



コンピュータグラフィックスS

第10回 演習(3):座標変換

システム創成情報工学科 尾下 真樹

2018年度 Q2

今日の内容

- 前回の復習
- 前回の演習の復習
- 視点操作の実現方法(復習)
- 視点操作の拡張
- 変換行列によるアニメーション
- 演習課題

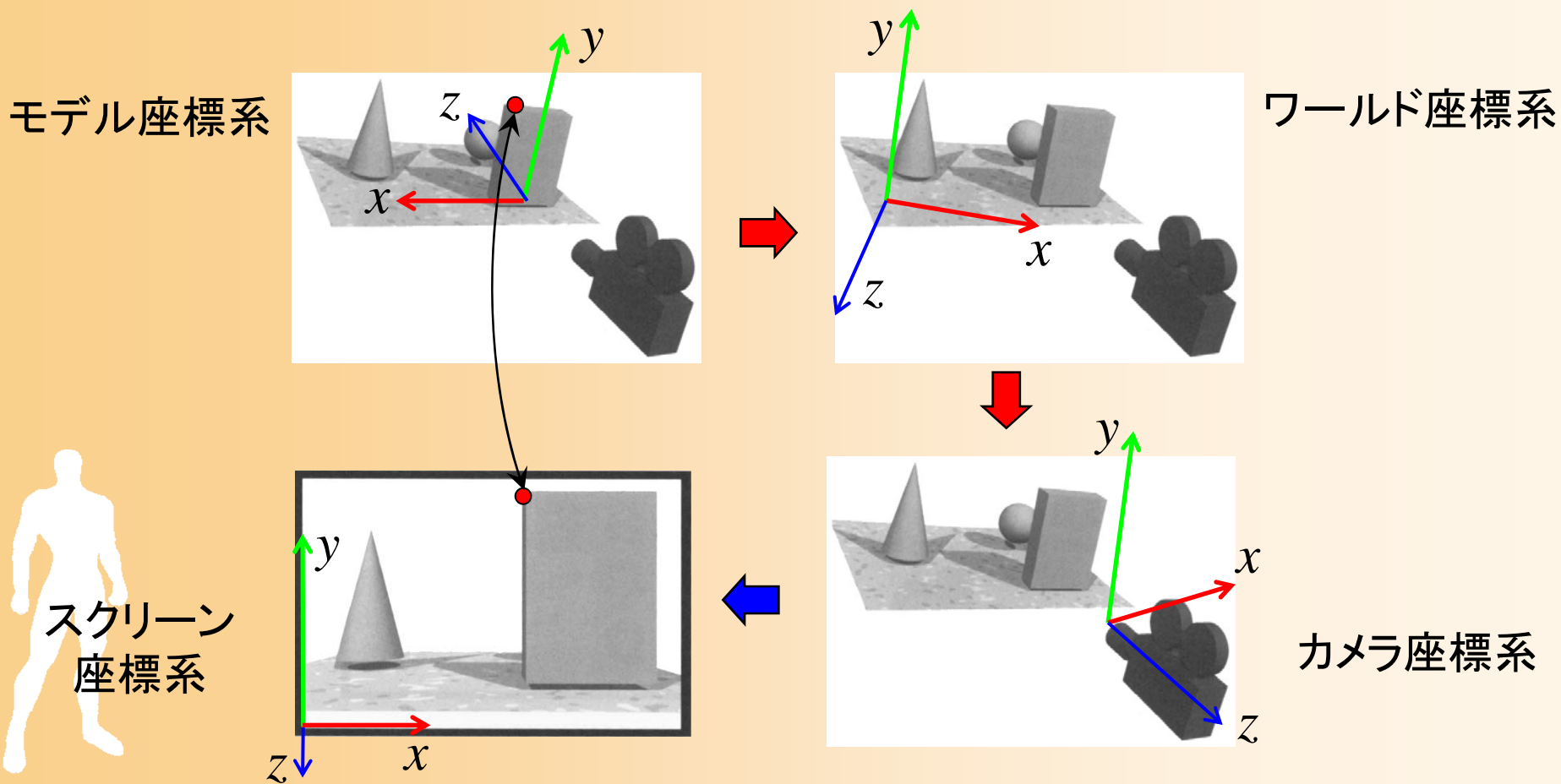


前回の復習



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 $(x'/w' \ y/w' \ z/w')$ 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



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



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

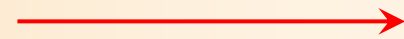


変換行列の変更

- 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、実際には先に適用される

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

プログラムに
記述する順番



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

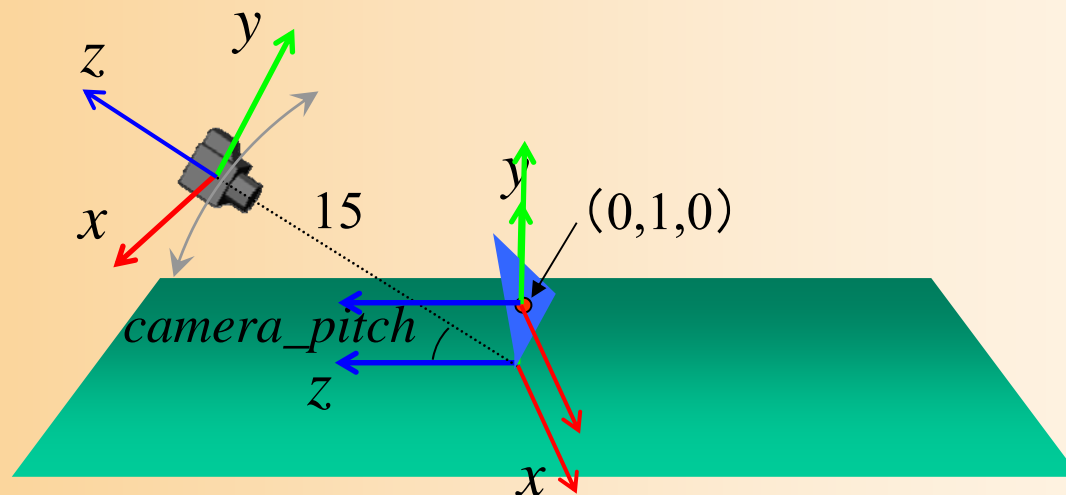


適用される順番



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_pitch`
 - 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

$$\begin{matrix} \textcircled{3} & & \textcircled{2} & & \textcircled{1} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & & & \\ 0 & \cos(-camera_pitch) & -\sin(-camera_pitch) & \\ 0 & \sin(-camera_pitch) & \cos(-camera_pitch) & \\ 0 & & & 0 \end{pmatrix} & \begin{pmatrix} & & & 0 \\ & & & 0 \\ & & & 0 \\ 1 & & & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \end{matrix}$$

