



# コンピューターグラフィックスS

## 第7回 演習(2):ポリゴンモデルの描画

システム創成情報工学科 尾下 真樹

2019年度 Q2

# 今日の内容

- 前回の演習の復習
- 前回の復習 (ポリゴンの描画方法)
- 基本オブジェクトの描画
- ポリゴンモデルの描画
- 別のポリゴンモデルの描画
- 演習課題

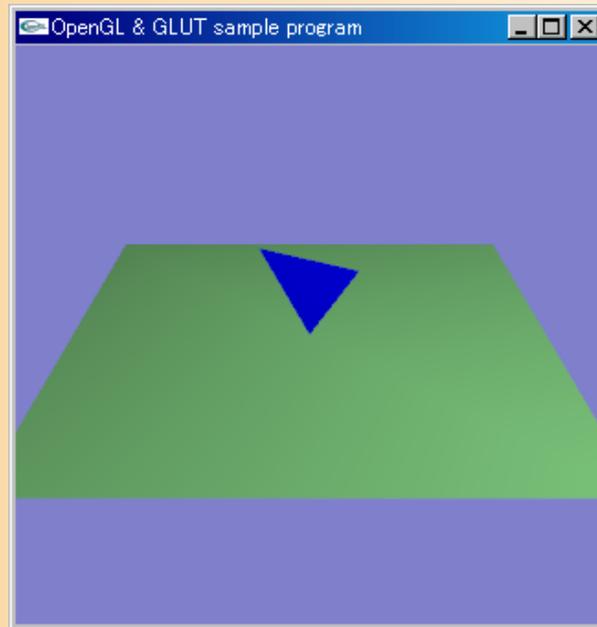




前回の演習の復習

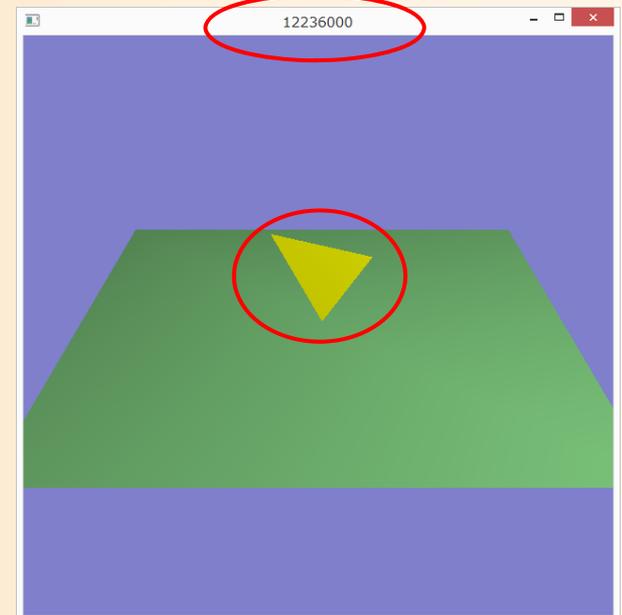
# サンプルプログラム

- `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



# 前回の演習課題

1. コンパイル・実行できることを確認する
2. プログラムを修正して、以下の修正を行う
  - ウィンドウのタイトルに、自分の学生番号が表示されるようにする
  - 三角形の色を、青から黄に変更する
3. 修正が終わったら、Moodleからプログラムを提出



# 演習資料(3種類)(確認)

- 演習資料(OpenGL演習)
  - この資料に従って、プログラムを拡張していく
  - 次回以降の分の説明は、逐次追加する
- コンパイル方法の説明資料
  - コンパイル方法の詳しい説明
  - CL端末や自宅での方法も一応説明
- OpenGL関数 簡易リファレンス
  - OpenGLの関数を簡単に説明した資料



