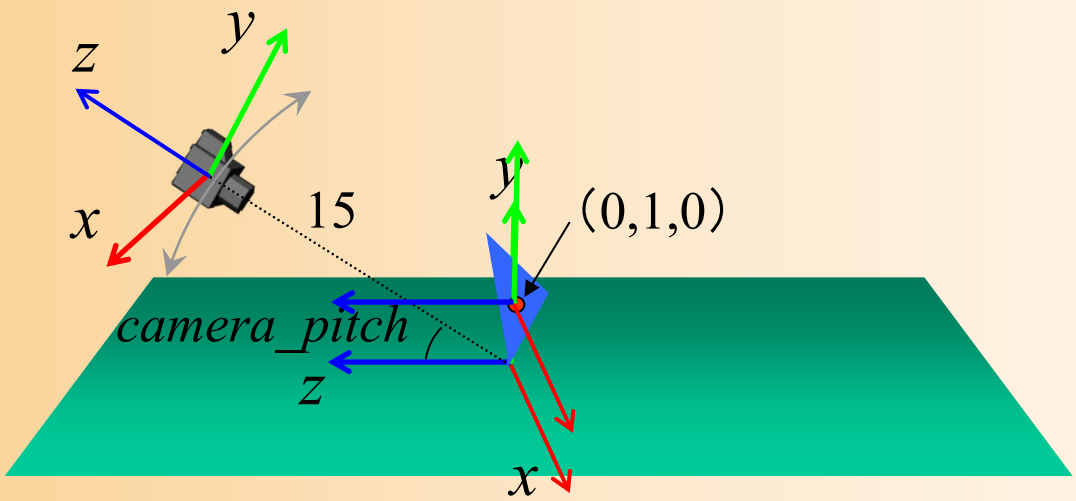
- 座標変換(Transformation)
  - 行列演算を用いて、ある座標系から、別の座標系に、頂点座標やベクトルを変換する技術
    - カメラから見た画面を描画するためには、モデルの頂点座標をカメラ座標系(最終的にはスクリーン座標系)に変換する必要がある



モデル座標系　　カメラ座標系　　スクリーン座標系

変換行列の設定

- サンプルプログラムでのカメラ位置の設定



- 以下の変換行列により表せる

ポリゴンを基準とする座標系での頂点座標

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

カメラから見た頂点座標（描画に使う頂点座標）

変換行列の設定

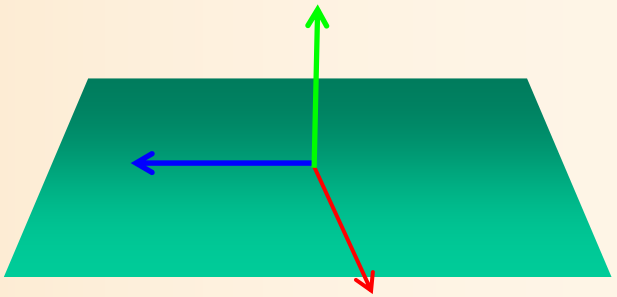
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

```
// 変換行列を設定（ワールド座標系→カメラ座標系）
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( 0.0,  0.0, - 15.0 );
glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

// 地面を描画
・・・・・・

// 変換行列を設定（物体のモデル座標系→カメラ座標系）
glTranslatef( 0.0, 1.0, 0.0 );

// 物体（１枚のポリゴン）を描画
・・・・・・
```

描画関数(2/4)

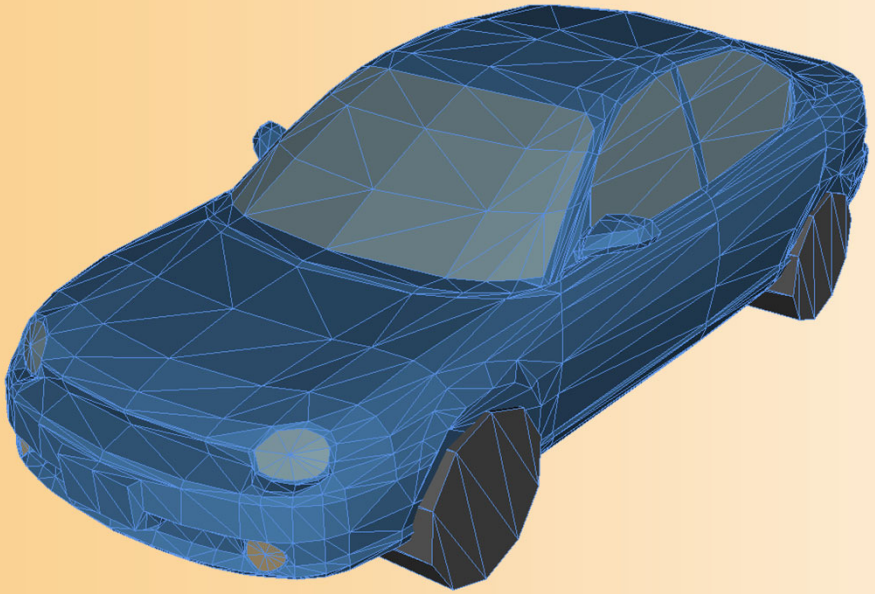
- 1枚の四角形として地面を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - 真上(0,1,0)を向き、水平方向の長さ10の四角形

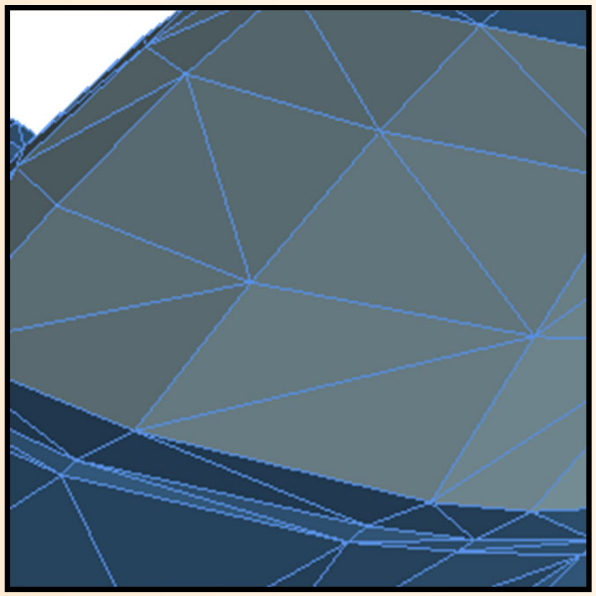
```
// 地面を描画
glBegin( GL_POLYGON );
        glNormal3f( 0.0, 1.0, 0.0 );
        glColor3f( 0.5, 0.8, 0.5 );

        glVertex3f( 5.0, 0.0, 5.0 );
        glVertex3f( 5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0, 5.0 );
glEnd();
```



ポリゴンモデル（復習）

- 物体の表面の形状を、多角形（ポリゴン）の集まりによって表現する方法
  - 最も一般的なモデリング技術
  - 本講義の演習でも、ポリゴンモデルを扱う

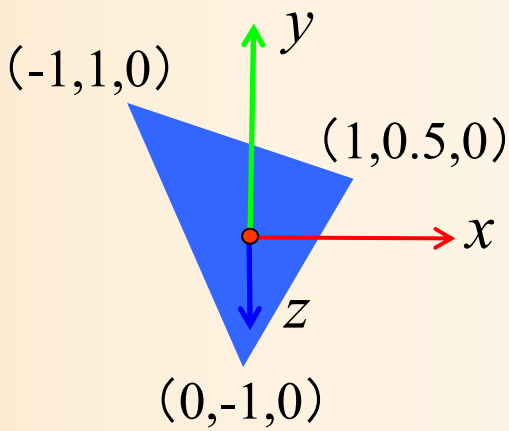




描画関数(3/4)

- 同じく、1枚の三角形を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - ポリゴンを基準とする座標系(モデル座標系)で頂点位置・法線を指定

```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );
        glVertex3f( 0.0,-1.0, 0.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```



参考：複雑なポリゴンモデルの描画

- プログラムに直接頂点座標等を記述するのではなく、以下のように、配列を使ってデータを管理するのが一般的（詳しくは後日説明）

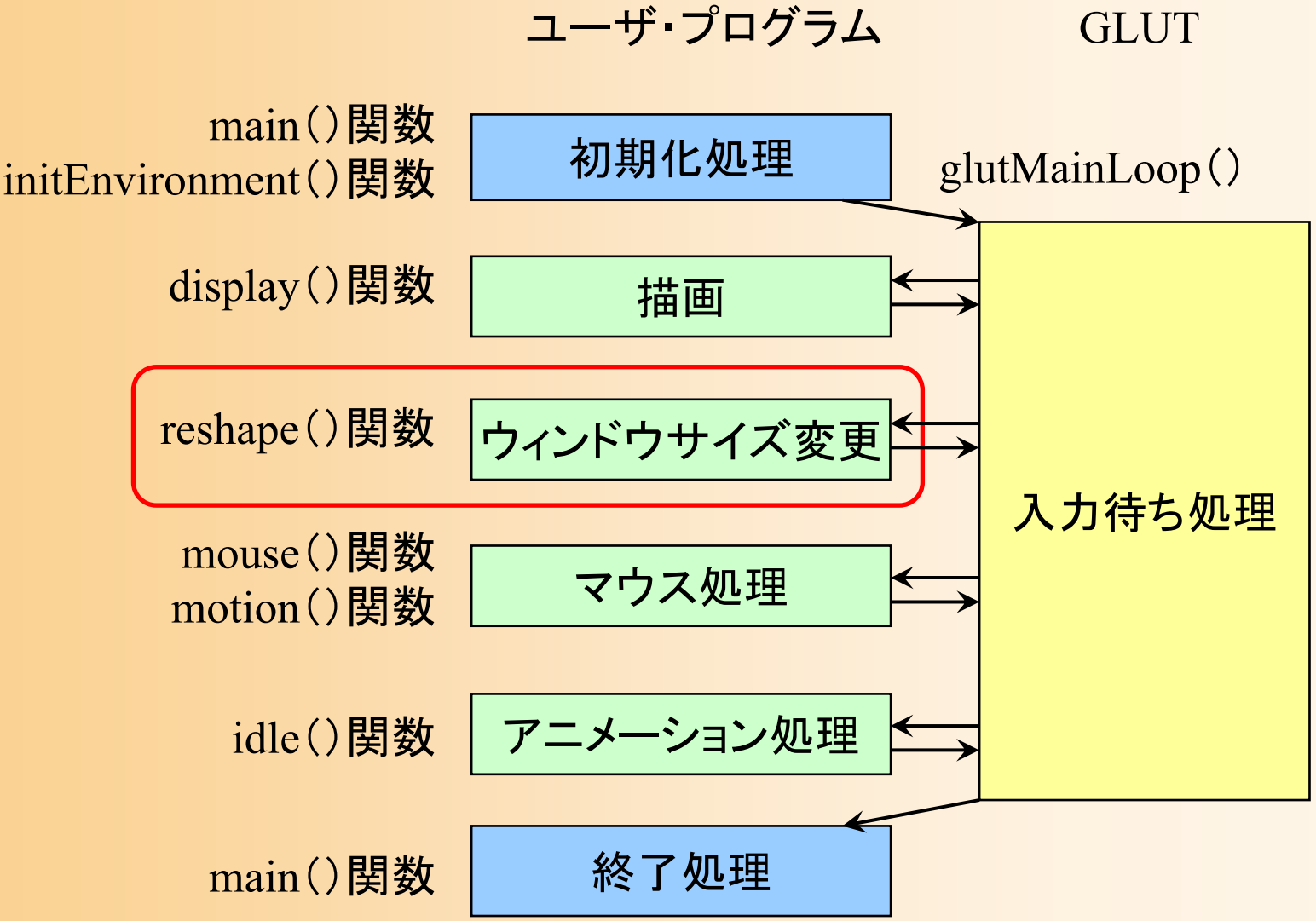
```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 },
          {-1.0,-0.8, 1.0 }, {-1.0,-0.8,-1.0 }
};

// 三角面インデックス（各三角面を構成する頂点の頂点番号）の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};
```

描画関数(4/4)

- 描画完了
  - 描画途中の画面が表示されることを避けるために、描画は裏画面(バックバッファ)に行い、描画が完了したところで、表画面(フロントバッファ)に表示する

```
        ……

        // バックバッファに描画した画面をフロントバッファに表示
        glutSwapBuffers();
}
```

サンプルプログラムの構成



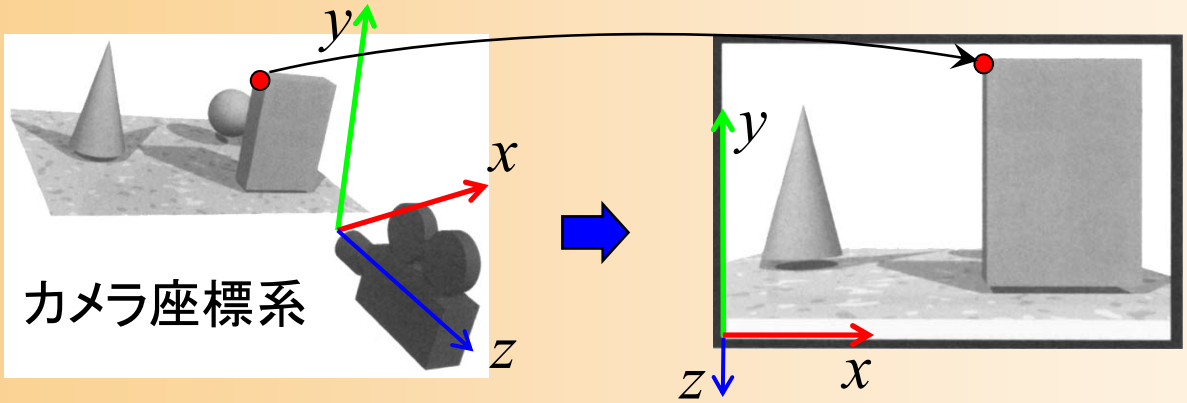
ウィンドウサイズ変更時の処理

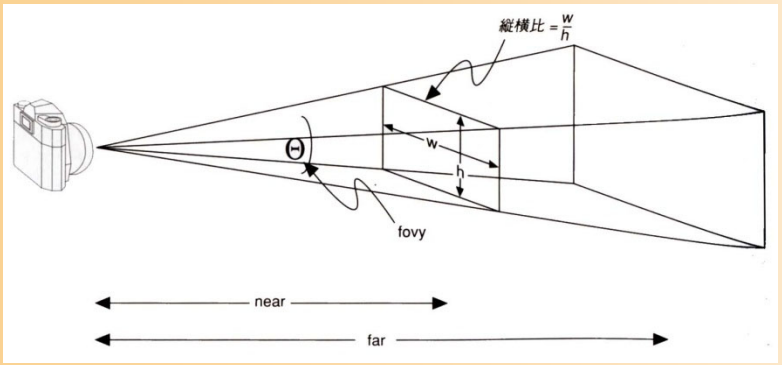
- 画面全体に描画を行うよう設定
- 射影変換行列の設定（視野角を45度とする）
  - 通常は、この設定のままで、変更は必要ない

```
void  reshape( int w, int h )
{
        // ウィンドウ内の描画を行う範囲を設定
        //（ウィンドウ全体に描画するよう設定）
        glViewport(0, 0, w, h);

        // カメラ座標系→スクリーン座標系への変換行列を設定
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();
        gluPerspective( 45, (double)w/h, 1, 500 );
}
```

参考：射影変換の設定

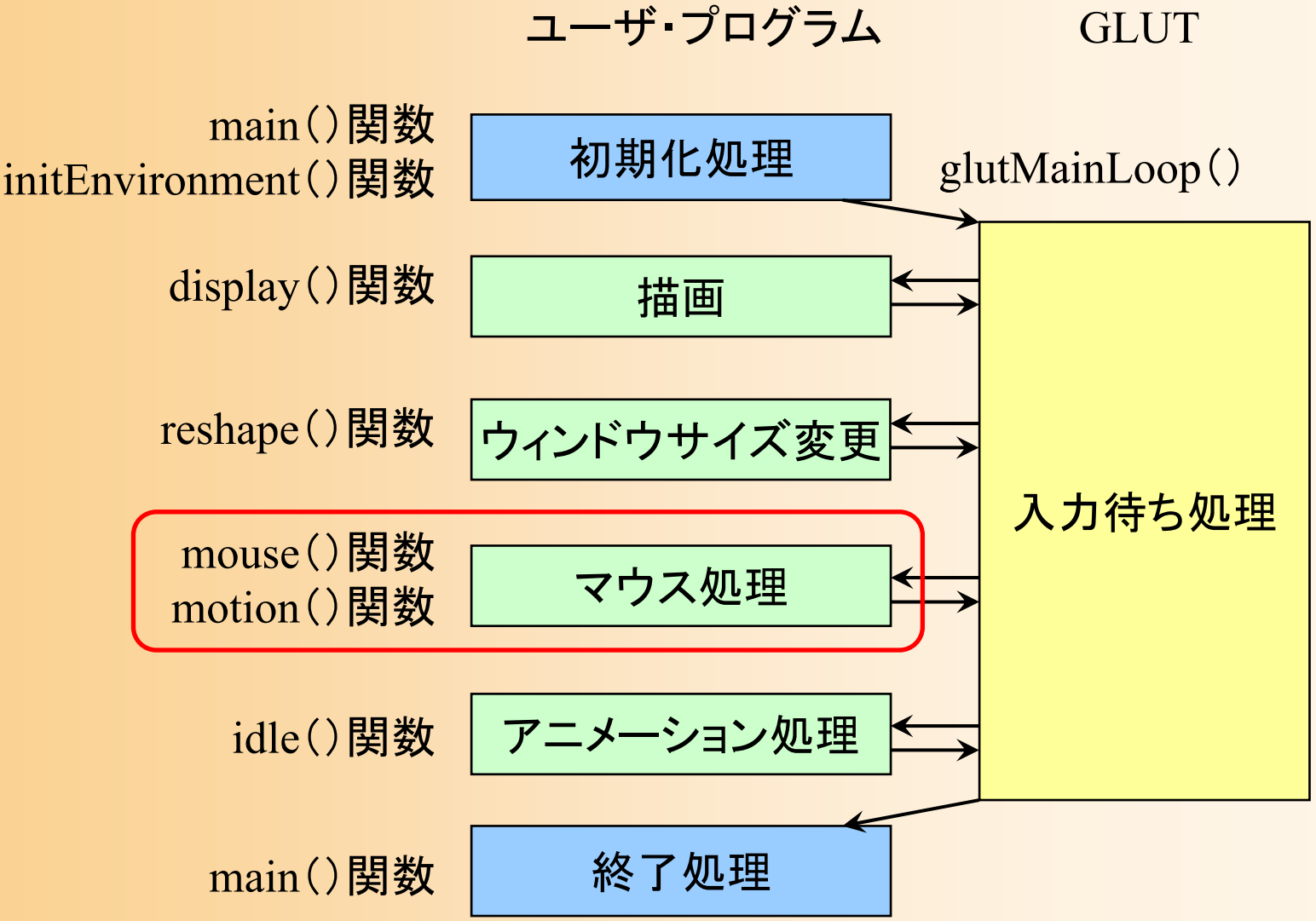
- カメラ座標系からスクリーン座標系への座標変換（射影変換）の設定





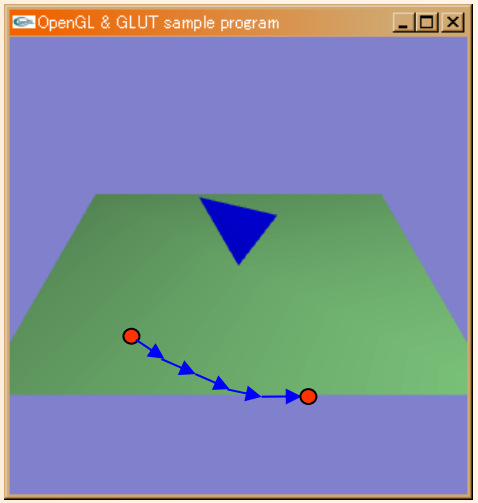
遠くにあるものほど小さく描画されるような変換（透視射影変換）を適用

サンプルプログラムの構成



マウス操作時の処理

- マウス操作のコールバック関数
  - mouse()関数
    - マウスのボタンが、押されたとき、または、離されたときに呼ばれる
  - motion()関数
    - マウスのボタンが押された状態で、マウスが動かされたとき(ドラッグ時)に定期的に呼ばれる
    - ボタンが押されない状態で、マウスが動かされたときに呼ばれる関数もある(今回は使用しない)



マウス操作時の処理(クリック処理関数)

- 右ボタンがクリックされたことを記録
  - 変数 drag_mouse_r に状態を格納

```
// マウスクリック時に呼ばれるコールバック関数
void  mouse( int button, int state, int mx, int my )
{
        // 右ボタンが押されたらドラッグ開始のフラグを設定
        if ( ( button == GLUT_RIGHT_BUTTON ) && ( state == GLUT_DOWN ) )
                drag_mouse_r = 1;
        // 右ボタンが離されたらドラッグ終了のフラグを設定
        else if ( ( button == GLUT_RIGHT_BUTTON ) && ( state == GLUT_UP ) )
                drag_mouse_r = 0;

        // 現在のマウス座標を記録
        last_mouse_x = mx;
        last_mouse_y = my;
}
```

マウス操作時の処理(ドラッグ処理関数1)

- ドラッグされた距離に応じて視点を変更
  - 視点の方位角 camera_pitch を変化
    - 前回と今回のマウス座標の差から計算

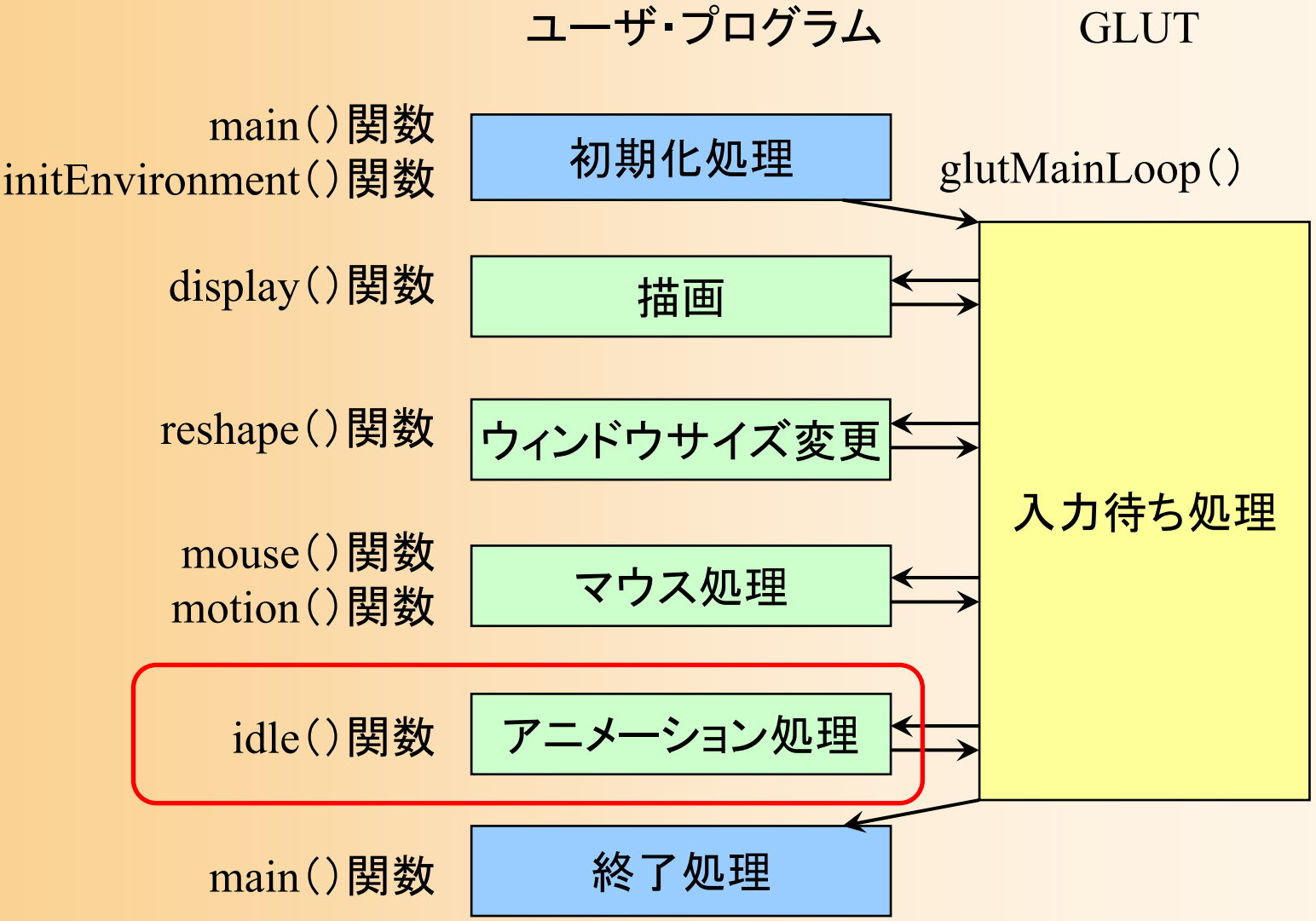
```
void  motion( int mx, int my )
{
        // 右ボタンのドラッグ中であれば、
        // マウスの移動量に応じて視点を回転する
        if ( drag_mouse_r == 1 )
        {
                // マウスの縦移動に応じてX軸を中心に回転
                camera_pitch -= ( my - last_mouse_y ) * 1.0;
                if ( camera_pitch < -90.0 )
                        camera_pitch = -90.0;
                else if ( camera_pitch > 0.0 )
                        camera_pitch = 0.0;
        }
        ......
```

マウス操作時の処理(ドラッグ処理関数2)

- 再描画の指示を行う
  - 視点の方位角 camera_pitch の変化に応じて、画面を再描画するため