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

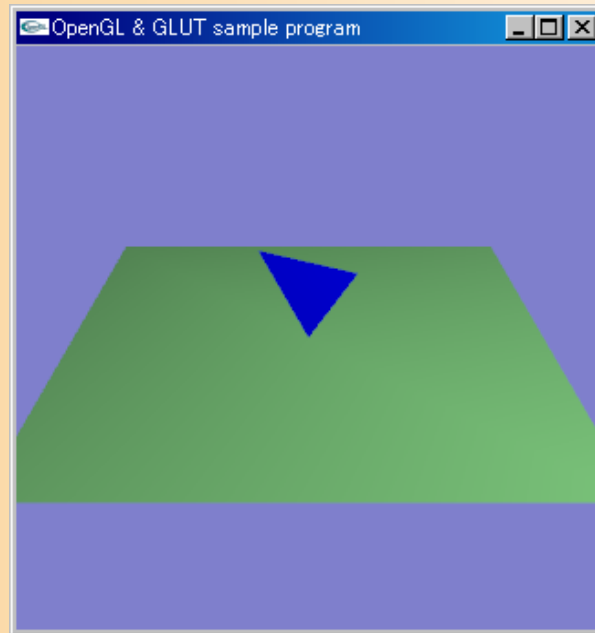


前回の演習の復習

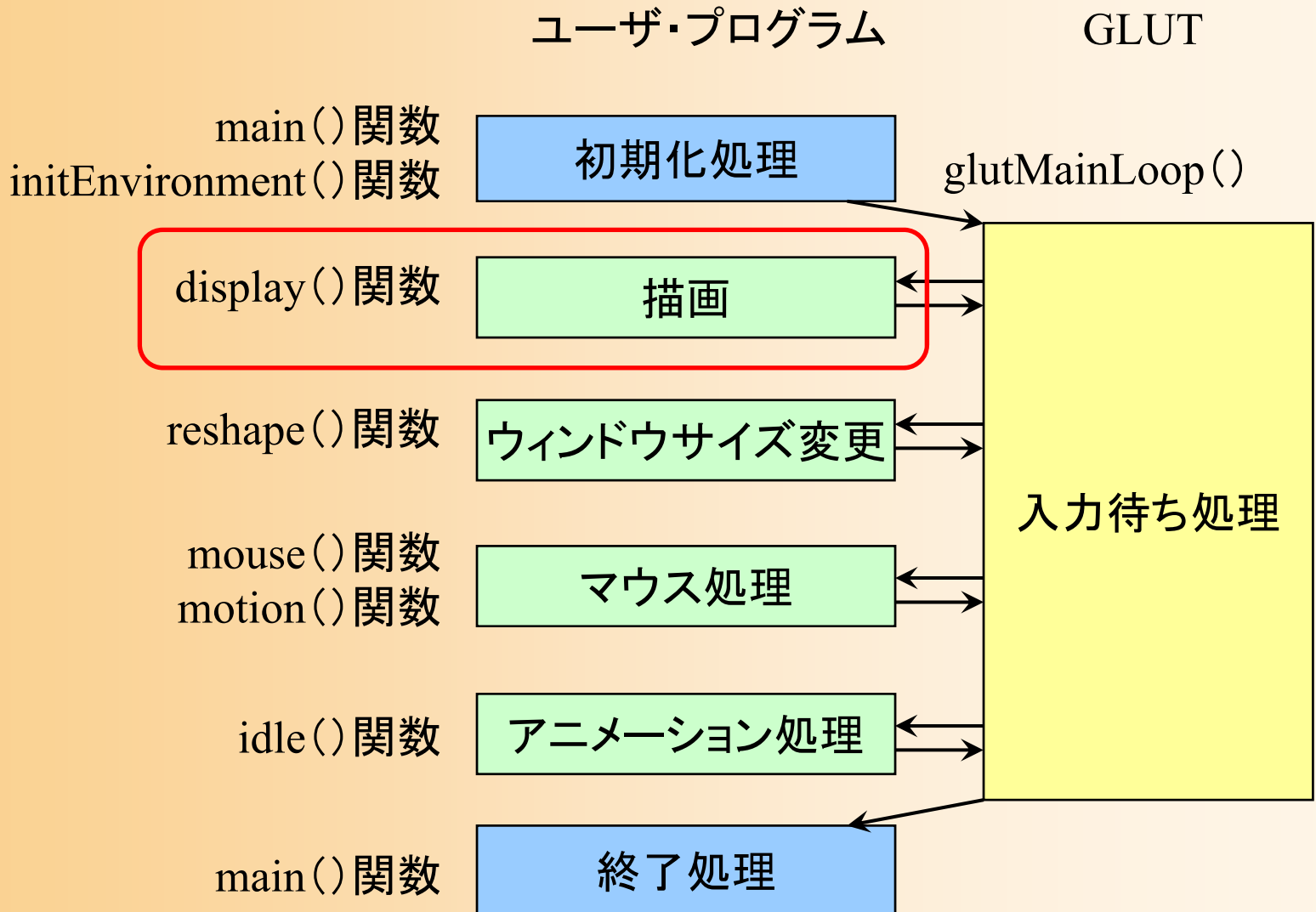


サンプルプログラム

- `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



サンプルプログラムの構成



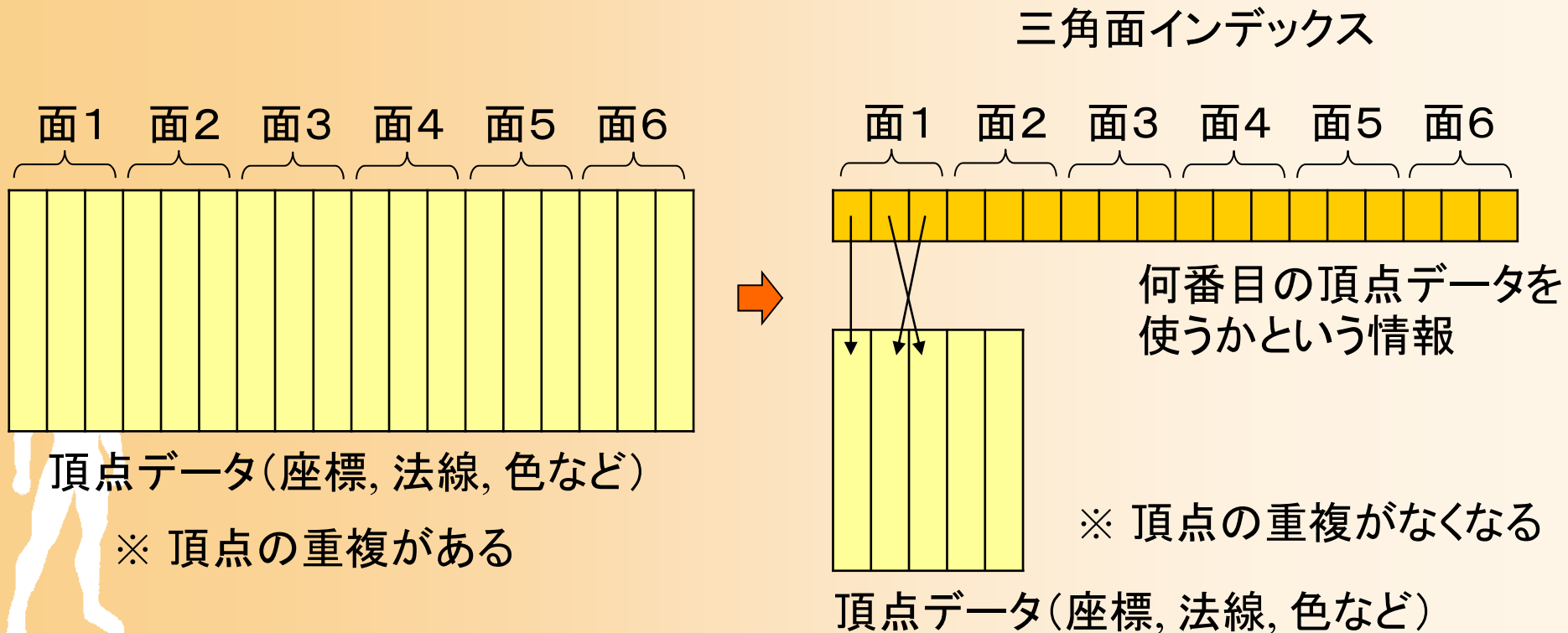
ポリゴンモデルの描画方法

- 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



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();
}
```

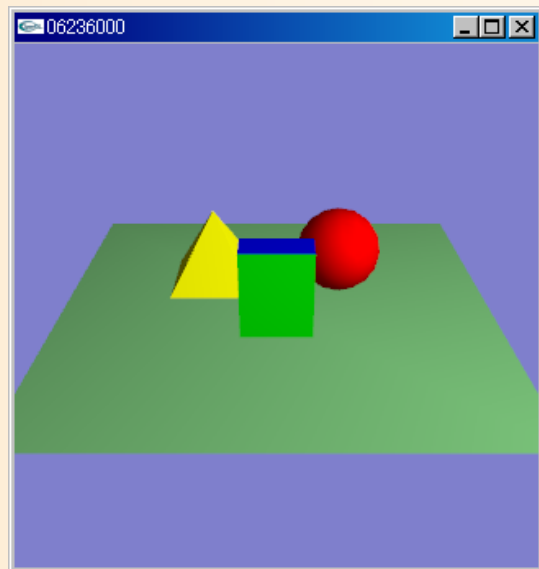
面の法線を指定
(i番目の面のデータを指定)

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)

演習課題

- 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



視点操作の実現方法(復習)



現在の視点操作の実現方法

- マウスの右ボタンを押しながら、上下にドラッグすると、カメラの回転角度(仰角)が変化
- カメラの回転角度を表す変数 `camera_pitch` を定義
- マウス操作に応じて、`camera_pitch` の値を変化
- `camera_pitch` に応じて、変換行列を設定



視点操作のための変数

- 視点操作のための変数の定義
 - グローバル変数(全ての関数からアクセス可能な変数)として定義