# 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()`

opengl\_sample.c

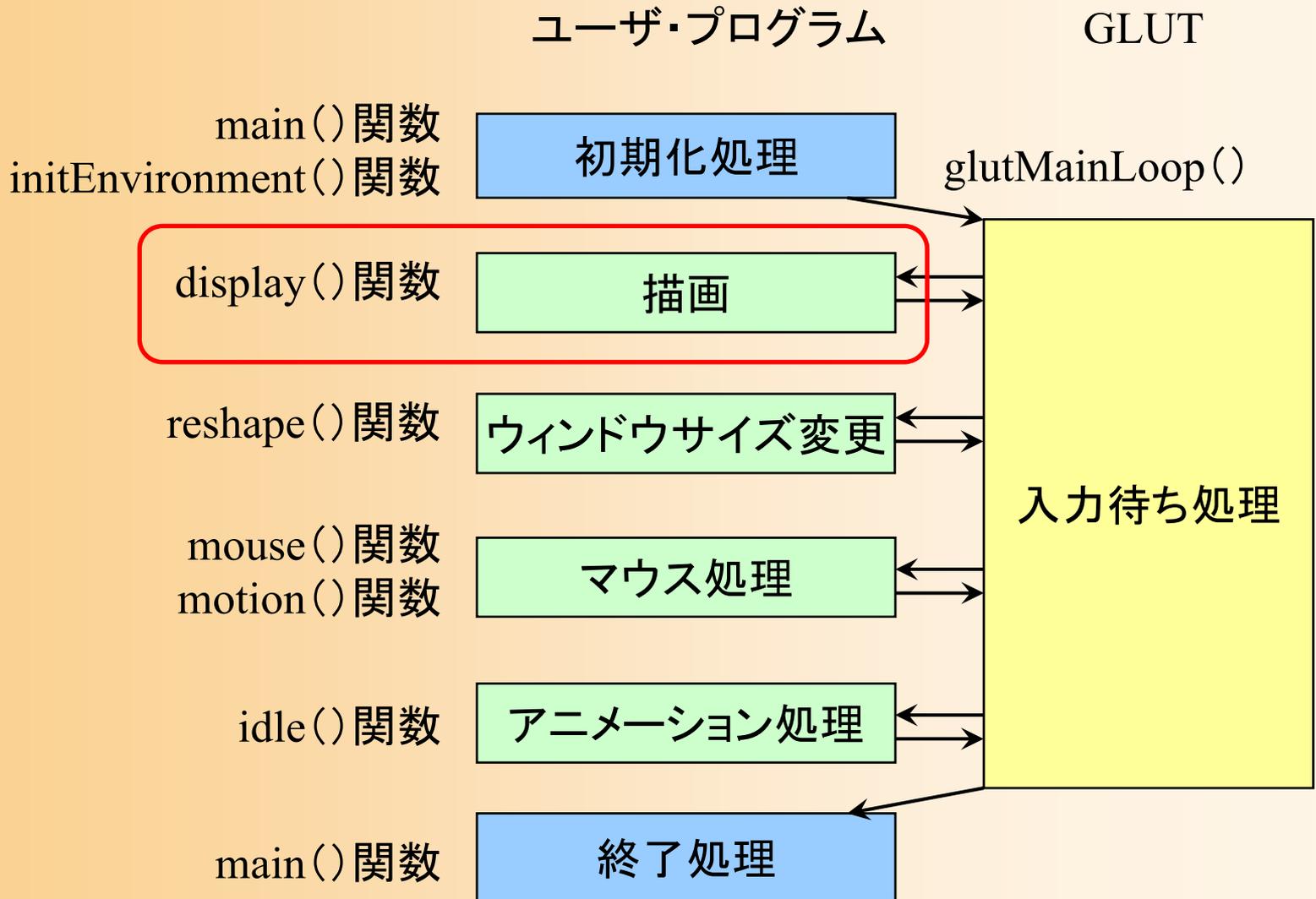


# OpenGLの関数

- **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



# サンプルプログラムの構成

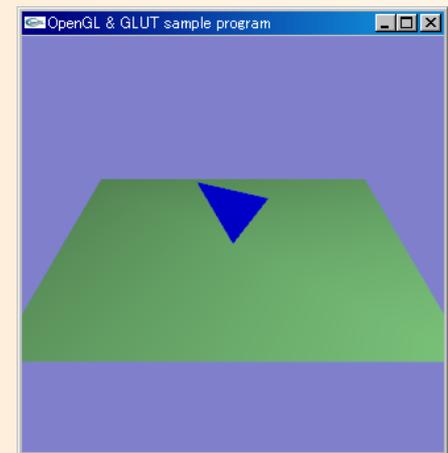
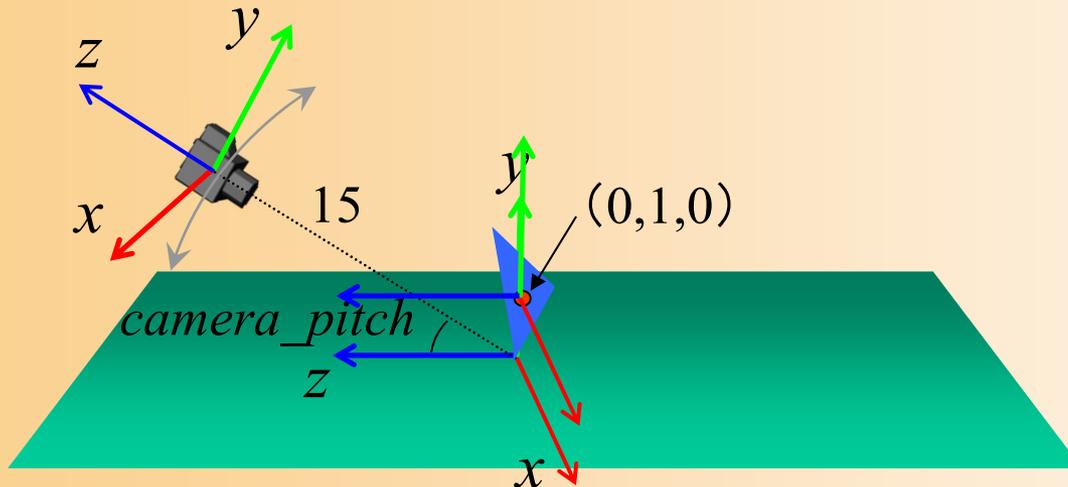


# 描画関数の流れ

```
//  
// 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  
}
```

# 今回の演習

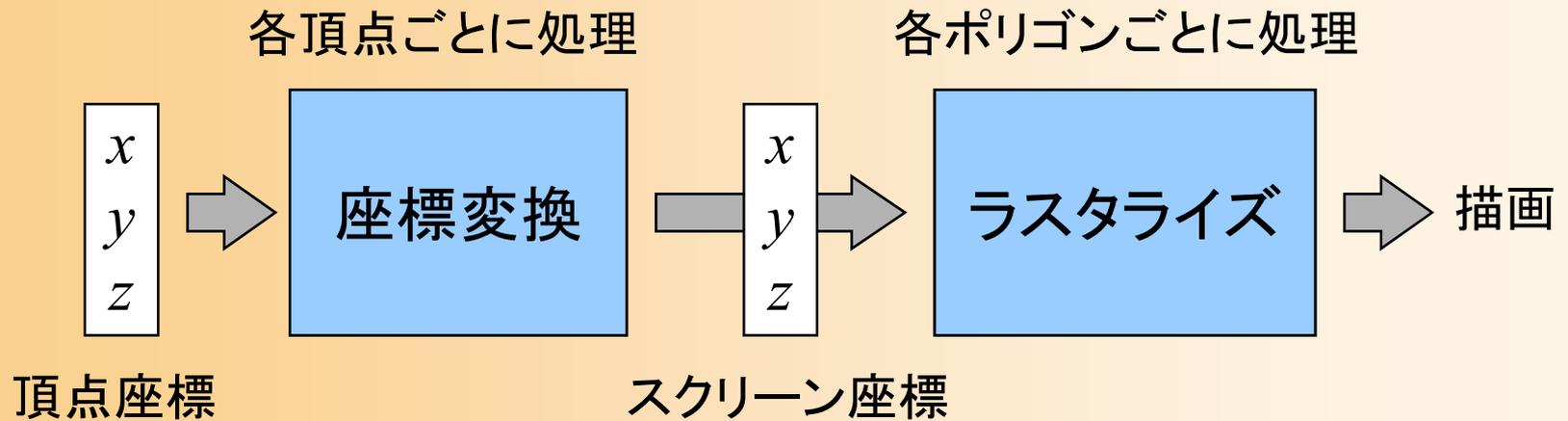
- 前回のプログラムの描画処理を、1枚のポリゴンの代わりに、ポリゴンモデルを描画するように変更
- 変換行列については、基本的に変更しない  
(詳しくは後日の講義で学習してから)





前回の復習

# レンダリング・パイプライン



## • レンダリング・パイプライン

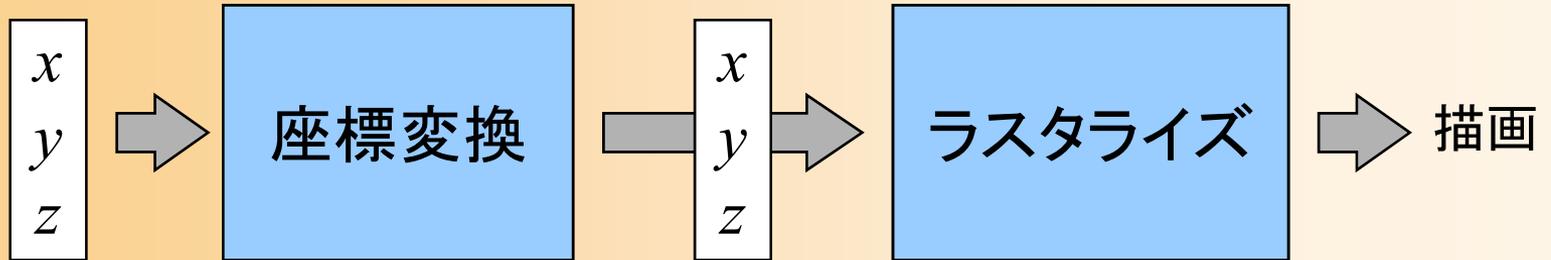
- 入力されたデータを、流れ作業(パイプライン)で処理し、順次、画面に描画
- 3次元空間のポリゴンのデータ(頂点データの配列)を入力
- いくつかの処理を経て、画面上に描画される