```
        // 今回のマウス座標を記録
        last_mouse_x = mx;
        last_mouse_y = my;

        // 再描画の指示を出す
        glutPostRedisplay();
}
```

サンプルプログラムの構成



アイドル時の処理

- 描画やマウス入力を処理する必要がないときに定期的に呼ばれる関数
  - 物体の位置・向きを少しずつ変化させるといった、アニメーションを実現するために利用できる
  - サンプルプログラムでは、現在は何も処理を行っていない（今後処理を追加する）

```
void  idle( void )
{
        // 現在は、何も処理を行なわない
}
```

描画処理(確認)

- dysplay()関数
  - 画面のクリア(glClear()関数)
  - 変換行列の設定(ワールド座標系→カメラ座標系)
  - 光源位置の設定
  - 地面のポリゴンの描画
  - 変換行列の設定(モデル座標系→カメラ座標系)
  - ポリゴンの描画
  - 描画画面を表示(glSwapBuffers()関数)
- 変換行列の設定、ポリゴン描画については、後で詳しく説明

描画処理の詳しい説明

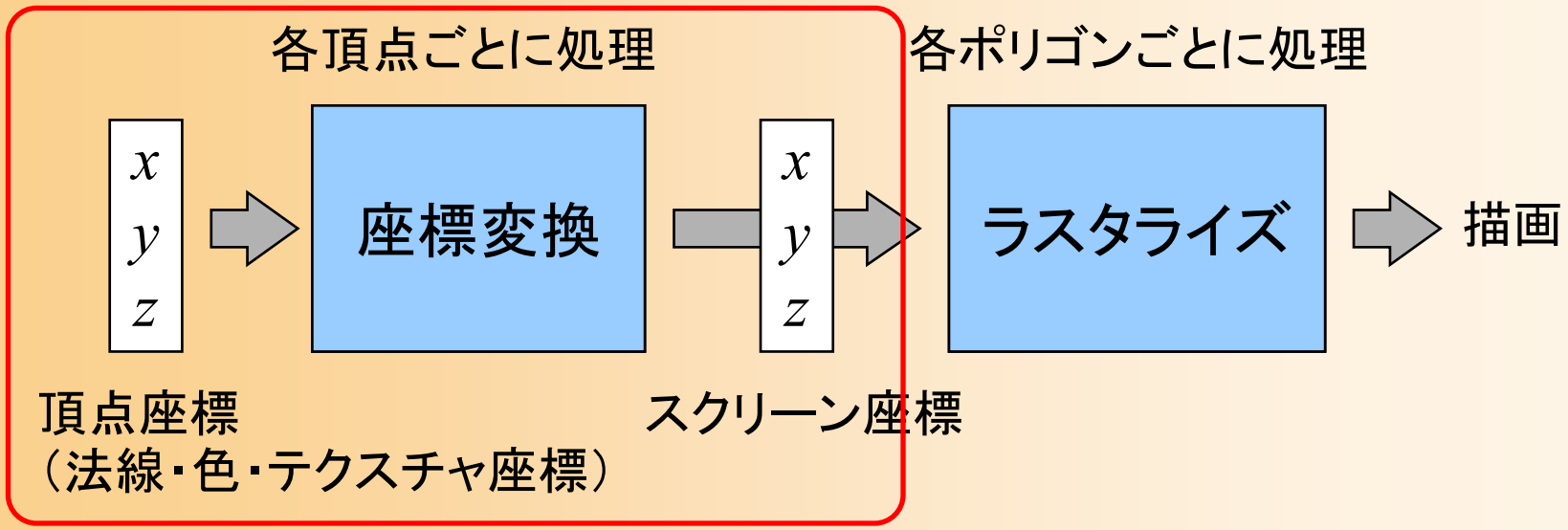
- 描画関数（display（）関数）の詳しい説明
  - 変換行列の設定
  - ポリゴンの描画
    - この後で説明
- 光源の設定
    - 今回は省略

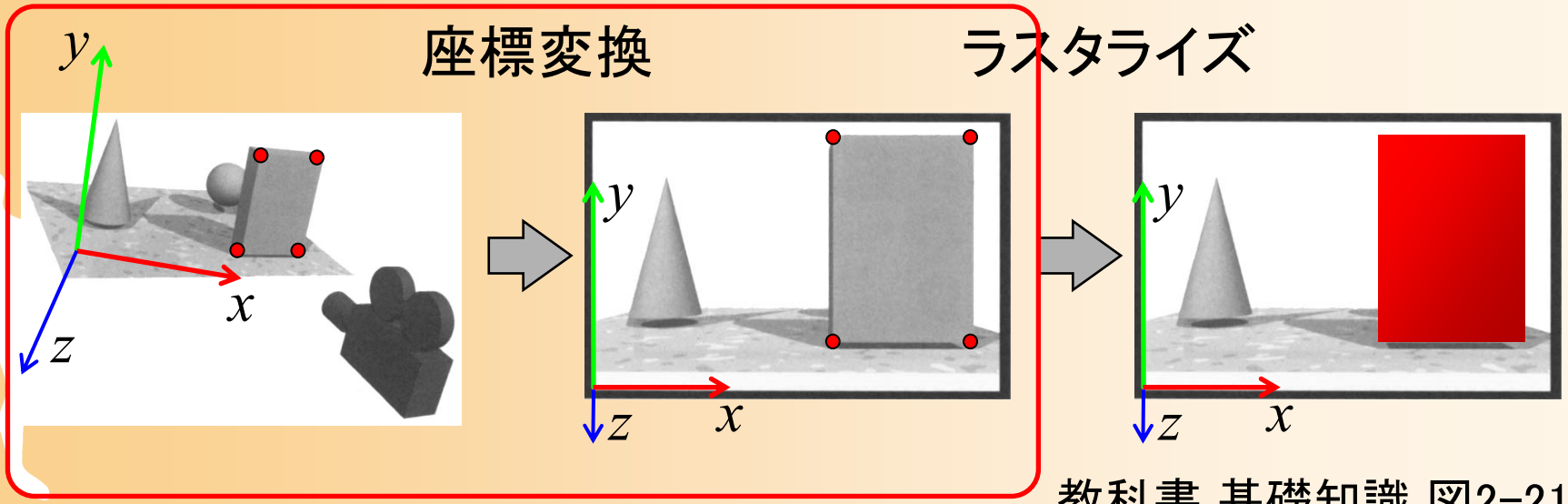
今日の内容

- OpenGL＆GLUTの概要
- サンプルプログラムの概要
- 座標変換
- 変換行列の設定
- ポリゴンモデルの描画

座標変換

座標変換





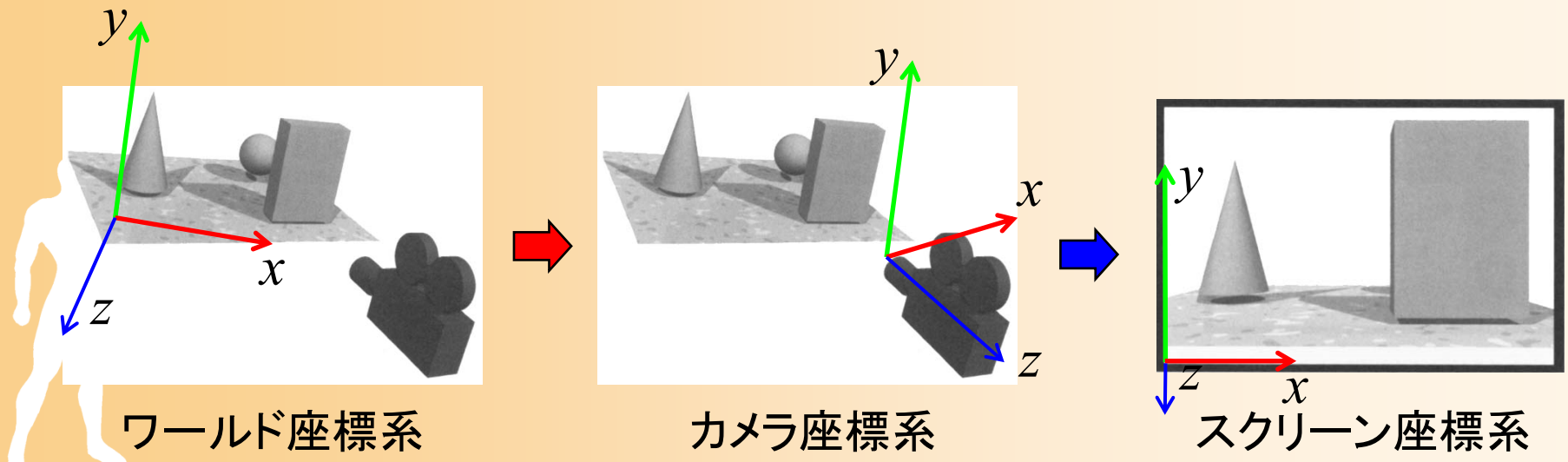
教科書 基礎知識 図2-21

座標変換

- 座標変換の概要
- 座標系
- 視野変換
- 射影変換
- 座標変換のまとめ

座標変換

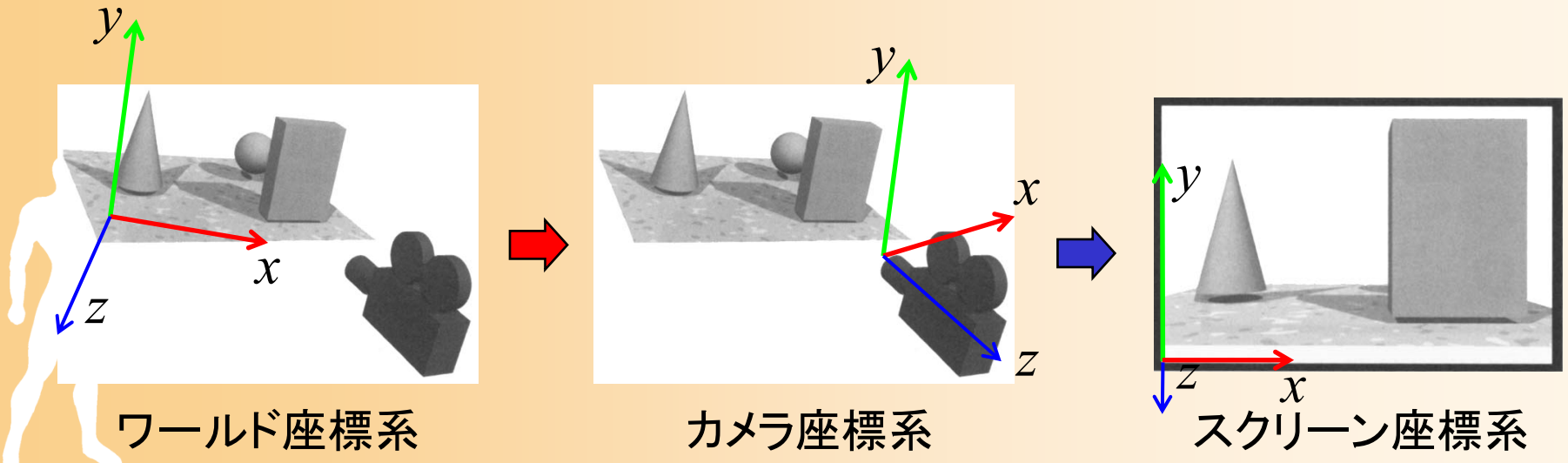
- 座標変換
  - ワールド座標系（モデル座標系）で表された頂点座標を、スクリーン座標系での頂点座標に変換する



ワールド座標系　　カメラ座標系　　スクリーン座標系

座標変換

- 2段階の座標変換により実現
  - ワールド座標からカメラ座標系への視野変換
  - カメラ座標系からスクリーン座標系への射影変換
    - 行列計算（同次座標変換）によって、上記の2種類の変換を実現する



ワールド座標系　　カメラ座標系　　スクリーン座標系

座標変換

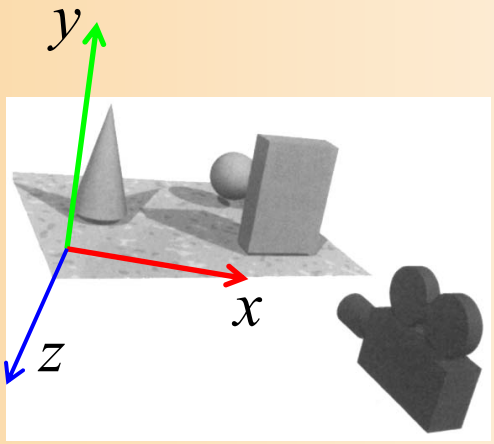
- 座標変換の概要
- 座標系
- 視野変換
- 射影変換
- 座標変換のまとめ

座標系の種類

- 原点と座標軸の取り方により、さまざまな座標系がある
  - モデル座標系
  - ワールド座標系
  - カメラ座標系
  - スクリーン座標系
- 座標系の軸の取り方に違いがある
  - 右手座標系
  - 左手座標系

ワールド座標系

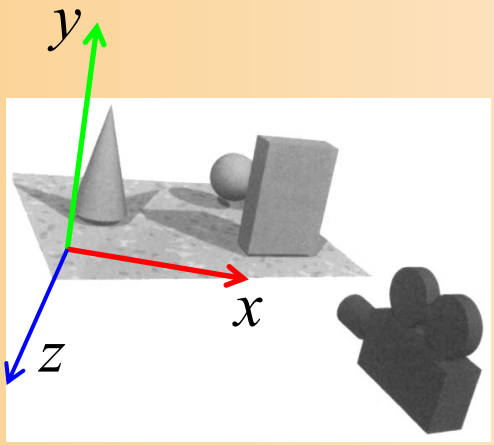
- 3次元空間の座標系
  - 物体や光源やカメラなどを配置する座標系
  - 原点や軸方向は適当にとって構わない
    - カメラと描画対象の相対位置・向きのみが重要
  - 単位も統一さえされていれば自由に設定して構わない（メートル、センチ、etc）



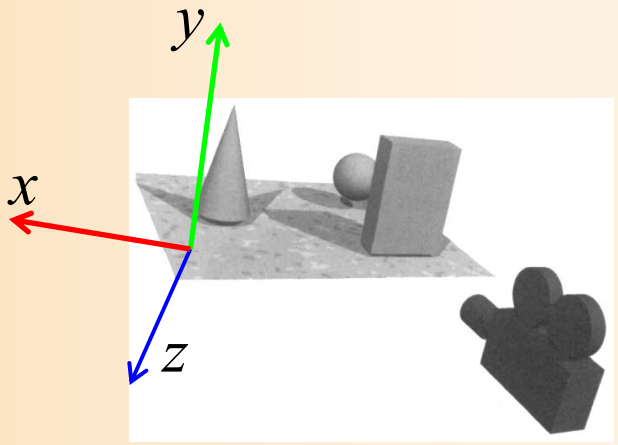
ワールド座標系

右手座標系と左手座標系

- 右手座標系と左手座標系
  - 座標系の軸の取り方の違い
  - 親指をX軸、人差し指をY軸、中指をZ軸とすると
    - 右手の指で表されるのが右手系（OpenGLなど）
    - 左手の指で表されるのが左手系（DirectXなど）



右手座標系



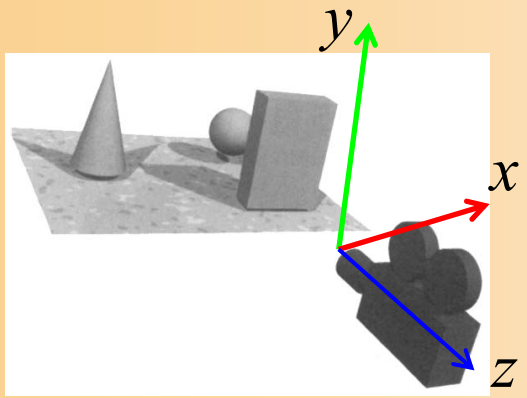
左手座標系

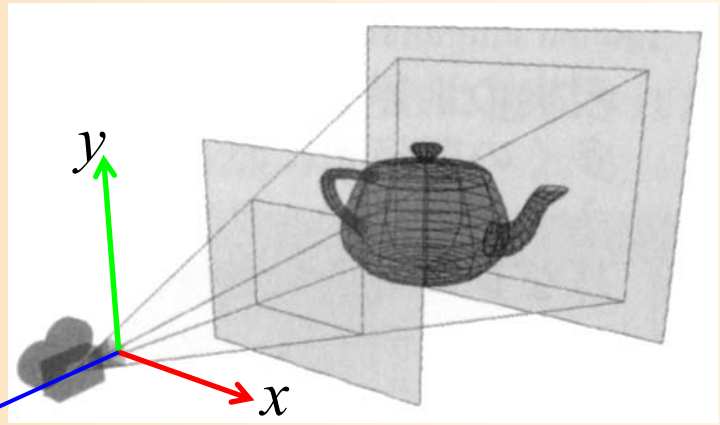
右手座標系と左手座標系（続き）

- 右手座標系と左手座標系の違い
  - 基本的にはほとんど同じ
  - 外積の定義が異なる
    - 外積の計算式は、右手座標系で定義されたもの
    - 左手座標系で外積を計算するときには、符号を反転する必要がある
      - 剛体の運動計算や電磁気などの物理計算では重要になる（この講義では扱わない）
  - 異なる座標系で定義されたモデルデータを利用する時には、変換が必要
    - 左右反転、面の方向を反転

カメラ座標系

- カメラを中心とする座標系
  - X軸・Y軸がスクリーンのX軸・Y軸に相当
  - 奥行きがZ軸に相当

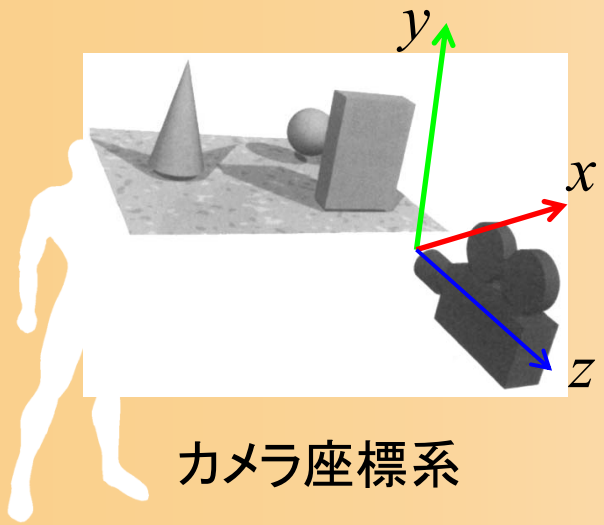




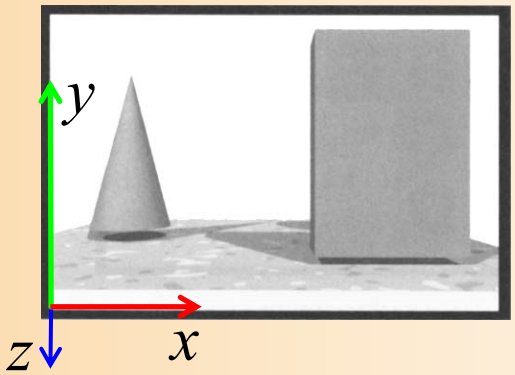
カメラ座標系

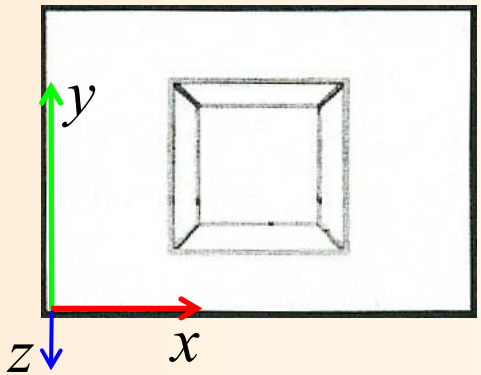
スクリーン座標系

- スクリーン上の座標
  - 射影変換（透視変換）を適用した後の座標
    - 奥にあるものほど中央に描画されるように座標計算
  - スクリーン座標も奥行き値（Z座標）も持つことに注意 → Zバッファ法で使用



カメラ座標系

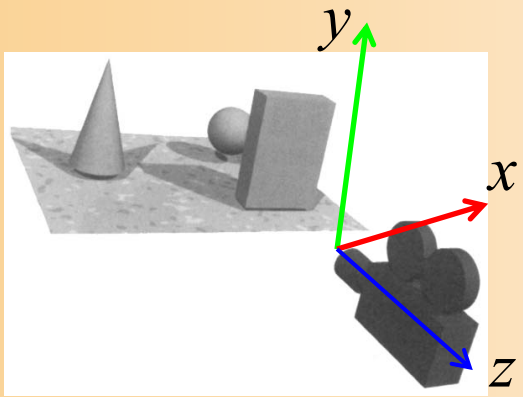




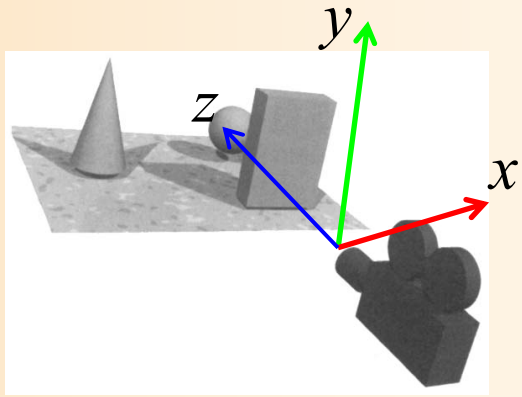
スクリーン座標系

右手座標系と左手座標系

- カメラ座標系・スクリーン座標系も、軸の取り方によって、座標系は異なる
  - 手前がZ軸の正方向（OpenGL）
  - 奥がZ軸の正方向（DirectX）
- こちらも基本的にはどちらでも構わない



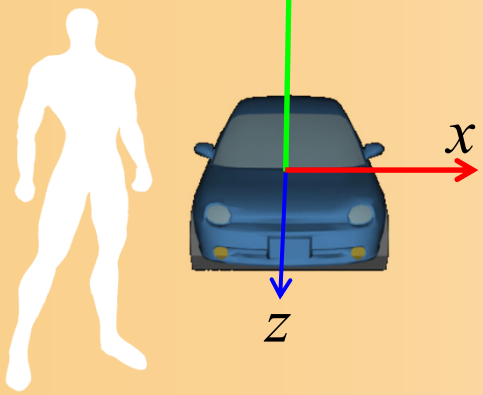
手前がZ軸の正方向

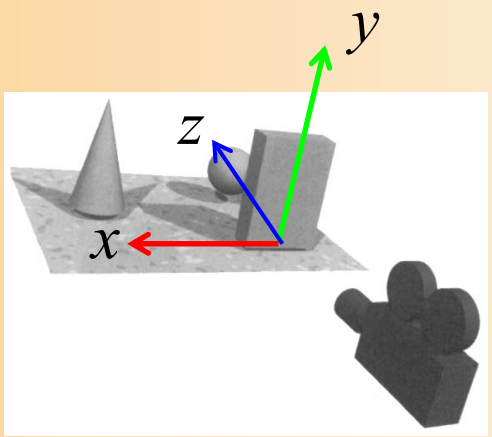


奥がZ軸の正方向

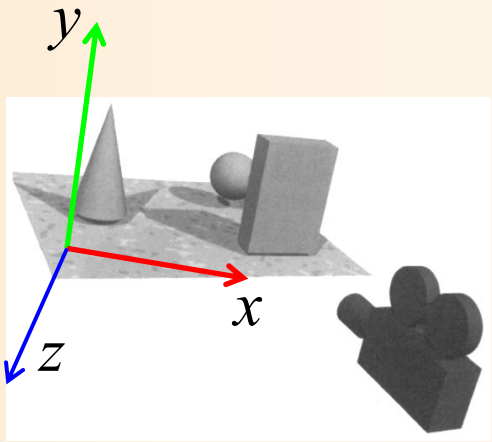
モデル座標系

- 物体のローカル座標
  - ポリゴンモデルの頂点はモデル内部の原点を基準とするモデル座標系で定義される
  - 正面方向をZ軸にとる場合が多い
  - ワールド座標系にモデルを配置





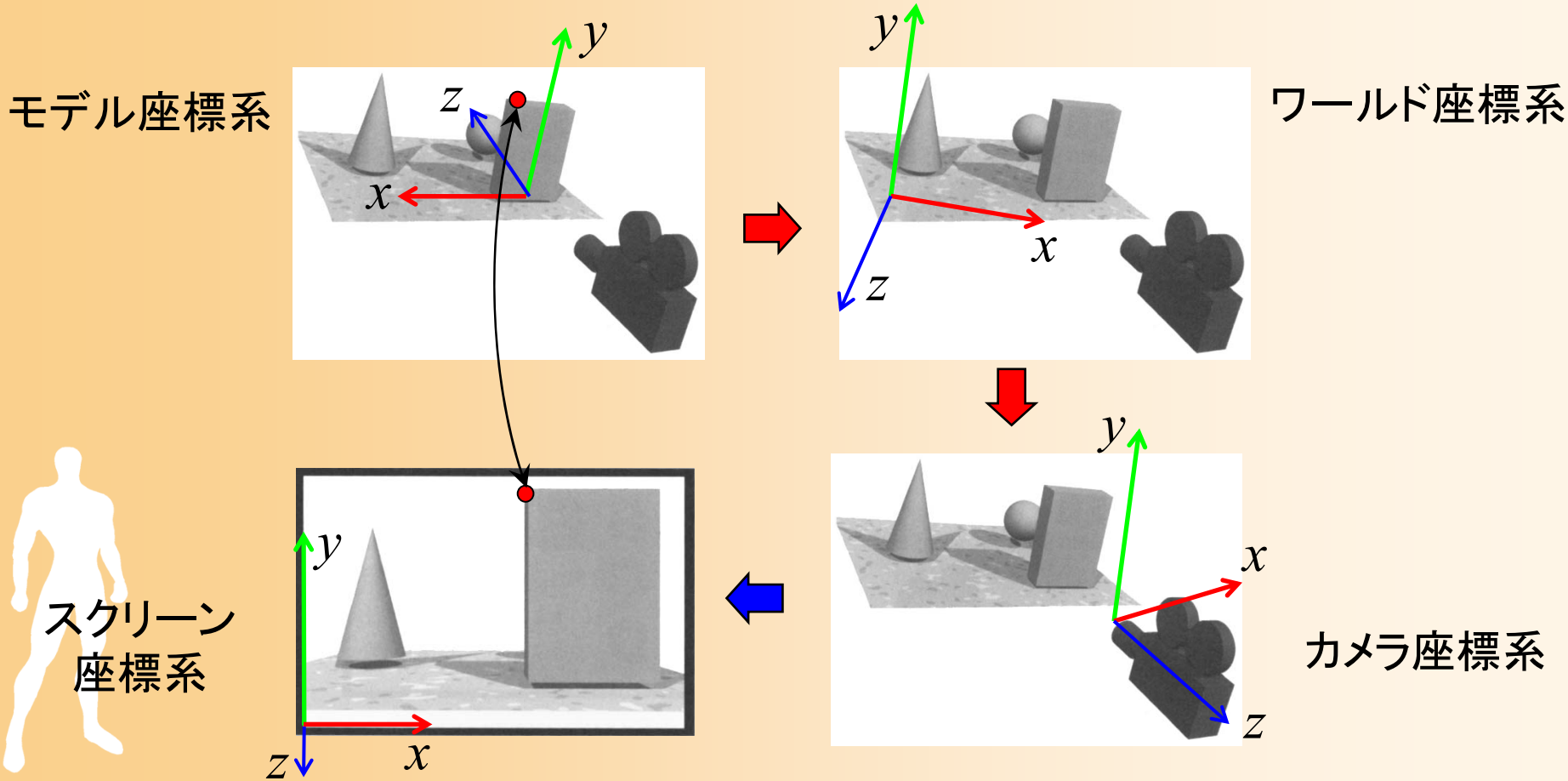
モデル座標系



ワールド座標系

座標変換の流れ（詳細）

- モデル座標系からスクリーン座標系に変換

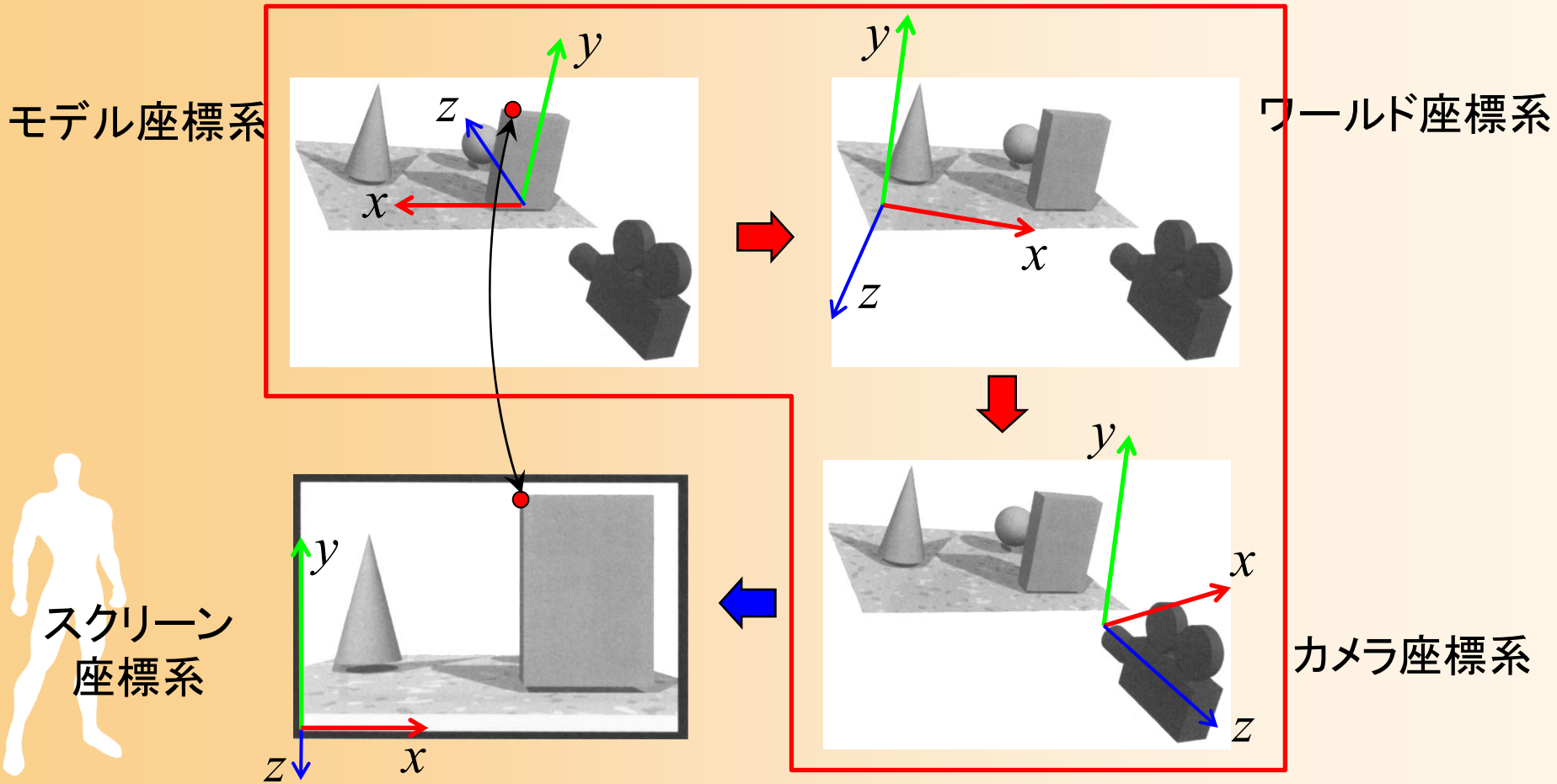


座標変換

- 座標変換の概要
- 座標系
- 視野変換
- 射影変換
- 座標変換のまとめ

視野変換

- モデル座標系からカメラ座標系に変換



同次座標変換

- 同次座標変換
  - ４×４行列の演算により、３次元空間における平行移動・回転・拡大縮小（アフィン変換）などの操作を統一的に実現
    - (x, y, z, w)の４次元座標値（同次座標）を扱う
    - ３次元座標値は$(x/w, y/w, z/w)$で計算（通常は $w = 1$）

$$\begin{pmatrix} R_{00}S_x & R_{01} & R_{02} & T_x \\ R_{10} & R_{11}S_y & R_{12} & T_y \\ R_{20} & R_{21} & R_{22}S_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

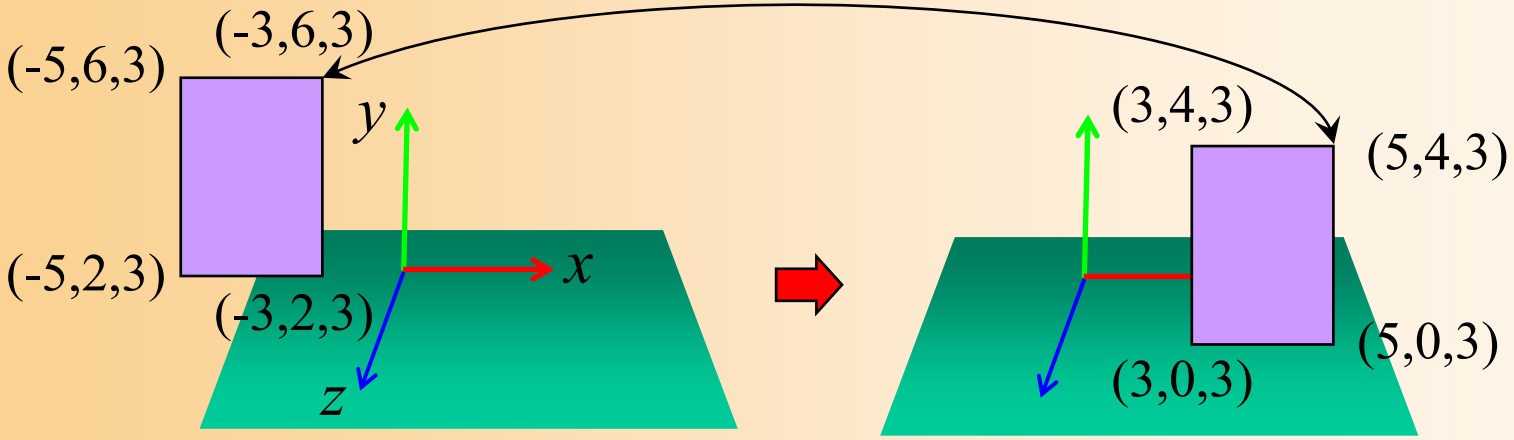
平行移動

- 平行移動
  - (Tx,Ty,Tz)の平行移動
    - 4×4行列を用いることで、平行移動を適用することができる

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+T_x \\ y+T_y \\ z+T_z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

平行移動の例

- (8,-2,0)平行移動



$$\begin{pmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+8 \\ y-2 \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

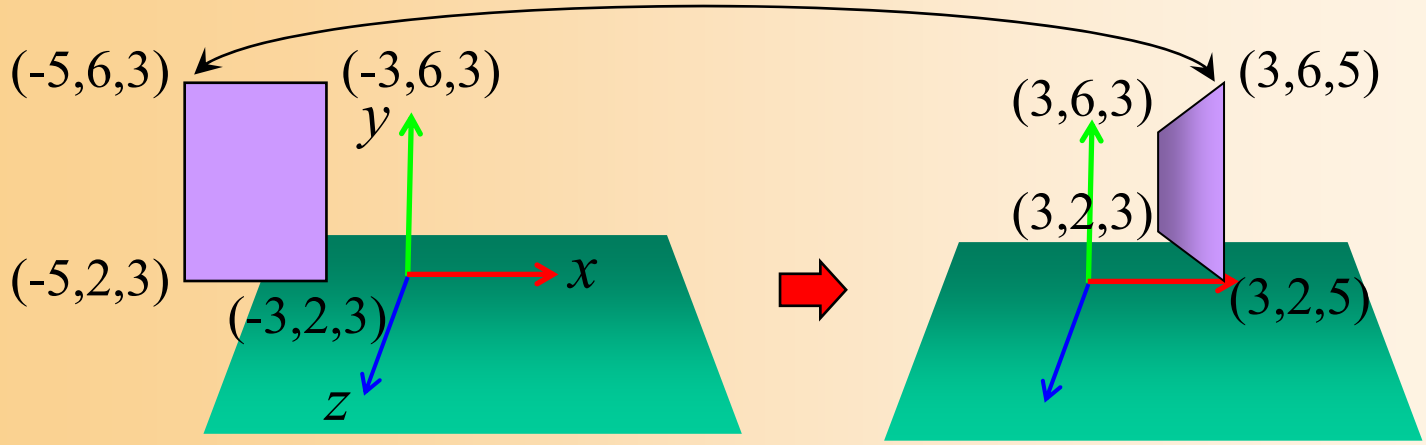
回転

- 回転
  - 原点を中心とする回転を表す