```
// 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
```

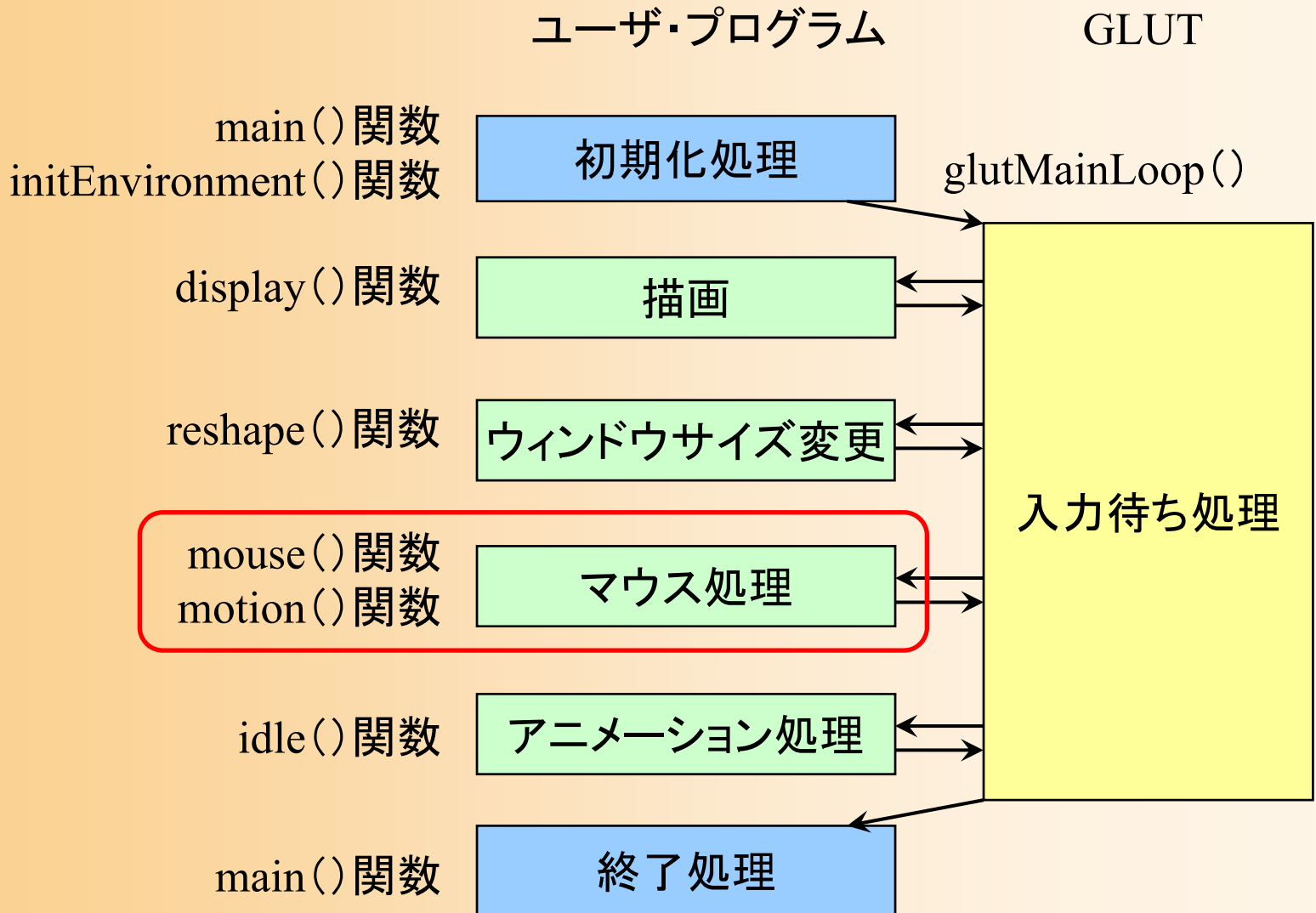
```
int drag_mouse_r = 0; // 右ボタンをドラッグ中かどうかのフラグ  
(0:非ドラッグ中,1:ドラッグ中)
```

```
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
```

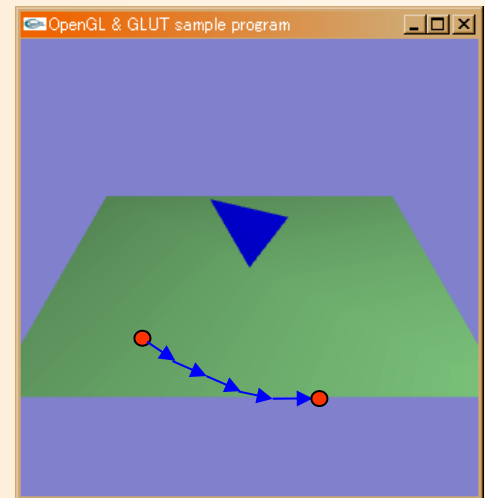


サンプルプログラムの構成



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
 - GLUTに再描画を指示(適切なタイミングで再描画が実行される)

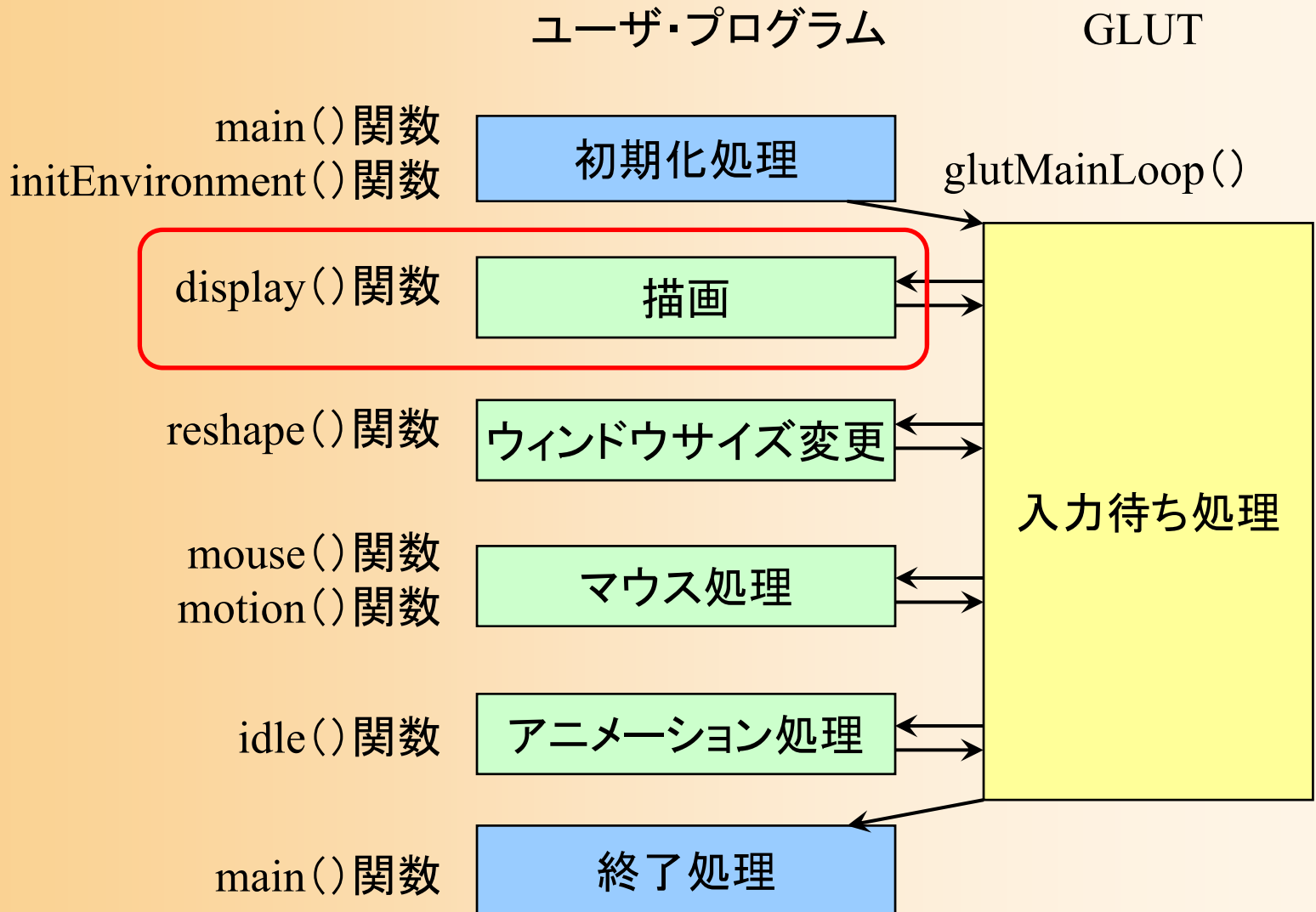
```
// 今回のマウス座標を記録
last_mouse_x = mx;
last_mouse_y = my;