# 処理の流れ

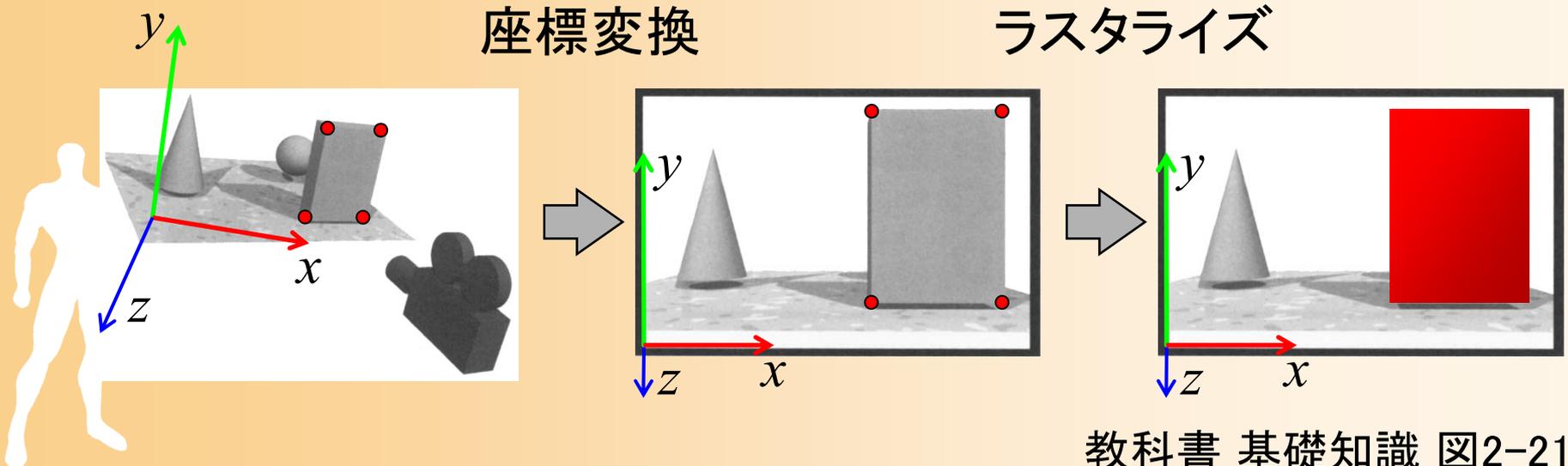
各頂点ごとに処理

各ポリゴンごとに処理



頂点座標  
(法線・色・テクスチャ座標)

スクリーン座標



# レンダリング

- Zバッファ法によるレンダリング

- 基本的には、OpenGLが自動的にZバッファ法を用いたレンダリングを行うので、自分のプログラムでは特別な処理は必要ない
- 最初に、Zテストを有効にするように、設定する必要がある

```
void initEnvironment( void )
{
    .....

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

    .....
}
```

# ポリゴンの描画

- glBegin() ~ glEnd() 関数を使用

```
glBegin( 図形の種類 );
```

この間に図形を構成する頂点データを指定

```
glEnd();
```

※ 頂点データの指定では、一つの関数で、図形を構成する頂点の座標・色・法線などの情報の一つを指定

- 図形の種類（各種の点・線・面が指定可能）

- 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

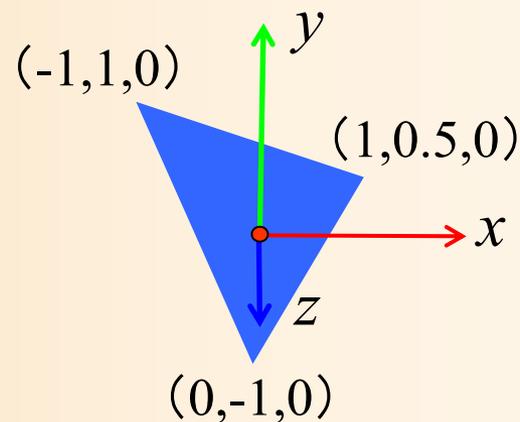
- `glColor3f( r, g, b )`
  - これ以降の頂点の色を設定
- `glNormal3f( nx, ny, nz )`
  - これ以降の頂点の法線を設定
- `glVertex3f( x, y, z )`
  - 頂点座標を指定
  - 色・法線は、最後に指定したものが使用される



# 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();
```

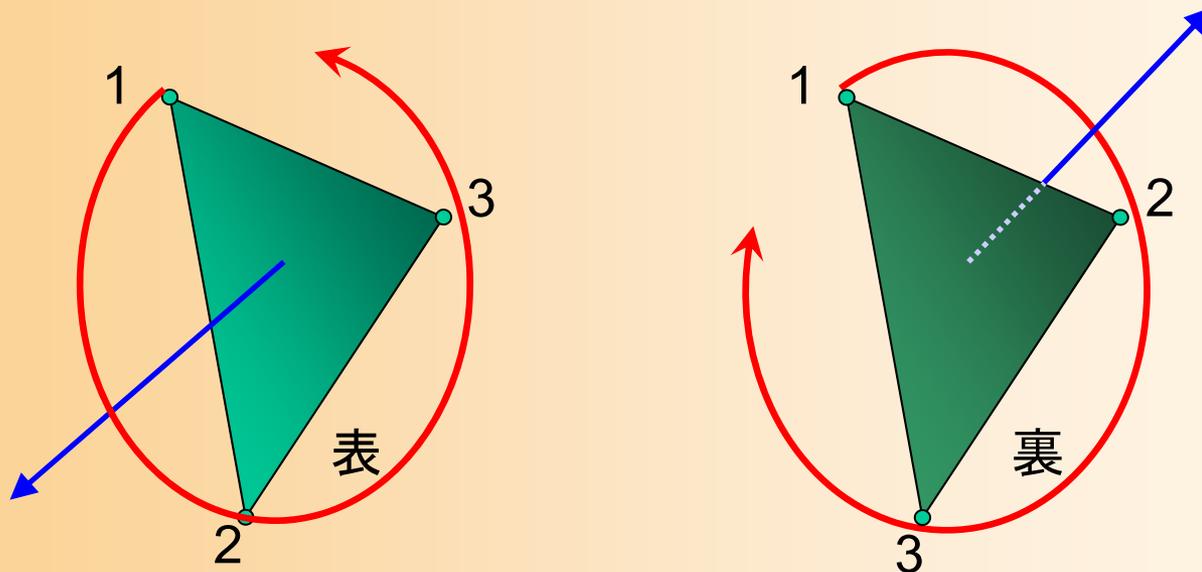


GL\_TRIANGLES が指定されているので、3つの頂点をもとに、1枚の三角面を描画(6つの頂点が指定されたら、2枚描画)



# 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





# 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: `glutWireCube()`, `glutSolidCube()`
  - 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する関数



# 立方体の描画

- `glutSolidCube(size)`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
    */

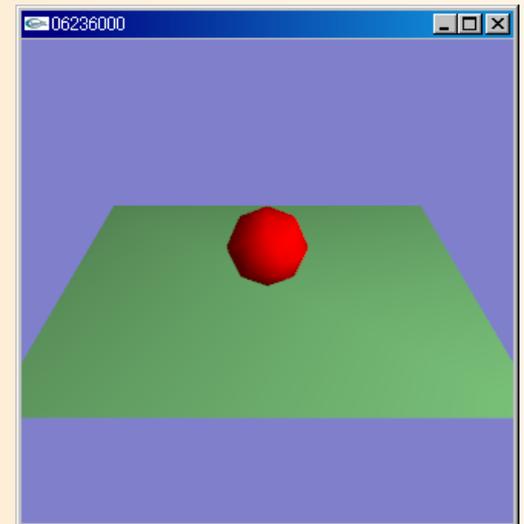
    // 立方体を描画
    glColor3f( 1.0, 0.0, 0.0 );
    glutSolidCube( 1.5f );
    .....
}
```