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} & 0 \\ R_{10} & R_{11} & R_{12} & 0 \\ R_{20} & R_{21} & R_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} R_{00}x + R_{01}y + R_{02}z \\ R_{10}x + R_{11}y + R_{12}z \\ R_{20}x + R_{21}y + R_{22}z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

回転の例

- Y軸を中心として 90度回転



$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} z \\ y \\ -x \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

回転変換の行列

- 回転変換の行列の導出方法
  - 各軸を中心として右ねじの方向の回転（軸の元から見て反時計回り方向の回転）を通常使用
  - yz平面、xz平面、xy平面での回転を考えれば、2次元平面での回転変換と同様に求められる
    - 2次元平面での回転行列は、高校の数学の内容

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

X軸を中心とする回転変換

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

Y軸を中心とする回転変換

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

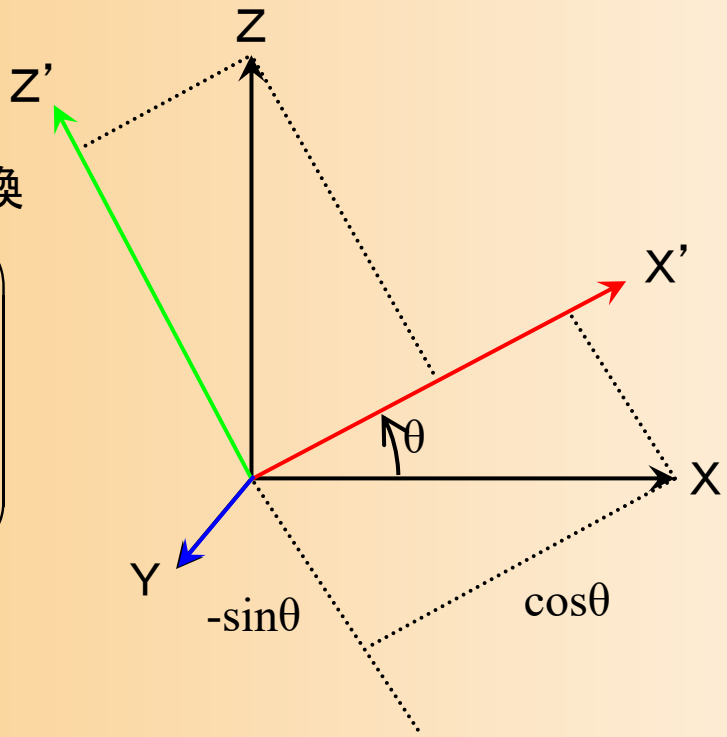
Z軸を中心とする回転変換

回転変換の行列（続き）

- 回転変換の行列の導出方法の例
  - 例えば、y軸周りの回転行列は、xz平面での回転を考えれば、導出できる

Y軸を中心とする回転変換

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



変換前のX軸・Z軸方向の単位ベクトルの、変換後の座標系での座標

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

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

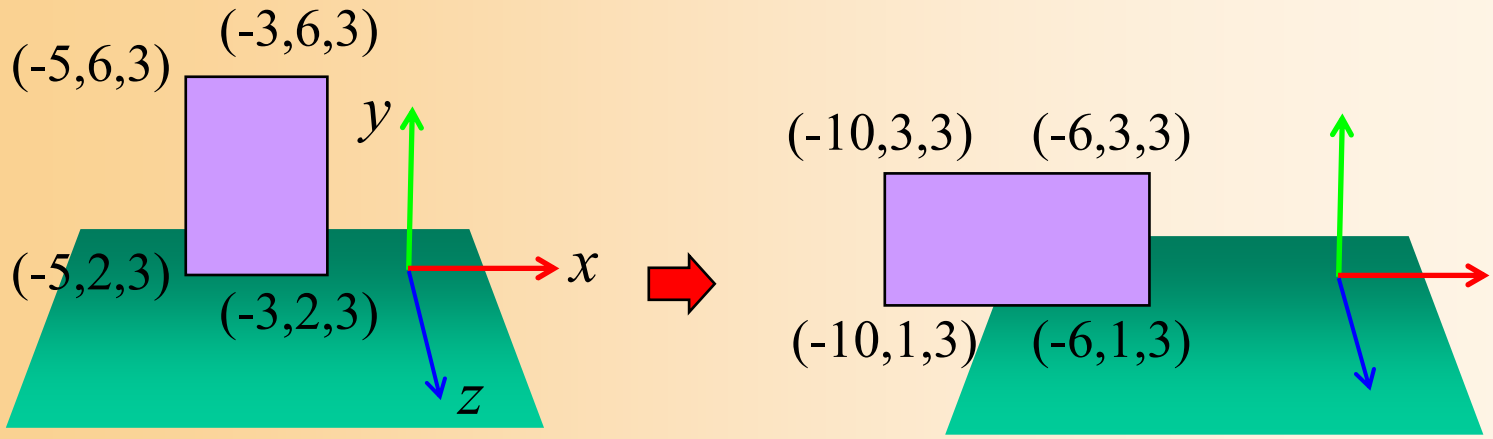
拡大縮小

- 拡大縮小
  - (Sx,Sy,Sz)倍のスケーリング

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x x \\ S_y y \\ S_z z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

拡大縮小の例

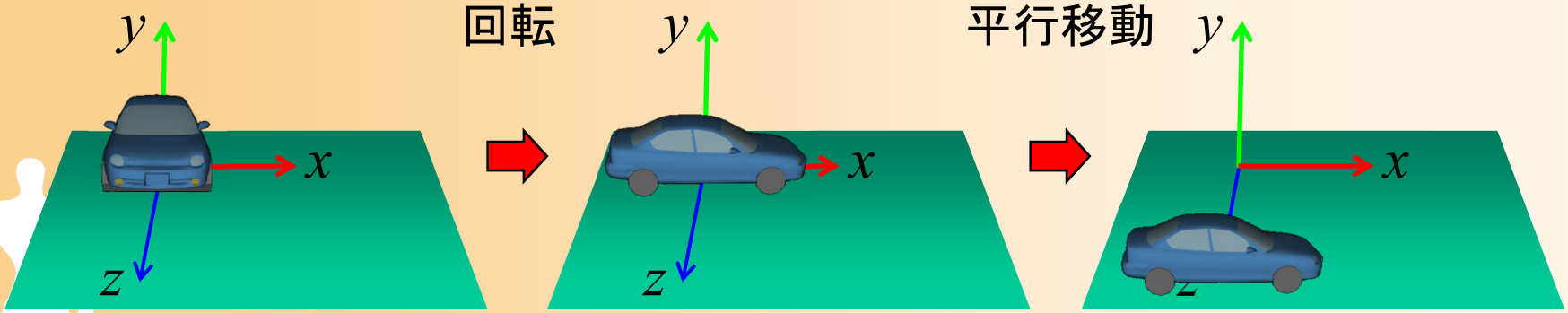
- (2, 0.5, 1)倍に拡大縮小



$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x \\ 0.5y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

変換行列の適用

- 1つの行列かけ算で各種の変換を適用可能
- 複数の行列を順番にかけていくことで、複数の変換を連続して適用できる
  - 回転･移動の組み合わせの例

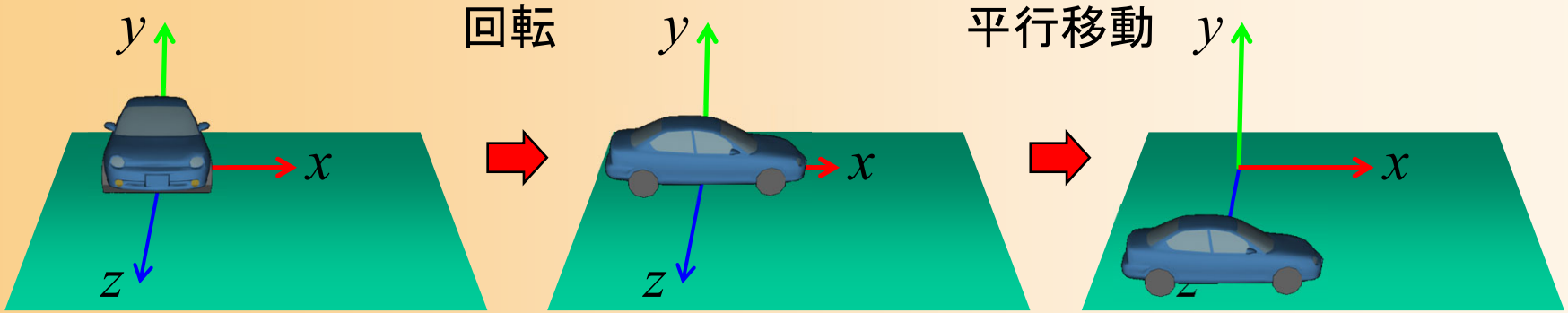


$$\begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

複数の変換行列の適用例(1)

- 回転･移動の組み合わせの例



平行移動　　回転

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

先に適用する方が右側になることに注意！

行列計算の適用順序

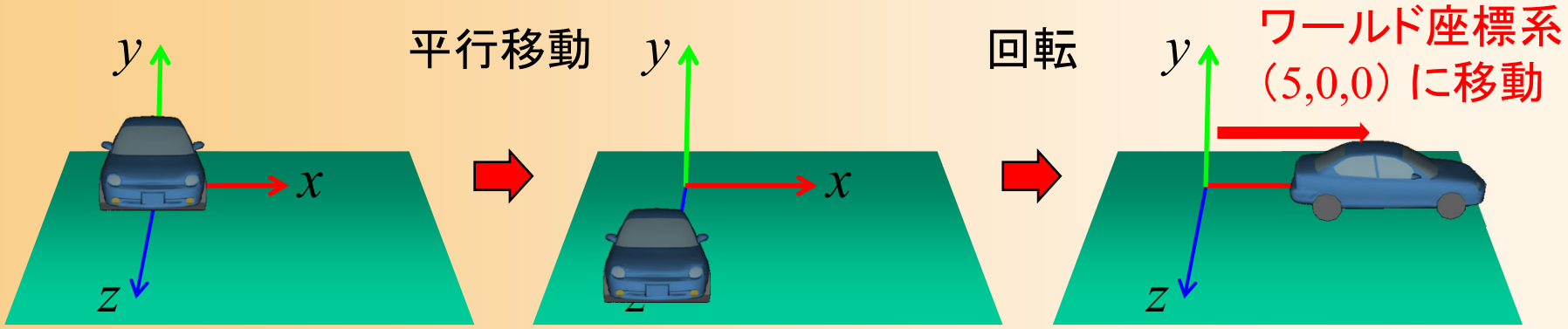
- 行列演算では可換則は成り立たないことに注意！

$$AB \neq BA$$

- 行列の適用順序によって結果が異なる
  - 例:
    - 回転 → 平行移動
    - 平行移動 → 回転

複数の変換行列の適用例(2)

- 移動→回転の順番で適用したときの例



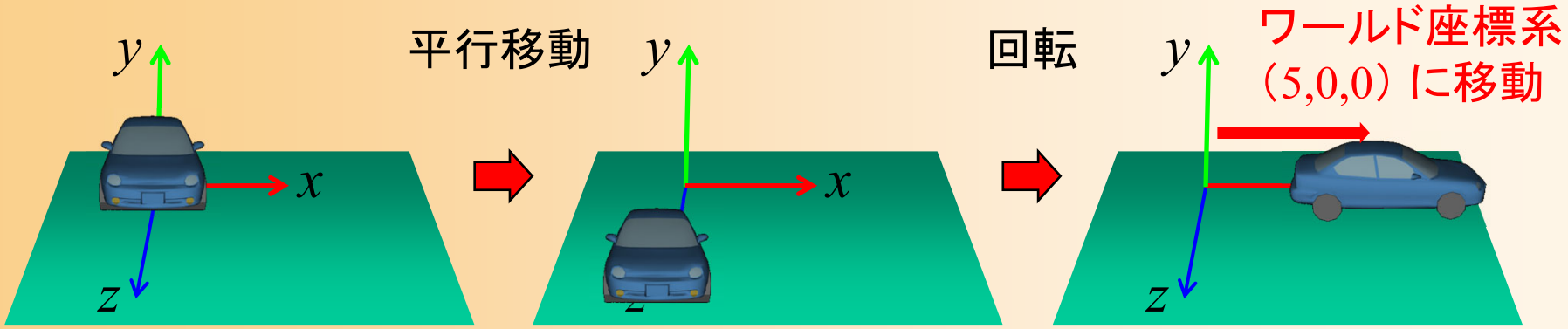
回転　　平行移動

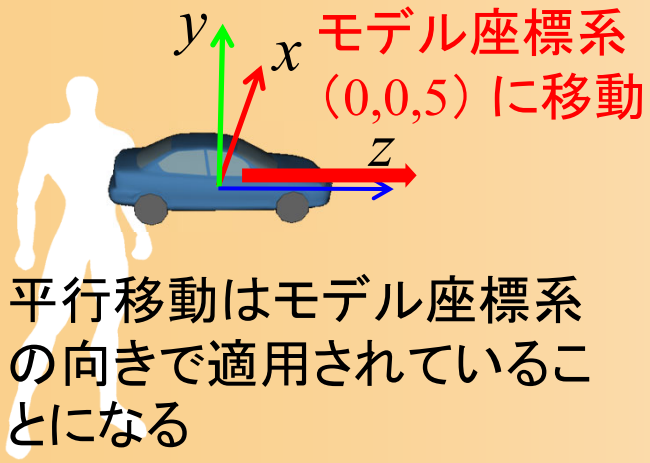
$$\begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 5 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

この場合は平行移動成分にも回転がかかる

複数の変換行列の適用例(2)

- 移動→回転の順番で適用したときの例





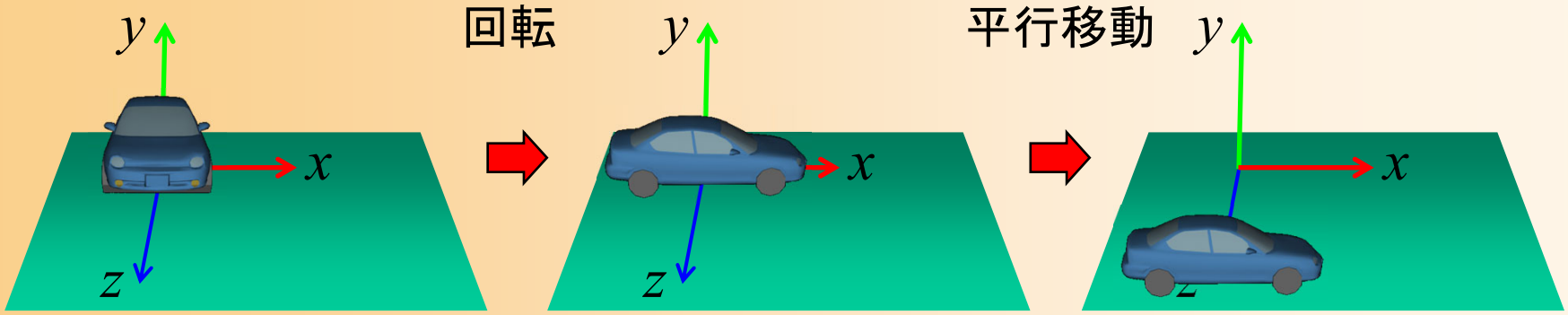
平行移動はモデル座標系の向きで適用されていることになる

回転　　平行移動

$$\begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

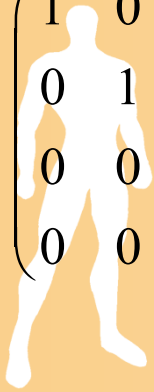
複数の変換行列の適用例(1)

- 回転→移動の順番で適用(さきほどの例)



平行移動　　回転

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$



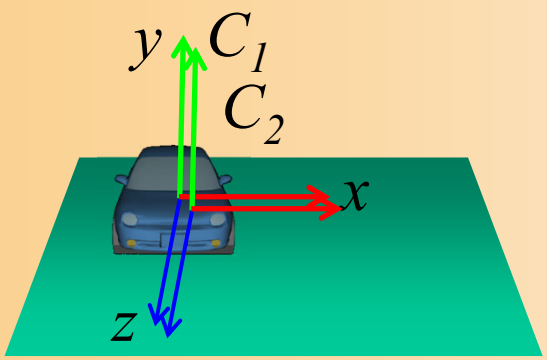
こちらの順番の方が普通に使う場合が多い

座標変換の考え方

- 座標変換の考え方
  - ある座標系内での回転・平行移動・拡大縮小の変換と考えることもできるし、
  - ある座標系から別の座標系への座標系の変換と考えることもできる

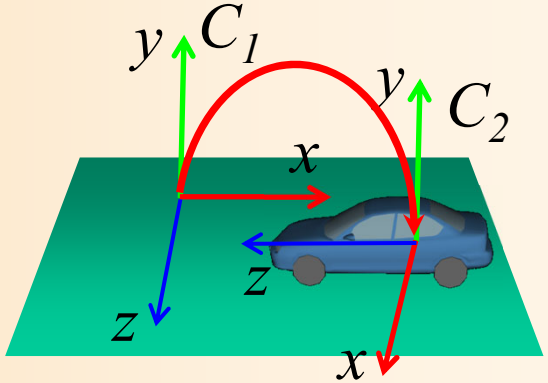
モデルをC1→C2に移動・回転 ＝
C2→C1の変換行列を求める

変換行列を適用しない状態では、
移動や回転はなし



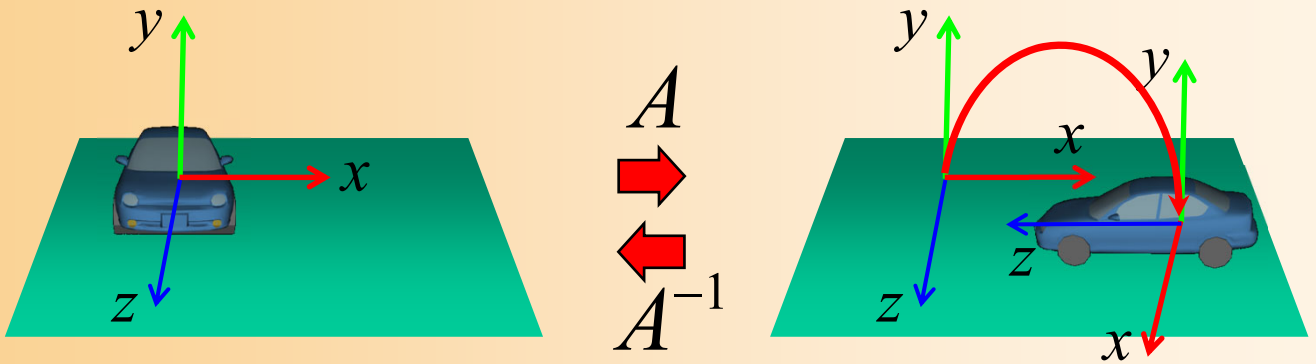
回転→移動の
変換を適用

A



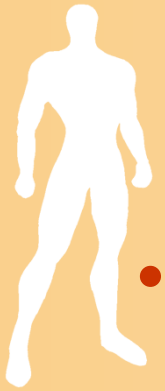
座標変換の逆変換

- 逆行列を計算すれば、反対方向の変換も求まる
- アフィン変換（回転・平行移動・拡大縮小）の行列は、正則であるため、常に逆行列が存在する



同次座標変換のメリット

- 行列演算だけでさまざまな処理を行える
  - 同次座標変換を使わずとも、回転・平行移動・拡大縮小など各処理に応じて計算することは可能
    - それぞれの処理だけをみればこの方が高速
  - 各種処理を統一的に扱えることに意味がある
- 複数の変換をまとめて一つの行列にできる
  - 最初に一度全行列を計算してしまえば、後は各頂点につき1回の行列演算だけで処理できる
- CG以外の分野でも広く用いられている



同次座標変換の表記方法

- 2通りの書き方がある
  - どちらの書き方で考えても良い
    - 本講義では、左から行列を掛ける表記を使用
  - 使用するライブラリによって行列データの渡し方が異なるので注意

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} & T_x \\ R_{10} & R_{11} & R_{12} & T_y \\ R_{20} & R_{21} & R_{22} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

左から行列を掛けていく表記（OpenGL）

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}^t \begin{pmatrix} R_{00} & R_{10} & R_{20} & 0 \\ R_{01} & R_{11} & R_{21} & 0 \\ R_{02} & R_{12} & R_{22} & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

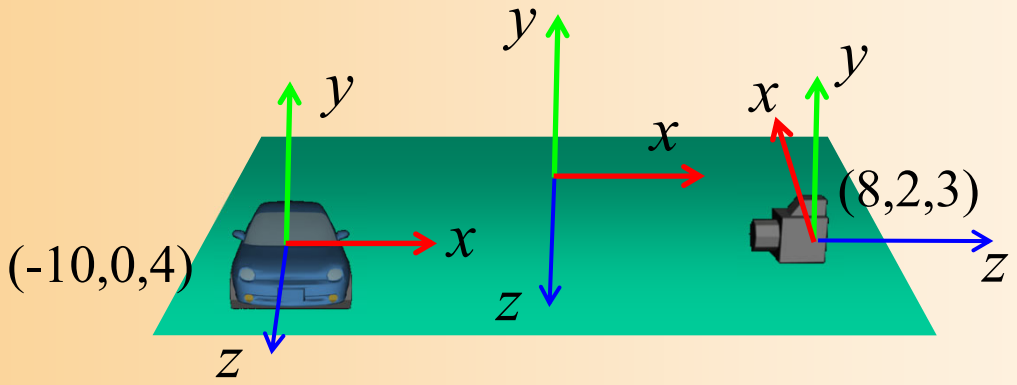
右から行列を掛けていく表記（DirectX）

2次元空間での同次座標変換

- 2次元空間（平面）でも、同様に同次座標変換は定義される
  - 3次元空間での同次座標変換・・・4×4行列
  - 2次元空間での同次座標変換・・・3×3行列
- 2次元空間の同次座標変換については、参考書を参照（本講義では扱わない）

座標変換の例