// 再描画の指示を出す
glutPostRedisplay();
```

```
}
```

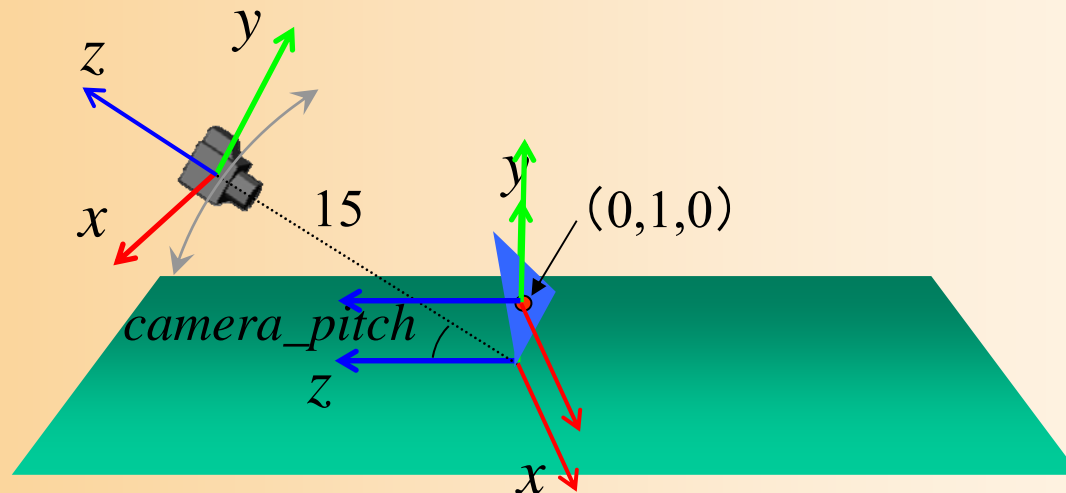


サンプルプログラムの構成



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_pitch`
 - 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

$$\begin{matrix} \textcircled{3} & & \textcircled{2} & & \textcircled{1} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & & & \\ & \cos(-camera_pitch) & & \\ & \sin(-camera_pitch) & & \\ & & & 1 \end{pmatrix} & \begin{pmatrix} & & & \\ & & & \\ & & & \\ 1 & & & \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \end{matrix}$$

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

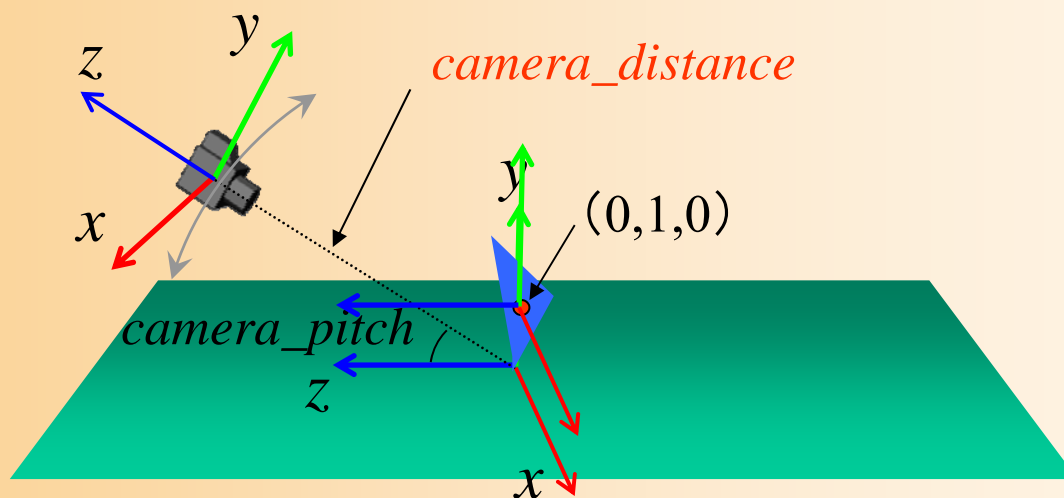


視点操作の拡張



視点操作の拡張

- 左ドラッグでカメラと注視点の距離を操作できるように拡張
 - カメラと注視点の距離を変数 `camera_distance` で表す



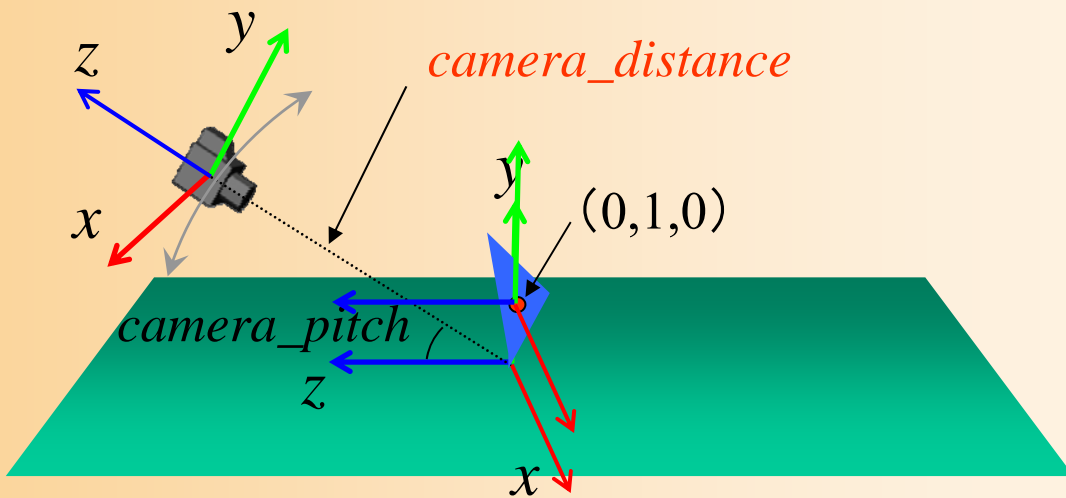
視点操作の拡張

- 変換行列

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

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



視点操作の拡張

- プログラムの修正箇所(多いので注意)
 - 左ボタンの押下状態を記録する変数を追加
 - カメラと原点の距離を記録する変数を追加
 - mouse()関数に、左ボタンの押下状態を更新する処理を追加
 - motion()関数に、左ドラッグに応じて camera_distance を変更する処理を追加
 - 一定値以上は近づかないように制限
 - display()関数を、camera_distance に応じて変換行列を設定するように変更





変換行列によるアニメーション

変換行列によるアニメーション

- 変換行列を組み合わせることで、さまざまな運動を実現できる
- アニメーション処理 (idle()関数)
 - 運動を表す媒介変数の変化を記述
- 描画処理 (display()関数)
 - 媒介変数の値に応じて、回転角度や移動距離に応じた変換行列を設定



サンプルプログラムの構成

回転角度に応じた変換行列の設定、ポリゴンモデル描画の処理を追加

回転角度を変化させる処理を追加

ユーザ・プログラム

GLUT

main()関数

environment()関数

display()関数

mouse()関数

motion()関数

idle()関数

main()関数

初期化処理

描画

マウス処理

アニメーション処理

終了処理

glutMainLoop()

入力待ち処理



アニメーションに使用する変数

- 資料に従って、いくつかの媒介変数を追加
 - theta_cycle
 - 0~360 へ単調増加、360 になったら 0 に戻る
 - theta_repeat
 - 0~180 の間を往復、180 になったら減少を始める
 - theta_cycle から計算できる
 - move
 - 0~1 の間を加速度つきで往復
 - 1に近づくと速度が減少
 - theta_cycle2, theta_repeat2



ポリゴンの回転のための変数

- 変数定義(先頭)、変数の変化(idle()関数)