# 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

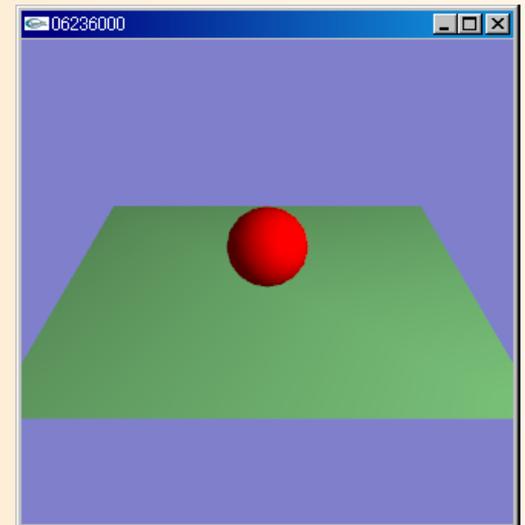
```
void display( void )  
{  
    .....  
    // 球を描画  
    glColor3f( 1.0, 0.0, 0.0 );  
    glutSolidSphere( 1.0, 8, 8 );  
    .....  
}
```

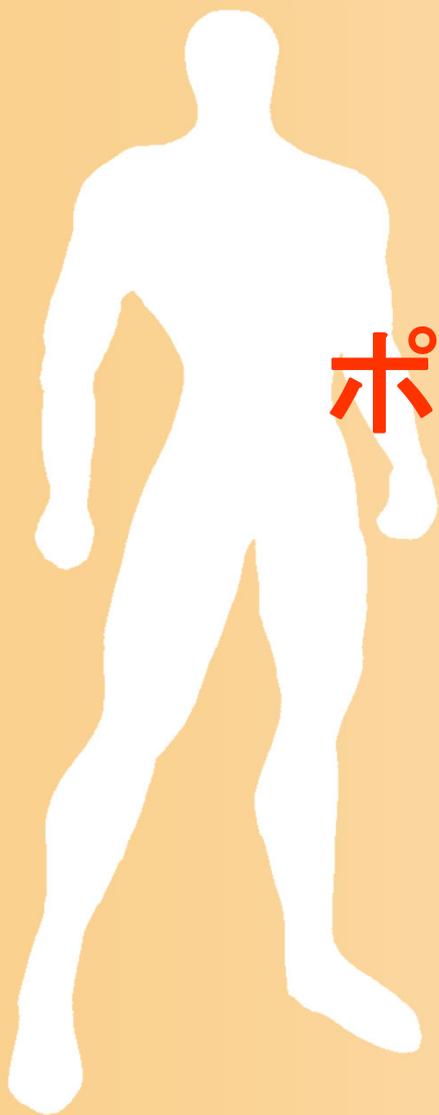


# 球の描画

- `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 );
    .....
}
```

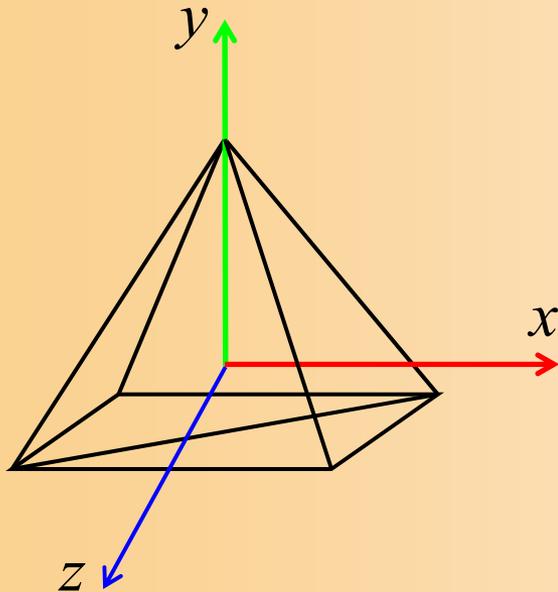




# ポリゴンモデルの描画

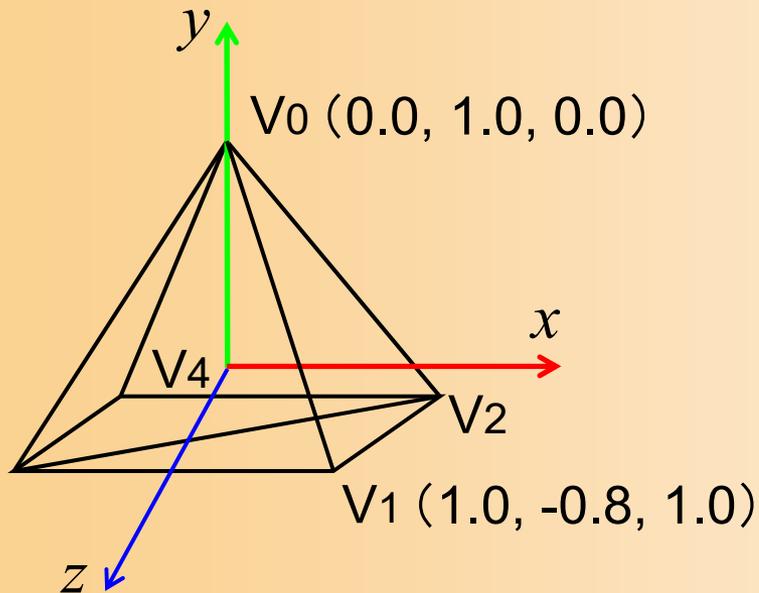
# 四角すいの描画

- 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

$V_0$   $(0.0, 1.0, 0.0)$

$V_1$   $(1.0, -0.8, 1.0)$

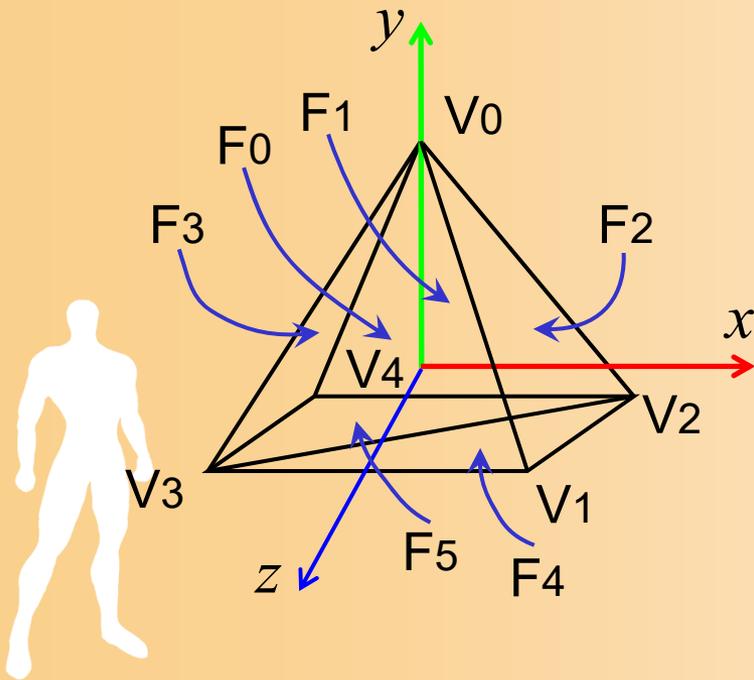
$V_2$   $(1.0, -0.8, -1.0)$

$V_3$   $(-1.0, -0.8, 1.0)$

$V_4$   $(-1.0, -0.8, -1.0)$

# 四角すいの描画

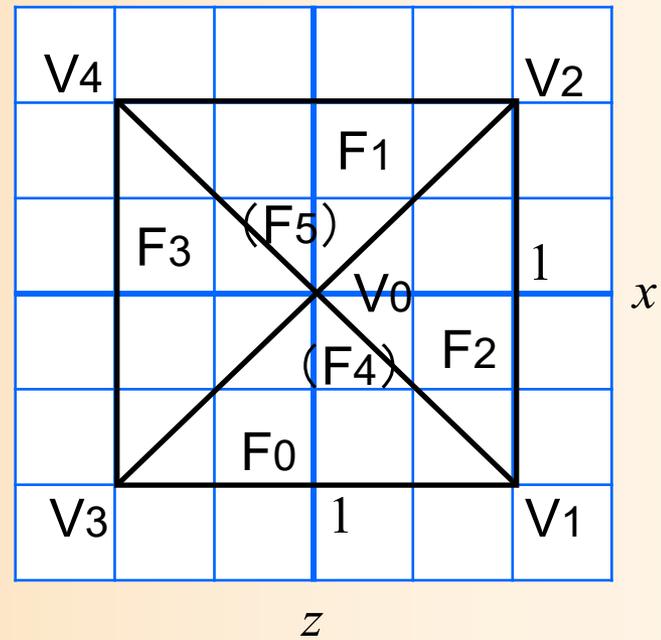
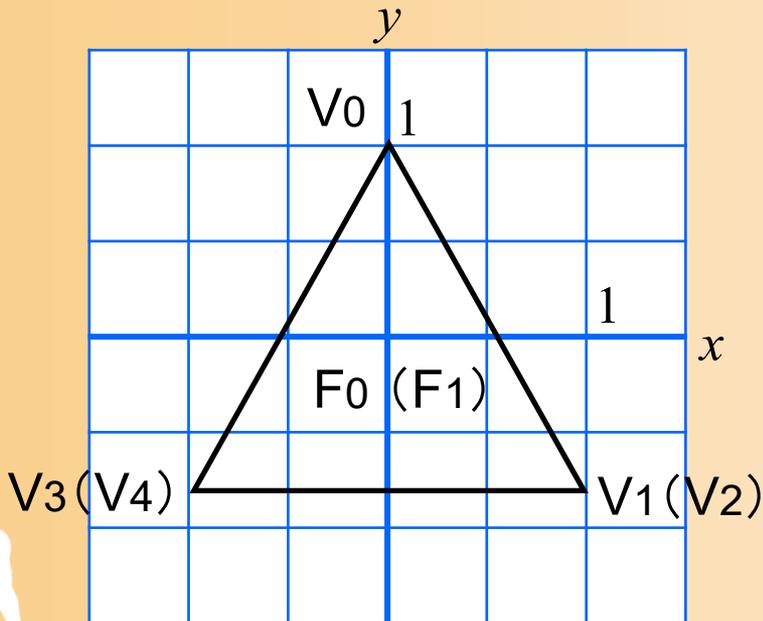
- 四角すいを構成する頂点と三角面
  - 頂点座標
  - 三角面の頂点、面の法線



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