- 下記のシーンにおける、モデル座標系からカメラ座標系への変換行列を計算せよ
  - 物体の位置が（-10,0,4）にあり、ワールド座標系と同じ向き
  - カメラの位置が（8,2,3）にあり、ワールド座標系のY軸を中心として90度回転している



座標変換の例

- 座標変換の考え方
  - モデル座標系→ワールド座標系 への変換行列
  - ワールド座標系→カメラ座標系 への変換行列

  の2つの変換を求めて、順に適用することで、
  モデル座標系→カメラ座標系 への変換を実現
    - カメラやモデルの位置・向きは、ワールド座標系で表されているため、全体を一度に求めることは難しい

$$\begin{pmatrix} ? \end{pmatrix}\begin{pmatrix} ? \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

ワールド → カメラ　モデル → ワールド

座標変換の例

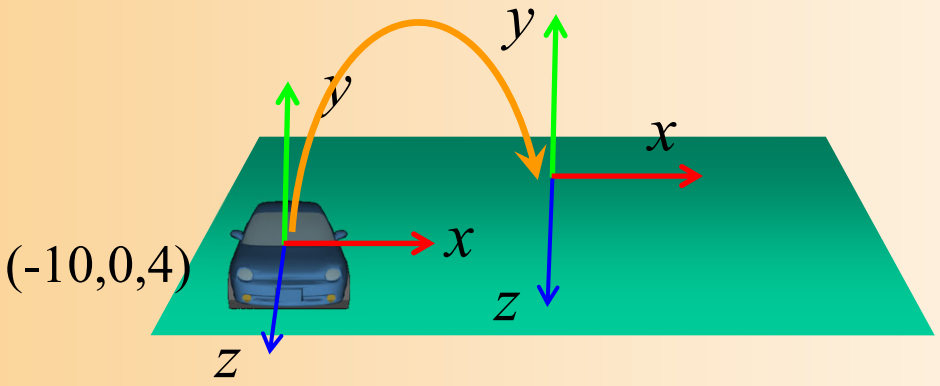
- モデル座標系 → ワールド座標系

$$\begin{pmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

平行移動のみ

回転が必要であれば、平行移動行列の前に回転行列を適用する必要がある（今回は向きが同じなので不要）

モデル座標系の原点 (0,0,0) はワールド座標系の (-10,0,4) に平行移動される



座標変換の例

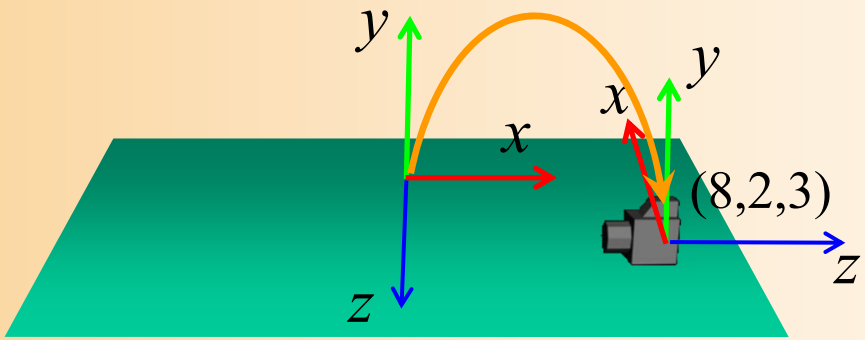
- ワールド座標系 → カメラ座標系

$$\begin{pmatrix} \cos(-90^\circ) & 0 & \sin(-90^\circ) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-90^\circ) & 0 & \cos(-90^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -8 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

カメラの座標系から見てワールド座標系は、
ワールド座標系のY軸を中心に -90 度回転

カメラの位置が(8,2,3)なので、
ワールド→カメラは(-8,-2,-3)

位置はワールド座標系で表されているので、先に平行移動を適用



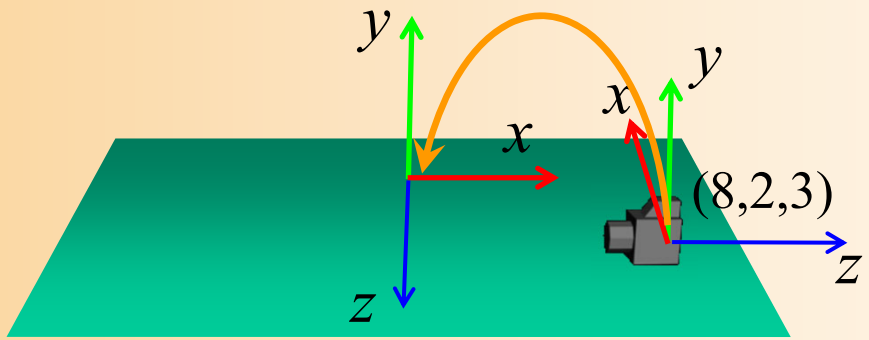
座標変換の例

- カメラ座標系 → ワールド座標系（参考）

$$\begin{pmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(90^\circ) & 0 & \sin(90^\circ) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(90^\circ) & 0 & \cos(90^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

モデル座標系 → ワールド座標系と同様の行列になる

回転→平行移動の順で適用、符号は反転の必要なし

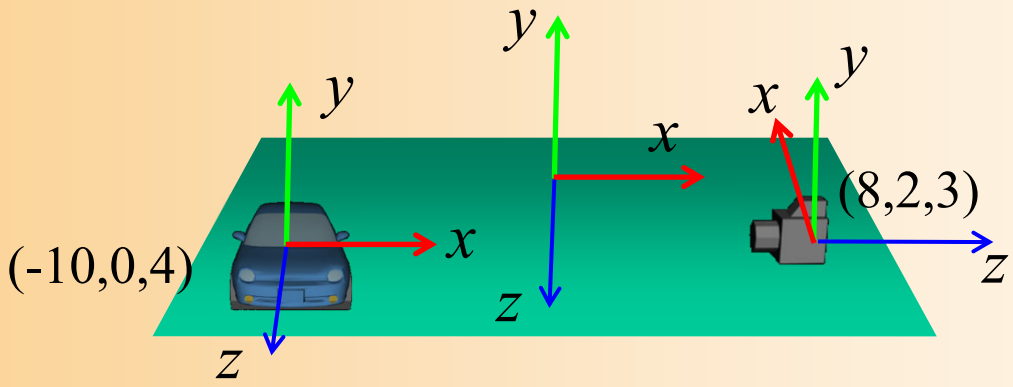


座標変換の例

- モデル座標系 → カメラ座標系

$$\begin{pmatrix} \cos(-90^\circ) & 0 & \sin(-90^\circ) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-90^\circ) & 0 & \cos(-90^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -8 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

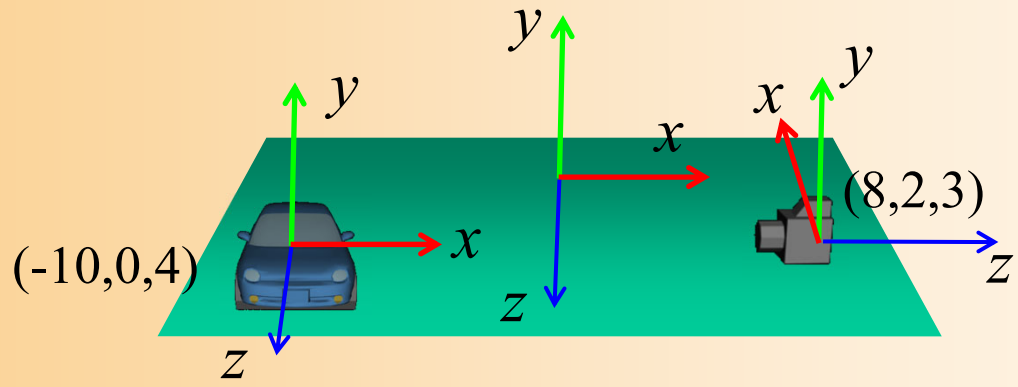
ワールド → カメラ　　モデル → ワールド



座標変換の順序に注意

- 回転と平行移動を適用する順序に注意！

$$\left(\begin{array}{c}\text{カメラ}\\ \text{座標での}\\ \text{平行移動}\end{array}\right)\left(\begin{array}{c}\text{ワールド}\\ \rightarrow\text{カメラ}\\ \text{の回転}\end{array}\right)\left(\begin{array}{c}\text{ワールド}\\ \text{座標での}\\ \text{平行移動}\end{array}\right)\left(\begin{array}{c}\text{モデル}\rightarrow\\ \text{ワールド}\\ \text{の回転}\end{array}\right)\left(\begin{array}{c}\text{モデル}\\ \text{座標での}\\ \text{平行移動}\end{array}\right)\begin{pmatrix}x\\ y\\ z\\ 1\end{pmatrix}=\begin{pmatrix}x'\\ y'\\ z'\\ 1\end{pmatrix}$$

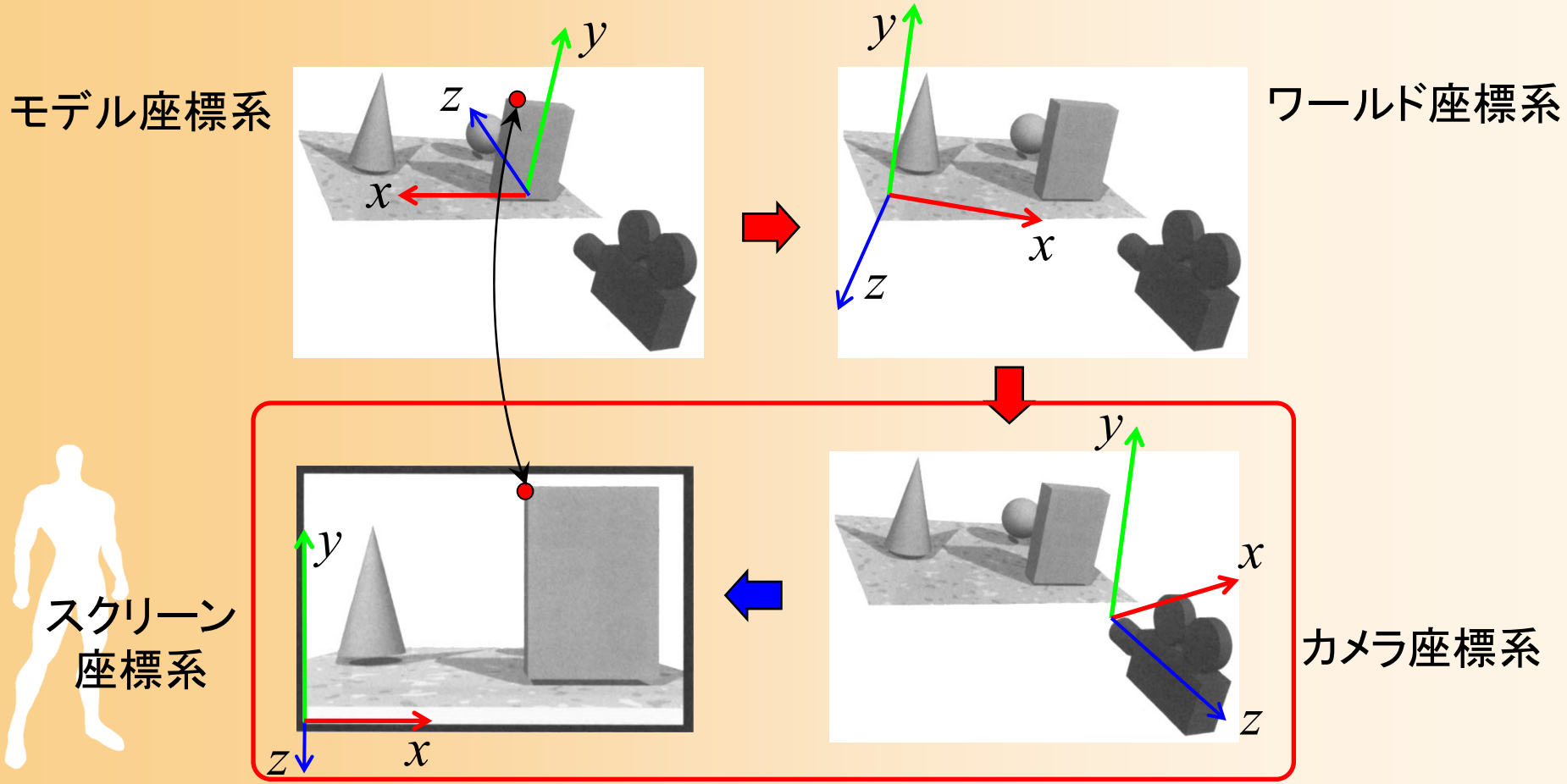


座標変換

- 座標変換の概要
- 座標系
- 視野変換
- 射影変換
- 座標変換のまとめ

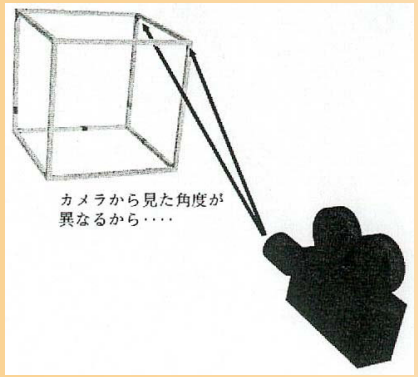
射影変換

- カメラ座標系からスクリーン座標系に変換

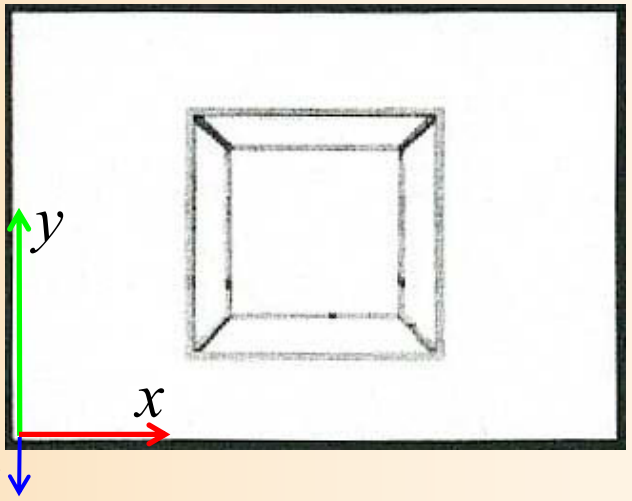


射影変換

- カメラ座標からスクリーン座標への射影変換
- 透視射影変換
  - 一般的な射影変換の方法
  - 奥にあるものほど中央に描画されるように計算

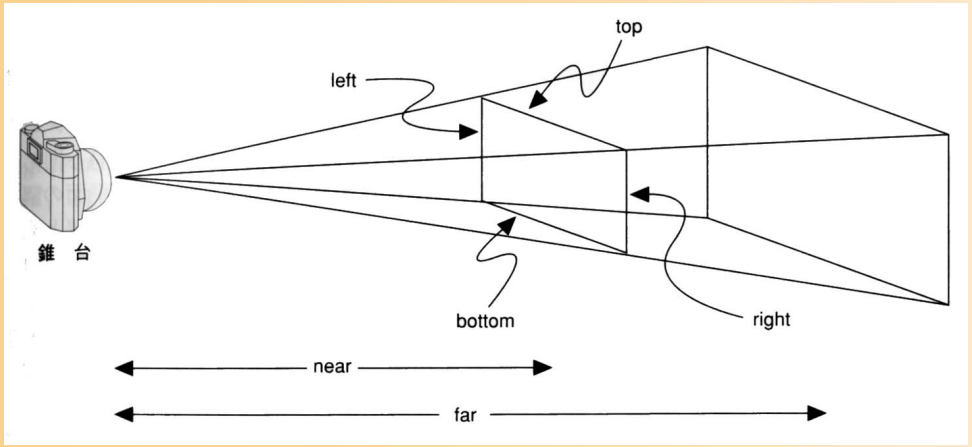


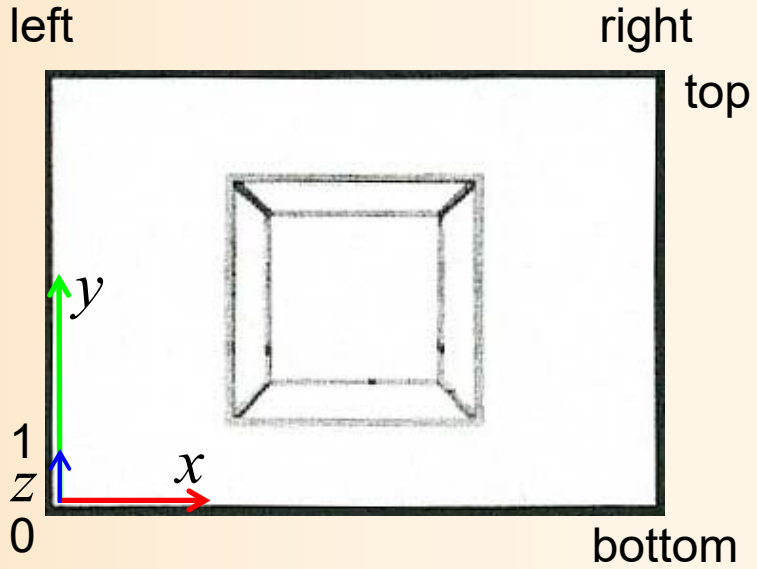
教科書 基礎知識 図2-28



透視射影変換

- 透視射影変換行列



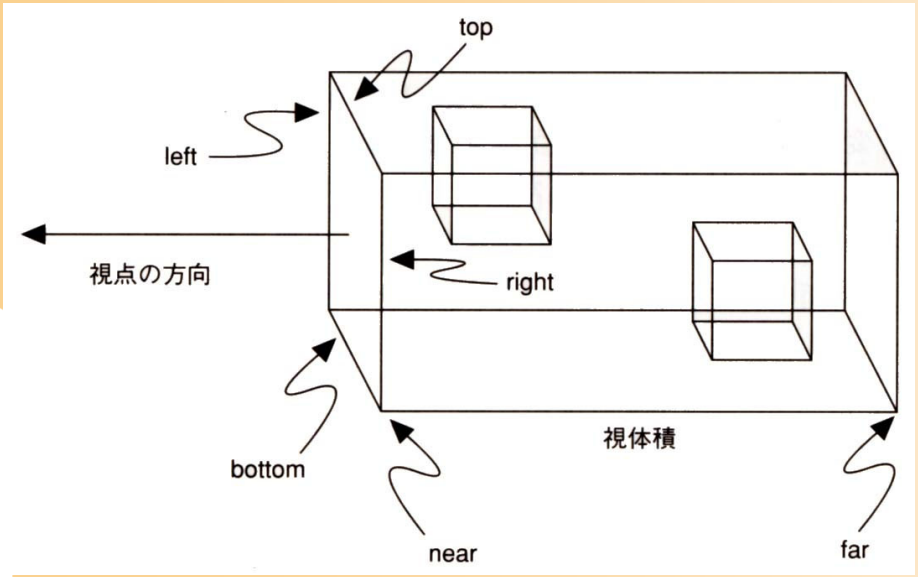


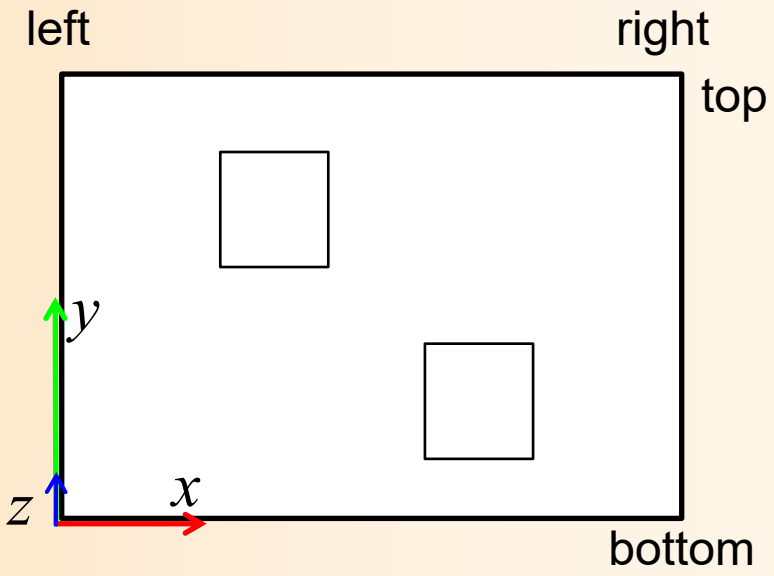
$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \quad \begin{pmatrix} x'/w' \\ y'/w' \\ z'/w' \end{pmatrix}$$

W'=－Z となり、Zで割ることになる
（Z値が大きくなるほど中央になる）

平行射影変換

- 平行射影変換
  - スクリーン対して平行に射影
  - 3面図などを描画するときに用いられる



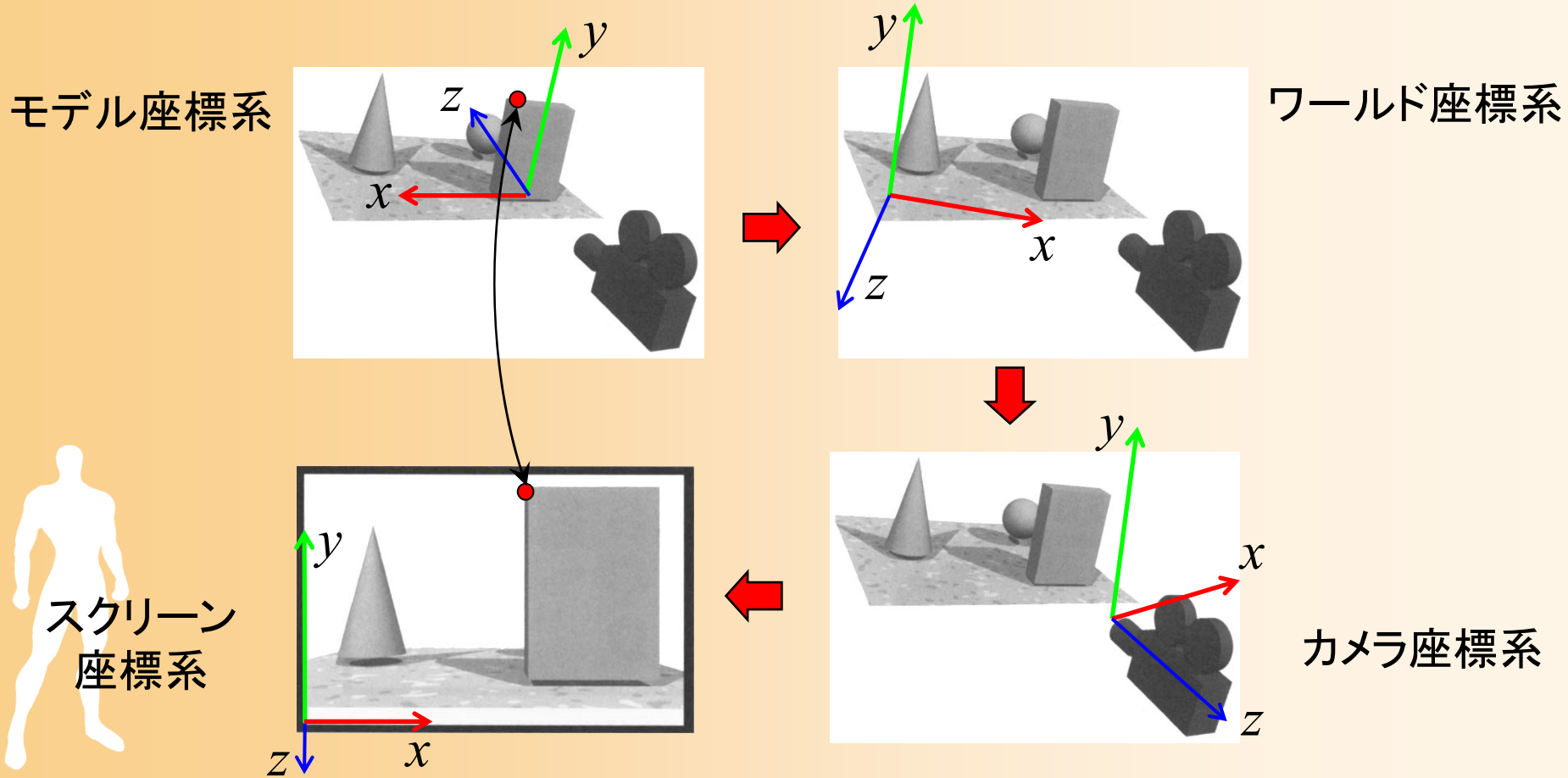


座標変換

- 座標変換の概要
- 座標系
- 視野変換
- 射影変換
- 座標変換のまとめ

座標変換の流れのまとめ

- モデル座標系からスクリーン座標系に変換



変換行列による座標変換の実現

- 視野変換＋射影変換
  - アフィン変換（視野変換）＋透視変換（射影変換）
  - 最終的なスクリーン座標は $(x'/w' \quad y/w' \quad z/w')$ となる

モデル座標系での頂点座標

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} R_{00}S_x & R_{01} & R_{02} & T_x \\ R_{10} & R_{11}S_y & R_{12} & T_y \\ R_{20} & R_{21} & R_{22}S_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

射影変換（カメラ→スクリーン）

視野変換（モデル→カメラ）

スクリーン座標系での頂点座標

座標変換の設定

- 自分のプログラムから OpenGLやDirectXに、2つの変換行列を設定する
  - ワールド座標からカメラ座標系への視野変換
    - カメラの位置・向きや、物体の位置向きに応じて、適切なアフィン変換行列を設定
    - さまざまな状況で、適切な変換行列を設定できるように、十分に理解しておく必要がある
  - カメラ座標系からスクリーン座標系への射影変換
    - 透視変換行列は、通常、固定なので、最初に一度だけ設定
    - 視野角やスクリーンサイズなどを適切に設定

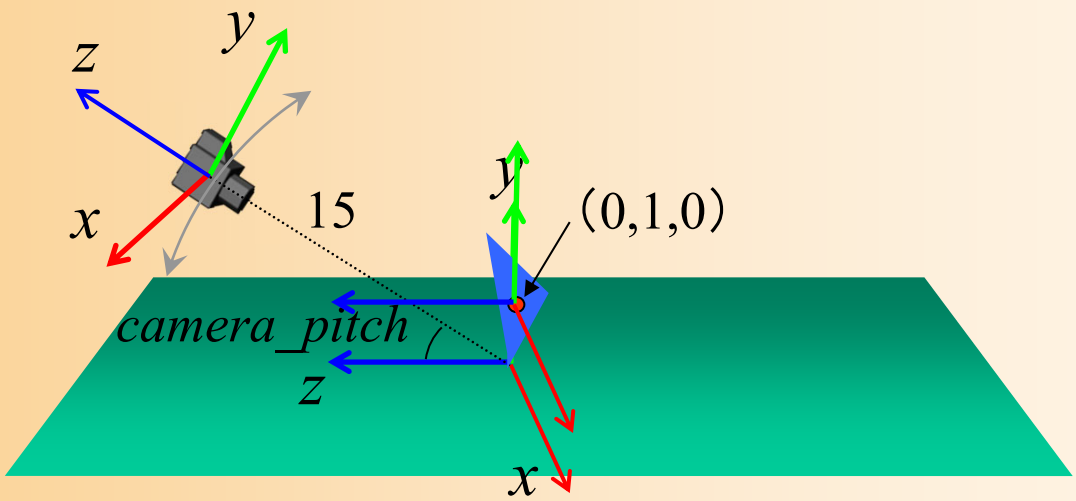
射影変換の設定（サンプルプログラム）

- ウィンドウサイズから変更された時に設定
  - 透視変換行列の設定（視野角を45度とする）
  - 各関数の詳細は、次回の演習で説明

