2015/6/23

コンピュータグラフィックスＳ 演習資料

第３回　変換行列

九州工業大学 情報工学部 システム創成情報工学科
講義担当：尾下真樹

1. 視点操作の拡張

今回の演習では、まず、前回までに演習で作成していたプログラム（ポリゴンモデルの描画を追加したプログラム）に、変換行列を使った視点操作の処理を拡張する。
最初のサンプルプログラムでは、マウスを前後に右ドラッグすることで視点を上下に回転させる処理が実現されていた。その処理を参考に、今度は、マウスを前後に左ドラッグすることで、視点と物体の距離を調節できるような機能を追加する。

まず、プログラムに、マウスの左ボタンが押されているかどうかの状態を記録するためのグローバル変数、中心からカメラ（視点）への距離を記録するグローバル変数、をそれぞれ追加する。

```
// マウスのドラッグのための変数
int       drag_mouse_l = 0;     // 左ボタンがドラッグ中かどうかのフラグ（1:ドラッグ中, 0:非ドラッグ中）
int       drag_mouse_r = 0;     // 右ボタンがドラッグ中かどうかのフラグ（1:ドラッグ中, 0:非ドラッグ中）
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
```

```
// 視点操作のための変数
float     camera_pitch = 0.0;       // Ｘ軸を中心とする回転角度
float     camera_distance = 15.0;   // 中心からカメラの距離
```

次に、マウスの操作に応じて押下状態を変更する処理、マウス操作に応じて視点の距離を変更する処理、を追加する。

```
//
//   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 )
{
        // 左ボタンが押されたらドラッグ開始
        if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_DOWN ) )
                drag_mouse_l = 1;
        // 左ボタンが離されたらドラッグ終了
        else if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_UP ) )
                drag_mouse_l = 0;

        // 右ボタンが押されたらドラッグ開始
        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
        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;
}
```

```
//
//   マウスドラッグ時に呼ばれるコールバック関数
//
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 )
        {
                // 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 > 90.0 )
                        camera_pitch = 90.0;
        }

        // 左ボタンのドラッグ中であれば、マウスの移動量に応じては視点の距離を変更する
        if ( drag_mouse_l )
        {
                // マウスの縦移動に応じて視点の距離を変更
                camera_distance += ( my - last_mouse_y ) * 0.2;
                if ( camera_distance < 2.0 )
                        camera_distance = 2.0;
        }

        // 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（この後で再描画のコールバック関数が呼ばれる）
        glutPostRedisplay();
}
```

視点の距離を変更するときに、視点が中心に近づきすぎたり、距離が負になってしまったりすると、不自然な画面となるので、距離が一定値以下（ここでは 2.0）にはならないように制限を加えている。
以上の処理によって、マウスを右ドラッグする度に、視点の距離（camera_distance）が変化する。

ここままでは、画面の描画には反映されないので、視点の距離（camera_distance）に応じて画面が描画されるように、変換行列を設定する。

```
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 );
        glLoadIdentity();
        glTranslatef( 0.0,   0.0, - camera_distance );
        glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

        ・・・・・・
}
```

プログラムをコンパイル・実行して、左ドラッグにより視点の距離が変化することを確認せよ。

2. 変換行列によるアニメーション

回転や移動の変換行列を組み合わせることで、さまざまな動きを実現できる。
ここでは、以前に作成した直方体の描画と、変換行列の組み合わせをつかって、アニメーションが実現する。

まずは、プログラムの先頭に、回転角度を記録するためのグローバル変数（theta_cycle）を追加する。なお、変数の初期値は 0 としている。

```
#include <windows.h>
#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>


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

// 視点操作のための変数
float    camera_pitch = 0.0;          // X 軸を中心とする回転角度
float    camera_distance = 15.0;   // 中心からカメラの距離
```

次に、アイドル関数（idle（）関数）に、この変数 theta_cycle を少しずつ変化させるような処理を追加する。
ここでは、回転を実現するため、theta_cycle を 0～360 に向かって少しずつ増加させることにする。また、theta_cycle が 360 になったら、0 に戻るようにする。
回転を行う度に画面全体が再描画されるように、GLUT に再描画を指示する glutPotRedisplay（）関数を実行するのを忘れないようにする。

```
//
//   アイドル時に呼ばれるコールバック関数
//
void   idle( void )
{
        // アニメーション用の変数を変化させる

        // 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（この後で再描画のコールバック関数が呼ばれる）
        glutPostRedisplay();
}
```

以上の追加により、変数 theta_cycle を 0～360 の間で変化させながら描画処理が行われるようになったので、次は、変数 theta_cycle に応じて立方体が回転した状態で描画されるように、変換行列の設定処理を変更する。

```
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, - camera_distance );
        glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );
        glTranslatef( 0.0, -1.0, 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 );

        // 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();
/*
        今までの描画処理は、全てコメントアウト
*/
        // 例 1：一定速度で回転運動
        glRotatef( theta_cycle, 0.0, 1.0, 0.0 );
        glTranslatef( 0.0, 0.0, 3.0 );
        renderCube();

        // 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();
}
```

直方体の描画には、前回の演習で作成した renderCube() 関数を利用する。
以上の修正を行って、プログラムをコンパイル・実行し、回転運動を行う直方体が正しく表示されることを確認せよ。

次に、上の直方体の描画処理の、回転行列と移動行列の適用順序を入れ替えて、以下のような描画処理に変更して実行し、どのように動きが変化するかを確認してみよ。
（例 1 の描画処理をコメントアウトして、以下の例 2 の描画処理に置き換える。）

```
        ・・・・・・

        // 例 2：一定位置で回転
        glTranslatef( 0.0, 0.0, 3.0 );
        glRotatef( theta_cycle, 0.0, 1.0, 0.0 );
        renderCube();

        ・・・・・・
```