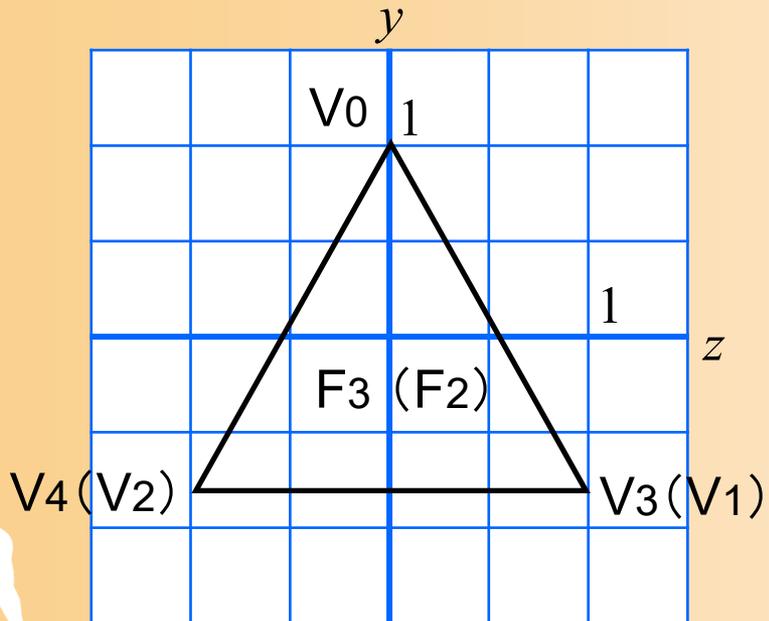


※ 括弧付きの頂点・面は裏側の頂点・面を表す



# 三面図

- $yz$ 平面

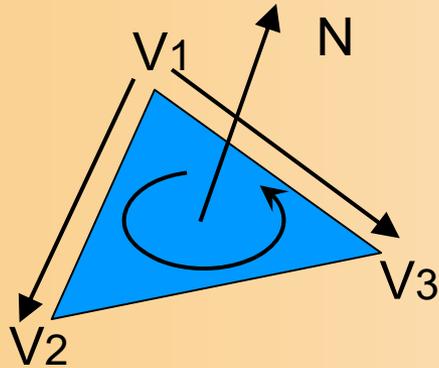


※ 括弧付きの頂点・面は裏側の頂点・面を表す



# 面の法線の計算方法

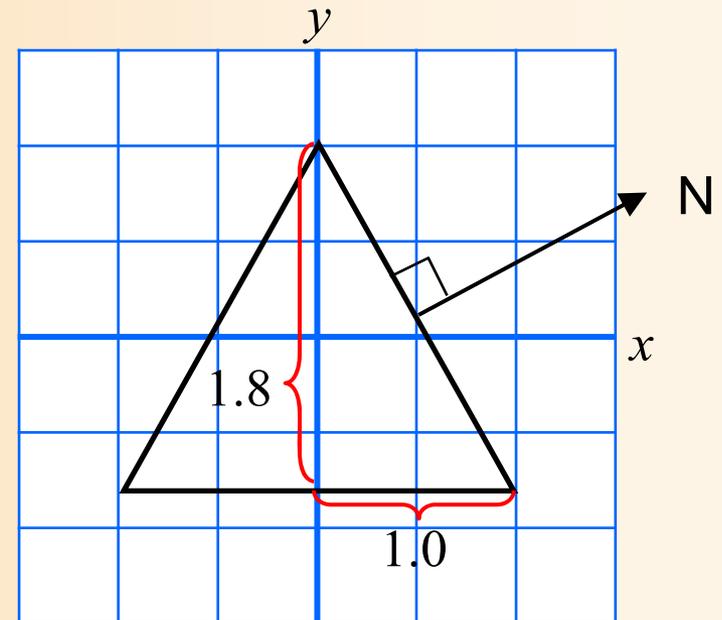
- ポリゴンの2辺の外積から計算できる



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

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



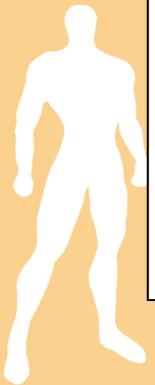
# 頂点データの配列を使用(1)

- 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す

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 $6 \times 3$ 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



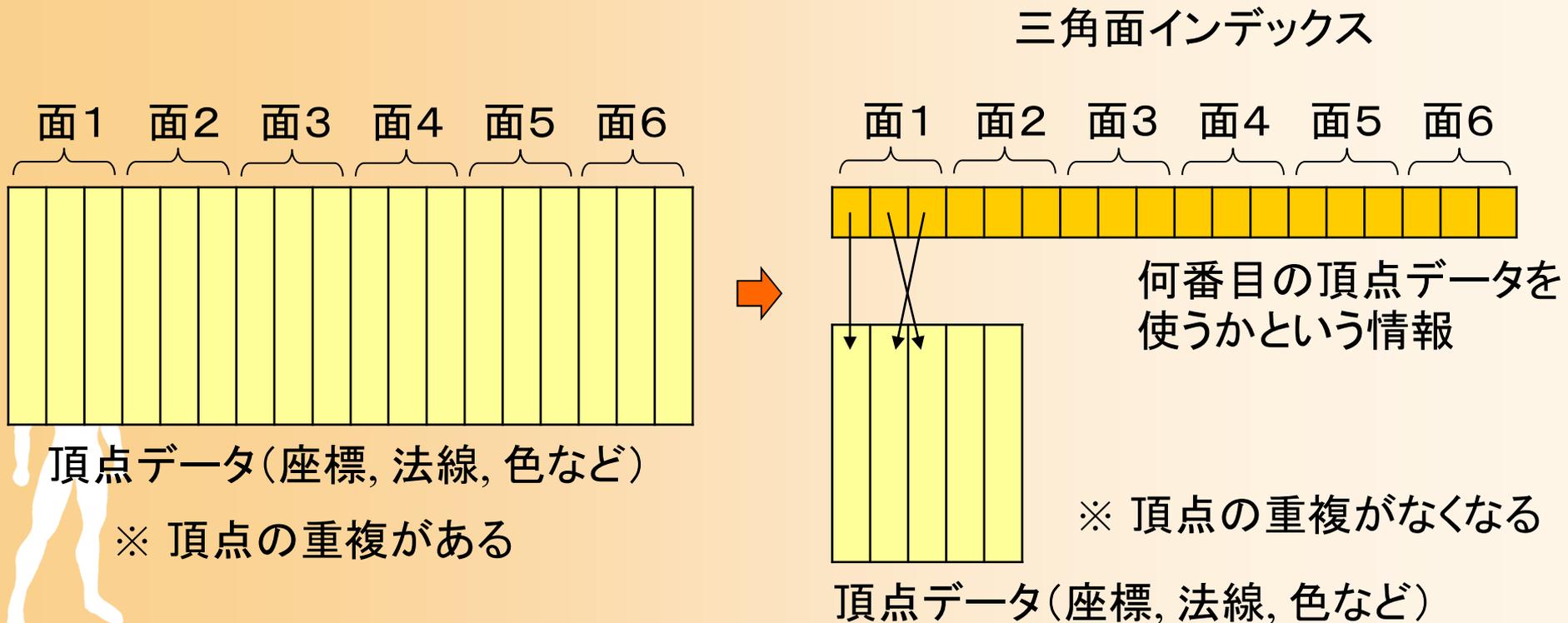
# 方法3 三角面インデックスを使用

- 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  $(x,y,z) \times$  頂点数
  - 三角面を構成する頂点番号  $(v_0,v_1,v_2) \times$  三角面数
  - 三角面の法線  $(x,y,z) \times$  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

各三角面ごとに繰り返す

三角面の各頂点ごとに繰り返す