```
void  reshape( int w, int h )
{
        ......
        // カメラ座標系→スクリーン座標系への変換行列を設定
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();
        gluPerspective( 45, (double)w/h, 1, 500 );
}
```

変換行列の設定（サンプルプログラム）

- サンプルプログラムでのカメラ位置の設定



- 以下の変換行列により表せる（詳細は後日説明）

ポリゴンを基準とする座標系での頂点座標

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

カメラから見た頂点座標（描画に使う頂点座標）

変換行列の設定（サンプルプログラム）

- 描画処理の中で設定
  - 各関数の詳細は、次回の演習で説明

```
// 変換行列を設定（ワールド座標系→カメラ座標系）
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( 0.0,  0.0, - 15.0 );
glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

// 地面を描画
・・・・・・

// 変換行列を設定（物体のモデル座標系→カメラ座標系）
glTranslatef( 0.0, 1.0, 0.0 );

// 物体（１枚のポリゴン）を描画
・・・・・・
```

今日の内容

- OpenGL＆GLUTの概要
- サンプルプログラムの概要
- 座標変換
- 変換行列の設定
- ポリゴンモデルの描画

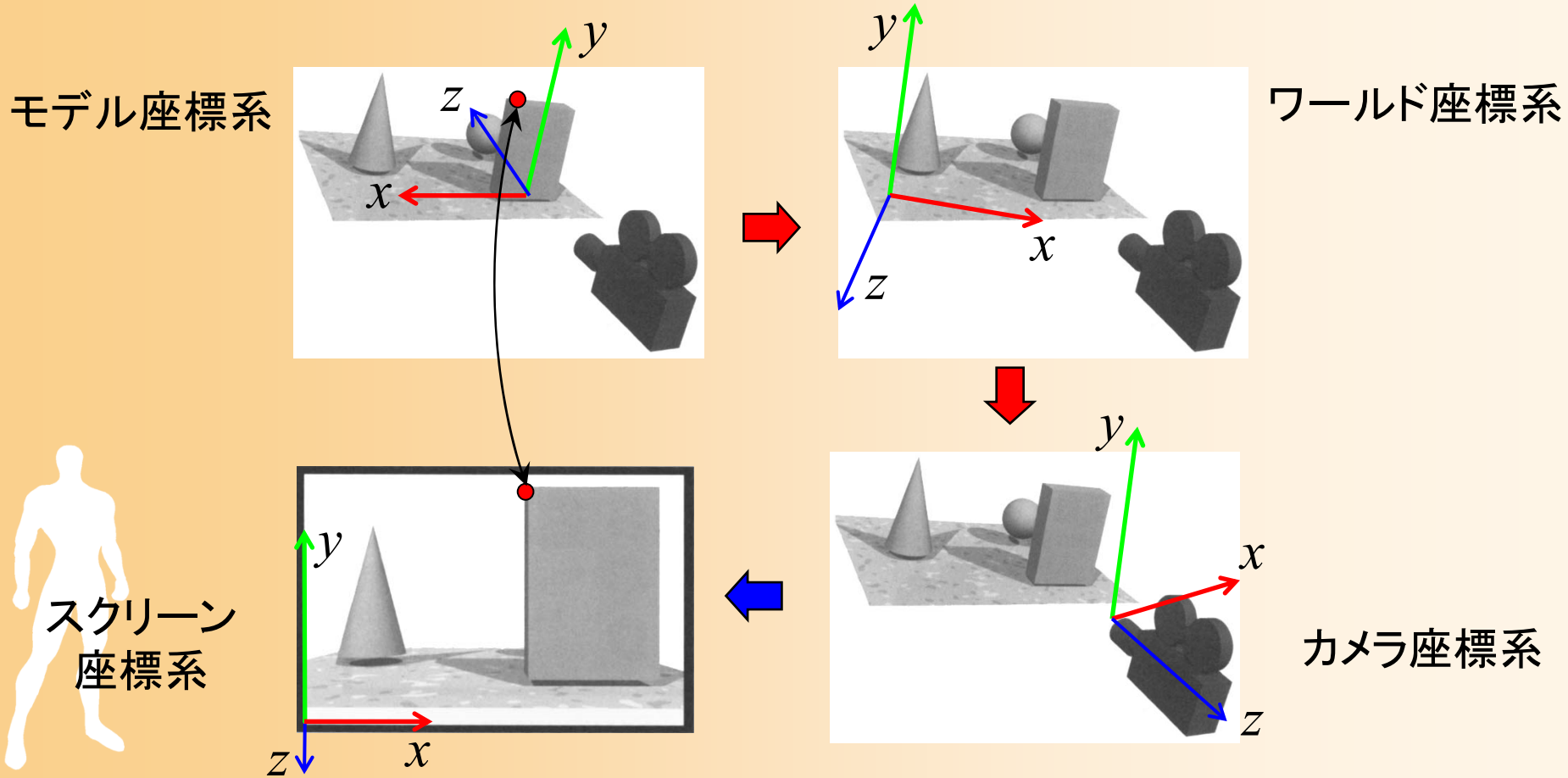
変換行列の設定

変換行列の設定

- OpenGLは、内部に変換行列を持っている
  - モデルビュー変換行列
  - 射影変換行列
    - 両者は別に扱った方が便利なので、別々に設定できるようになっている
- OpenGLの関数を呼び出すことで、これらの変換行列を変更できる

座標変換(復習)

- モデル座標系からスクリーン座標系への変換



変換行列による座標変換（復習）

- 視野変換＋射影変換
  - アフィン変換（視野変換）＋透視変換（射影変換）
  - 最終的なスクリーン座標は $(x'/w' \quad y/w' \quad z/w')$ となる

モデル座標系での頂点座標

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} R_{00}S_x & R_{01} & R_{02} & T_x \\ R_{10} & R_{11}S_y & R_{12} & T_y \\ R_{20} & R_{21} & R_{22}S_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

射影変換（カメラ→スクリーン）

視野変換（モデル→カメラ）

スクリーン座標系での頂点座標

座標変換の設定(復習)

- 自分のプログラムから OpenGLやDirectXに、2つの変換行列を設定する
  - ワールド座標からカメラ座標系への視野変換
    - カメラの位置・向きや、物体の位置向きに応じて、適切なアフィン変換行列を設定
    - さまざまな状況で、適切な変換行列を設定できるように、十分に理解しておく必要がある
  - カメラ座標系からスクリーン座標系への射影変換
    - 透視変換行列は、通常、固定なので、最初に一度だけ設定
    - 視野角やスクリーンサイズなどを適切に設定

変換行列の設定のための関数

- 設定を行う変換行列の指定
  - glMatrixMode()
    - どの変換行列を変更するのかを指定する
- 変換行列の設定
  - 主に視野変換の設定に使われる関数
    - glLoadIdentity()、glTranslate()、glRotate()、他
  - 射影変換行列の設定に使われる関数
    - gluPerspective(), glFrustrum(), glOrth()、他

変換行列の指定

- glMatrixMode(*mode*)
  - 設定する変換行列を指定する
  - GL_MODELVIE
    - モデルビュー変換(視野変換)
      (モデル座標系からカメラ座標系への変換)
  - GL_PROJECTION
    - 射影変換(投影変換)
      (カメラ座標系からスクリーン座標系への変換)

変換行列の設定のための関数

- 設定を行う変換行列の指定
  - glMatrixMode()
    - どの変換行列を変更するのかを指定する
- 変換行列の設定
  - 主に視野変換の設定に使われる関数
    - glLoadIdentity()、glTranslate()、glRotate()、他
  - 射影変換行列の設定に使われる関数
    - gluPerspective(), glFrustrum(), glOrth()、他

変換行列の変更

- glLoadIdentity()
  - 単位行列で初期化
- glTranslate(*x, y, z*)
  - 平行移動変換をかける
- glRotate(*angle, x, y, z*)
  - 指定した軸周りの回転変換をかける
  - angle は、1回転を360として指定

変換行列の変更

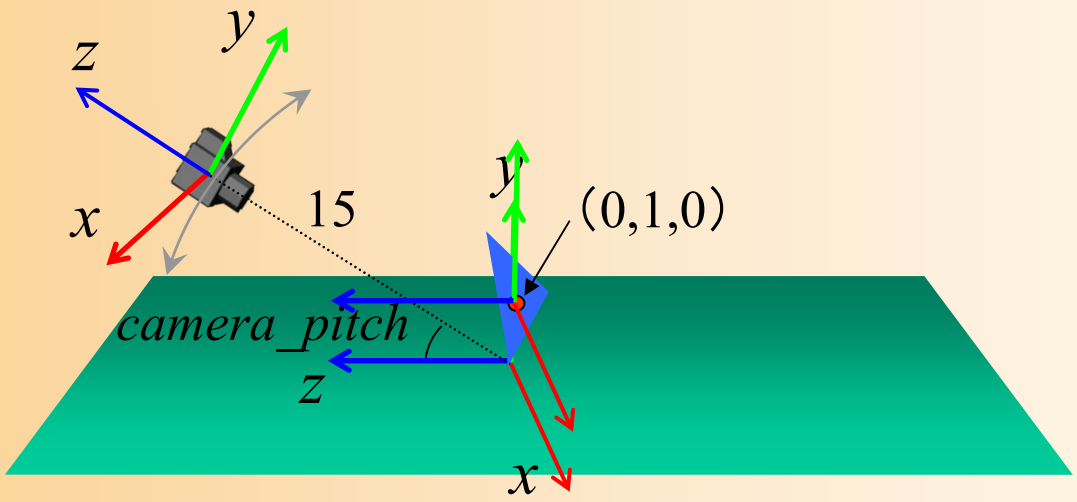
- 変換行列は順番に右側にかけられていく
  - プログラムで後から記述した変換行列の方が、実際には先に計算される

$$\begin{pmatrix} & \mathbf{A} & \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

$$\mathbf{A} = \mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\cdots\mathbf{A}_n$$

サンプルプログラムの視野変換行列

- サンプルプログラムのシーン設定
  - カメラと水平面の角度（仰角）は camera_ptich
  - カメラと中心の間の距離は 15
  - ポリゴンを（0,1,0）の位置に描画



サンプルプログラムの視野変換行列

- モデル座標系 → カメラ座標系 への変換行列

③ ②

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

① ワールド座標系→カメラ座標系　モデル座標系→ワールド座標系

  - x軸周りの回転
  - ２つの平行移動変換の位置に注意
    - 中心から15離れるということは、回転後の座標系でカメラを後方（z軸）に15下げることと同じ

サンプルプログラムの変換行列の設定

- 描画処理（display（）関数）

```
// 変換行列を設定（ワールド座標系→カメラ座標系）
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
③ glTranslatef( 0.0,  0.0, - 15.0 );
② glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