以上のアニメーションでは、物体自身が回転していた。物体が回転運動をして、なおかつ物体自身は回転させないようにするためには、回転運動の後に、もう一度逆方向の回転変換を適用し、回転運動による回転を打ち消すと良い。描画処理を以下のように変更して、実行してみよ。

```
        // 例 3：一定速度で回転運動（常に正面を向く）
        glRotatef( theta_cycle, 0.0, 1.0, 0.0 );
        glTranslatef( 0.0, 0.0, 3.0 );
        glRotatef( - theta_cycle, 0.0, 1.0, 0.0 );
        renderCube();
```

ここまでのアニメーションは、0～360 の間を繰り返し変化する theta_cycle 変数を使って繰り返し回転運動を実現した。次に、往復運動を実現するため、0～180 の間を往復変化する theta_repeat 変数を追加し、この変数を使ってアニメーションを実現する。
ここまでの修正と同様に、以下のようなグローバル変、idle 関数の処理、display 関数の処理をそれぞれ追加して、例 4 のアニメーションの実行結果を確認してみよ。

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

```
void   idle( void )
{
        // アニメーション用の変数を変化させる

        // 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;

        // 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;

        // 再描画の指示を出す（この後で再描画のコールバック関数が呼ばれる）
        glutPostRedisplay();
}
```

```
void   display( void )
{
        ・・・・・・

        // 例 4：一定速度で往復回転運動
        glRotatef( theta_repeat - 90, 0.0, 1.0, 0.0 );
        glTranslatef( 0.0, 0.0, 3.0 );
        renderCube();

        ・・・・・・
}
```

次に、同じ変数を使って、回転運動だけではなく、移動のアニメーションも試してみる。

```
void   display( void )
{
        ・・・・・・

        // 例 5：一定速度で上下に往復移動運動
        glTranslatef( 0.0f, theta_repeat / 180.0, 0.0 );
        renderCube();

        ・・・・・・
}
```

一定速度での運動だけではなく、上下に移動するときに、下に行くに従って動きが速くなるような放物運動を実現してみる。
ここでは、位置を表す変数 thete_repeat を一定の速度で変化させるのではなく、三角関数を使用して、移動量を表す変数 move を計算する。

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

```
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、一定速度でなはなく、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 ) );

        ・・・・・・
}
```

```
void   display( void )
{
        ・・・・・・

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

        ・・・・・・
}
```

最後に、変換行列の待避・復元（glPushMatrix() 関数・glPopMatrix() 関数）を使うことで、変換行列が異なる複数のオブジェクトのアニメーションを実現する方法を試してみる。
異なる周期でオブジェクトを回転させるために、新たな変数（theta_cycle2, theta_repeat2）を追加する。

```
// アニメーションのための変数
float    theta_cycle = 0.0;
float    theta_repeat = 0.0;
float    move = 0.0;
float    theta_cycle2 = 0.0;
float    theta_repeat2 = 0.0;
```

```
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 += 0.2;
        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;

        ・・・・・・
}
```

```
void   display( void )
{
        ・・・・・・
```

```
        // 例７：２つの物体を描画（異なる周期で回転運動）
        glPushMatrix();
                glRotatef( theta_cycle2, 0.0, 1.0, 0.0 );
                glTranslatef( 0.0, 0.0, 3.0 );
                renderCube();
        glPopMatrix();

        glRotatef( theta_cycle, 0.0, 1.0, 0.0 );
        glTranslatef( 0.0, 0.0, 1.5 );
        renderCube();

        ・・・・・・
}
```

glPopMatrix() 関数を呼び出すことで、glPushMatrix() 関数を呼び出したときの変換行列を復元することができ、結果的に、複数の物体の動きを独立に設定することが可能となる。
以下の２つの例を試して、glPushMatrix() 関数・glPopMatrix() 関数の有無により、全体のアニメーションがどのように変化するかを確認せよ。

```
void   display( void )
{
        ・・・・・・

        // 例８：２つの物体を描画（回転運動、加速度つきで上下に放物往復移動運動）
        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();

        ・・・・・・
}
```

```
void   display( void )
{
        ・・・・・・

        // 例９：２つの物体を描画（回転運動、加速度つきで上下に放物往復移動運動）
        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();

        ・・・・・・
}
```

3. 演習課題

最後に、ここまで学習した内容のまとめとして、以下のような、３つの直方体の運動を実現するように、プログラムを修正してみる。

- 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。

Moodle に、上記のアニメーションを実現したサンプルプログラム opengl_sample3 をアップロードしているので、参考にすること。Moodle からプログラムをダウンロードした後で、

chmod u+x opengl_sample3

のようなコマンドを実行し、ダウンロードしたファイルに実行権限を与えてから、実行する。

以下のように、3 つの直方体の動きを実現する処理を記述し、適切な位置に、glPushMatrix() 関数・glPopMatrix() 関数を入れることで、指定された動きを実現できる。
各立方体を描画する処理の中では、変換行列を適切に変更した後で、renderCube 関数を呼び出して直方体を描画する。
（ここまでに作成した変換行列の変更と物体描画のプログラムはコメントアウトして、以下の内容を新たに追加する。）

```
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

        ・・・・・・
}
```

ヒント：
全部を一度に実現できなければ、まずは、上からひとつずつ順番に実現していくと良い。
glPushMatrix() 関数と glPopMatrix() 関数の呼び出し回数は、必ず、一致させる必要がある。今回の例では、それぞれ、全体で１度ずつ使うだけで、指定されたアニメーションを実現できる。