```
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();
}
```

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



# 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
    - 渡すデータの量は同じでも、余分なオーバーヘッドを省けるため、処理を高速化できる
    - モバイル端末用の OpenGL ES では、この頂点配列を使った描画しかできない (`glVertex()` は使えない)
- 詳細は、本授業の演習では省略

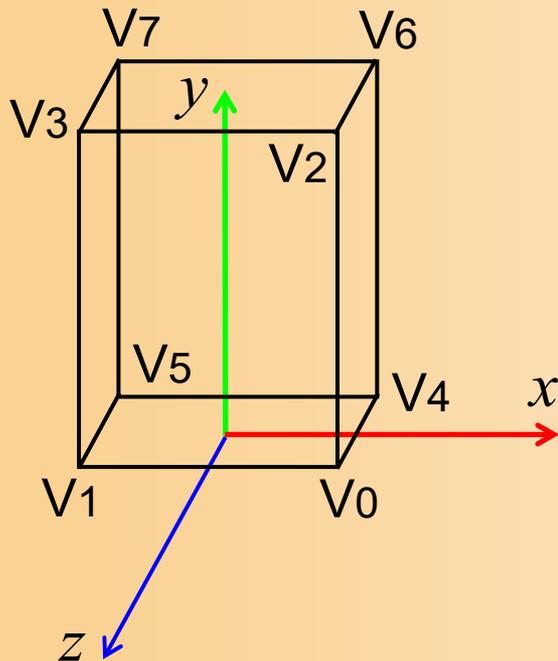




# 別のポリゴンモデルの描画

# 直方体の描画

- 別のポリゴンモデル(直方体)の描画
  - 四角面の集まりとして描画
  - 8つの頂点と6枚の四角面により構成



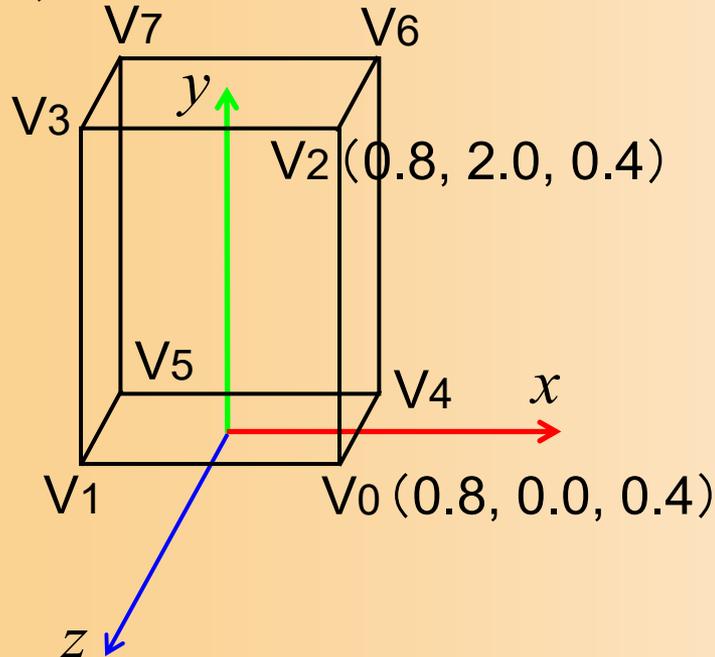
# 直方体の描画

- 別のポリゴンモデル(直方体)の描画

- 頂点座標

- 四角面の頂点、法線

(-0.8, 2.0, -0.4)



頂点座標

V0 (0.8, 0.0, 0.4)

V1 (-0.8, 0.0, 0.4)

V2 (0.8, 2.0, 0.4)

V3 (-0.8, 2.0, 0.4)

V4 (0.8, 0.0, -0.4)

V5 (-0.8, 0.0, -0.4)

V6 (0.8, 2.0, -0.4)

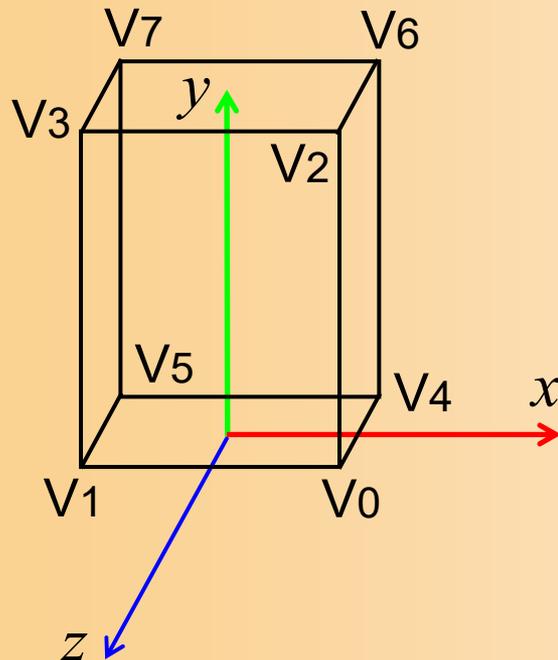
V7 (-0.8, 2.0, -0.4)

# 直方体の描画

- 別のポリゴンモデル(直方体)の描画