// 地面を描画（ワールド座標系で頂点位置を指定）
・・・・・・

// 変換行列を設定（モデル座標系→カメラ座標系）
① glTranslatef( 0.0, 1.0, 0.0 );

// ポリゴンを描画（モデル座標系で頂点位置を指定）
・・・・・・
```

以降、視野変換行列を変更することを指定

単位行列で初期化

平行移動行列・回転行列を順にかけることで、変換行列を設定

その他の変換行列の設定関数

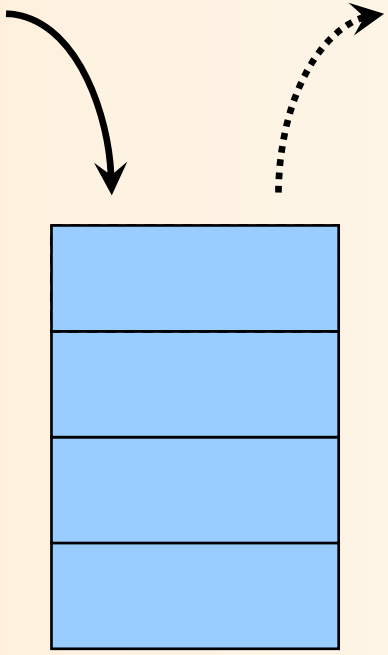
- glLookAt(カメラ位置, 目標位置, 上方ベクトル)
  - カメラと目標の位置で指定
  - 回転角度で向きを表す場合には向かない
- glLoadMatrix(配列), glMultMatrix(配列)
  - 配列を使って行列を直接設定 or かける
  - GL_double m[4][4];
    - m[i][j] が 行列の j行 i 列の要素を表す

※ やや特殊な設定方法なので、本講義の演習では、これらの関数は使用しない

変換行列の退避・復元

- 現在の変換行列を別の領域（スタック）に記録しておき、後から復元して利用できる
- glPushMatrix()
  - 現在の変換行列の退避
  - スタックに積む
- glPopMatrix()
  - 最後に退避した変換行列の回復
  - スタックから取り出す

※ 具体的な使用例は次回説明

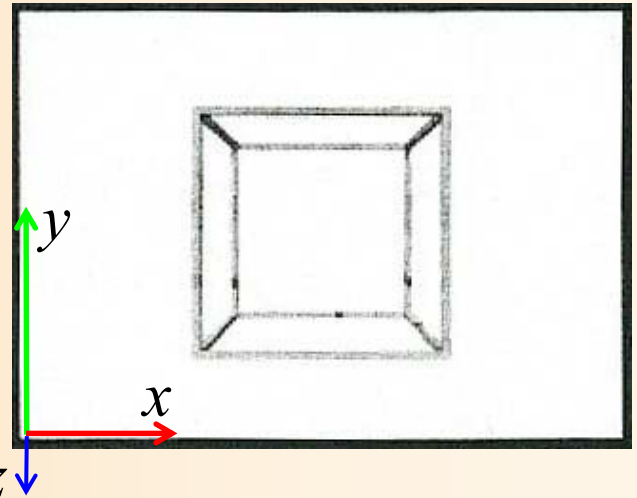


変換行列の設定のための関数

- 設定を行う変換行列の指定
  - glMatrixMode()
    - どの変換行列を変更するのかを指定する
- 変換行列の設定
  - 主に視野変換の設定に使われる関数
    - glLoadIdentity()、glTranslate()、glRotate()、他
  - 射影変換行列の設定に使われる関数
    - gluPerspective(), glFrustrum(), glOrth()、他

射影行列の変換

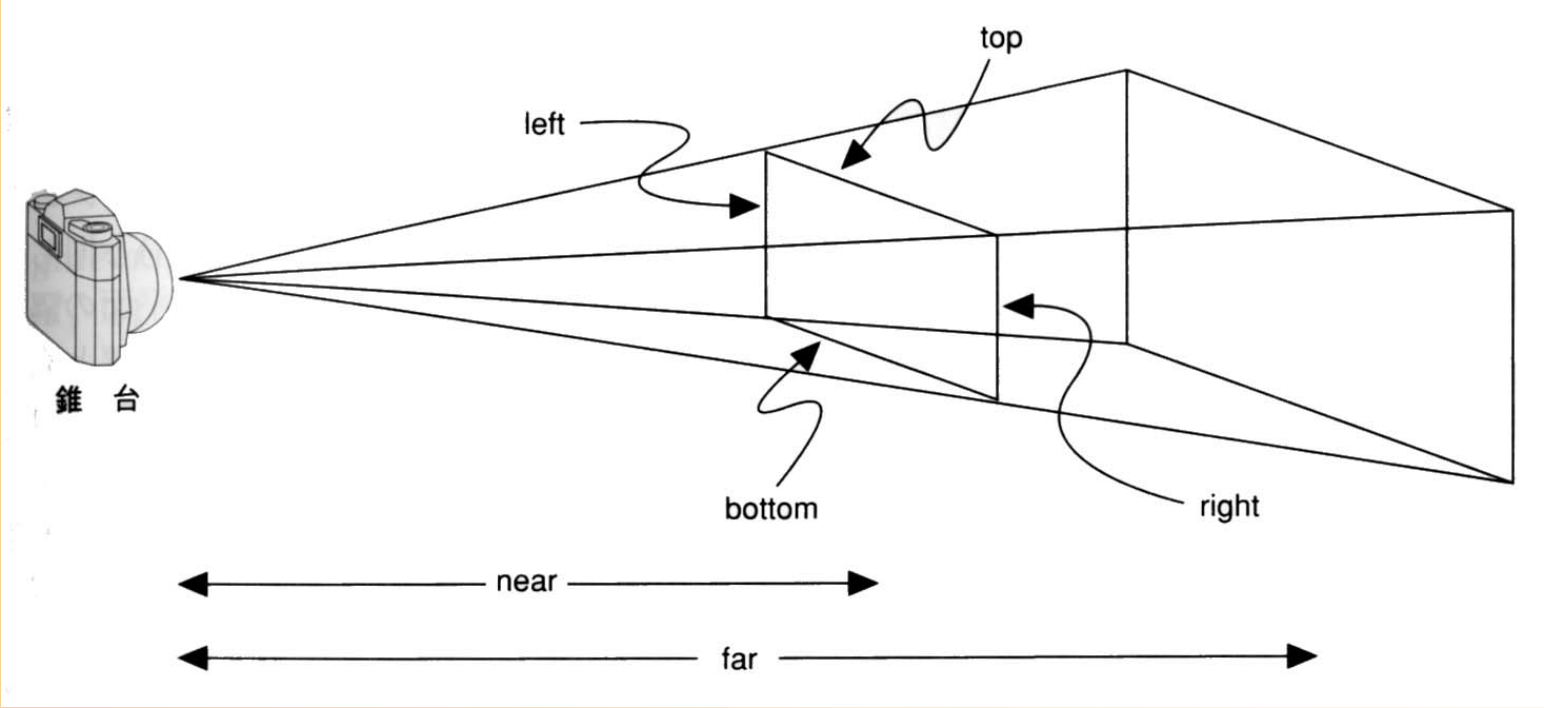
- 射影変換の種類
  - 透視射影
    - 現実の見え方をシミュレート
    - 遠くにあるものほど中央に寄って見える



  - 平行射影
    - 平行に射影
    - 図面などを描画する時に使用

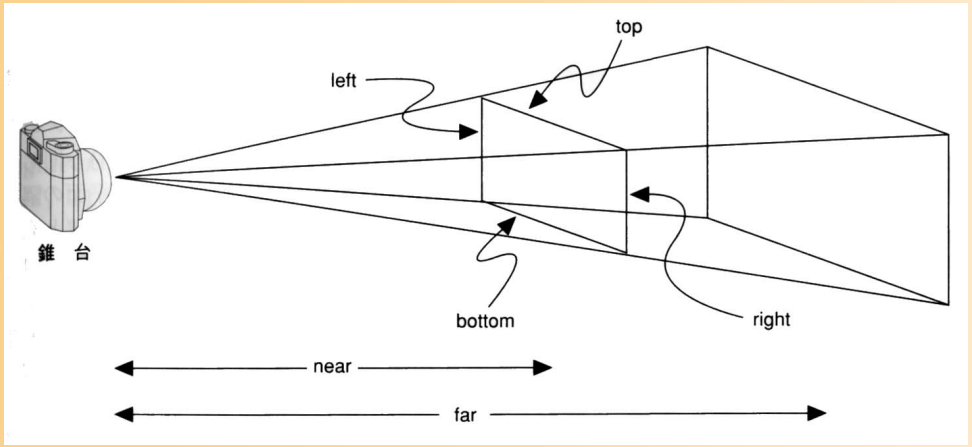
透視射影変換

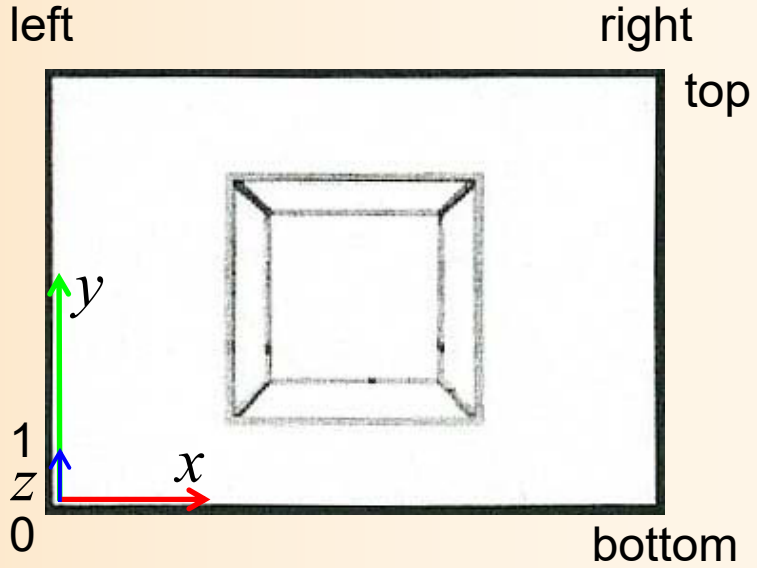
- glFrustum（手前面の大きさ, 手前面の距離, 奥面の距離）



透視変換(復習)

- 透視変換行列



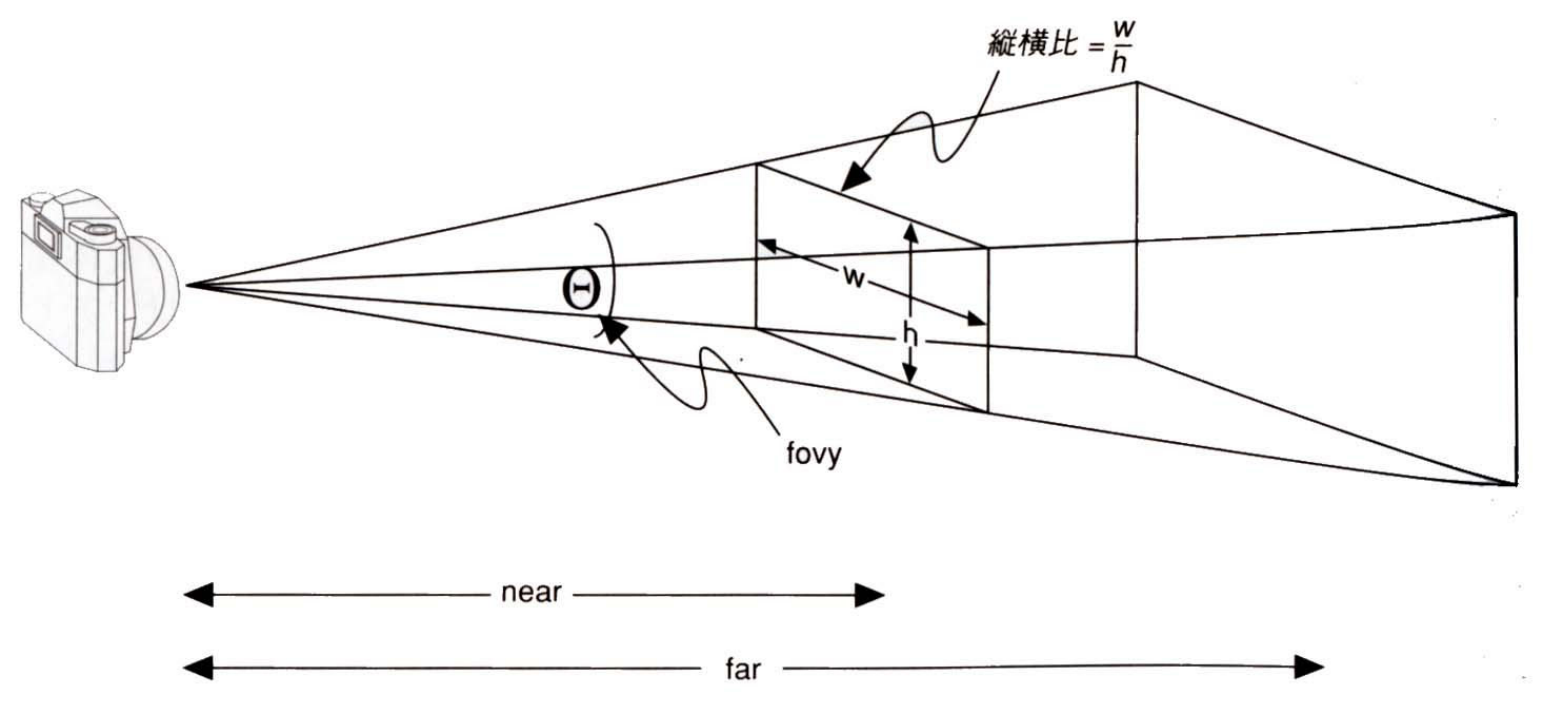


$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \quad \begin{pmatrix} x'/w' \\ y'/w' \\ z'/w' \end{pmatrix}$$

W'=−Z となり、Zで割ることになる
(Z値が大きくなるほど中央になる)

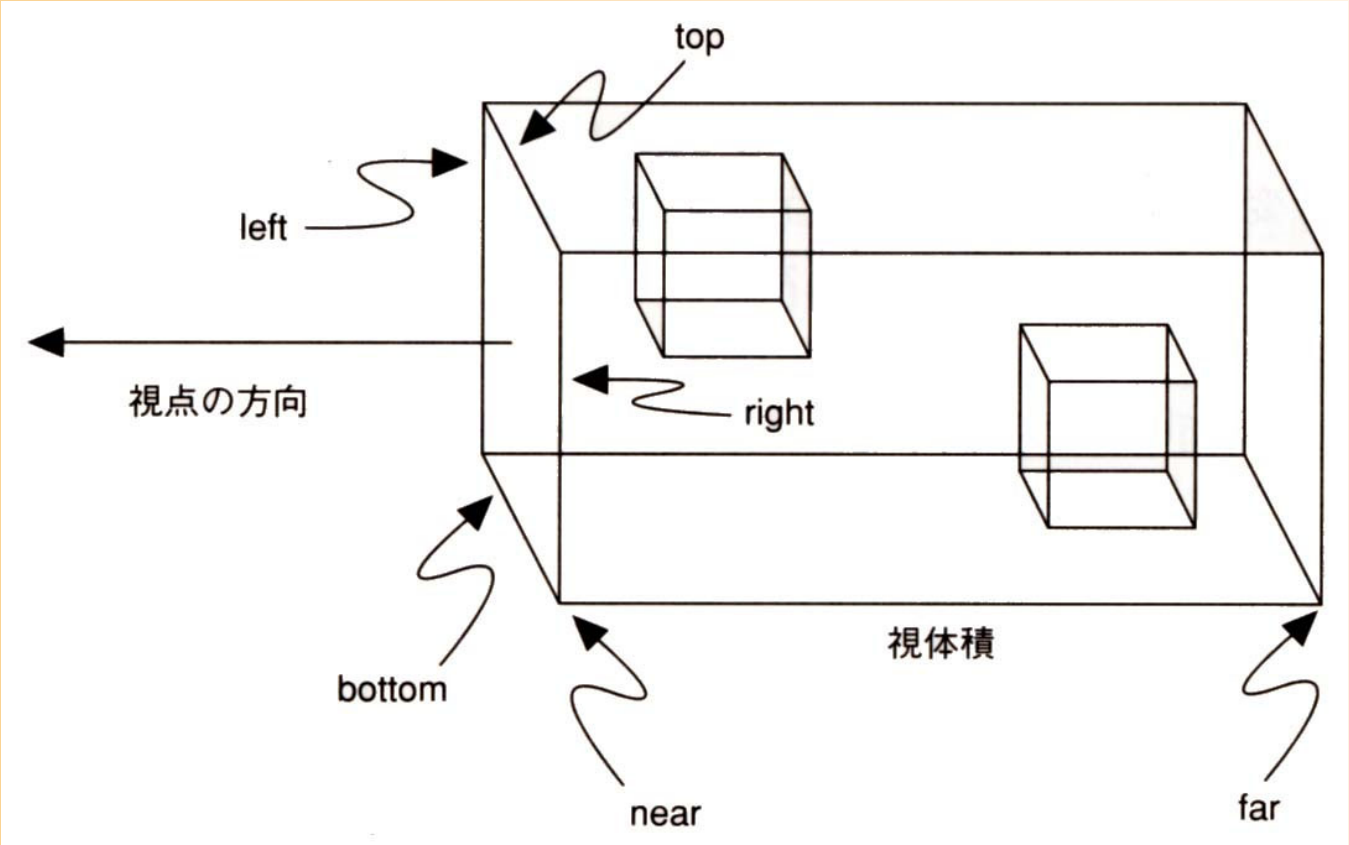
透視射影変換

- gluPerspective（視野角, 手前面の距離, 奥面の距離）
  - 視界領域が左右対称であるという前提で、より少ない引数で透視射影変換を設定する関数



平行射影変換

- glOrtho（描画範囲, 手前面の距離, 奥面の距離）



射影変換の設定（サンプルプログラム）

- ウィンドウサイズから変更された時に設定
  - 透視変換行列の設定（視野角を45度とする）

```
void  reshape( int w, int h )
{
        // ウィンドウ内の描画を行う範囲を設定（ウィンドウ全体に描画）
        glViewport(0, 0, w, h);

        // カメラ座標系→スクリーン座標系への変換行列を設
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();
        gluPerspective( 45, (double)w/h, 1, 500 );
}
```

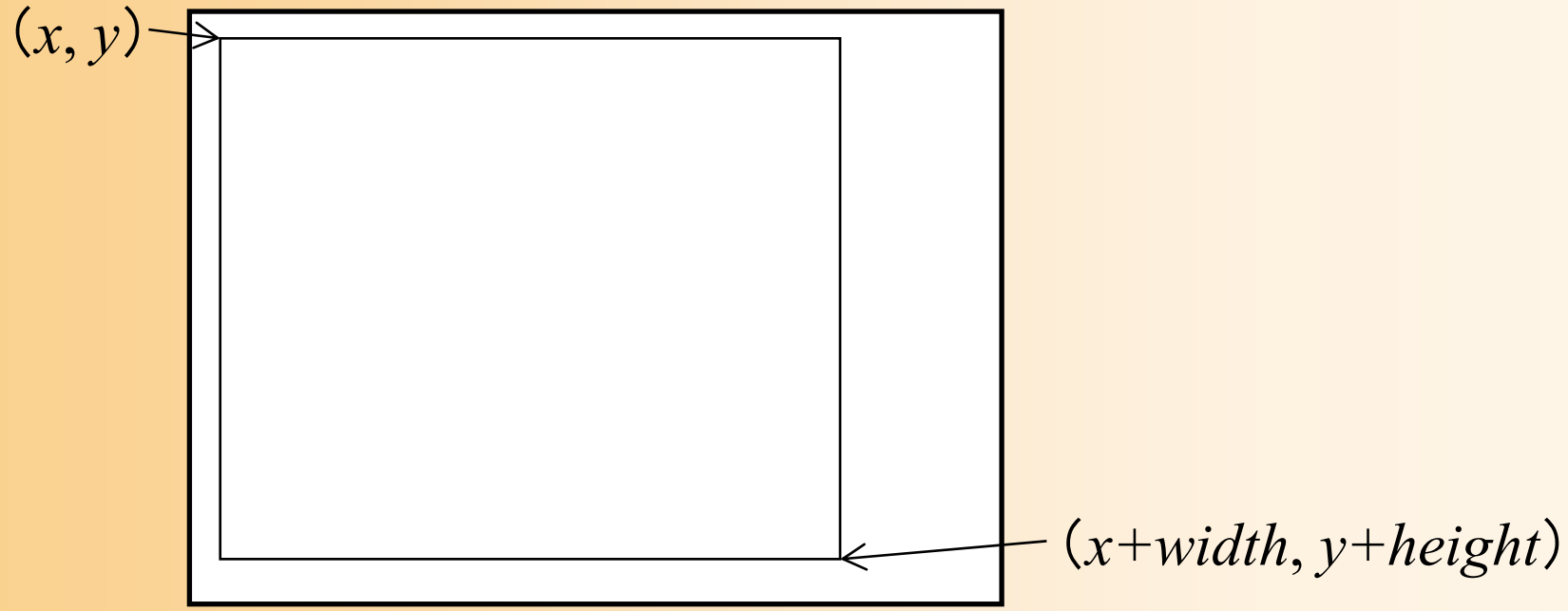
以降、射影変換行列を変更することを指定

単位行列で初期化

透視変換を設定

ビューポートの設定

- glViewport(*x*, *y*, *widht*, *height*)
  - ウィンドウ内のどの範囲に描画を行うかを設定



ポリゴンの描画

ポリゴンの描画

- glBegin() ～ glEnd() を使用

```
glBegin( プリミティブの種類 );
        この間にプリミティブを構成する頂点データを指定
glEnd();
```

※ 頂点データの指定は、一つの関数で、ポリゴンを構成するデータの一つのみを指定するようになっている

- プリミティブの種類
  - GL_POINTS（点）、GL_LINES（線分）、GL_TRIANGLES（三角面）、GL_QUADS（四角面）、GL_POLYGON（ポリゴン）、他

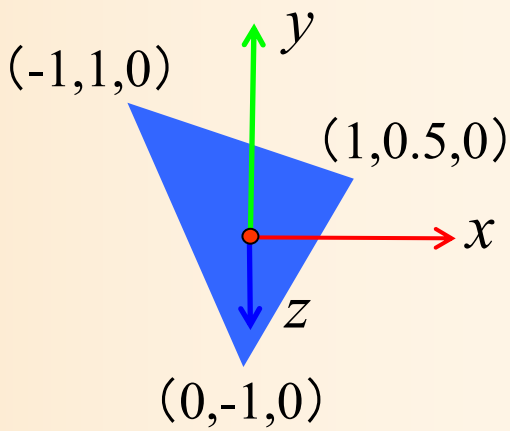
頂点データの指定

- glColor3f(r, g, b)
  - これ以降の頂点の色を設定
- glNormal3f(nx, ny, nz)
  - これ以降の頂点の法線を設定
- glVertex3f(x, y, z)
  - 頂点座標を指定
  - 色・法線は、最後に指定したものが使用される

ポリゴンの描画の例(1)

- 1枚の三角形を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - ポリゴンを基準とする座標系(モデル座標系)で頂点位置・法線を指定

```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );
        glVertex3f( 0.0,-1.0, 0.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```

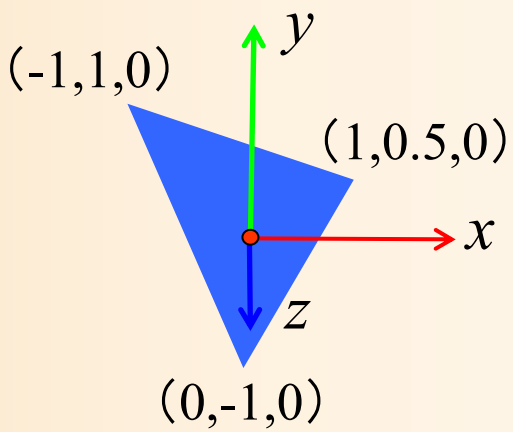


GL_TRIANGLES が指定されているので、3つの頂点をもとに、1枚の三角面を描画 (6つの頂点が指定されたら、2枚描画)

ポリゴンの描画の例(1)