```
// アニメーションのための変数  
float theta_cycle = 0.0;
```

```
void idle( void )  
{  
    // theta_cycle を 0~360 まで繰り返し変化させる  
    // (360まで来たら0に戻る)  
    theta_cycle += 0.1;  
    if ( theta_cycle > 360 )  
        theta_cycle -= 360;  
  
    // 再描画の指示を出す(描画関数が呼ばれる)  
    glutPostRedisplay();  
}
```

1回の呼び出しの度にどれだけ回転を行うか、実行環境に応じて適当な値を設定



アニメーションの例

- 一定速度で回転運動
- 一定位置で回転運動
- 一定速度で回転運動(常に正面を向く)
- 一定速度で往復回転運動
- 一定速度で上下に往復移動運動
- 加速度つきで上下に往復移動運動
- 複数の物体の運動の組み合わせ



例1:一定速度で回転運動

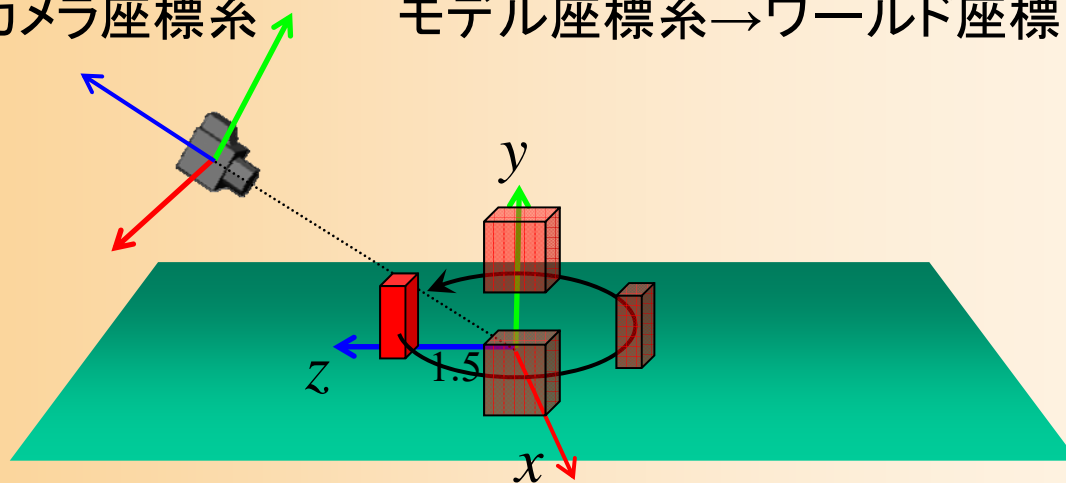
- 移動→回転の順に適用

- 移動にも回転が適用されるので、半径1.5で回転

$$\left(\begin{array}{c} \mathbf{M} \end{array} \right) \left(\begin{array}{cccc} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{array} \right) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

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



例1:一定速度で回転運動

- 行列と同じ順序で、回転・平行移動を適用

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} \cos(\theta_{cycle}) & 0 & \sin(\theta_{cycle}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{cycle}) & 0 & \cos(\theta_{cycle}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

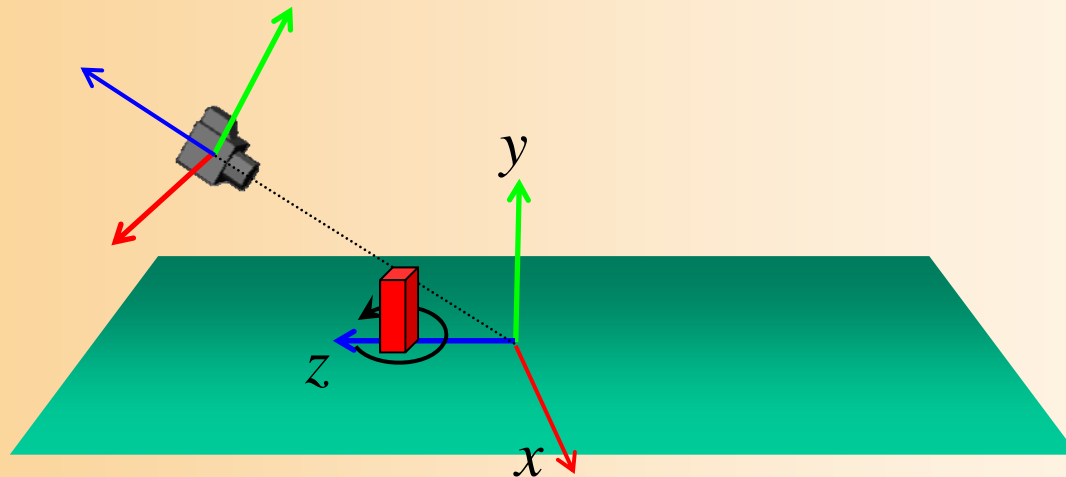
```
void display( void )
{
    .....
    // 例1:一定速度で回転運動
    glRotatef( theta_cycle, 0.0f, 1.0f, 0.0f );
    glTranslatef( 0.0f, 0.0f, 1.5f );
    renderCube();
    .....
}
```



例2:一定位置で回転運動

- 回転→移動の順に適用(順序を逆)
 - 常に同じ位置に移動するので、その場で回転

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta_{\text{cycle}}) & 0 & \sin(\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}}) & 0 & \cos(\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$



例2:一定位置で回転運動

- 回転→移動の順に適用(順序を逆)
 - 常に同じ位置に移動するので、その場で回転

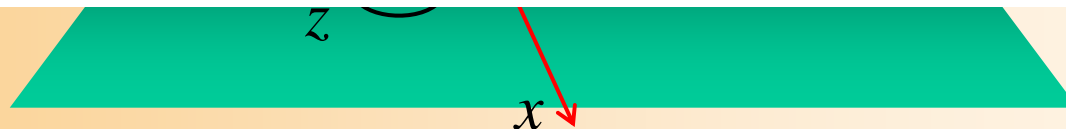
$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

// 例2:一定位置で回転