- 頂点座標

- 四角面の頂点、法線



四角面

法線

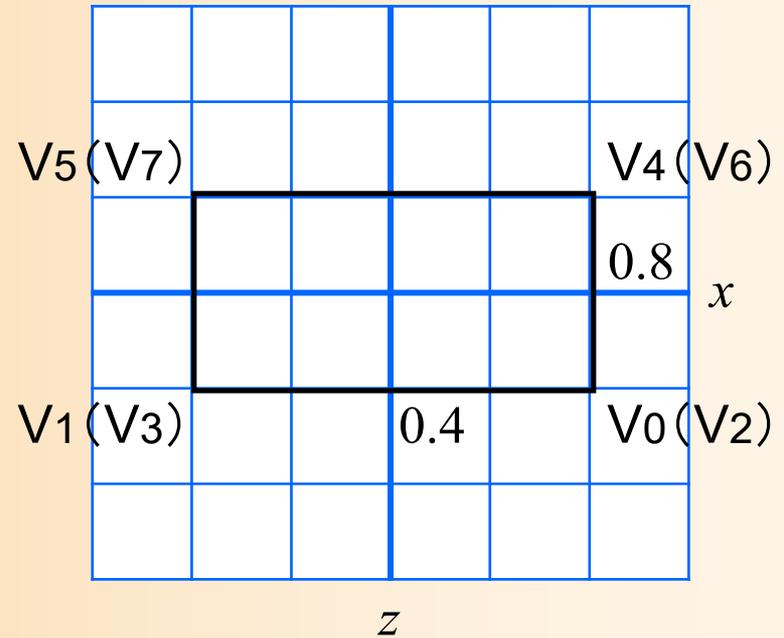
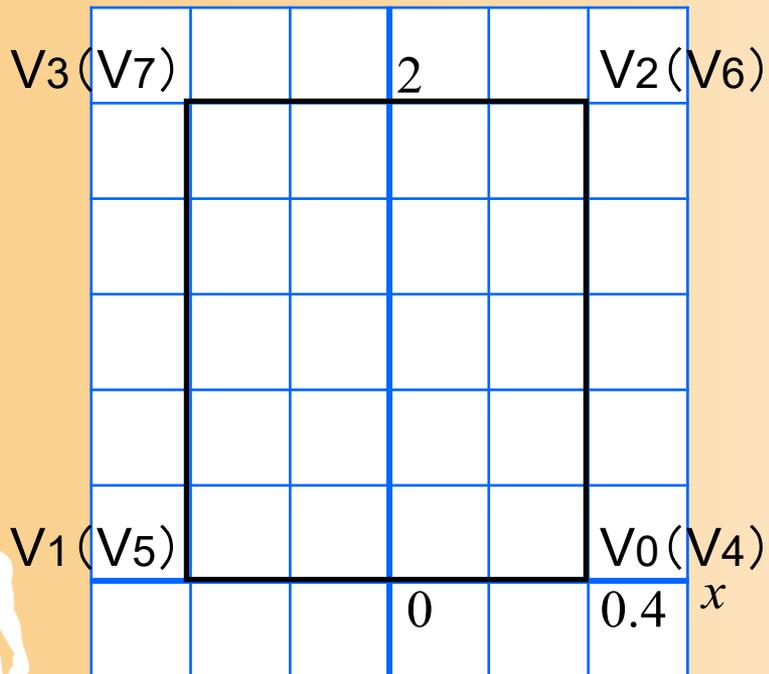
{ V2, V3, V1, V0 }	{ 0.0, 0.0, 1.0 }
{ V7, V6, V4, V5 }	{ 0.0, 0.0, -1.0 }
{ V2, V0, V4, V6 }	{ 1.0, 0.0, 0.0 }
{ V3, V7, V5, V1 }	{ -1.0, 0.0, 0.0 }
{ <span style="border: 1px solid red; padding: 2px;">?</span> }	{ <span style="border: 1px solid red; padding: 2px;">?</span> }
{ V0, V1, V5, V4 }	{ 0.0, -1.0, 0.0 }

1枚は空欄にしているので、各自、適切な頂点番号を考えて追加する

# 三面図

- $xy$ 平面、 $xz$ 平面

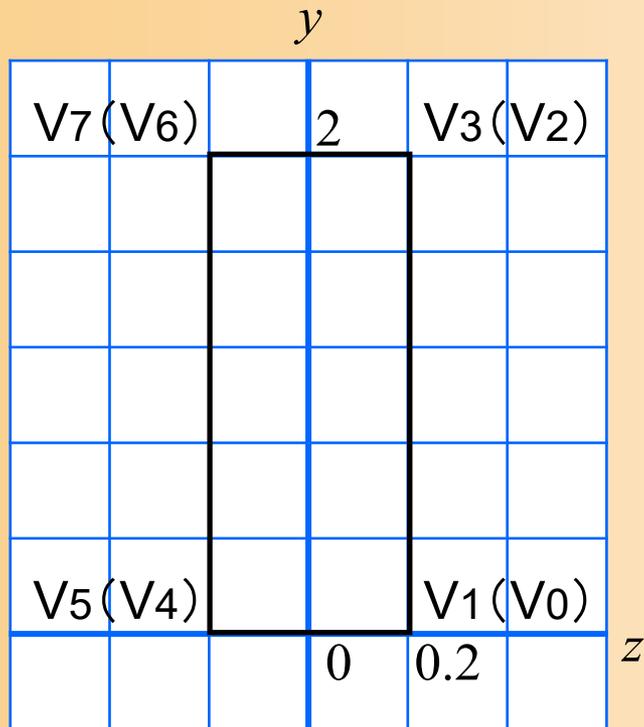
$y$



※ 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

# 配列を使った直方体の描画(1)

- 描画方法
  - 方法3を使用
  - 法線や色の情報は、面ごとに指定
  - 三角面の代わりに、四角面を使用する



# ポリゴンモデルの描画方法(確認)

- 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



# 配列を使った直方体の描画(2)

- 配列データの定義

- 四角面を使うので、各面の頂点数が4個になる