- 頂点の色・法線は、頂点ごとに指定可能
  - 指定しなければ、最後に指定したものが使われる

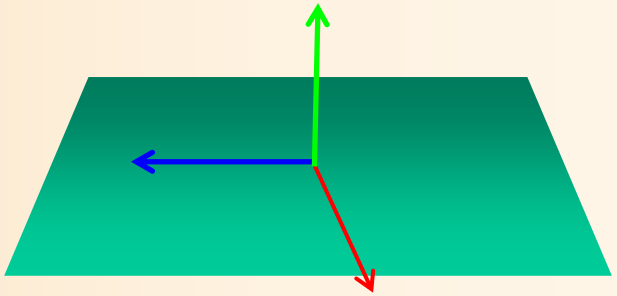
```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );

        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f( 0.0,-1.0, 0.0 );

        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```



ポリゴンの描画の例(2)

- 1枚の四角形として地面を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - 真上(0,1,0)を向き、水平方向の長さ10の四角形

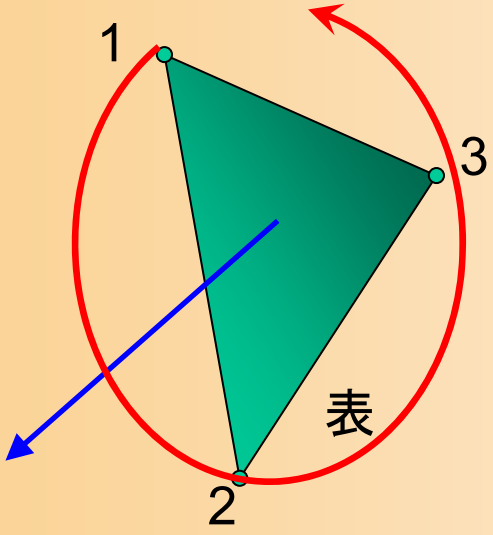
```
// 地面を描画
glBegin( GL_POLYGON );
        glNormal3f( 0.0, 1.0, 0.0 );
        glColor3f( 0.5, 0.8, 0.5 );

        glVertex3f( 5.0, 0.0, 5.0 );
        glVertex3f( 5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0, 5.0 );
glEnd();
```

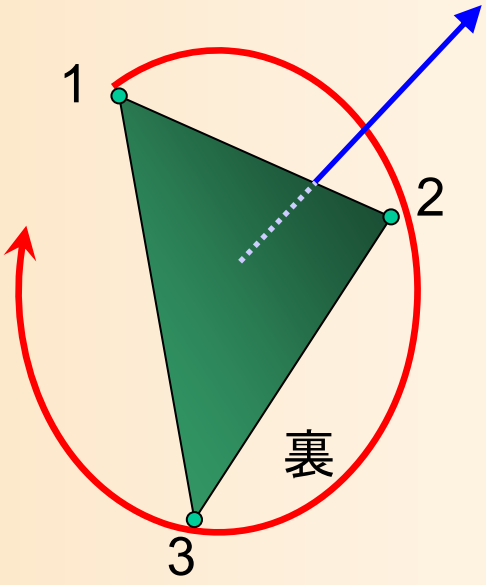


GL_POLYGON が指定されているので、n 個の頂点をもとに、n 角面を描画
(1度に1枚しか描画できない)

ポリゴンの向き

- 頂点の順番により、ポリゴンの向きを決定
  - 表から見て反時計回りの順序で頂点を与える
  - 視点と反対の向きでなら描画しない（背面除去）
    - 頂点の順序を間違えると、描画されないので、注意



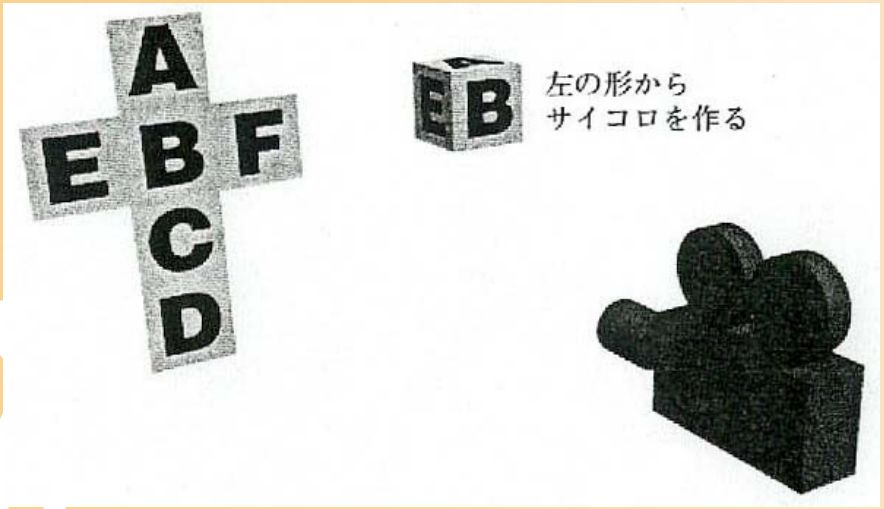


背面消去(復習)

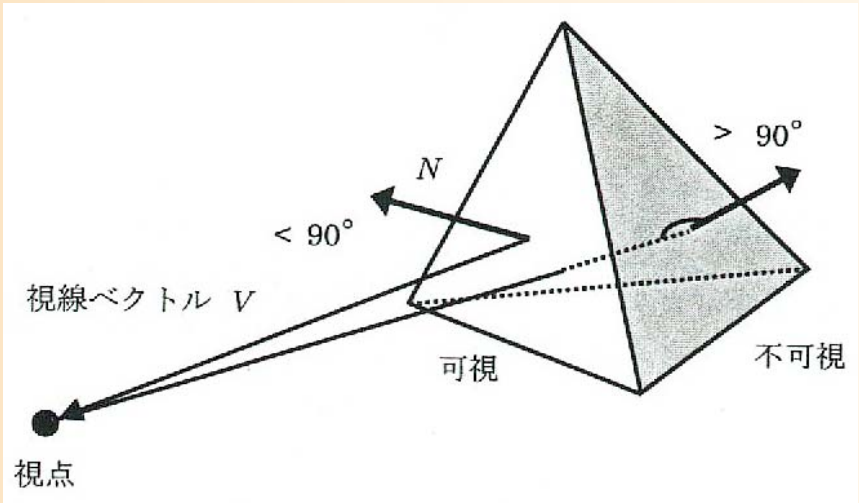
- 背面消去(後面消去、背面除去、後面除去)
  - バックフェースカリング、とも呼ぶ
- 後ろ向きの面の描画を省略する処理
- サーフェスモデルであれば、後ろ向きの面は描画は不要である点に注目する
  - 仮に描画したとしても、その後、手前側にある面で上書きされる
  - 裏向きの面の描画を省略することで処理を高速化できる(単純に考えると、約半分に減らせる)

背面消去(復習)

- 後ろ向きの面の判定方法
  - 視線ベクトル(カメラから面へのベクトル)と面の法線ベクトルの内積により判定



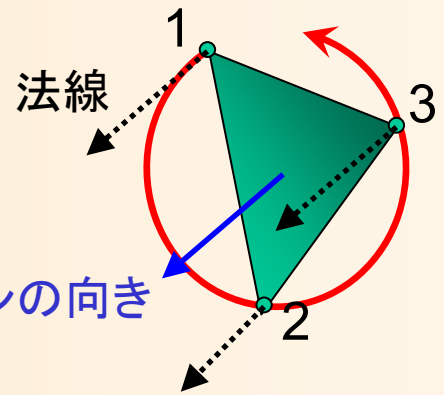
教科書 基礎知識 図2-22



教科書 基礎と応用 図3.5

法線とポリゴンの向き

- OpenGLでは、法線とポリゴンの向きは、独立の扱い（要注意）
- 法線
  - 頂点ごとに、関数（glNormal3f()）により指定
  - 光のモデルにより色を計算するために使用（詳細は後日の講義で説明）
- ポリゴンの向き
  - ポリゴンを描画するとき、頂点の順序により指定
  - 背面除去の判定に使用



基本オブジェクトの描画

基本オブジェクトの描画

- GLUTには、基本的なポリゴンモデルを描画する関数が用意されている
  - 立方体、球、円すい、16面体、円環体、ティーポット、等
  - あらかじめ用意されたポリゴンモデルを描画
- 例：glWireCube（size）, glSolidCube（size）
  - それぞれ、ワイヤーフレームとポリゴンモデルで立方体を描画

立方体の描画

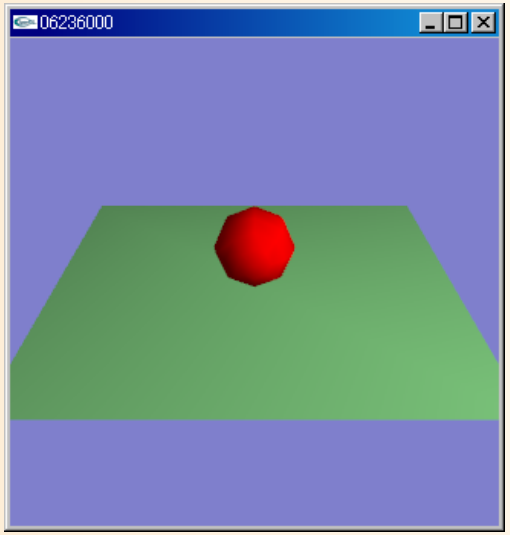
- glutSolidCube（）を使って立方体を描画

```
void  display( void )
{
        ・・・・・・
        // 変換行列を設定（物体のモデル座標系→カメラ座標系）
        //（物体が (0.0, 1.0, 0.0) の位置にあり、静止しているとする）
        glTranslatef( 0.0, 1.0, 0.0 );
/*
        ポリゴンの描画はコメントアウト
*/
        // 立方体を描画
        glColor3f( 1.0, 0.0, 0.0 );
        glutSolidCube( 1.5f );
        ・・・・・・
}
```

球の描画

- glutSolidSphere（ *radius*, *slices*, *stacks* ）
  - 球をポリゴンモデルで近似して描画
  - どれだけ細かいポリゴンで近似するかを、引数 *slices*, *stacks* で指定可能
    - 引数の値を変えて、ポリゴンモデルの変化を確認

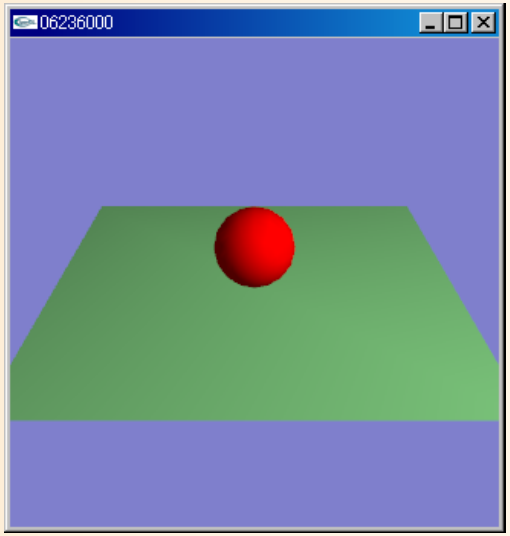
```
void  display( void )
{
        ......
        // 球を描画
        glColor3f( 1.0, 0.0, 0.0 );
        glutSolidSphere( 1.0, 8, 8 );
        ......
}
```

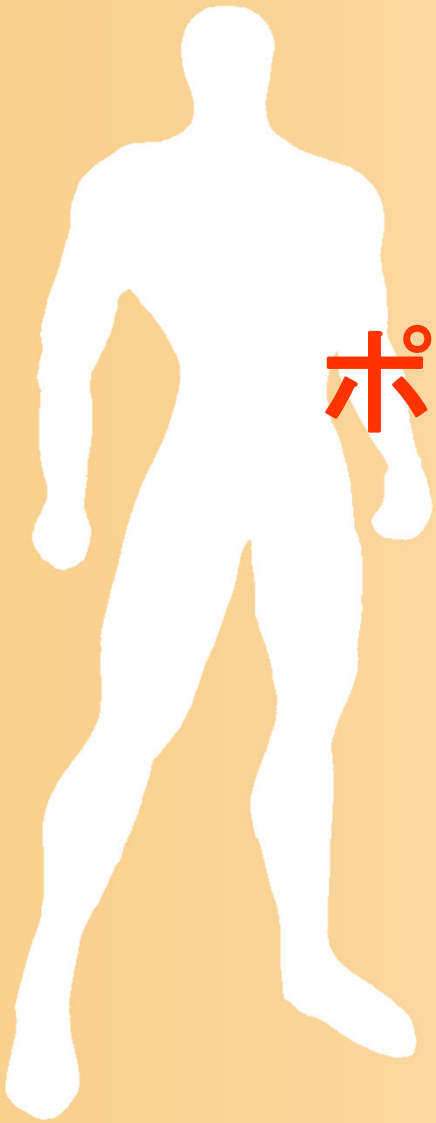


球の描画

- glutSolidSphere（ *radius*, *slices*, *stacks* ）
  - 球をポリゴンモデルで近似して描画
  - どれだけ細かいポリゴンで近似するかを、引数 *slices*, *stacks* で指定可能
    - 引数の値を変えて、ポリゴンモデルの変化を確認

```
void  display( void )
{
        ......
        // 球を描画
        glColor3f( 1.0, 0.0, 0.0 );
        glutSolidSphere( 1.0, 16, 16 );
        ......
}
```

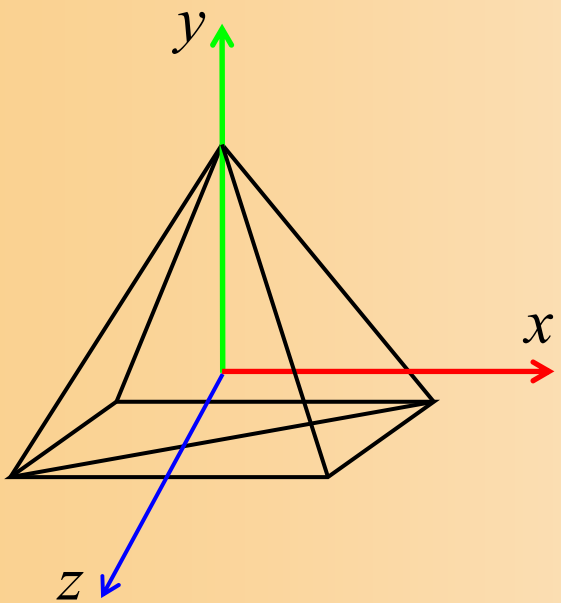




ポリゴンモデルの描画

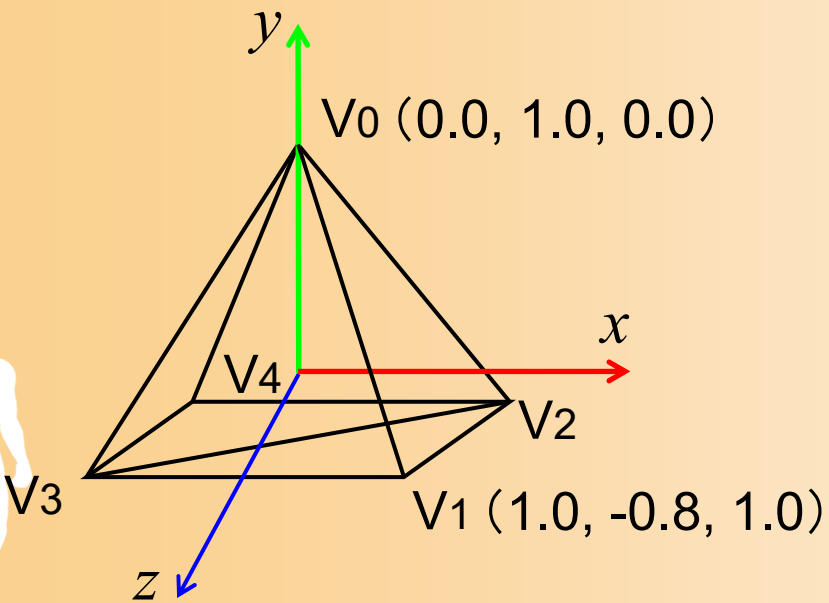
四角すいの描画

- 三角面の集まりとして描画
  - 5個の頂点と6枚の三角面により構成
  - 底面（四角形）は2枚の三角形に分割して表す



四角すいの描画

- 四角すいを構成する頂点と三角面
  - 頂点座標
  - 三角面の頂点、面の法線



頂点座標

V0 (0.0, 1.0, 0.0)

V1 (1.0, -0.8, 1.0)

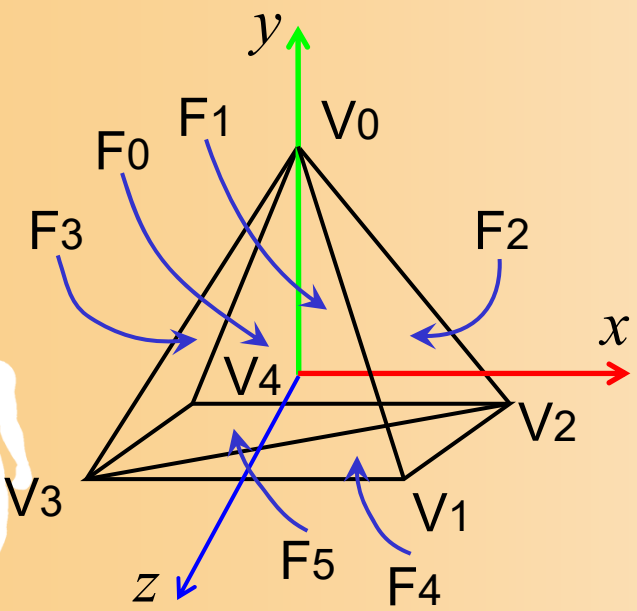
V2 (1.0, -0.8, -1.0)

V3 (-1.0, -0.8, 1.0)

V4 (-1.0, -0.8, -1.0)

四角すいの描画

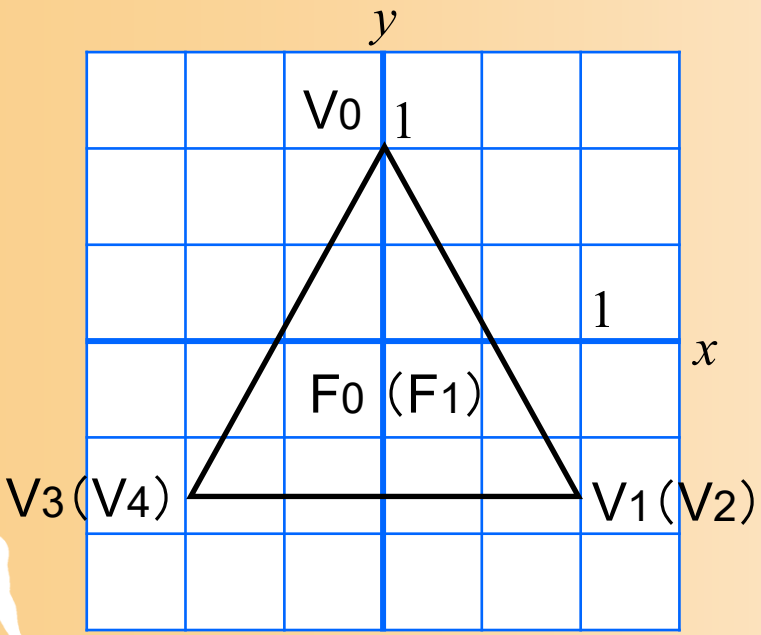
- 四角すいを構成する頂点と三角面
  - 頂点座標
  - 三角面の頂点、面の法線

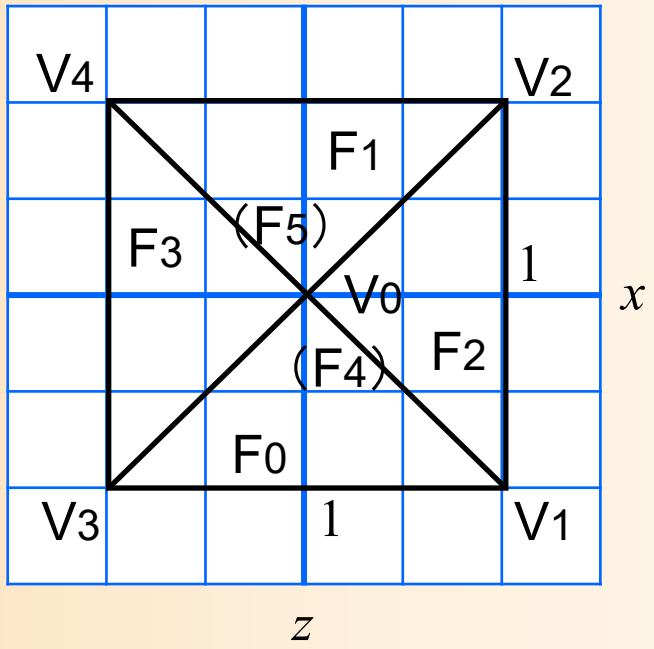


三角面	法線
F0 { V0, V3, V1 }	{ 0.0, 0.53, 0.85 }
F1 { V0, V2, V4 }	{ 0.0, 0.53, -0.85 }
F2 { V0, V1, V2 }	{ 0.85, 0.53, 0.0 }
F3 { V0, V4, V3 }	{ -0.85, 0.53, 0.0 }
F4 { V1, V3, V2 }	{ 0.0, -1.0, 0.0 }
F5 { V4, V2, V3 }	{ 0.0, -1.0, 0.0 }

三面図

- xy平面、xz平面

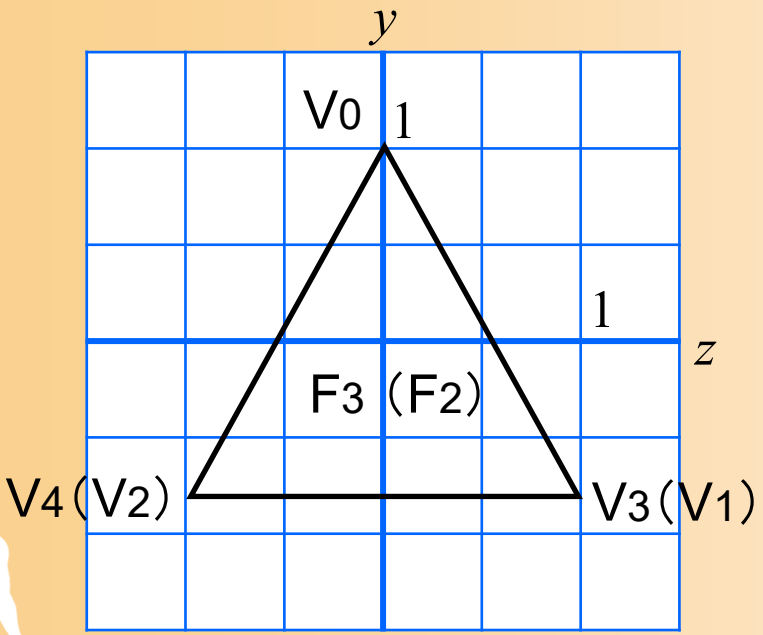




※ 括弧付きの頂点・面は裏側の頂点・面を表す

三面図

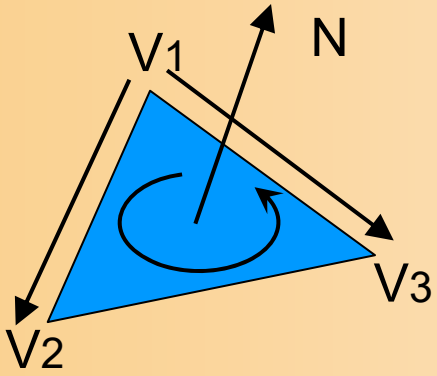
- yz平面



※ 括弧付きの頂点・面は裏側の頂点・面を表す

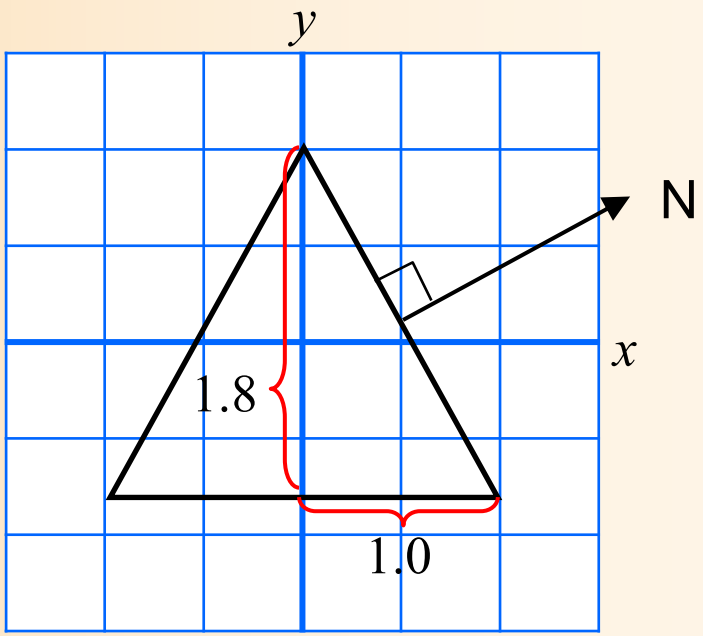
面の法線の計算方法

- ポリゴンの2辺の外積から計算できる



$$N = (V_3 - V_1) \times (V_2 - V_1)$$

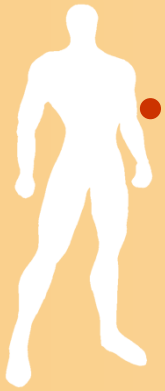
長さが 1 になるように正規化



- 四角すいの場合、
  断面で考えれば、
  より簡単に求まる

ポリゴンモデルの描画方法

- いくつかの描画方法がある
  - プログラムからOpenGLに頂点データを与える方法として、いろいろなやり方がある
- 形状データの表現方法の違い
  - 頂点データのみを使う方法と、頂点データ＋面インデックスデータを使う方法がある
  - 後者の方が、データをコンパクトにできる
- OpenGLへのデータの渡し方の違い
  - OpenGLの頂点配列の機能を使うことで、より高速に描画できる



ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ(直接記述)、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ(直接記述)、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す

方法1 最も基本的な描画方法

- サンプルプログラムと同様の描画方法
  - glVertex() 関数の引数に直接頂点座標を記述
  - ポリゴン数×各ポリゴンの頂点数の数だけ glVertex()関数を呼び出す

四角すいの描画(1)

- 四角すいを描画する新たな関数を追加