```
glTranslatef( 0.0f, 0.0f, 1.5f );
```

```
glRotatef( theta_cycle, 0.0f, 1.0f, 0.0f );
```

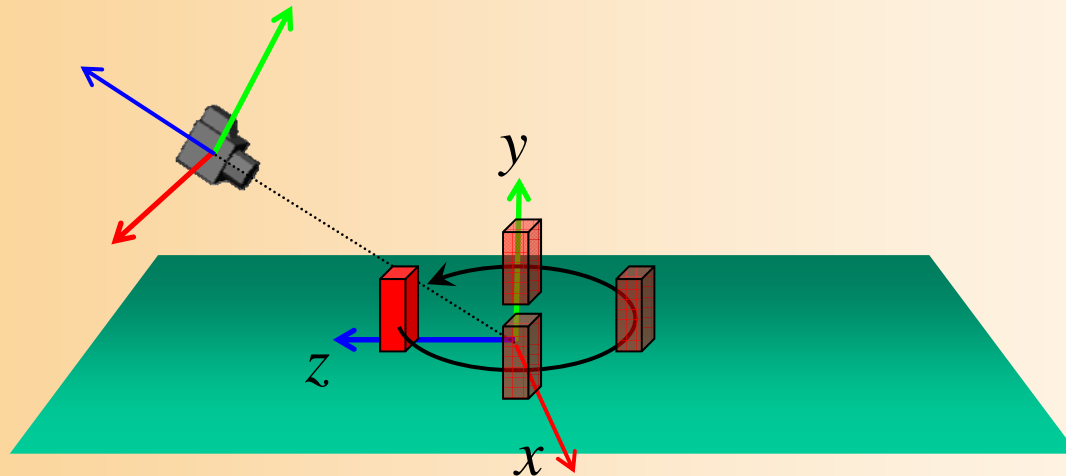
```
renderCube();
```



例3:一定速度で回転運動2

- 常に正面を向くようにするためには？
 - 最初に逆方向に回転しておくことで、次の回転をキャンセル（移動にのみ回転がかかる）

$$\mathbf{M} \begin{pmatrix} \cos(\theta_{\text{cycle}}) & 0 & \sin(\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}}) & 0 & \cos(\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(-\theta_{\text{cycle}}) & 0 & \sin(-\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\theta_{\text{cycle}}) & 0 & \cos(-\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$



例3:一定速度で回転運動2

- 常に正面を向くようにするためには？
 - 最初に逆方向に回転しておくことで、次の回転をキャンセル（移動にのみ回転がかかる）

$$\mathbf{M} \begin{pmatrix} \cos(\theta_{\text{cycle}}) & 0 & \sin(\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}}) & 0 & \cos(\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(-\theta_{\text{cycle}}) & 0 & \sin(-\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\theta_{\text{cycle}}) & 0 & \cos(-\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

// 例3:一定速度で回転運動(常に正面を向く)

glRotatef(theta_cycle, 0.0f, 1.0f, 0.0f);

glTranslatef(0.0f, 0.0f, 1.5f);

glRotatef(- theta_cycle, 0.0f, 1.0f, 0.0f);

renderCube();

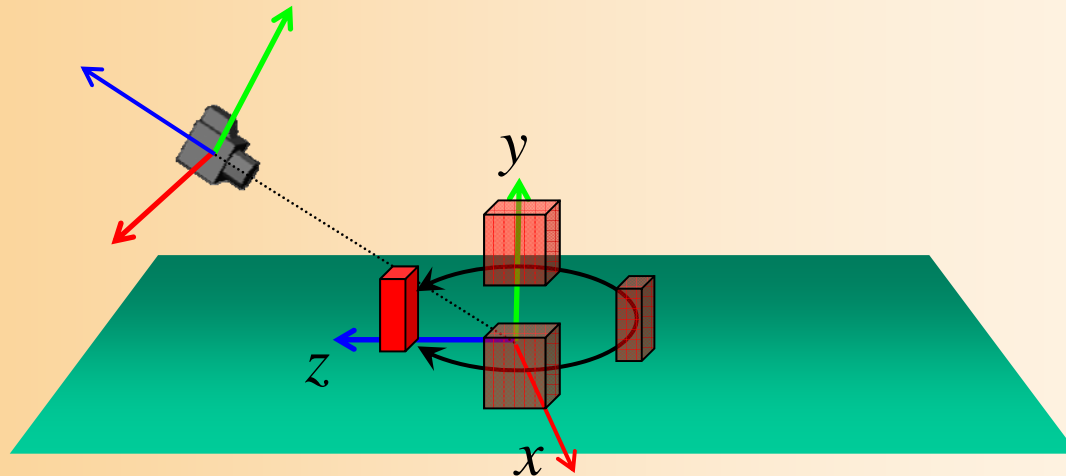


x

例4:一定速度で往復回転運動

- 変換行列は例1と同じ、異なる変数を使用
 - 変数の変化 (`idle()` 関数) と変換行列の設定 (`display()` 関数) の組み合わせが重要

$$\mathbf{M} \begin{pmatrix} \cos(\theta_{repeat}) & 0 & \sin(\theta_{repeat}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{repeat}) & 0 & \cos(\theta_{repeat}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$



例4: 一定速度で往復回転運動

- 変換行列は例1と同じ、異なる変数を使用
 - 変数の変化 (idle() 関数) と変換行列の設定 (display() 関数) の組み合わせが重要

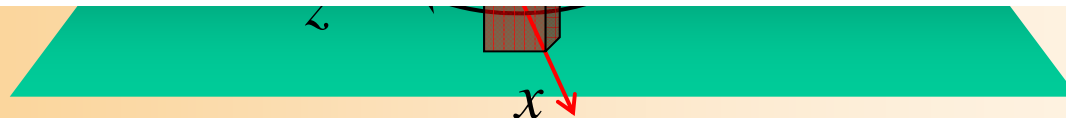
$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} \cos(\theta_{repeat}) & 0 & \sin(\theta_{repeat}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{repeat}) & 0 & \cos(\theta_{repeat}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

// 例4: 一定速度で往復回転運動

glRotatef(theta_repeat, 0.0f, 1.0f, 0.0f);

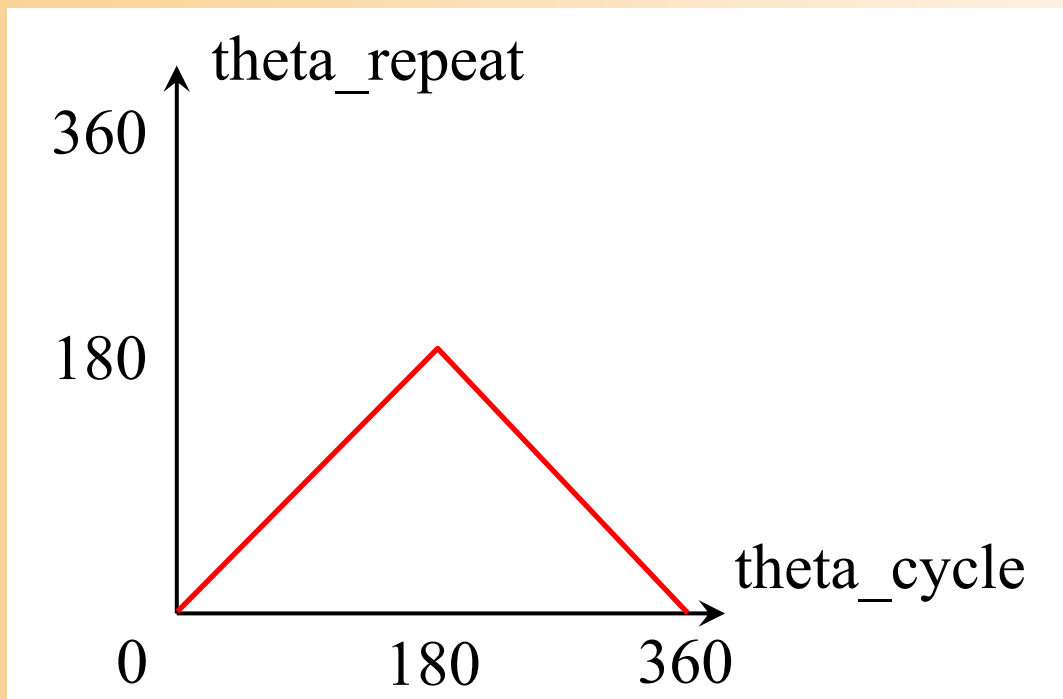
glTranslatef(0.0f, 0.0f, 1.5f);

renderCube();



例4:一定速度で往復回転運動

- アニメーション用の変数を追加
 - 0~180 の間で反復変化 (θ_{repeat})
 - 0~360 で繰り返し変化する変数 (θ_{cycle}) から計算



例4:一定速度で往復回転運動

- アニメーション用の変数を追加
 - 0~180 の間で反復変化 (theta_repeat)
 - 0~360 で繰り返し変化する変数 (theta_cycle) から計算

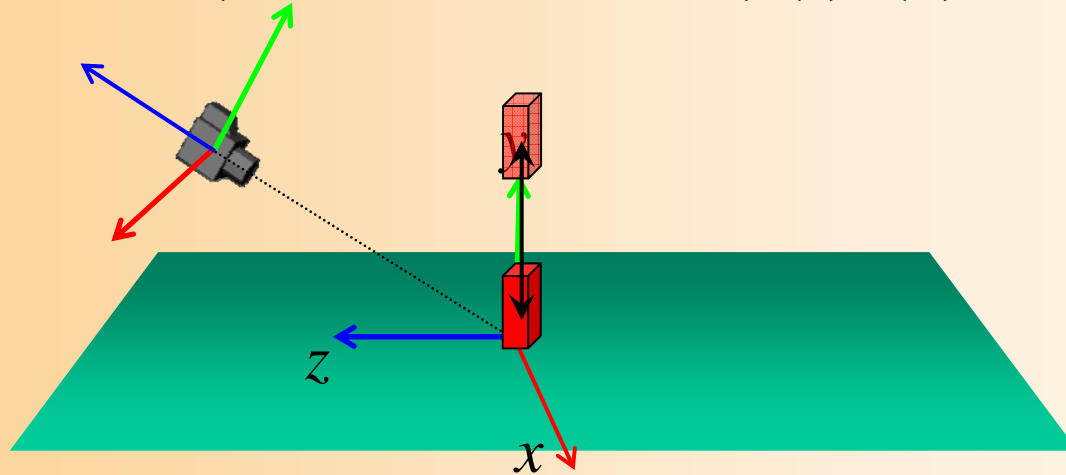
```
void idle( void )
{
    .....
    // theta_repeat を 0~180 の間で反復変化させる
    // (180まで増加したら0まで減少する)
    if ( theta_cycle <= 180 )
        theta_repeat = theta_cycle;
    else
        theta_repeat = 360 - theta_cycle;
    .....
}
```



例5:一定速度で上下に往復移動

- 回転だけではなく、位置に変数を使用することもできる
 - めり込みを避けるために y座標値を +1 している

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \text{theta_repeat}/180+1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$



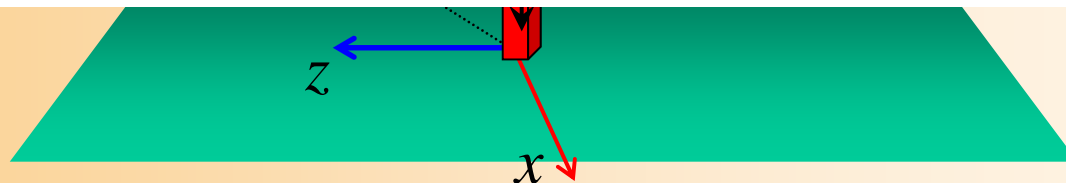
例5:一定速度で上下に往復移動

- 回転だけではなく、位置に変数を使用することもできる
 - めり込みを避けるために y座標値を +1 している

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \text{theta_repeat}/180+1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

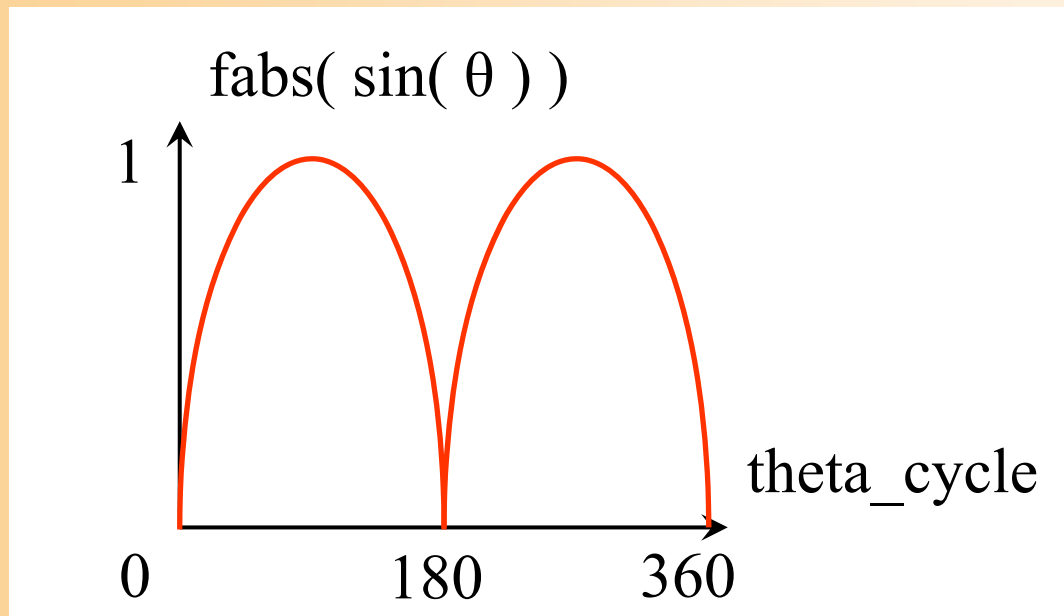
// 例5:一定速度で上下に往復移動運動