```
const int num_cube_vertices = 8; // 頂点数
const int num_cube_quads = 6;   // 四角面数

// 頂点座標の配列
float cube_vertices[ num_cube_vertices ][ 3 ] = {
    { 0.8, 0.0, 0.4 }, // 0
    .....
    { -0.8, 2.0, -0.4 }, // 7
};

// 四角面インデックス(各四角面を構成する頂点の頂点番号)の配列
int cube_index[ num_cube_quads ][ 4 ] = {
    { 2, 3, 1, 0 }, { 7, 6, 4, 5 }, { 2, 0, 4, 6 }, { 3, 7, 5, 1 },
    { ? }, { 0, 1, 5, 4 }
};
```



# 配列を使った直方体の描画(3)

- 配列データの定義(続き)
  - 今回は各面の色も指定する

```
// 四角面の法線ベクトルの配列(四角面を構成する頂点座標から計算)
float cube_normals[ num_cube_quads ][ 3 ] = {
    { 0.00, 0.00, 1.00 },
    { 0.00, 0.00, -1.00 },
    .....
    { ? },
    { 0.00, -1.00, 0.00 } };
```

```
// 四角面のカラーの配列
float cube_colors[ num_cube_quads ][ 3 ] = {
    { 0.00, 1.00, 0.00 },
    { 1.00, 0.00, 1.00 },
    .....
    { 1.00, 1.00, 0.00 } };
```



# 配列を使った直方体の描画(4)

- 配列データを参照しながら四角面を描画

```
void renderCube()
{
    int i, j, v_no;
    glBegin( GL_QUADS );
    for ( i=0; i<num_cube_quads; i++ )
    {
        glNormal3f( cube_normals[i][0], ··[i][1], ·· [i][2] );
        glColor3f( cube_colors[i][0], ···[i][1], ···[i][2] );
        for ( j=0; j<4; j++ )
        {
            v_no = cube_index[ i ][ j ];
            glVertex3f( cube_vertices[v_no][0], ··[v_no][1], ··[v_no][2] );
        }
    }
    glEnd();
}
```

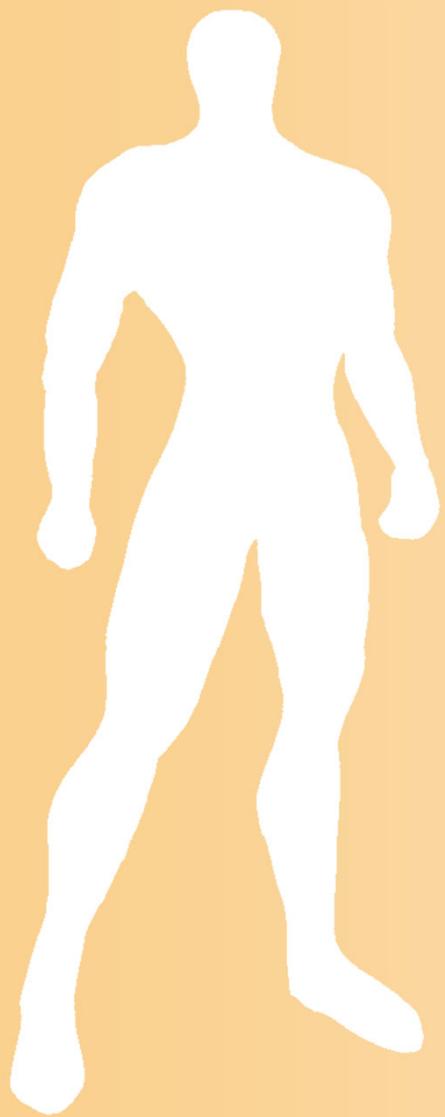


# 配列を使った直方体の描画(5)

- 描画関数から直方体の描画関数を呼び出し
  - 色の指定は不要

```
void display( void )
{
    .....
    // 直方体を描画
    renderCube();
    .....
}
```

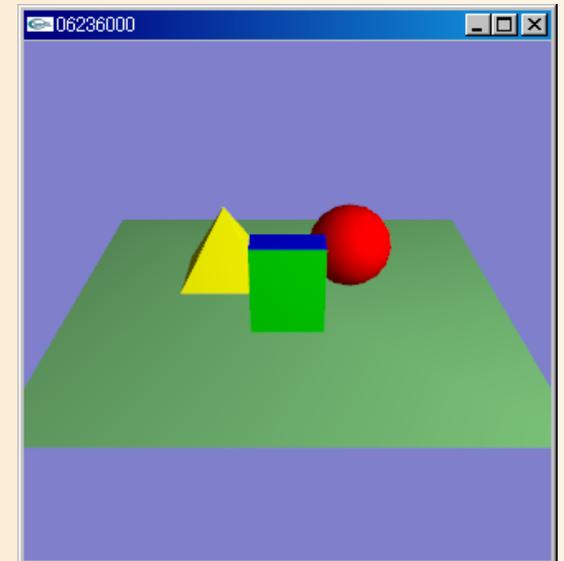




# 演習課題

# ポリゴンモデルの描画

- ここまでのポリゴンモデルをまとめて描画
  - 変換行列を利用して、3つのポリゴンモデルを同時に描画
    - 変換行列については、後日学習するので、今回は、サンプルプログラムをそのまま使用しておく
  - 右のスクリーンショットと同じ画面になるように、プログラムの空欄を埋める



• 前回の演習課題のプログラムをもとに変更を加える



# ポリゴンモデルの描画

- ここまでのポリゴンモデルをまとめて描画
  - 変換行列を利用して、3つのポリゴンモデルを同時に描画

```
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 );

        これまでのポリゴンの描画はコメントアウト

    */
    .....
}
```



```
.....  
// 球を描画  
glPushMatrix();  
    glTranslatef( 1.5, 1.0, -1.0 );  
    glColor3f( 1.0, 0.0, 0.0 );  
    ?  
glPopMatrix();  
  
// 角すいの描画  
glPushMatrix();  
    glTranslatef( -1.5, 1.0, -1.0 );  
    glColor3f( ? );  
    renderPyramid3();  
glPopMatrix();  
  
// 直方体の描画  
glPushMatrix();  
    glTranslatef( 0.0, 0.0, 1.0 );  
    renderCube();  
glPopMatrix();  
.....  
}
```



# プログラムの提出

- 作成したプログラムを「学生番号.cpp」のファイル名で、Moodleから提出
  - 本講義時間中に終わらせて、提出
  - 終わらなかった人は、下記の締め切りまでに提出
- **締め切り：7月19日（金）18:00**
  - 演習問題と同様、成績に加える
    - 未完成のプログラムが提出されていれば、減点
  - 今回の内容で、演習のやり方をきちんとおさえておかないと、次回以降の演習が全くできないため、必ず課題を行って提出すること



# 文字化けについての注意

- 日本語の文字化けが生じることがある
  - プログラムのコメント中の日本語の文字化け
    - Moodleに提出したプログラムをダウンロードして編集
    - Windowsでファイルを編集
    - 文字コードの異なる他のソフトからコピー&ペーストなどのことを行うと生じる可能性がある
  - 文字化けした状態で保存すると、復元は不可能
- 文字化けのないプログラムを提出すること
  - 万一、文字化けが生じたら、途中で保存したファイルや最初のファイルまで戻って、やり直す



# まとめ

- 前回の演習の復習
- 前回の復習 (ポリゴンの描画方法)
- 基本オブジェクトの描画
- ポリゴンモデルの描画
- 別のポリゴンモデルの描画
- 演習課題



# 次回予告

- 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