```
void  renderPyramid1()
{
        glBegin( GL_TRIANGLES );
                // +Z方向の面
                glNormal3f( 0.0, 0.53, 0.85 );
                glVertex3f( 0.0, 1.0, 0.0 );
                glVertex3f(-1.0,-0.8, 1.0 );
                glVertex3f( 1.0,-0.8, 1.0 );

                ......
                以下、残りの5枚分のデータを記述
                ......
        glEnd();
}
```

四角すいの描画(2)

- 描画関数から四角すいの描画関数を呼び出し
  - 修正の場所を間違えないように注意
  - renderPyramid()関数では色は指定されていないので、呼び出す前に色を設定している

```
void  display( void )
{
        ......
        // 角すいの描画
        glColor3f( 1.0, 0.0, 0.0 );
        renderPyramid1();
        ......
}
```

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ(直接記述)、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す

方法2 頂点データの配列を使用

- 配列を使う方法
  - 頂点データを配列として定義しておく
  - glVertex() 関数の引数として配列データを順番に与える
- 利点
  - モデルデータが配列になってるので扱いやすい

頂点データの配列を使用(1)

- 配列データの定義

```
// 全頂点数
const int  num_full_vertices = 18;

// 全頂点の頂点座標
static float  pyramid_full_vertices[][ 3 ] = {
    { 0.0, 1.0, 0.0 }, {-1.0,-0.8, 1.0 }, { 1.0,-0.8, 1.0 },
    ・・・・
    {-1.0,-0.8,-1.0 }, { 1.0,-0.8,-1.0 }, {-1.0,-0.8, 1.0 } };

// 全頂点の法線ベクトル
static float  pyramid_full_normals[][ 3 ] = {
    { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 },
    ・・・・
    { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 } };
```

頂点データの配列を使用(2)

- 各頂点の配列データを呼び出す

```
void  renderPyramid2()
{
        int  i;
        glBegin( GL_TRIANGLES );
        for ( i=0; i<num_full_vertices; i++ )
        {
                glNormal3f( pyramid_full_normals[i][0],
                            pyramid_full_normals[i][1],
                            pyramid_full_normals[i][2] );
                glVertex3f( pyramid_full_vertices[i][0],
                            pyramid_full_vertices[i][1],
                            pyramid_full_vertices[i][2] );
        }
        glEnd();
}
```

各頂点ごとに繰り返し

法線・頂点を指定（i番目の頂点のデータを指定）

頂点データの配列を使用(3)

- 描画関数から描画関数を呼び出し
  - 新しく追加した方の関数を使って描画するように修正
  - 実行結果の画像は変化しないことを確認

```
void  display( void )
{
        ......
        // 角すいの描画
        glColor3f( 1.0, 0.0, 0.0 );
//      renderPyramid1();
        renderPyramid2();
        ......
}
```

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ(直接記述)、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す

ここまでの方法の問題点

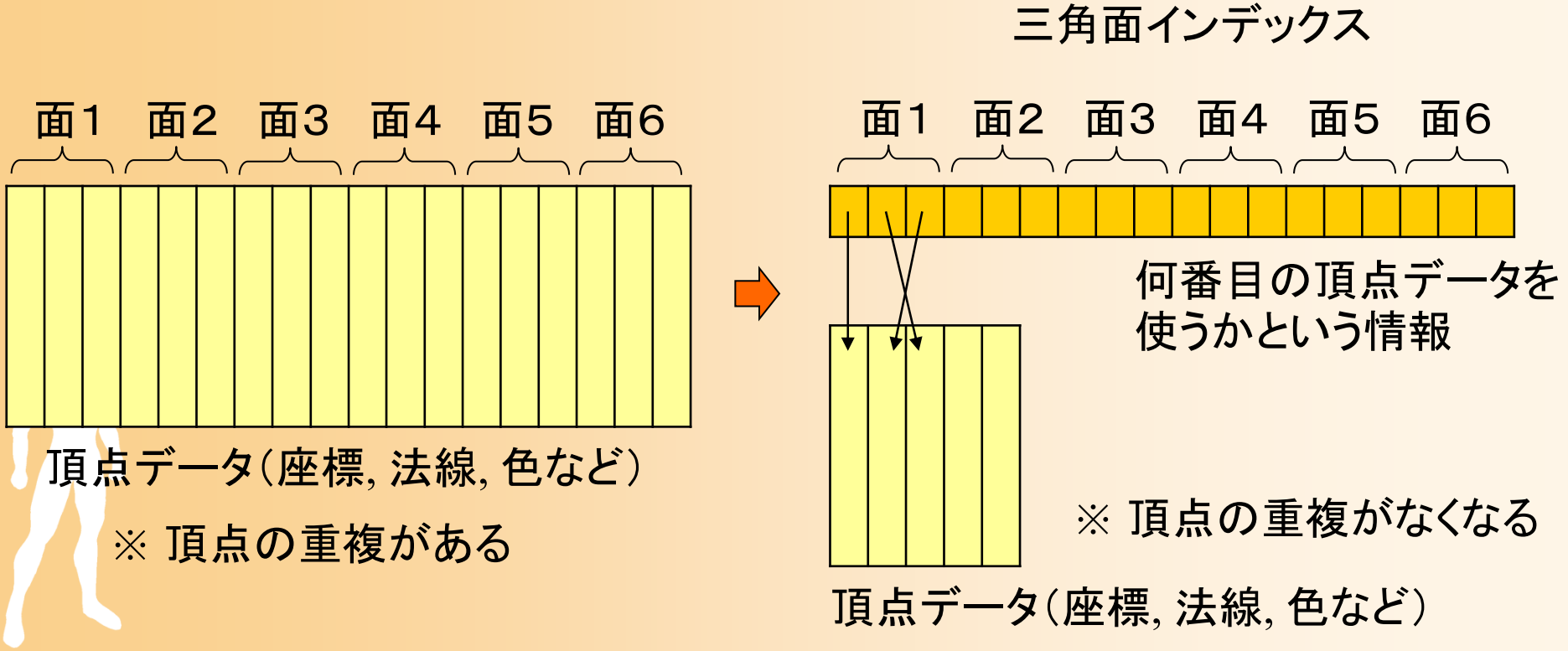
- 別の問題点
  - ある頂点を複数のポリゴンが共有している時、各ポリゴンごとに同じ頂点のデータを何度も記述する必要がある
    - 例：四角すいの頂点数は5個だが、これまでの方法では、三角面数6×3個＝18個の頂点を記述する必要がある
  - OpenGLは、与えられた全ての頂点に座標変換などの処理を適用するので、同じモデルを描画する時でも、なるべく頂点数が少ない方が高速

方法3 三角面インデックスを使用

- 頂点とポリゴンの情報を別々の配列に格納
  - 頂点データの数を最小限にできる
- 描画関数では、配列のデータを順に参照しながら描画
- 必要な配列（サンプルプログラムの例）
  - 頂点座標（x,y,z）×頂点数
  - 三角面を構成する頂点番号（v0,v1,v2）×三角面数
  - 三角面の法線（x,y,z）×三角面数

三角面インデックス

- 頂点データの配列と、三角面インデックスの配列に分けて管理する



配列を使った四角すいの描画(1)

- 配列データの定義

```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 }, ・・・・・・
};

// 三角面インデックス(各三角面を構成する頂点の頂点番号)の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};

// 三角面の法線ベクトルの配列(三角面を構成する頂点座標から計算)
float  pyramid_tri_normals[ num_pyramid_triangles ][ 3 ] = {
          { 0.00, 0.53, 0.85 }, // +Z方向の面
          ・・・・・・
```

配列を使った四角すいの描画(2)

- 配列データを参照しながら三角面を描画

```
void  renderPyramid3()
{
    int  i, j, v_no;
    glBegin( GL_TRIANGLES );
    for ( i=0; i<num_pyramid_triangles; i++ )
    {
        glNormal3f( pyramid_tri_normals[i][0], [i][1], [i][2] );
        for ( j=0; j<3; j++ )
        {
            v_no = pyramid_tri_index[ i ][ j ];
            glVertex3f( pyramid_vertices[ v_no ][0], [ v_no ][1], 
        }
    }
    glEnd();
}
```

各三角面ごとに繰り返し

三角面の各頂点ごとに繰り返し

面の法線を指定（i番目の面のデータを指定）

頂点番号を取得（i番目の面のj番目の頂点が、何番目の頂点を使うかを取得）

頂点座標を指定（v_no番目の頂点のデータを指定）

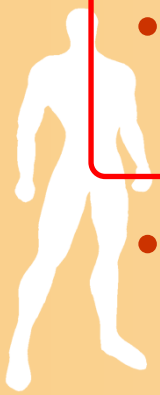
配列を使った四角すいの描画(3)

- 描画関数から描画関数を呼び出し
  - 新しく追加した方の関数を使って描画するように修正
  - 実行結果の画像は変化しないことを確認

```
void  display( void )
{
        ......
        // 角すいの描画
        glColor3f( 1.0, 0.0, 0.0 );
//      ......
        renderPyramid3();
        ......
}
```

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ(直接記述)、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す



ここまでの方法の問題点

- 問題点
  - 頂点ごとに glVertex(), glNormal() 関数を呼び出す必要がある
  - 一般に関数呼び出しにはオーバーヘッドがかかるので、なるべく関数呼び出しの回数は少なくしたい
- OpenGLの頂点配列の機能を使えば、この問題を解決できる

頂点配列を使った描画方法

- 頂点配列
  - 配列データを一度に全部 OpenGL に渡して描画を行う機能
  - 頂点ごとに OpenGL の関数を呼び出して、個別にデータを渡す必要がなくなる
    - 処理を高速化できる
    - 渡すデータの量は同じでも、頂点配列を利用することで、処理を高速化できる

方法4 頂点配列を使った描画方法

- 頂点配列・・・配列データを一度に全部 OpenGL に渡して描画する機能
- 頂点配列を使った描画の手順
  1. 頂点の座標・法線などの配列データを用意
  2. OpenGLに配列データを指定（配列の先頭アドレス）
     - glVertexPointer()関数、glNormalPointer()関数、等
  3. どの配列データを使用するかを設定
     - 頂点の座標、色、法線ベクトル、テクスチャ座標など
     - glEnableClientState()関数
  4. 配列の使用する範囲を指定して一気に描画
     - 配列データをレンダリング・パイプラインに転送
     - glDrawArrays()関数

頂点配列を使用した描画(1)

- 配列データの定義(従来の描画方法と同じ)

```
// 全頂点数
const int  num_full_vertices = 18;

// 全頂点の頂点座標
static float  pyramid_full_vertices[][ 3 ] = {
    { 0.0, 1.0, 0.0 }, {-1.0,-0.8, 1.0 }, { 1.0,-0.8, 1.0 },
    ・・・・
    {-1.0,-0.8,-1.0 }, { 1.0,-0.8,-1.0 }, {-1.0,-0.8, 1.0 } };

// 全頂点の法線ベクトル
static float  pyramid_full_normals[][ 3 ] = {
    { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 },
    ・・・・
    { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 } };
```

頂点配列を使用した描画(2)

- 頂点配列の機能を利用して描画

```
void  renderPyramid()
{
        glVertexPointer( 3, GL_FLOAT, 0, pyramid_full_vertices );
        glNormalPointer( GL_FLOAT, 0, pyramid_full_normals );

        glEnableClientState( GL_VERTEX_ARRAY );
        glEnableClientState( GL_NORMAL_ARRAY );

        glDrawArrays( GL_TRIANGLES, 0, num_full_vertices );
}
```

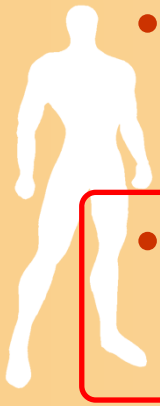
配列の先頭アドレスを
OpenGLに渡す

設定した配列を
有効化

設定した配列を使って
複数のポリゴンを描画

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ(直接記述)、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す



方法5 頂点配列＋インデックス配列

- 頂点配列に加えて三角面インデックスを指定し、OpenGLに一度にデータを渡して描画する方法
- 描画の手順
  - 基本的には、先ほどの頂点配列を使用する場合と同様に、配列データを設定する
  - 最後の描画時に、三角面インデックスを指定して描画
    - glDrawArrays()関数の代わりに glDrawElements()関数を使用

頂点配列を使用した描画(1)

- 配列データの定義(従来の描画方法と同じ)

```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 }, ・・・・・・
};

// 三角面インデックス(各三角面を構成する頂点の頂点番号)の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};

// 三角面の法線ベクトルの配列(三角面を構成する頂点座標から計算)
float  pyramid_tri_normals[ num_pyramid_triangles ][ 3 ] = {
          { 0.00, 0.53, 0.85 }, // +Z方向の面
          ・・・・・・
```

頂点配列を使用した描画(2)

- 頂点配列の機能を利用して描画

```
void  renderPyramid()
{
        glVertexPointer( 3, GL_FLOAT, 0, pyramid_full_vertices );
        glNormalPointer( GL_FLOAT, 0, pyramid_full_normals );

        glEnableClientState( GL_VERTEX_ARRAY );
        glEnableClientState( GL_NORMAL_ARRAY );

        glDrawElements( GL_TRIANGLES, num_pyramid_triangles * 3,
                                GL_UNSIGNED_INT, pyramid_tri_index );
}
```

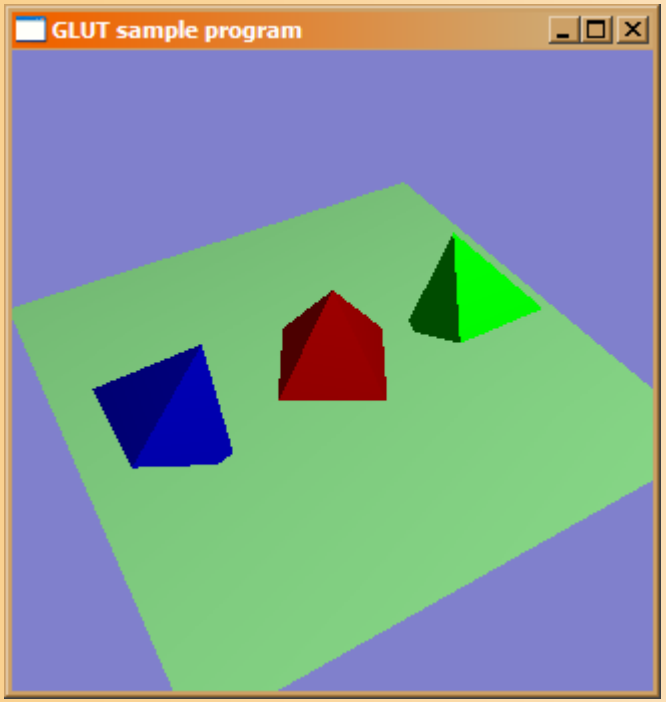

pyramid_tri_index で指定した頂点番号の配列に従って、頂点データを参照し、複数のポリゴンを描画

この方法の問題点

- 頂点の法線の問題
  - 隣接するポリゴンが共有する頂点を一つにまとめることができる
    - 頂点データの数を減らすことができる
  - ただし、頂点の座標だけでなく、法線やテクスチャ座標も一緒にする必要がある
    - 同じ頂点で、面ごとに法線を変えるようなことはできない
    - どうしても別の法線にしたければ、頂点データを分ける必要がある

2種類の法線による結果の例

同一頂点でも面ごとに
異なる法線を使用



同一頂点には全部の面で
同じ法線を使用

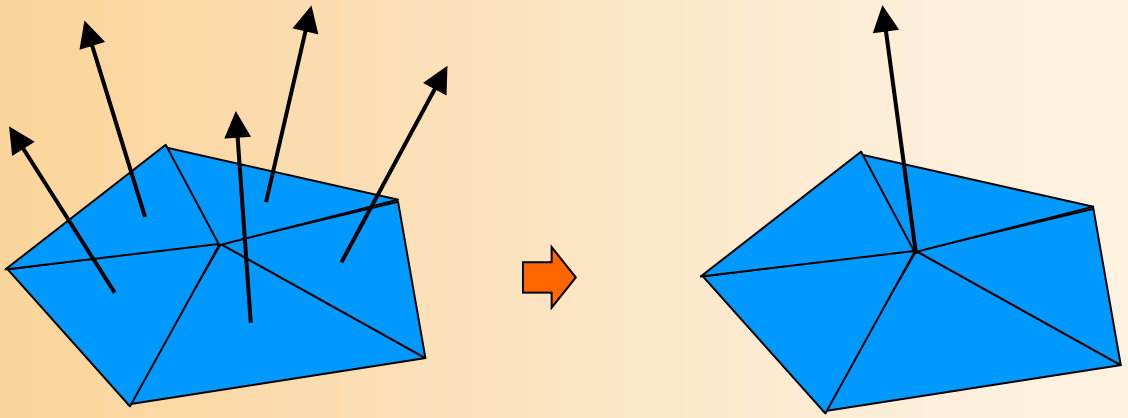
※ 面と面の境界がおなじ色になる

頂点の法線の使い分け

- 頂点を共有する面全部で共通の法線
  - 人体などの一般的な物体では、シェーディングによって表面をなめらかに見せるため、頂点を共有する面全部で頂点の法線を共通にするのが普通
  - 法線データは頂点の数だけ必要
- 面ごとに別々の法線
  - 今回の例の四角すいのように、角のある物体については、面ごとに法線を分けるのが自然
  - 法線データは面の数（×３）だけ必要になる
- 両者の混在は困難なのでどちらかに決めて適用

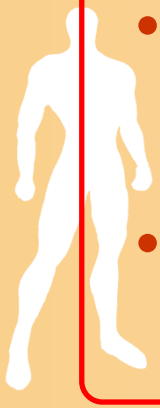
頂点の法線の計算方法

- 頂点の法線を自動的に計算する方法
  - 頂点の法線を全て零ベクトルにする
  - それぞれの面ごとに面の法線を計算
    - 面の法線を、面を構成する全頂点の法線に加算
  - 最後に、それぞれの頂点の法線を正規化する

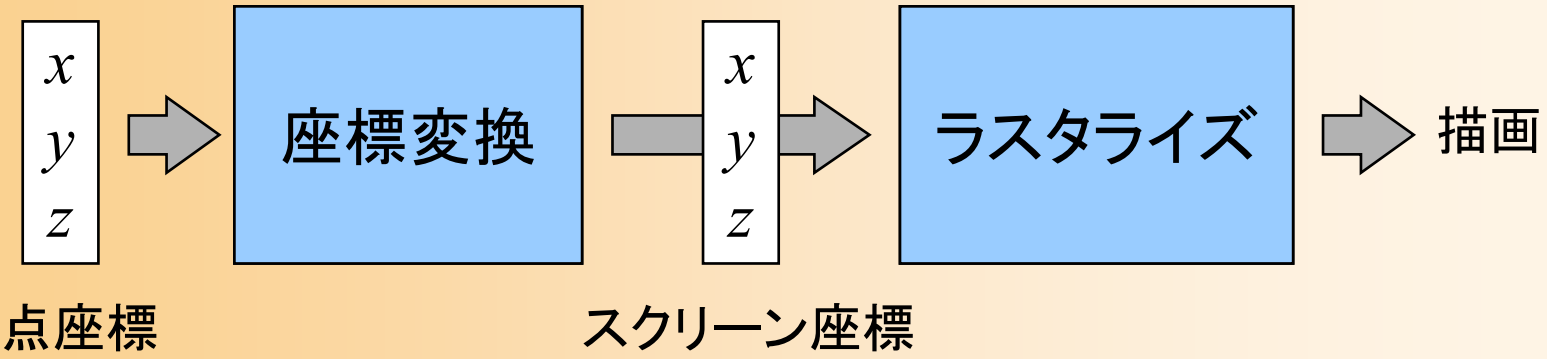


ポリゴンモデルの描画方法（まとめ）

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、OpenGLにまとめて渡す



補足: レンダリングの高速化



- レンダリング時にボトルネックとなりうる処理
  - ラスタライズ
  - 頂点データの座標変換、頂点データの転送
  - 各描画関数の呼び出し
- 環境やプログラムによりボトルネックは異なる

レンダリングの高速化

- ラスタライズの問題
  - 特に、ハードウェアがサポートしていない環境
  - Zソートなどと併用することで、高速化できる
- 頂点データの座標変換・転送の問題
  - 特に、ハードウェアがサポートしていない環境
  - 同じ頂点は共有するなど、データ量を減らすことで高速化できる
- 関数呼び出しのオーバーヘッドの問題
  - 頂点配列・インデックス配列を使うことで高速化

まとめ

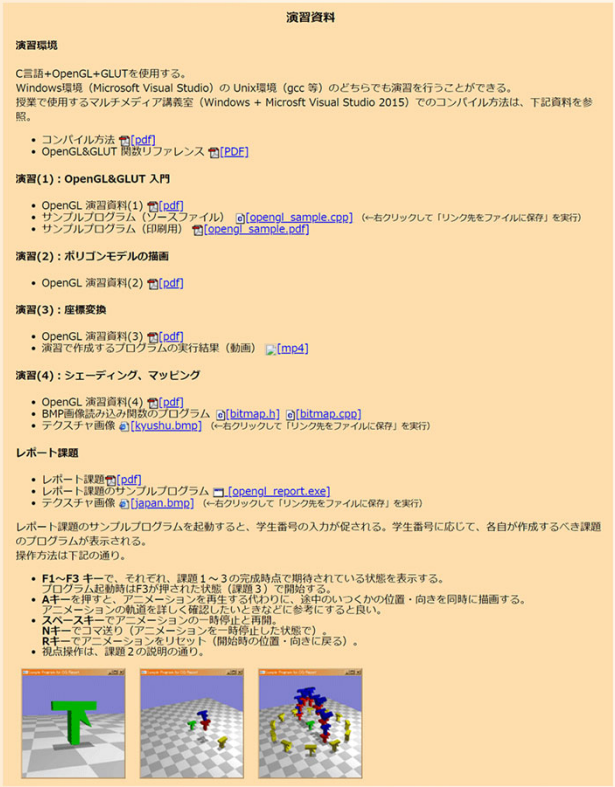
- OpenGLプログラミングの基礎
  - C言語＋OpenGL＋GLUT によるプログラミング
- 座標変換の基礎
  - 同次座標変換（アフィン変換）にもとづく視野変換行列の設定

  - いずれも、学部の講義（レベルの内容）の復習

プログラミング演習課題

- 学部の授業の演習課題・レポート課題
  - 基礎の理解が不十分な人は、各自で取り組む
  - システム創成情報工学科「コンピュータグラフィックスS」の講義・演習資料

    http://www.cg.ces.kyutech.ac.jp/lecture/cg/
  - 演習課題
    1. ポリゴンモデルの描画
    2. 座標変換によるアニメーション
    3. テクスチャマッピング
  - レポート課題



まとめ

- OpenGL＆GLUTの概要
- サンプルプログラムの概要
- 座標変換
- 変換行列の設定
- ポリゴンモデルの描画

次回予告

- 視点操作
  - 利用者が視点を操作して、仮想空間や物体を適切な位置・方向から見るための機能
  - 変換行列による、代表的な視点操作の実現方法
- 第1回 レポート課題