```
glTranslatef( 0.0f, theta_repeat / 180.0 + 1.0f, 0.0f );  
renderCube();
```



例6: 加速度つきで上下に往復運動

- 変数の変化を工夫することで、移動速度を変化させるようなこともできる
 - ここでは三角関数の絶対値 (fabs関数) を利用
 - 0~360 から 0~1 の間で変化する変数に変換



例6: 加速度つきで上下に往復運動

- 変数の変化を工夫することで、移動速度を変化させるようなこともできる
 - ここでは三角関数の絶対値 (fabs関数) を利用
 - sin関数の引数の範囲は $0 \sim 2\pi$ なので、 $0 \sim 360$ の範囲を $0 \sim 2\pi$ の範囲に変換


```
void idle( void )
{
    // move を 0~1 の間で反復変化させる
    //(三角関数を用いることで、一定速度でではなく、
    // 0 の近くで速度が小さく
    // 180 の近くで速度が大きくなるように変化させる)
    move = fabs( sin( theta_cycle * 3.1415926 / 180.0 ) );
}
```



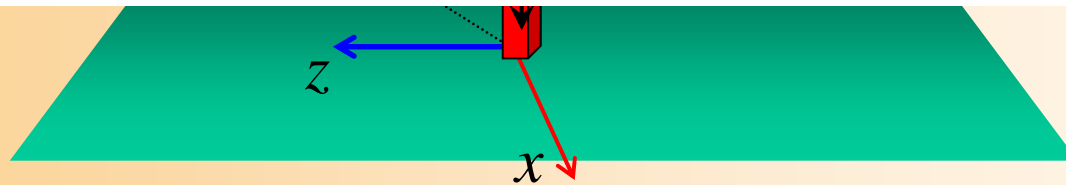
例6: 加速度つきで上下に往復運動

- 例5と同様の変換行列を使用

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & move+1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$



```
// 例6: 加速度つきで上下に放物往復移動運動  
glTranslatef( 0.0f, move + 1.0f, 0.0f );  
renderCube();
```



例7: 複数の物体の運動

- それぞれ異なる変換行列を使用して描画

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

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} \cos(\theta_{\text{cycle}}) & 0 & \sin(\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}}) & 0 & \cos(\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} \cos(\theta_{\text{cycle}2}) & 0 & \sin(\theta_{\text{cycle}2}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}2}) & 0 & \cos(\theta_{\text{cycle}2}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

ワールド座標系 → カメラ座標系 (共通)



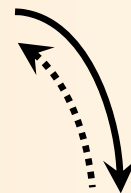
変換行列の退避・復元(1)

- 現在の変換行列を別領域(スタック)に記録しておき、後から復元することができる

- `glPushMatrix()`

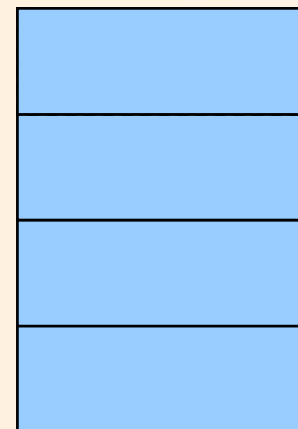
- 現在の変換行列の退避
- スタックに積む

現在OpenGLに
設定されている
変換行列



- `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



変換行列の退避・復元(2)

- `glPushMatrix()`, `glPopMatrix()` を複数回呼び出すと、保存と逆の順番で復元される
 - プログラミングの講義で学習したスタックを覚えていなければ、復習しておくこと
 - 変換行列の設定に適している
- `glPushMatrix()`, `glPopMatrix()` を使用する
ときの注意
 - 必ず両者の呼び出しの数が同じになるようにすること
 - Popをしないまま、Pushを呼び出し続けると、スタックが溢れてエラーとなる



変換行列の退避・復元の例

- ワールド座標系からカメラ座標系への変換行列を設定

$$\left(\begin{array}{l} \text{World} \rightarrow \\ \text{Camera} \end{array} \right)$$

- 地面を描画
- 行列を退避

- 物体1からワールド座標系への変換行列

$$\left(\begin{array}{l} \text{World} \rightarrow \\ \text{Camera} \end{array} \right) \left(\begin{array}{l} \text{Obj1} \rightarrow \\ \text{World} \end{array} \right)$$

- 物体1を描画

- 行列を回復

$$\left(\begin{array}{l} \text{World} \rightarrow \\ \text{Camera} \end{array} \right)$$

- 物体2からワールド座標系への変換行列

$$\left(\begin{array}{l} \text{World} \rightarrow \\ \text{Camera} \end{array} \right) \left(\begin{array}{l} \text{Obj2} \rightarrow \\ \text{World} \end{array} \right)$$

- 物体2を描画



例7: 複数の物体の運動

- それぞれ異なる変換行列を使用して描画

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

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} \cos(\theta_{\text{cycle}}) & 0 & \sin(\theta_{\text{cycle}}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}}) & 0 & \cos(\theta_{\text{cycle}}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{M} \end{pmatrix} \begin{pmatrix} \cos(\theta_{\text{cycle}2}) & 0 & \sin(\theta_{\text{cycle}2}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_{\text{cycle}2}) & 0 & \cos(\theta_{\text{cycle}2}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

ワールド座標系 → カメラ座標系 (共通)



例7: 複数の物体の運動

- アニメーション用の変数を追加

```
void idle( void )
{
    .....
    // theta_cycle2 を theta_cycle と同様に2倍の速度で変化させる
    theta_cycle2 += 2.0;
    if ( theta_cycle2 > 360 )
        theta_cycle2 -= 360;

    // theta_repeat2 を theta_repeat と同様に2倍の速度で変化させる
    if ( theta_cycle2 <= 180 )
        theta_repeat2 = theta_cycle2;
    else
        theta_repeat2 = 360 - theta_cycle2;
}
```

例7: 複数の物体の運動

```
// 例7: 2つの物体を描画(異なる周期で往復回転運動)
```

```
glPushMatrix();
```

```
glRotatef( theta_cycle2, 0.0, 1.0, 0.0 );
```

```
glTranslatef( 0.0, 0.0, 3.0 );
```

```
renderCube();
```

```
glPopMatrix();
```

```
glRotatef( theta_cycle, 0.0f, 1.0f, 0.0f );
```

```
glTranslatef( 0.0f, 0.0f, 1.5f );
```

```
renderCube();
```



例8: 複数の物体の運動2(1)

- Push, Pop の有無による違いを確認
 - 物体1: 回転運動
 - 物体2: 加速度つきで上下に放物往復移動運動

```
// 例8: 2つの物体を描画  
glRotatef( theta_cycle2, 0.0, 1.0, 0.0 );  
glTranslatef( 0.0, 0.0, 3.0 );  
renderCube();
```

```
glTranslatef( 0.0f, move + 2, 0.0 );  
renderCube();
```



例8: 複数の物体の運動2(2)

- Push, Pop の有無による違いを確認
 - 物体1: 回転運動
 - 物体2: 加速度つきで上下に放物往復移動運動

```
// 例9: 2つの物体を描画
glPushMatrix();
glRotatef( theta_cycle2, 0.0, 1.0, 0.0 );
glTranslatef( 0.0, 0.0, 3.0 );
renderCube();
glPopMatrix();

glTranslatef( 0.0f, move + 2, 0.0 );
renderCube();
```





演習課題

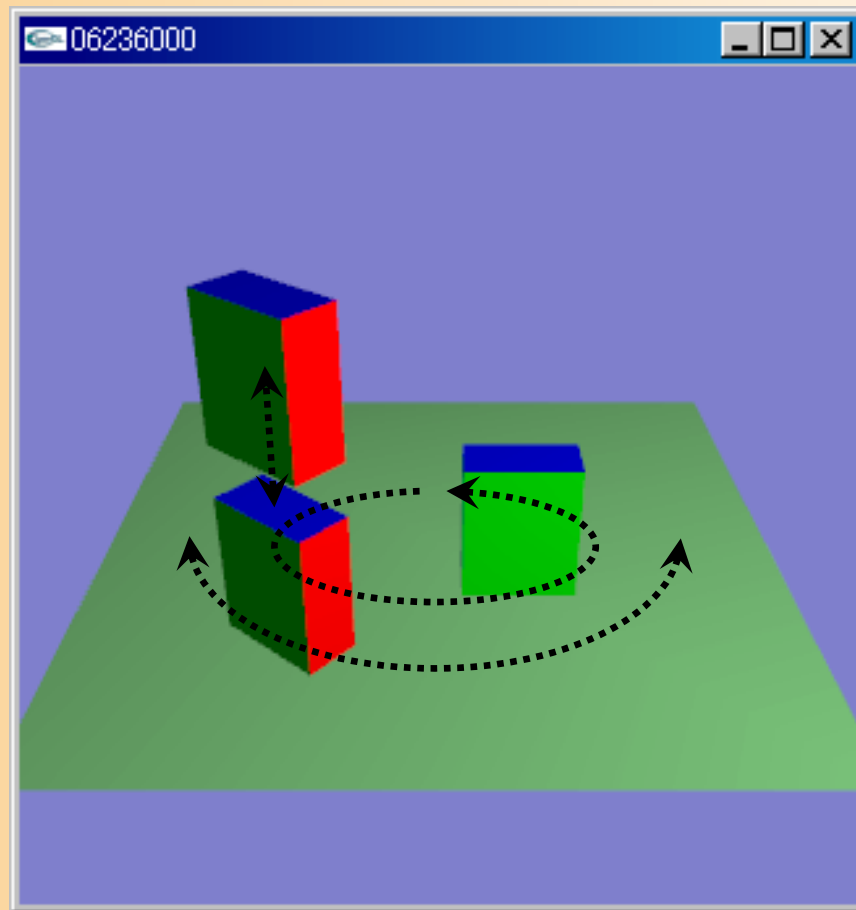
演習課題

- 今までの内容を踏まえて、以下のような、3つの直方体のアニメーションを実現する
 - 1つ目の直方体は、常に正面を向いたまま、原点を中心とする半径1.5の円周上を、等速回転運動する。
 - 2つ目の直方体は、原点を中心とする半径3.0の半円上を、等速往復回転運動する。
 - 3つ目の直方体は、2つ目の直方体の上で、上下に方物往復移動運動する。



演習課題

- 演習課題のアニメーション



演習課題

```
void display( void )
```

```
{
```

```
    glPushMatrix() or glPopMatrix() or どちらも入れない
```

```
    // 1つ目の直方体の描画（回転運動）
```

```
    ?
```

```
    glPushMatrix() or glPopMatrix() or どちらも入れない
```

```
    // 2つ目の直方体の描画（往復回転運動）
```

```
    ?
```

```
    glPushMatrix() or glPopMatrix() or どちらも入れない
```

```
    // 3つ目の直方体の描画（方物往復移動運動）
```

```
    ?
```

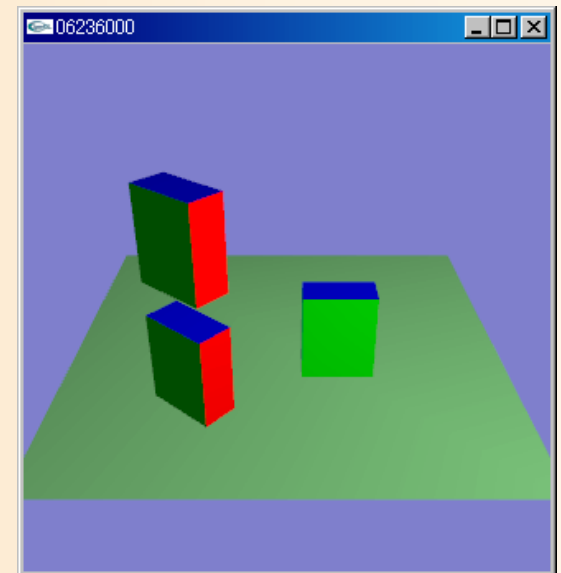
```
    glPushMatrix() or glPopMatrix() or どちらも入れない
```

```
}
```



演習課題

- ここまでの演習をテキストに従って行う
 - 前回の演習のファイルを、引き続き修正
 - プログラムの空欄は、自分で考えて、指定どおりのアニメーションが実現されるようにする



演習課題の提出

- 視点操作の拡張
- アニメーションの実現
 - 例1～9のアニメーションもコメントアウトして残す
- 上記の両方の課題を実現した、一つのプログラムを、Moodleから提出
 - ファイル名は、学生番号.cとする
 - 必ず、両方とも完成したプログラムを提出すること(部分点はなし)
 - 時間内に終わらなければ、締め切りまでに提出
- 締め切り 7月20日(金) 18:00 (厳守)



まとめ

- 前回の復習
- 前回の演習の復習
- 視点操作の実現方法(復習)
- 視点操作の拡張
- 変換行列によるアニメーション
- 演習課題



次回予告

- シェーディング
 - 光のモデル
 - スムーズシェーディング
- レポート課題

